PROFESSIONAL CMAKE

A PRACTICAL GUIDE

CRAIG SCOTT

Exclusively for sbmain17@gmail.com Transaction [illegible]

Professional CMake: A Practical Guide

11th Edition

ISBN 978-1-925904-16-1

© 2018-2021 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.

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

Exclusively for sbmain17@gmail.com Transaction: 0098315591

Table of Contents

Preface . . . 1
Acknowledgments . . . 1
Part I: Fundamentals . . . 3
1. Introduction . . . 4
2. Setting Up A Project . . . 6
2.1. In-source Builds . . . 6
2.2. Out-of-source Builds . . . 7
2.3. Generating Project Files . . . 7
2.4. Running The Build Tool . . . 8
2.5. Recommended Practices . . . 9
3. A Minimal Project . . . 10
3.1. Managing CMake Versions . . . 10
3.2. The project() Command . . . 12
3.3. Building A Basic Executable . . . 13
3.4. Commenting . . . 13
3.5. Recommended Practices . . . 14
4. Building Simple Targets . . . 15
4.1. Executables . . . 15
4.2. Defining Libraries . . . 16
4.3. Linking Targets . . . 17
4.4. Linking Non-targets . . . 19
4.5. Old-style CMake . . . 19
4.6. Recommended Practices . . . 20
5. Variables . . . 22
5.1. Variable Basics . . . 22
5.2. Environment Variables . . . 24
5.3. Cache Variables . . . 24
5.4. Potentially Surprising Behavior Of Variables . . . 26
5.5. Manipulating Cache Variables . . . 27
5.5.1. Setting Cache Values On The Command Line . . . 28
5.5.2. CMake GUI Tools . . . 29
5.6. Printing Variable Values . . . 32
5.7. String Handling . . . 33
5.8. Lists . . . 35
5.8.1. Problems With Unbalanced Square Brackets . . . 37
5.9. Math . . . 38
5.10. Recommended Practices . . . 39
6. Flow Control . . . 40

Exclusively for sbmain17@gmail.com Transaction: 0098315591

6.1. The if() Command . . . 40
6.1.1. Basic Expressions . . . 40
6.1.2. Logic Operators . . . 42
6.1.3. Comparison Tests . . . 42
6.1.4. File System Tests . . . 44
6.1.5. Existence Tests . . . 45
6.1.6. Common Examples . . . 46
6.2. Looping . . . 48
6.2.1. foreach() . . . 48
6.2.2. while() . . . 50
6.2.3. Interrupting Loops . . . 50
6.3. Recommended Practices . . . 51
7. Using Subdirectories . . . 52
7.1. add_subdirectory() . . . 52
7.1.1. Source And Binary Directory Variables . . . 53
7.1.2. Scope . . . 54
7.1.3. When To Call project() . . . 56
7.2. include() . . . 57
7.3. Project-relative Variables . . . 59
7.4. Ending Processing Early . . . 61
7.5. Recommended Practices . . . 62
8. Functions And Macros . . . 64
8.1. The Basics . . . 64
8.2. Argument Handling Essentials . . . 65
8.3. Keyword Arguments . . . 67
8.4. Scope . . . 73
8.5. Overriding Commands . . . 75
8.6. Special Variables For Functions . . . 77
8.7. Other Ways Of Invoking CMake Code . . . 78
8.8. Problems With Argument Handling . . . 81
8.8.1. Parsing Arguments Robustly . . . 83
8.8.2. Forwarding Command Arguments . . . 84
8.8.3. Special Cases For Argument Expansion . . . 86
8.9. Recommended Practices . . . 89
9. Properties . . . 90
9.1. General Property Commands . . . 90
9.2. Global Properties . . . 93
9.3. Directory Properties . . . 94
9.4. Target Properties . . . 95
9.5. Source Properties . . . 96
9.6. Cache Variable Properties . . . 98

Exclusively for sbmain17@gmail.com Transaction: 0098315591

9.7. Other Property Types . . . 98
9.8. Recommended Practices . . . 99
10. Generator Expressions . . . 100
10.1. Simple Boolean Logic . . . 101
10.2. Target Details . . . 103
10.3. General Information . . . 104
10.4. Utility Expressions . . . 105
10.5. Recommended Practices . . . 106
11. Modules . . . 108
11.1. Checking Existence And Support . . . 109
11.2. Other Modules . . . 116
11.3. Recommended Practices . . . 116
12. Policies . . . 118
12.1. Policy Control . . . 118
12.2. Policy Scope . . . 121
12.3. Recommended Practices . . . 122
13. Debugging And Diagnostics . . . 124
13.1. Log Messages . . . 124
13.1.1. Log Levels . . . 124
13.1.2. Message Indenting . . . 126
13.1.3. Message Contexts . . . 127
13.1.4. Check Messages . . . 129
13.2. Print Helpers . . . 131
13.3. Tracing Variable Access . . . 132
13.4. Debugging Generator Expressions . . . 132
13.5. Profiling CMake Calls . . . 134
13.6. Recommended Practices . . . 134
Part II: Builds In Depth . . . 136
14. Build Type . . . 137
14.1. Build Type Basics . . . 137
14.1.1. Single Configuration Generators . . . 138
14.1.2. Multiple Configuration Generators . . . 138
14.2. Common Errors . . . 139
14.3. Custom Build Types . . . 140
14.4. Recommended Practices . . . 144
15. Compiler And Linker Essentials . . . 146
15.1. Target Properties . . . 146
15.1.1. Compiler Flags . . . 146
15.1.2. Linker Flags . . . 148
15.1.3. Target Property Commands . . . 150

Exclusively for sbmain17@gmail.com Transaction: 0098315591

15.2. Directory Properties And Commands . . . 153
15.3. De-duplicating Options . . . 156
15.4. Compiler And Linker Variables . . . 157
15.4.1. Compiler And Linker Variables Are Single Strings, Not Lists . . . 158
15.4.2. Distinguish Between Cache And Non-cache Variables . . . 159
15.4.3. Prefer Appending Over Replacing Flags . . . 160
15.4.4. Understand When Variable Values Are Used . . . 160
15.5. Language-specific Compiler Flags . . . 161
15.6. Recommended Practices . . . 162
16. Language Requirements . . . 164
16.1. Setting The Language Standard Directly . . . 164
16.2. Setting The Language Standard By Feature Requirements . . . 167
16.2.1. Detection And Use Of Optional Language Features . . . 168
16.3. Recommended Practices . . . 169
17. Target Types . . . 171
17.1. Executables . . . 171
17.2. Libraries . . . 172
17.2.1. Basic Library Types . . . 172
17.2.2. Object Libraries . . . 172
17.2.3. Imported Libraries . . . 173
17.2.4. Interface Libraries . . . 175
17.2.5. Interface Imported Libraries . . . 177
17.2.6. Library Aliases . . . 178
17.3. Promoting Imported Targets . . . 180
17.4. Recommended Practices . . . 181
18. Custom Tasks . . . 184
18.1. Custom Targets . . . 184
18.2. Adding Build Steps To An Existing Target . . . 187
18.3. Commands That Generate Files . . . 188
18.4. Configure Time Tasks . . . 192
18.5. Platform Independent Commands . . . 195
18.6. Combining The Different Approaches . . . 198
18.7. Recommended Practices . . . 199
19. Working With Files . . . 201
19.1. Manipulating Paths . . . 201
19.1.1. cmake_path() . . . 201
19.1.2. Older Commands . . . 204
19.2. Copying Files . . . 207
19.3. Reading And Writing Files Directly . . . 215
19.4. File System Manipulation . . . 219
19.5. File Globbing . . . 221

Exclusively for sbmain17@gmail.com Transaction: 0098315591

19.6. Downloading And Uploading . . . 223
19.7. Recommended Practices . . . 225
20. Specifying Version Details . . . 227
20.1. Project Version . . . 227
20.2. Source Code Access To Version Details . . . 229
20.3. Source Control Commits . . . 232
20.4. Recommended Practices . . . 235
21. Libraries . . . 237
21.1. Build Basics . . . 237
21.2. Linking Static Libraries . . . 238
21.3. Shared Library Versioning . . . 238
21.4. Interface Compatibility . . . 240
21.5. Symbol Visibility . . . 245
21.5.1. Specifying Default Visibility . . . 246
21.5.2. Specifying Individual Symbol Visibilities . . . 247
21.6. Mixing Static And Shared Libraries . . . 252
21.7. Recommended Practices . . . 255
22. Toolchains And Cross Compiling . . . 257
22.1. Toolchain Files . . . 258
22.2. Defining The Target System . . . 259
22.3. Tool Selection . . . 260
22.4. System Roots . . . 263
22.5. Compiler Checks . . . 264
22.6. Examples . . . 265
22.6.1. Raspberry Pi . . . 265
22.6.2. GCC With 32-bit Target On 64-bit Host . . . 265
22.7. Android . . . 266
22.7.1. Historical Context . . . 266
22.7.2. Using The NDK . . . 267
22.7.3. Android Studio . . . 268
22.7.4. ndk-build . . . 269
22.7.5. Visual Studio Generators . . . 269
22.8. Recommended Practices . . . 269
23. Apple Features . . . 271
23.1. CMake Generator Selection . . . 271
23.2. Application Bundles . . . 273
23.2.1. Bundle Structure . . . 273
23.2.2. Bundle Info.plist Files . . . 274
23.2.3. Sources, Resources And Other Files . . . 276
23.3. Frameworks . . . 278
23.3.1. Framework Structure . . . 278

Exclusively for sbmain17@gmail.com Transaction: 0098315591

23.3.2. Framework Info.plist Files . . . 280
23.3.3. Headers . . . 280
23.4. Loadable Bundles . . . 281
23.5. Build Settings . . . 282
23.5.1. SDK Selection . . . 282
23.5.2. Deployment Target . . . 283
23.5.3. Device Families . . . 284
23.5.4. Compiler Test Workarounds . . . 284
23.6. Code Signing . . . 285
23.6.1. Signing Identity And Development Team . . . 286
23.6.2. Provisioning Profiles . . . 287
23.6.3. Entitlements . . . 287
23.7. Creating And Exporting Archives . . . 288
23.8. Universal Binaries . . . 290
23.8.1. Device-only Bundles . . . 290
23.8.2. Intel And Apple Silicon Binaries . . . 292
23.8.3. Combined Device And Simulator Binaries . . . 293
23.9. Linking Frameworks . . . 293
23.10. Embedding Frameworks . . . 295
23.11. Limitations . . . 298
23.12. Recommended Practices . . . 298
Part III: The Bigger Picture . . . 301
24. Finding Things . . . 302
24.1. Finding Files And Paths . . . 302
24.1.1. Apple-specific Behavior . . . 307
24.1.2. Cross-compilation Controls . . . 307
24.2. Finding Paths . . . 309
24.3. Finding Programs . . . 309
24.4. Finding Libraries . . . 310
24.5. Finding Packages . . . 312
24.5.1. Package Registries . . . 319
24.5.2. FindPkgConfig . . . 321
24.6. Debugging find_...() Calls . . . 323
24.7. Recommended Practices . . . 324
25. Testing . . . 327
25.1. Defining And Executing A Simple Test . . . 327
25.2. Pass / Fail Criteria And Other Result Types . . . 331
25.3. Test Grouping And Selection . . . 335
25.3.1. Regular Expressions . . . 335
25.3.2. Test Numbers . . . 336

Exclusively for sbmain17@gmail.com Transaction: 0098315591

25.3.3. Labels . . . 336
25.3.4. Repeating Tests . . . 337
25.4. Parallel Execution . . . 339
25.5. Managing Test Resources . . . 340
25.5.1. Defining Test Resource Requirements . . . 342
25.5.2. Specifying Available System Resources . . . 343
25.5.3. Using Resources Allocated To A Test . . . 345
25.6. Test Dependencies . . . 346
25.7. Cross-compiling And Emulators . . . 349
25.8. Build And Test Mode . . . 350
25.9. CDash Integration . . . 352
25.9.1. Key CDash Concepts . . . 353
25.9.2. Executing Pipelines And Actions . . . 354
25.9.3. CTest Configuration . . . 356
25.9.4. Test Measurements And Results . . . 360
25.10. Output Control . . . 362
25.11. GoogleTest . . . 363
25.12. Recommended Practices . . . 368
26. Installing . . . 371
26.1. Directory Layout . . . 372
26.1.1. Relative Layout . . . 372
26.1.2. Base Install Location . . . 374
26.2. Installing Project Targets . . . 376
26.2.1. Interface Properties . . . 382
26.2.2. RPATH . . . 383
26.2.3. Apple-specific Targets . . . 386
26.3. Installing Exports . . . 389
26.4. Installing Imported Targets . . . 393
26.5. Installing Files And Directories . . . 394
26.6. Custom Install Logic . . . 397
26.7. Installing Dependencies . . . 398
26.7.1. Runtime Dependency Sets . . . 398
26.7.2. InstallRequiredSystemLibraries Module . . . 400
26.7.3. BundleUtilities And GetPrerequisites . . . 402
26.8. Writing A Config Package File . . . 402
26.8.1. Config Files For CMake Projects . . . 403
26.8.2. Config Files For Non-CMake Projects . . . 411
26.9. Executing An Install . . . 413
26.10. Recommended Practices . . . 413
27. Packaging . . . 417
27.1. Packaging Basics . . . 417

Exclusively for sbmain17@gmail.com Transaction: 0098315591

27.2. Components . . . 422
27.3. Multi Configuration Packages . . . 426
27.3.1. Multiple Build Directories . . . 426
27.3.2. Pass Multiple Configurations To cpack . . . 427
27.4. Package Generators . . . 428
27.4.1. Simple Archives . . . 429
27.4.2. Qt Installer Framework (IFW) . . . 430
27.4.3. WIX . . . 436
27.4.4. NSIS . . . 437
27.4.5. DragNDrop . . . 439
27.4.6. productbuild . . . 441
27.4.7. RPM . . . 442
27.4.8. DEB . . . 446
27.4.9. FreeBSD . . . 448
27.4.10. Cygwin . . . 448
27.4.11. NuGet . . . 448
27.4.12. External . . . 448
27.5. Recommended Practices . . . 448
28. External Content . . . 451
28.1. ExternalProject . . . 451
28.1.1. Tour Of Main Features . . . 452
28.1.2. Step Management . . . 458
28.1.3. Miscellaneous Features . . . 461
28.1.4. Common Issues . . . 463
28.2. FetchContent . . . 465
28.2.1. Developer Overrides . . . 470
28.2.2. Other Uses For FetchContent . . . 472
28.2.3. Restrictions . . . 473
28.3. ExternalData . . . 473
28.4. Recommended Practices . . . 474
29. Project Organization . . . 476
29.1. Superbuild Structure . . . 476
29.2. Non-superbuild Structure . . . 478
29.3. Common Top Level Subdirectories . . . 482
29.4. IDE Projects . . . 483
29.5. Defining Targets . . . 486
29.5.1. Target Sources . . . 487
29.5.2. Target Outputs . . . 491
29.6. Windows-specific Issues . . . 492
29.7. Cleaning Files . . . 494
29.8. Re-running CMake On File Changes . . . 495

Exclusively for sbmain17@gmail.com Transaction: 0098315591

29.9. Injecting Files Into Projects . . . 495
29.10. Presets . . . 497
29.11. Recommended Practices . . . 506
30. Build Performance . . . 509
30.1. Unity Builds . . . 509
30.1.1. BATCH Mode . . . 510
30.1.2. GROUP Mode . . . 512
30.2. Precompiled Headers . . . 513
30.3. Compiler Caches . . . 514
30.4. Debug-related Improvements . . . 518
30.5. Alternative Linkers . . . 520
30.6. Optimizing Build Dependencies . . . 520
30.7. Recommended Practices . . . 522
31. Working With Qt . . . 524
31.1. Basic Setup . . . 524
31.2. Build Details . . . 526
31.2.1. Standard Library Implementation . . . 526
31.2.2. Position Independent Code . . . 526
31.2.3. Windows GUI Applications . . . 527
31.3. Autogen . . . 527
31.3.1. Moc . . . 528
31.3.2. Widgets . . . 532
31.3.3. Resources . . . 533
31.4. Translations . . . 535
31.5. Deployment . . . 538
31.5.1. Qt Deployment Tools . . . 538
31.5.2. Deploying Translation Files . . . 540
31.5.3. Linux Considerations . . . 542
31.6. Transition To Qt 6 . . . 542
31.7. Recommended Practices . . . 543
Index . . . 545

Exclusively for sbmain17@gmail.com Transaction: 0098315591

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 classic pivotal moment which I now look back on with some 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, but more importantly I was really struck by the positive experience I had while 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.

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 and 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, both 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 handling 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 many 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 number of people generously agreed to review the material in this book, without whom both technical accuracy and readability would have suffered. Any remaining errors and deficiencies are squarely my own. Fellow CMake developers Gregor Jasny and Christian Pfeiffer were valuable

Exclusively for sbmain17@gmail.com Transaction: 0098315591

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 Hetherton, both of whom provided constructive feedback and reminded me of the importance of a fresh set of eyes.

A number of people have generously reviewed chapters for subsequent editions. Many of these people are also CMake contributors or maintainers, and I am grateful for both their time reviewing this book and in improving CMake itself. Thanks to Cristian Adam for reviewing Chapter 30, *Build Performance* and Chapter 31, *Working With Qt*, and to Tobias Hunger, Alexandru Croitor, Joerg Bornemann, Sebastian Holtermann and Burkhard Stubert, all of whom also reviewed the material on Qt at various stages. Alex Turbov and Marc Chevrier also provided valuable comments and suggestions for new and updated material in the 7th edition.

The helpful feedback received from readers is also much appreciated. I'd like to make particular mention of the efforts of Declan Moran and Ganesh A. Hegde, both of whom highlighted a number of improvements and corrections.

A special mention is deserved for my past colleague Mike Wake. Much of the material in this book was thrashed out and tested in real, actively developed production projects. There have been wrong turns and many technical discussions on how to improve things from both a usability and 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 have been 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.

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 of the 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 end 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.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

Part I: Fundamentals

Attempting to use any tool before understanding at least the basics of what it does and how it is meant to be used is most likely going to result in 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. This first part of the book follows a logical progression through CMake's more fundamental features and concepts and is structured to enable the reader to immediately experiment and to do increasingly useful things with each chapter. The goal is to incrementally build up the base knowledge needed to use CMake effectively, with an emphasis on being able to put that knowledge into practice right away.

The initial focus in the first few chapters is on building a basic executable or library, covering just enough to give a new developer a quick introduction to CMake. Subsequent chapters expand that knowledge to demonstrate how to get the most out of what CMake has to offer. The techniques presented are aimed at real world use, with the intention of establishing good habits and teaching sound methods which scale to very large projects and can handle more complex scenarios.

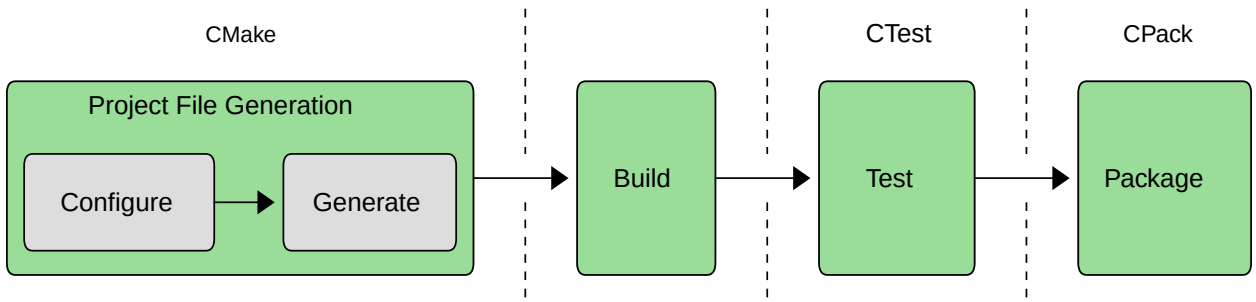
The later parts of the book all rely heavily on the material covered in this first part. Those who have already been using CMake for some time may find the topics relatively familiar, but the material also includes hard-won knowledge from real world projects and interaction with the CMake community. Even experienced users should find at least the Recommended Practices section at the end of each chapter to be a useful read.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

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 (e.g. 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 for managing the later testing and packaging stages respectively. The entire process from start to finish can be driven from CMake itself, with the testing and packaging steps available simply as additional targets in the build. Even the build tool can be invoked by CMake.

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 (eg 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 install. Developers are encouraged to take advantage of this and remain relatively close to the most recent stable CMake release.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

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, etc. and 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.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

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 (e.g. `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 (e.g. Xcode or Visual Studio project files, Makefiles, etc.) are also created in the build directory and 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

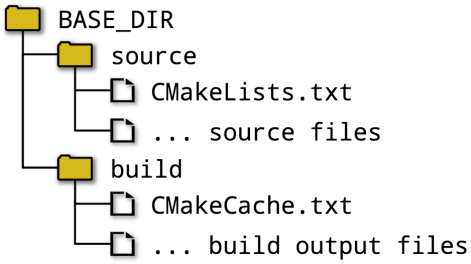
It is possible, though discouraged, for the source and build directories to be the same. This arrangement is called an *in-source* build. Developers at the beginning of their career often start out using this approach because of the perceived simplicity. The main difficulty with in-source builds, however, 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 either 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.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

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, which allows builds to be set up with different sets of options, such as debug and release versions, etc.

This book will always use out-of-source builds and will follow 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 used by some developers 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 it does still carry with it some of the disadvantages of an in-source arrangement. Unless there is a good reason to structure things that way, keeping the build directory completely outside of the source tree instead is recommended.

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 16 2019	Yes
	Visual Studio 15 2017	
	⋮	
Xcode	Xcode	Yes
Ninja	Ninja	No
	Ninja Multi-Config	Yes
Makefiles	Unix Makefiles	No
	MSYS Makefiles	
	MinGW Makefiles	
	NMake Makefiles	

Exclusively for sbmain17@gmail.com Transaction: 0098315591

Some of the generators produce projects which support multiple configurations (e.g. Debug, Release, etc.). These allow the developer to choose between different build configurations without having to re-run CMake, which is more suitable for generators creating 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 often have good support in IDE environments not so closely associated with a particular compiler (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 10, *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, but the introduction of the GUI application is deferred to Chapter 5, *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, read by command line tools, etc. Alternatively, `cmake` can invoke the build tool on the developer's behalf like so:

Exclusively for sbmain17@gmail.com Transaction: 0098315591

```
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 14, *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, all that needs to be done is change the argument given to the `-G` CMake option and 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.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

Chapter 3. A Minimal Project

All CMake projects start with a file called `CMakeLists.txt` and it is expected to be placed 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 the next few chapters, 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 producing 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 more common 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

Exclusively for sbmain17@gmail.com Transaction: 0098315591

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 be the first line of the `CMakeLists.txt` file so 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 will issue a warning if the `CMakeLists.txt` file does not call `cmake_minimum_required()` before any other command. It 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 12, *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 3.2 is perhaps the oldest any new project should consider, since it provides a reasonably complete feature set for modern CMake techniques. Version 2.8.12 has a reduced feature coverage, lacking a number of useful features but it may be workable for older projects. Versions before that lack substantial features that would make using many modern CMake techniques impossible. If working with fast-moving platforms such as iOS, quite recent versions of CMake may be needed in order to support the latest OS releases, etc.

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

Exclusively for sbmain17@gmail.com Transaction: 0098315591

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 12, *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 projects to update themselves.

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 (eg Xcode and Visual Studio) and it is also used in various other parts of the project, such as to act as defaults for packaging and documentation metadata, to provide project-specific variables and so on. 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 20, *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. 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.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

The `project()` command does much more than just 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 22, *Toolchains And Cross Compiling* discusses this area in much greater detail, including the various ways to influence toolchain selection and configuration. Chapter 7, *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 build directory in the `CMakeCache.txt` file. 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 9, *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 will be 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 similar commenting conventions as 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. The following shows a few comment examples and brings together the concepts introduced in this chapter:

Exclusively for sbmain17@gmail.com Transaction: 0098315591

```
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)
```

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 the later the version, the more CMake features the project will be able to use. 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 creating a project intended to be built and distributed as part of the operating system itself (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.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

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 the remainder of 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 23.2, “Application Bundles”. On non-Apple platforms, the `MACOSX_BUNDLE` keyword is ignored.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

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 being used, the name may be slightly different, such as `ALL_BUILD` for Xcode). 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 17, *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

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

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 23.3, “Frameworks”.

MODULE

Specifies a library that is somewhat like a shared library, but is intended to be loaded

Exclusively for sbmain17@gmail.com Transaction: 0098315591

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 be 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 5, *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 17, *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, however, 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.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

`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 17.2.4, "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 17.2, "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 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).

Exclusively for sbmain17@gmail.com Transaction: 0098315591

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.

In addition to the above, for historical reasons, any item 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 or not the build is configured as a debug build (see Chapter 14, *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.

4.5. Old-style CMake

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

Exclusively for sbmain17@gmail.com Transaction: 0098315591

situations, they may instead be treated as PRIVATE. In particular, if a project defines a chain of library dependencies with a mix of both old and new forms of the command, 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. It is common to see tutorials and examples use a variable for the project name and reuse that variable for the name of an executable target like so:

```
# Poor practice, but very common
set(projectName MyExample)
project(${projectName})
add_executable(${projectName} ...)
```

This only works for the most basic of projects and encourages a number of bad habits. Consider the project name and executable name as being separate, even if initially they start out the same. 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 and assume the project will eventually need to define more than one target. This reinforces better habits which will be important when working on more complex multi-target projects.

When naming targets for libraries, resist the temptation to start or end the name with lib. On many platforms (i.e. just about all 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 resultant library file names end up with the form liblibsomething⋯., which people often assume to be a mistake.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

Unless there are strong reasons to do so, try to 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()`.

Aim to 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.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

Chapter 5. Variables

The preceding chapters showed how to define basic targets and produce build outputs. On its own, this is already useful, but CMake comes with a whole host of other features which bring great flexibility and convenience. This chapter covers one of the most fundamental parts of CMake, namely the use of variables.

5.1. Variable Basics

Like any computing language, variables are a cornerstone of getting things done in CMake. The most basic way of defining a variable is with the `set()` command. A normal variable can be defined in a `CMakeLists.txt` file as follows:

```
set(varName value... [PARENT_SCOPE])
```

The name of the variable, `varName`, can contain letters, numbers and underscores, with letters being case-sensitive. The name may also contain the characters `./-+` but these are rarely seen in practice. Other characters are also possible via indirect means, but again, these are not typically seen in normal use.

In CMake, a variable has a particular scope, much like how variables in other languages have scope limited to a particular function, file, etc. A variable cannot be read or modified outside of its scope. Compared to other languages, variable scope is a little more flexible in CMake, but for now, in the simple examples in this chapter, consider the scope of a variable as being global. Chapter 7, *Using Subdirectories* and Chapter 8, *Functions And Macros* introduce the situations where local scopes arise and show how the `PARENT_SCOPE` keyword is used to promote the visibility of a variable into the enclosing scope.

CMake treats all variables as strings. In various contexts, variables may be interpreted as a different type, but ultimately, they are just strings. When setting a variable's value, CMake doesn't require those values to be quoted unless the value contains spaces. If multiple values are given, the values will be joined together with a semicolon separating each value - the resultant string is how CMake represents lists. The following should help to demonstrate the behavior.

```
set(myVar a b c)    # myVar = "a;b;c"
set(myVar a;b;c)    # myVar = "a;b;c"
set(myVar "a b c")  # myVar = "a b c"
set(myVar a b;c)    # myVar = "a;b;c"
set(myVar a "b c")  # myVar = "a;b c"
```

The value of a variable is obtained with `${myVar}`, which can be used anywhere a string or variable is expected. CMake is particularly flexible in that it is also possible to use this form recursively or to specify the name of another variable to set. In addition, CMake doesn't require variables to be defined before using them. Use of an undefined variable simply results in an empty string being substituted, similar to the way Unix shell scripts behave. By default, no warning is issued for use of an undefined variable, but the `--warn-uninitialized` option can be given to the `cmake` command to

Exclusively for sbmain17@gmail.com Transaction: 0098315591

enable such warnings. Be aware, however, that such use is very common and is not necessarily a symptom of a problem, so the option's usefulness may be limited.

```
set(foo ab)               # foo   = "ab"
set(bar ${foo}cd)         # bar   = "abcd"
set(baz ${foo} cd)        # baz   = "ab;cd"
set(myVar ba)             # myVar = "ba"
set(big "${${myVar}r}ef") # big   = "${bar}ef" = "abcdef"
set(${foo} xyz)           # ab    = "xyz"
set(bar ${notSetVar})     # bar   = ""
```

Strings are not restricted to being a single line, they can contain embedded newline characters. They can also contain quotes, which require escaping with backslashes.

```
set(myVar "goes here")
set(multiLine "First line ${myVar}
Second line with a \"quoted\" word")
```

If using CMake 3.0 or later, an alternative to quotes is to use the lua-inspired bracket syntax where the start of the content is marked by `[=[` and the end with `]=]`. Any number of `=` characters can appear between the square brackets, including none at all, but the same number of `=` characters must be used at the start and the end. If the opening brackets are immediately followed by a newline character, that first newline is ignored, but subsequent newlines are not. Furthermore, no further transformation of the bracketed content is performed (i.e. no variable substitution or escaping).

```
# Simple multi-line content with bracket syntax,
# no = needed between the square bracket markers
set(multiLine [[
First line
Second line
]])

# Bracket syntax prevents unwanted substitution
set(shellScript [=[
#!/bin/bash

[[ -n "${USER}" ]] && echo "Have USER"
]=])

# Equivalent code without bracket syntax
set(shellScript
"#!/bin/bash

[[ -n \"\${USER}\" ]] && echo \"Have USER\"
")
```

As the above example shows, bracket syntax is particularly well suited to defining content like Unix shell scripts. Such content uses the ${⋯} syntax for its own purpose and frequently contains quotes, but using bracket syntax means these things do not have to be escaped, unlike the traditional

Exclusively for sbmain17@gmail.com Transaction: 0098315591

quoting style of defining CMake content. The flexibility to use any number of = characters between the [and] markers also means embedded square brackets do not get misinterpreted as markers. Chapter 19, *Working With Files* includes further examples which highlight situations where bracket syntax can be a better alternative.

A variable can be unset either by calling `unset()` or by calling `set()` with no value for the named variable. The following are equivalent, with no error or warning if `myVar` does not already exist:

```
set(myVar)
unset(myVar)
```

In addition to variables defined by the project for its own use, the behavior of many of CMake's commands can be influenced by the value of specific variables at the time the command is called. This is a common pattern used by CMake to tailor command behavior or to modify defaults so they don't have to be repeated for every command, target definition, etc. The CMake reference documentation for each command typically lists any variables which can affect that command's behavior. Later chapters of this book also highlight a number of useful variables and the way they affect or give information about the build.

5.2. Environment Variables

CMake also allows the value of environment variables to be retrieved and set using a modified form of the CMake variable notation. The value of an environment variable is obtained using the special form `$ENV{varName}` and this can be used anywhere a regular `${varName}` form can be used. Setting an environment variable can be done in a similar way to a CMake variable, except with `ENV{varName}` instead of just `varName` as the variable to set. For example:

```
set(ENV{PATH} "$ENV{PATH}:/opt/myDir")
```

Note, however, that setting an environment variable like this only affects the currently running CMake instance. As soon as the CMake run is finished, the change to the environment variable is lost. In particular, the change to the environment variable will not be visible at build time. Therefore, setting environment variables within the `CMakeLists.txt` file like this is rarely useful.

5.3. Cache Variables

In addition to normal variables discussed above, CMake also supports *cache* variables. Unlike normal variables which have a lifetime limited to the processing of the `CMakeLists.txt` file, cache variables are stored in the special file called `CMakeCache.txt` in the build directory and they persist between CMake runs. Once set, cache variables remain set until something explicitly removes them from the cache. The value of a cache variable is retrieved in exactly the same way as a normal variable (i.e. with the `${myVar}` form), but the `set()` command is different when used to set a cache variable:

```
set(varName value... CACHE type "docstring" [FORCE])
```


Exclusively for sbmain17@gmail.com Transaction: 0098315591

When the `CACHE` keyword is present, the `set()` command will apply to a cache variable named `varName` instead of a normal variable. Cache variables have more information attached to them than a normal variable, including a nominal type and a documentation string. Both must be provided when setting a cache variable, although the `docstring` can be empty. The documentation string does not affect how CMake treats the variable, it is used only by GUI tools to provide things like help details, tooltips, etc.

CMake will always treat the variable as a string during processing. The `type` is used mostly to improve the user experience in GUI tools, with some notable exceptions discussed in the next section. The `type` must be one of the following:

`BOOL`

The cache variable is a boolean `on/off` value. GUI tools use a checkbox or similar to represent the variable. The underlying string value held by the variable will conform to one of the ways CMake represents booleans as strings (`ON/OFF`, `TRUE/FALSE`, `1/0`, etc. - see Section 6.1.1, "Basic Expressions" for full details).

`FILEPATH`

The cache variable represents a path to a file on disk. GUI tools present a file dialog to the user for modifying the variable's value.

`PATH`

Like `FILEPATH`, but GUI tools present a dialog that selects a directory rather than a file.

`STRING`

The variable is treated as an arbitrary string. By default, GUI tools use a single-line text edit widget for manipulating the value of the variable. Projects may use cache variable properties to provide a pre-defined set of values for GUI tools to present as a combobox or similar instead (see Section 9.6, "Cache Variable Properties").

`INTERNAL`

The variable is not intended to be made available to the user. Internal cache variables are sometimes used to persistently record internal information by the project, such as caching the result of an intensive query or computation. GUI tools do not show `INTERNAL` variables. `INTERNAL` also implies `FORCE` (see further below).

GUI tools typically use the `docstring` as a tooltip for the cache variable or as a short one line description when the variable is selected. The `docstring` should be short and consist of plain text (i.e. no HTML markup, etc.).

Setting a boolean cache variable is such a common need that CMake provides a separate command for it. Rather than the somewhat verbose `set()` command, developers can use `option()` instead:

```
option(optVar helpString [initialValue])
```

If `initialValue` is omitted, the default value `OFF` will be used. If provided, the `initialValue` must conform to one of the boolean values accepted by the `set()` command. For reference, the above can be thought of as more or less equivalent to:

Exclusively for sbmain17@gmail.com Transaction: 0098315591

```
set(optVar initialValue CACHE BOOL helpString)
```

Compared to `set()`, the `option()` command more clearly expresses the behavior for boolean cache variables, so it would generally be the preferred command to use. Be aware, however, that the effect of the two commands can be different in certain situations (see the next section).

An important difference between normal and cache variables is that the `set()` command will only overwrite a cache variable if the `FORCE` keyword is present, unlike normal variables where the `set()` command will always overwrite a pre-existing value. The `set()` command acts more like set-if-not-set when used to define cache variables, as does the `option()` command (which has no `FORCE` capability). The main reason for this is that cache variables are primarily intended as a customization point for developers. Rather than hard-coding the value in the `CMakeLists.txt` file as a normal variable, a cache variable can be used so that the developer can override the value without having to edit the `CMakeLists.txt` file. The variable can be modified by interactive GUI tools or by scripts without having to change anything in the project itself.

5.4. Potentially Surprising Behavior Of Variables

A point that is often not well understood is that normal and cache variables are two separate things. It is possible to have a normal variable and a cache variable with the same name, but holding different values. In such cases, CMake will retrieve the normal variable's value rather than the cache variable when using `${myVar}`. Put another way, normal variables take precedence over cache variables. The exception to this is that when setting a cache variable's value, any normal variable of the same name is removed from the current scope in the following situations (subject to policy settings discussed further below):

- The cache variable did not exist before the call to `set()` or `option()`.
- The cache variable existed before the call to `set()` or `option()`, but it did not have a defined type (see Section 5.5.1, "Setting Cache Values On The Command Line" for how this can occur).
- The `FORCE` or `INTERNAL` option was used in the call to `set()`.

In the first two cases above, this means it is possible to get different behavior between the first and subsequent CMake runs. In the first run, the cache variable won't exist or won't have a defined type, but in subsequent runs it will. Therefore, in the first run, a normal variable would be hidden, but in subsequent runs, it would not. An example should help illustrate the problem:

```
set(myVar foo)                  # Local myVar
set(result ${myVar})            # result = foo
set(myVar bar CACHE STRING "")  # Cache myVar

set(result ${myVar})            # First run:       result = bar
                                # Subsequent runs: result = foo

set(myVar fred)
set(result ${myVar})            # result = fred
```


Exclusively for sbmain17@gmail.com Transaction: 0098315591

Chapter 7, *Using Subdirectories* and Chapter 8, *Functions And Macros* contain further discussions of how a variable's scope can influence the value that `${myVar}` would return.

In CMake 3.13, the behavior of `option()` was changed such that if a normal variable already exists with the same name, the command does nothing. This newer behavior is typically what developers intuitively expect. A similar change was made for the `set()` command in CMake 3.21, but note the following differences in the new behavior for both commands:

- For `set()`, the cache variable is still set if it didn't exist previously, but for `option()` it is not.
- If `INTERNAL` or `FORCE` is used with `set()`, the cache variable will always be set or updated.

Developers should be mindful of these inconsistencies and the different CMake versions that provide the new behaviors. Policies `CMP0077` and `CMP0126` control the actual behavior (see Chapter 12, *Policies* for an understanding of how these can be manipulated).

The interaction between cache and non-cache variables can also lead to other potentially unexpected behavior. Consider the following three commands:

```
unset(foo)
set(foo)
set(foo "")
```

One might be tempted to think that the evaluation of `${foo}` would always give an empty string after any of these three cases, but only the last is guaranteed to do so. Both `unset(foo)` and `set(foo)` remove a non-cache variable from the current scope. If there is also a *cache* variable called `foo`, that cache variable is left alone and `${foo}` would provide the value of that cache variable. In this sense, `unset(foo)` and `set(foo)` both effectively unmask the `foo` cache variable, if one exists. On the other hand, `set(foo "")` doesn't remove a non-cache variable, it explicitly sets it to an empty value, so `${foo}` will always then evaluate to an empty string regardless of whether there is also a cache variable called `foo`. Therefore, setting a variable to an empty string rather than removing it is likely to be the more robust way of achieving the developer's intention.

For those rare situations where a project may need to get the value of a cache variable and ignore any non-cache variable of the same name, CMake 3.13 added documentation for the `$CACHE{someVar}` form. Projects should not generally make use of this other than for temporary debugging, since it breaks with the long-established expectation that normal variables will override values set in the cache.

5.5. Manipulating Cache Variables

Using `set()` and `option()`, a project can build up a useful set of customization points for its developers. Different parts of the build can be turned on or off, paths to external packages can be set, flags for compilers and linkers can be modified and so on. Later chapters cover these and other uses of cache variables, but first, the ways to manipulate such variables need to be understood. There are two primary ways developers can do this, either from the `cmake` command line or using a GUI tool.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

5.5.1. Setting Cache Values On The Command Line

CMake allows cache variables to be manipulated directly via command line options passed to `cmake`. The primary workhorse is the `-D` option, which is used to define the value of a cache variable.

```
cmake -D myVar:type=someValue ...
```

`someValue` will replace any previous value of the `myVar` cache variable. The behavior is essentially as though the variable was being assigned using the `set()` command with the `CACHE` and `FORCE` options. The command line option only needs to be given once, since it is stored in the cache for subsequent runs and therefore does not need to be provided every time `cmake` is run. Multiple `-D` options can be provided to set more than one variable at a time on the `cmake` command line.

When defining cache variables this way, they do not have to be set within the `CMakeLists.txt` file (i.e. no corresponding `set()` command is required). Cache variables defined on the command line have an empty `docstring`. The `type` can also be omitted, in which case the variable will have an undefined type, or more accurately, it is given a special type that is similar to `INTERNAL` but which CMake interprets to mean undefined. The following shows various examples of setting cache variables via the command line.

```
cmake -D foo:BOOL=ON ...
cmake -D "bar:STRING=This contains spaces" ...
cmake -D hideMe=mysteryValue ...
cmake -D helpers:FILEPATH=subdir/helpers.txt ...
cmake -D helpDir:PATH=/opt/helpThings ...
```

Note how the entire value given with the `-D` option should be quoted if setting a cache variable with a value containing spaces.

There is a special case for handling values initially declared without a type on the `cmake` command line. If the project's `CMakeLists.txt` file then tries to set the same cache variable and specifies a type of `FILEPATH` or `PATH`, then if the value of that cache variable is a relative path, CMake will treat it as being relative to the directory from which `cmake` was invoked and automatically convert it to an absolute path. This is not particularly robust, since `cmake` could be invoked from any directory, not just the build directory. Therefore, developers are advised to always include a type if specifying a variable on the `cmake` command line for a variable that represents some kind of path. It is a good habit to always specify the type of the variable on the command line in general anyway so that it is likely to be shown in GUI applications in the most appropriate form. It will also prevent one of the scenarios mentioned earlier in Section 5.4, "Potentially Surprising Behavior Of Variables".

It is also possible to remove variables from the cache with the `-U` option, which can be repeated as necessary to remove more than one variable. Note that the `-U` option supports * and ? wildcards, but care needs to be taken to avoid deleting more than was intended and leaving the cache in an unbuildable state. In general, it is recommended to only remove specific entries without wildcards unless it is absolutely certain the wildcards used are safe.

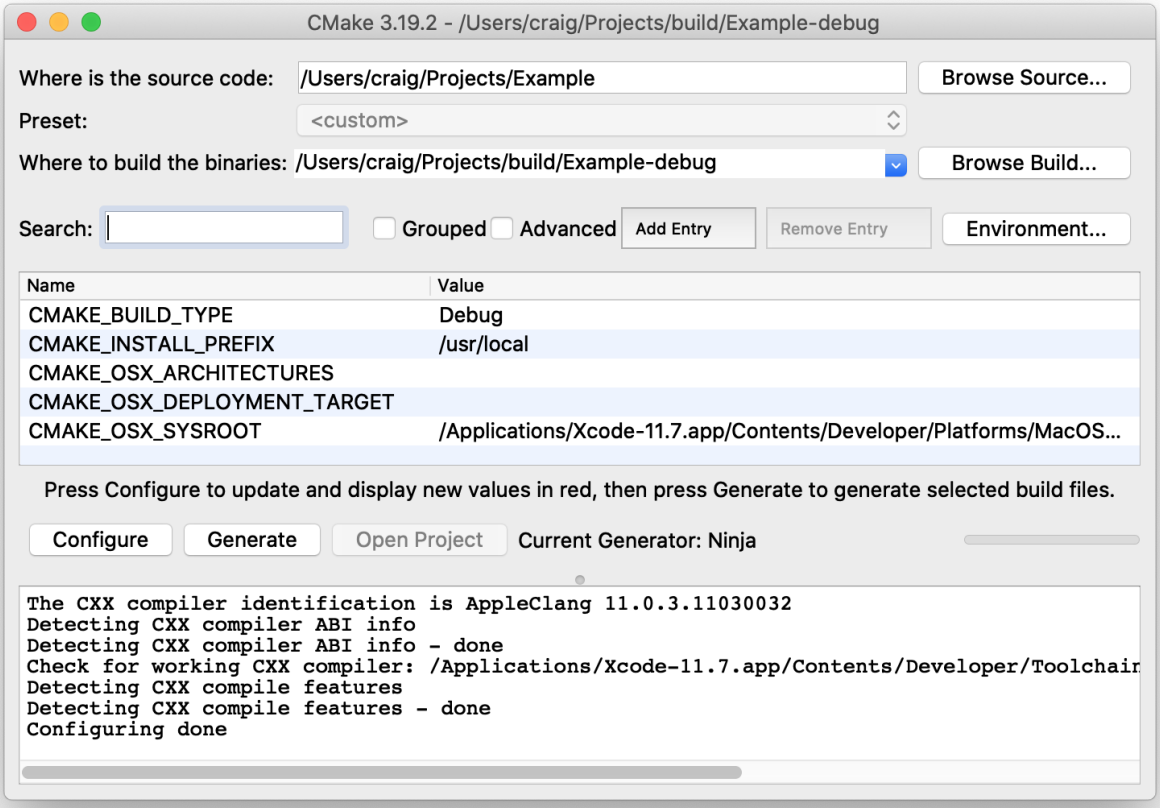
```
cmake -U 'help*' -U foo ...
```


Exclusively for sbmain17@gmail.com Transaction: 0098315591

5.5.2. CMake GUI Tools

Setting cache variables via the command line is an essential part of automated build scripts and anything else driving CMake via the `cmake` command. For everyday development, however, the GUI tools provided by CMake often present a better user experience. CMake provides two equivalent GUI tools, `cmake-gui` and `ccmake`, which allow developers to manipulate cache variables interactively. `cmake-gui` is a fully functional GUI application supported on all major desktop platforms, whereas `ccmake` uses a curses-based interface which can be used in text-only environments such as over a `ssh` connection. `cmake-gui` is included in the official CMake release packages for all platforms, `ccmake` is included for all platforms except Windows. If using system-provided packages on Linux rather than the official releases, note that many distributions split `cmake-gui` out into its own package.

The `cmake-gui` user interface is shown in the figure below. The top section allows the project's source and build directories to be defined. The middle section is where the cache variables can be viewed and edited. At the bottom are the `Configure` and `Generate` buttons, followed by a log area that shows the output from those operations.



The source directory must be set to the directory containing the `CMakeLists.txt` file at the top of the project's source tree. The build directory is where CMake will generate all build output (recommended directory layouts were discussed in Chapter 2, *Setting Up A Project*). For new projects, both must be set, but for existing projects, setting the build directory will also update the source directory, since the source location is stored in the build directory's cache.

CMake's two-stage setup process was introduced in Section 2.3, "Generating Project Files". In the first stage, the `CMakeLists.txt` file is read and a representation of the project is built up in memory.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

This is called the *configure* stage. If the configure stage is successful, the *generate* stage can then be executed to create the build tool's project files in the build directory. When running `cmake` from the command line, both stages are executed automatically, but in the GUI application, they are triggered separately with the `Configure` and `Generate` buttons. Each time the configure step is initiated, the cache variables shown in the middle of the UI are updated. Any variables which were newly added or which changed value from the previous run will be highlighted in red (when a project is first loaded, all variables are shown highlighted). Good practice is to re-run the configure stage until there are no changes, since this ensures robust behavior for more complex projects where enabling some options may add further options which could require another configure pass. Once all cache variables are shown without red highlighting, the generate stage can be run. The example in the previous screenshot shows typical log output after the configure stage has been run and no changes were made to any of the cache variables.

Hovering the mouse over any cache variable will show a tooltip containing the `docstring` for that variable. New cache variables can also be added with the `Add Entry` button, which is equivalent to issuing a `set()` command with an empty `docstring`. Cache variables can be removed with the `Remove Entry` button, although CMake will most likely recreate that variable on the next run.

Clicking on a variable allows its value to be edited in a widget tailored to the variable type. Booleans are shown as a checkbox, files and paths have a browse filesystem button and strings are usually presented as a text line edit. As a special case, cache variables of type `STRING` can be given a set of values to show in a combobox in CMake GUI instead of showing a simple text entry widget. This is achieved by setting a cache variable's `STRINGS` property (covered in detail in Section 9.6, "Cache Variable Properties", but shown here for convenience):

```
set(trafficLight Green CACHE STRING "Status of something")
set_property(CACHE trafficLight PROPERTY STRINGS Red Orange Green)
```

In the above, the `trafficLight` cache variable will initially have the value `Green`. When the user attempts to modify `trafficLight` in `cmake-gui`, they will be given a combobox containing the three values `Red`, `Orange` and `Green` instead of a simple line edit widget which would otherwise have allowed them to enter any arbitrary text. Note that setting the `STRINGS` property on the variable doesn't prevent that variable from having other values assigned to it, it only affects the widget used by `cmake-gui` when editing it. The variable can still be given other values via `set()` commands in the `CMakeLists.txt` file or by other means such as manually editing the `CMakeCache.txt` file.

Cache variables can also have a property marking them as advanced or not. This too only affects the way the variable is displayed in `cmake-gui`, it does not in any way affect how CMake uses the variable during processing. By default, `cmake-gui` only shows non-advanced variables, which typically presents just the main variables a developer would be interested in viewing or modifying. Enabling the `Advanced` option shows all cache variables except those marked `INTERNAL` (the only way to see `INTERNAL` variables is to edit the `CMakeCache.txt` file with a text editor, since they are not intended to be manipulated directly by developers). Variables can be marked as advanced with the `mark_as_advanced()` command within the `CMakeLists.txt` file:

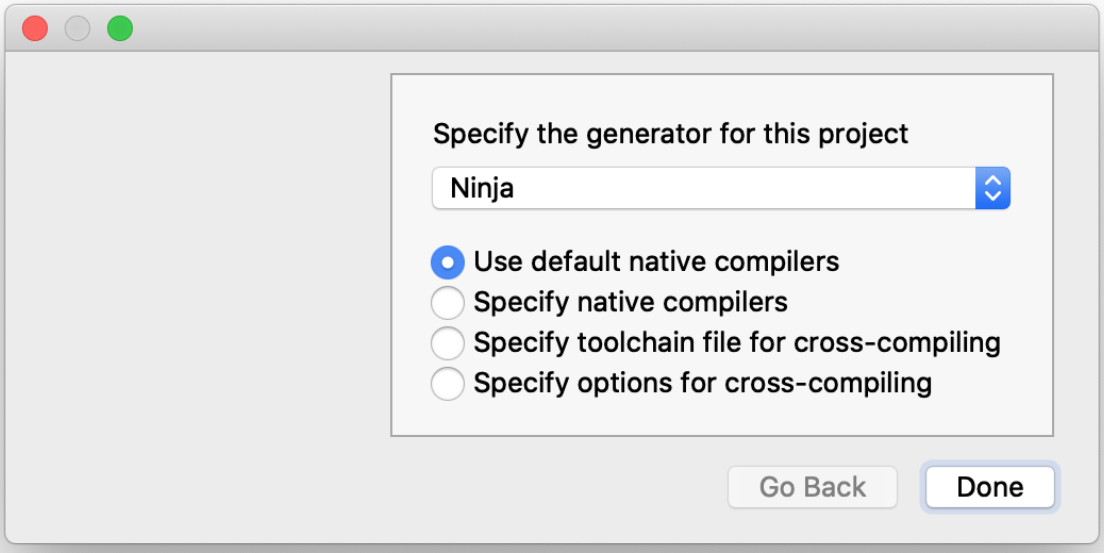
```
mark_as_advanced([CLEAR|FORCE] var1 [var2...])
```


Exclusively for sbmain17@gmail.com Transaction: 0098315591

The `CLEAR` keyword ensures the variables are not marked as advanced, while the `FORCE` keyword ensures the variables are marked advanced. Without either keyword, the variables will only be marked as advanced if they don't already have an advanced/non-advanced state set.

Selecting the `Grouped` option can make viewing advanced variables easier by grouping variables together based on the start of the variable name up to the first underscore. Another way to filter the list of variables shown is to enter text in the `Search` area, which results in only showing variables with the specified text in their name or value.

When the configure stage is run for the first time on a new project, the developer is presented with a dialog similar to that shown in the next screenshot:



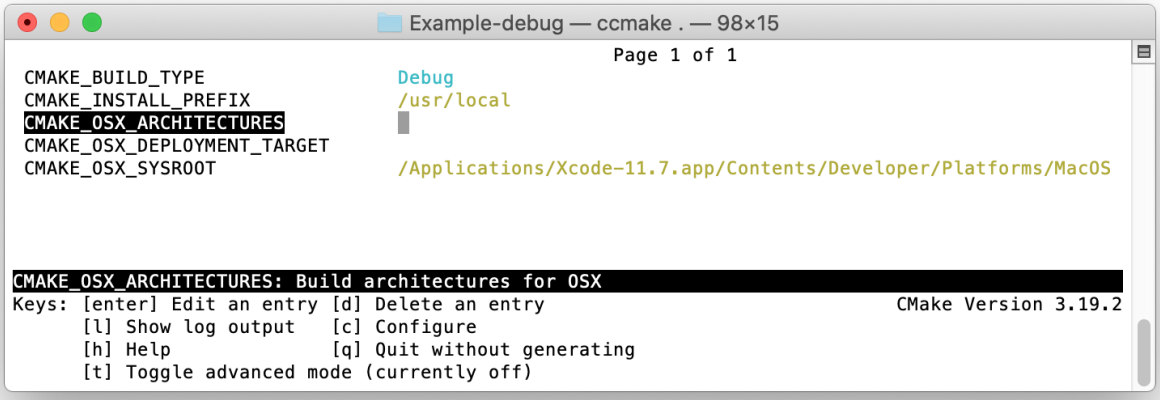
This dialog is where the CMake generator and toolchain are specified. The choice of generator is usually up to the developer's personal preference, with available options provided in the combobox. Depending on the project, the choice of generator may be more restricted than what the combobox options allow, such as if the project relies on generator-specific functionality. A common example of this is a project that requires the Xcode generator due to the Apple platform's unique features, such as code signing and iOS/tvOS/watchOS support. Once a generator has been selected for a project, it cannot be changed without deleting the cache and starting again, which can be done from the `File` menu if required.

For the toolchain options presented, each one requires progressively more information from the developer. Using the default native compilers is the usual choice for ordinary desktop development and selecting that option requires no further details. If more control is required, developers can instead override the native compilers, with the paths to the compilers being given in a follow-up dialog. If a separate toolchain file is available, that can be used to customize not just the compilers but also the target environment, compiler flags and various other things. Using a toolchain file is typical when cross-compiling, which is covered in detail in Chapter 22, *Toolchains And Cross Compiling*. Lastly, for ultimate control, developers can specify the full set of options for cross-

Exclusively for sbmain17@gmail.com Transaction: 0098315591

compiling, but this is not recommended for normal use. A toolchain file can provide the same information but has the advantage that it can be re-used as needed.

The `ccmake` tool offers most of the same functionality as the `cmake-gui` application, but it does so through a text-based interface:



Rather than selecting the source and build directories like with `cmake-gui`, the source or build directory has to be specified on the `ccmake` command line, just like for the `cmake` command.

A minor drawback of the `ccmake` interface is that there is no ability to filter the variables shown. The methods for editing a variable are also not as rich as with `cmake-gui`. Nevertheless, the `ccmake` tool is a useful alternative when the full `cmake-gui` application is not practical or not available, such as over a terminal connection that cannot support UI forwarding.

5.6. Printing Variable Values

As projects get more complicated or when investigating unexpected behavior, it can be useful to print out diagnostic messages and variable values during a CMake run. This is generally achieved using the `message()` command, which is covered in detail in Chapter 13, *Debugging And Diagnostics*. For now, it is enough to know that in its simplest form, all the `message()` command does is print its arguments to CMake's output. It adds no separator between arguments if more than one argument is given and a newline is automatically appended to the end of the message. Newlines can also be explicitly included using the common `\n` notation. A variable's value can be included in the message by using the usual `${myVar}` notation.

```
set(myVar HiThere)
message("The value of myVar = ${myVar}\nAnd this "
        "appears on the next line")
```

This will give the following output:

```
The value of myVar = HiThere
And this appears on the next line
```


Exclusively for sbmain17@gmail.com Transaction: 0098315591

5.7. String Handling

As project complexity grows, in many cases so too does the need to implement more involved logic for how variables are managed. A core tool CMake provides for this is the `string()` command, which provides a wide range of useful string handling functionality. This command enables projects to perform find and replace operations, regular expression matching, upper/lower case transformations, strip whitespace and other common tasks. Some of the more frequently used functionality is presented below, but the CMake reference documentation should be considered the canonical source of all available operations and their behavior.

The first argument to `string()` defines the operation to be performed and subsequent arguments depend on the operation being requested. These arguments will generally require at least one input string and since CMake commands cannot return a value, an output variable for the result of the operation. In the material below, this output variable will generally be named `outVar`.

```
string(FIND inputString subString outVar [REVERSE])
```

`FIND` searches for `subString` in `inputString` and stores the index of the found `subString` in `outVar` (the first character is index 0). The first occurrence is found unless `REVERSE` is specified, in which case the last occurrence will be found instead. If `subString` does not appear in `inputString`, then `outVar` will be given the value -1.

```
string(FIND abcdefabcdef def fwdIndex)
string(FIND abcdefabcdef def revIndex REVERSE)

message("fwdIndex = ${fwdIndex}\n"
        "revIndex = ${revIndex}")
```

This results in the following output:

```
fwdIndex = 3
revIndex = 9
```

Replacing a simple substring follows a similar pattern:

```
string(REPLACE matchString replaceWith outVar input...)
```

The `REPLACE` operation will replace every occurrence of `matchString` in the `input` strings with `replaceWith` and store the result in `outVar`. When multiple `input` strings are given, they are joined together without any separator between each string before searching for substitutions. This can sometimes lead to unexpected matches and typically developers would provide just the one `input` string in most situations.

Regular expressions are also well supported by the `REGEX` operation, with a few different variants available as determined by the second argument:

Exclusively for sbmain17@gmail.com Transaction: 0098315591

```
string(REGEX MATCH    regex outVar input...)
string(REGEX MATCHALL regex outVar input...)
string(REGEX REPLACE  regex replaceWith outVar input...)
```

The regular expression to match, `regex`, can make use of typical basic regular expression syntax (see the CMake reference documentation for the full specification), although some common features such as negation are not supported. The `input` strings are concatenated before substitution. The `MATCH` operation finds just the first match and stores it in `outVar`. `MATCHALL` finds all matches and stores them in `outVar` as a list. `REPLACE` will return the entire `input` string with each match replaced by `replaceWith`. Matches can be referred to in `replaceWith` using the usual notation \1, \2, etc., but note that the backslashes themselves must be escaped unless bracket notation is used. The following example and its output demonstrate the above points:

```
string(REGEX MATCH    "[ace]"           matchOne abcdefabcdef)
string(REGEX MATCHALL "[ace]"           matchAll abcdefabcdef)
string(REGEX REPLACE  "([de])" "X\\1Y"  replVar1 abc def abcdef)
string(REGEX REPLACE  "([de])" [[X\1Y]] replVar2 abcdefabcdef)

message("matchOne = ${matchOne}\n"
        "matchAll = ${matchAll}\n"
        "replVar1 = ${replVar1}\n"
        "replVar2 = ${replVar2}")
```

```
matchOne = a
matchAll = a;c;e;a;c;e
replVar1 = abcXdYXeYfabcXdYXeYf
replVar2 = abcXdYXeYfabcXdYXeYf
```

Extracting a substring is also possible:

```
string(SUBSTRING input index length outVar)
```

The `index` is an integer defining the start of the substring to extract from `input`. Up to `length` characters will be extracted, or if `length` is -1, the returned substring will contain all characters up to the end of the `input` string. Note that in CMake 3.1 and earlier, an error was reported if `length` pointed past the end of the string.

String length can be trivially obtained and strings can easily be converted to upper or lower case. It is also straightforward to strip whitespace from the start and end of a string. The syntax for these operations all share the same form:

```
string(LENGTH  input outVar)
string(TOLOWER input outVar)
string(TOUPPER input outVar)
string(STRIP   input outVar)
```


Exclusively for sbmain17@gmail.com Transaction: 0098315591

In the case of LENGTH, for historical reasons the command counts bytes rather than characters. For strings containing multi-byte characters, this means the reported length will be different to the number of characters.

CMake provides other operations, such as string comparison, hashing, timestamps, JSON handling and more, but their use is less common in everyday CMake projects. The interested reader should consult the CMake reference documentation for the string() command for full details.

5.8. Lists

Lists are used heavily in CMake. Ultimately, lists are just a single string with list items separated by semicolons (with one exception, which is covered in Section 5.8.1, "Problems With Unbalanced Square Brackets" further below). This can make it less convenient to manipulate individual list items. CMake provides the list() command to facilitate such tasks. Like for the string() command, list() expects the operation to perform as its first argument. The second argument is always the list to operate on and it must be a variable (i.e. passing a raw list like a;b;c is not permitted).

The most basic list operations are counting the number of items and retrieving one or more items from the list:

```
list(LENGTH listVar outVar)
list(GET    listVar index [index...] outVar)
```

Example usage:

```
set(myList a b c)        # Creates the list "a;b;c"

list(LENGTH myList len)
message("length  = ${len}")

list(GET myList 2 1 letters)
message("letters = ${letters}")
```

The output of the above example would be:

```
length  = 3
letters = c;b
```

Inserting, appending and prepending items is also a common task:

```
list(INSERT  listVar index item [item...])
list(APPEND  listVar item [item...])
list(PREPEND listVar item [item...])  # Requires CMake 3.15 or later
```

Unlike the LENGTH and GET cases, INSERT, APPEND and PREPEND act directly on the listVar and modify it in-place, as demonstrated by the following example:

Exclusively for sbmain17@gmail.com Transaction: 0098315591

```
set(myList a b c)

list(INSERT myList 2 X Y Z)
message("myList (first)  = ${myList}")

list(APPEND myList d e f)
message("myList (second) = ${myList}")

list(PREPEND myList P Q R)
message("myList (third)  = ${myList}")
```

Which gives the following output:

```
myList (first)  = a;b;X;Y;Z;c
myList (second) = a;b;X;Y;Z;c;d;e;f
myList (third)  = P;Q;R;a;b;X;Y;Z;c;d;e;f
```

Finding a particular item in the list follows the expected pattern:

```
list(FIND myList value outVar)
```

Example usage:

```
set(myList a b c d e)
list(FIND myList d index)
message("index = ${index}")
```

Resultant output:

```
index = 3
```

Three operations are provided for removing items, all of which modify the list directly:

```
list(REMOVE_ITEM       myList value [value...])
list(REMOVE_AT         myList index [index...])
list(REMOVE_DUPLICATES myList)
```

The `REMOVE_ITEM` operation can be used to remove all instances of one or more items from a list. If the item is not in the list, it is not an error. `REMOVE_AT` on the other hand, specifies one or more indices to remove and CMake will halt with an error if any of the specified indices are past the end of the list. `REMOVE_DUPLICATES` will ensure the list contains only unique items.

CMake 3.15 added support for popping items from the front or back of a list and optionally storing the popped items:

Exclusively for sbmain17@gmail.com Transaction: 0098315591

```
# Requires CMake 3.15 or later
list(POP_FRONT myList [outVar1 [outVar2...]])
list(POP_BACK myList [outVar1 [outVar2...]])
```

When no `outVar` is given, a single item is popped from the front or back and discarded. If one or more `outVar` names are given, popped items will be stored in those variables, with the number of items popped equal to the number of variable names provided.

List items can also be reordered with `REVERSE` or `SORT` operations:

```
list(REVERSE myList)
list(SORT    myList [COMPARE method] [CASE case] [ORDER order])
```

All of the optional keywords for `list(SORT)` are only available with CMake 3.13 or later. If the `COMPARE` option is present, the `method` must be one of the following:

`STRING`

Sort alphabetically. This is the default behavior when the `COMPARE` option is not given.

`FILE_BASENAME`

Sort by assuming that each item is a path and that items should be ordered according to the basename part of the path only.

`NATURAL`

Similar to `STRING`, except contiguous digits within an item are sorted numerically. This is most useful for sorting strings that contain embedded version numbers. The sorting rules are the same as for the `strverscmp()` C function (a GNU extension). This sorting method is only available with CMake 3.18 or later.

The `CASE` keyword requires `SENSITIVE` or `INSENSITIVE` for the `case`, while the `ORDER` keyword requires either `ASCENDING` or `DESCENDING` for the `order`.

For all list operations taking an index, a negative index indicates that counting starts from the end of the list. When used this way, the last item in the list has index -1, the second last -2, and so on.

The above describes most of the available `list()` sub-commands. Those mentioned are all supported since at least CMake 3.0 unless otherwise noted, so projects should generally be able to expect them to be available. For the full list of supported sub-commands, the reader should consult the CMake documentation.

5.8.1. Problems With Unbalanced Square Brackets

There is one exception to the way CMake usually treats semicolons as list separators. For historical reasons, if a list item contains an opening square bracket `[`, it must also have a matching closing square bracket `]`. CMake will consider any semicolon between these square brackets to be part of the list item instead of as a list separator. If one tries to construct a list with unbalanced square brackets, the list won't be interpreted as expected. The following demonstrates the behavior:

Exclusively for sbmain17@gmail.com Transaction: 0098315591

```
set(noBrackets   "a_a" "b_b")
set(withBrackets "a[a" "b]b")

list(LENGTH noBrackets   lenNo)
list(LENGTH withBrackets lenWith)

list(GET noBrackets   0 firstNo)
list(GET withBrackets 0 firstWith)

message("No brackets:   Length=${lenNo} --> First_element=${firstNo}")
message("With brackets: Length=${lenWith} --> First_element=${firstWith}")
```

The output from the above would be:

```
No brackets:   Length=2 --> First_element=a_a
With brackets: Length=1 --> First_element=a[a;b]b
```

Discussion in Section 8.8.3, “Special Cases For Argument Expansion” covers further aspects of this peculiarity.

5.9. Math

One other common form of variable manipulation is math computation. CMake provides the `math()` command for performing basic mathematical evaluation:

```
math(EXPR outVar mathExpr [OUTPUT_FORMAT format])
```

The first argument must be the keyword `EXPR`, while `mathExpr` defines the expression to be evaluated and the result will be stored in `outVar`. The expression may use any of the following operators which all have the same meaning as they would in C code: `+ - * / % | & ^ ~ << >>`. Parentheses are also supported and have their usual mathematical meaning. Variables can be referenced in the `mathExpr` with the usual `${myVar}` notation.

If using CMake 3.13 or later, the `OUTPUT_FORMAT` keyword can be given to control how the result is stored in `outVar`. The `format` should be either `DECIMAL`, which is the default behavior, or `HEXADECIMAL`.

```
set(x 3)
set(y 7)
math(EXPR zDec "(${x}+${y}) * 2")
message("decimal = ${zDec}")

# Requires CMake 3.13 or later for HEXADECIMAL
math(EXPR zHex "(${x}+${y}) * 2" OUTPUT_FORMAT HEXADECIMAL)
message("hexadecimal = ${zHex}")
```

The above produces the following output:

Exclusively for sbmain17@gmail.com Transaction: 0098315591

```
decimal = 20
hexadecimal = 0x14
```

5.10. Recommended Practices

Where the development environment allows it, the CMake GUI tool is a useful way to quickly and easily understand the build options for a project and to modify them as needed during development. A little bit of time spent getting familiar with it will simplify working with more complex projects later. It also gives developers a good base to work from should they need to experiment with things like compiler settings, since these are easily found and modified within the GUI environment.

Prefer to provide cache variables for controlling whether to enable optional parts of the build instead of encoding the logic in build scripts outside of CMake. This makes it trivial to turn them on and off in the CMake GUI and other tools which understand how to work with the CMake cache (a growing number of IDE environments are acquiring this capability).

Try to avoid relying on environment variables being defined, apart from perhaps the ubiquitous `PATH` or similar operating system level variables. The build should be predictable, reliable and easy to set up, but if it relies on environment variables being set for things to work correctly, this can be a point of frustration for new developers as they wrestle to get their build environment set up. Furthermore, the environment at the time CMake is run can change compared to when the build itself is invoked. Therefore, prefer to pass information directly to CMake through cache variables instead wherever possible.

Try to establish a variable naming convention early. For cache variables, consider grouping related variables under a common prefix followed by an underscore to take advantage of how CMake GUI groups variables based on the same prefix automatically. Also consider that the project may one day become a sub-part of some larger project, so a name beginning with the project name or something closely associated with the project may be desirable.

Try to avoid defining non-cache variables in the project which have the same name as cache variables. The interaction between the two types of variables can be unexpected for developers new to CMake. Later chapters also highlight other common errors and misuses of regular variables that share the same name as cache variables.

CMake provides a large number of pre-defined variables that provide details about the system or influence certain aspects of CMake's behavior. Some of these variables are heavily used by projects, such as those that are only defined when building for a particular platform (`WIN32`, `APPLE`, `UNIX`, etc.). It is therefore recommended for developers to occasionally make a quick scan through the CMake documentation page listing the pre-defined variables to help become familiar with what is available.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

Chapter 6. Flow Control

A common need for most CMake projects is to apply some steps only in certain circumstances. Projects may want to use certain compiler flags only with a particular compiler or when building for a particular platform, for example. In other cases, the project may need to iterate over a set of values or to keep repeating some set of steps until a certain condition is met. These examples of flow control are well supported by CMake in ways which should be familiar to most software developers. The ubiquitous `if()` command provides the expected if-then-else behavior and looping is provided through the `foreach()` and `while()` commands. All three commands provide the traditional behavior as implemented by most programming languages, but they also have added features specific to CMake.

6.1. The if() Command

The modern form of the `if()` command is as follows (multiple `elseif()` clauses can be provided):

```
if(expression1)
    # commands ...
elseif(expression2)
    # commands ...
else()
    # commands ...
endif()
```

Very early versions of CMake required `expression1` to be repeated as an argument to the `else()` and `endif()` clauses, but this has not been required since CMake 2.8.0. While it is still not unusual to encounter projects and example code using that older form, it is discouraged for new projects since it can be somewhat confusing to read. New projects should leave the `else()` and `endif()` arguments empty, as shown above.

The expressions in `if()` and `elseif()` commands can take a variety of different forms. CMake offers the traditional boolean logic as well as various other conditions such as file system tests, version comparison and testing for the existence of things.

6.1.1. Basic Expressions

The most basic of all expressions is a single constant value:

```
if(value)
```

CMake's logic for what it considers true and false is a little more involved than most programming languages. For a single unquoted `value`, the rules are as follows:

- If `value` is a quoted or unquoted constant with value `ON`, `YES`, `TRUE` or `Y`, it is treated as true. The test is case-insensitive.
- If `value` is a quoted or unquoted constant with value `OFF`, `NO`, `FALSE`, `N`, `IGNORE`, `NOTFOUND`, an empty string or a string that ends in `-NOTFOUND`, it is treated as false. Again, the test is case-insensitive.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

- If `value` is a (possibly floating-point) number, it will be converted to a `bool` following usual `C` rules, although values other than 0 or 1 are not often used in this context.
- If none of the above cases apply, it will be treated as a variable name (or possibly as a string) and evaluated further as described below.

In the following examples, only the `if(…)` part of the command is shown for illustration purposes, the corresponding body and `endif()` is omitted:

```
# Examples of quoted and unquoted constants
if(YES)
if("True")
if(0)
if(TRUE)

# These are also treated as unquoted constants because the
# variable evaluation occurs before if() sees the values
set(A YES)
set(B 0)
if(${A})  # Evaluates to true
if(${B})  # Evaluates to false

# Does not match any of the true or false constants, so proceed
# to testing as a variable name in the fall-through case below
if(someLetters)

# Quoted value that doesn't match any of the true or false constants,
# so again fall through to testing as a variable name or string
if("someLetters")
```

The CMake documentation refers to the fall through case as the following form:

```
if(<variable|string>)
```

What this means in practice is the if-expression is either:

- An unquoted name of a (possibly undefined) variable.
- A quoted string.

When an unquoted variable name is used, the variable's value is compared against the false constants. If none of those match the value, the result of the expression is true. An undefined variable will evaluate to an empty string, which matches one of the false constants and will therefore yield a result of false.

```
# Common pattern, often used with variables defined
# by commands such as option(enableSomething "...")
if(enableSomething)
    # ...
endif()
```

Exclusively for sbmain17@gmail.com Transaction: 0098315591

Note that environment variables do not count as variables in this discussion. A statement like `if(ENV{some_var})` will always evaluate to false, regardless of whether an environment variable called `some_var` exists or not.

When the if-expression is a quoted string, the behavior is more involved:

- A quoted string that doesn't match any of the defined true constants always evaluates to false in CMake 3.1 or later, regardless of the string's value (but this can be overridden with policy setting `CMP0054`, see Chapter 12, *Policies*).
- Before CMake 3.1, if the value of the string matched the name of an existing variable, then the quoted string is effectively replaced by that variable name (unquoted) and the test is then repeated.

Both of the above can be a surprise to developers, but at least the CMake 3.1 behavior is always predictable. The pre-3.1 behavior would occasionally lead to unexpected string substitutions when the string value happened to match a variable name, possibly one defined somewhere quite far from that part of the project. The potential confusion around quoted values means it is generally advisable to avoid using quoted arguments with the `if(something)` form. There are usually better comparison expressions that handle strings more robustly, which are covered in Section 6.1.3, "Comparison Tests" further below.

6.1.2. Logic Operators

CMake supports the usual `AND`, `OR` and `NOT` logical operators, as well as parentheses to control order of precedence.

```
# Logical operators
if(NOT expression)
if(expression1 AND expression2)
if(expression1 OR expression2)

# Example with parentheses
if(NOT (expression1 AND (expression2 OR expression3)))
```

Following usual conventions, expressions inside parentheses are evaluated first, beginning with the innermost parentheses.

6.1.3. Comparison Tests

CMake separates comparison tests into three distinct categories: *numeric*, *string* and *version numbers*, but the syntax forms all follow the same pattern:

```
if(value1 OPERATOR value2)
```

The two operands, `value1` and `value2`, can be either variable names or (possibly quoted) values. If a value is the same as the name of a defined variable, it will be treated as a variable. Otherwise, it is treated as a string or value directly. Once again though, quoted values have ambiguous behavior similar to that in basic unary expressions. Prior to CMake 3.1, a quoted string with a value that

Exclusively for sbmain17@gmail.com Transaction: 0098315591

matched a variable name would be replaced by that variable's value. The behavior of CMake 3.1 and later uses the quoted value without substitution, which is what developers intuitively expect.

All three comparison categories support the same set of operations, but the `OPERATOR` names are different for each category. The following table summarizes the supported operators:

Numeric	String	Version numbers
`LESS`	`STRLESS`	`VERSION_LESS`
`GREATER`	`STRGREATER`	`VERSION_GREATER`
`EQUAL`	`STREQUAL`	`VERSION_EQUAL`
`LESS_EQUAL`[1]	`STRLESS_EQUAL`[1]	`VERSION_LESS_EQUAL`[1]
`GREATER_EQUAL`[1]	`STRGREATER_EQUAL`[1]	`VERSION_GREATER_EQUAL`[1]

[1] Only available with CMake 3.7 and later.

Numeric comparison works as one would expect, comparing the value of the left against the right. Note, however, that CMake does not typically raise an error if either operand is not a number and its behavior does not fully conform to the official documentation when values contain more than just digits. Depending on the mix of digits and non-digits, the result of the expression may be true or false.

```
# Valid numeric expressions, all evaluating as true
if(2 GREATER 1)
if("23" EQUAL 23)
set(val 42)
if(${val} EQUAL 42)
if("${val}" EQUAL 42)

# Invalid expression that evaluates as true with at
# least some CMake versions. Do not rely on this behavior.
if("23a" EQUAL 23)
```

Version number comparisons are somewhat like an enhanced form of numerical comparisons. Version numbers are assumed to be in the form `major[.minor[.patch[.tweak]]]` where each component is expected to be a non-negative integer. When comparing two version numbers, the `major` part is compared first. Only if the `major` components are equal will the `minor` parts be compared (if present) and so on. A missing component is treated as zero. In all of the following examples, the expression evaluates to true:

```
if(1.2   VERSION_EQUAL   1.2.0)
if(1.2   VERSION_LESS    1.2.3)
if(1.2.3 VERSION_GREATER 1.2  )
if(2.0.1 VERSION_GREATER 1.9.7)
if(1.8.2 VERSION_LESS    2    )
```

The version number comparisons have the same robustness caveats as numeric comparisons. Each version component is expected to be an integer, but the comparison result is essentially undefined if this restriction does not hold.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

For strings, values are compared lexicographically. No assumptions are made about the contents of the strings, but be mindful of the potential for the variable/string substitution situation described earlier. String comparisons are one of the most common situations where such unexpected substitutions occur.

CMake also supports testing a string against a regular expression:

```
if(value MATCHES regex)
```

The `value` again follows the variable-or-string rules defined above and is compared against the `regex` regular expression. If the `value` matches, the expression evaluates to true. While the CMake documentation doesn't define the supported regular expression syntax for `if()` commands, it does define it elsewhere for other commands (e.g. see the `string()` command documentation). Essentially, CMake supports basic regular expression syntax only.

Parentheses can be used to capture parts of the matched value. The command will set variables with names of the form `CMAKE_MATCH_<n>` where `<n>` is the group to match. The entire matched string is stored in group 0.

```
if("Hi from ${who}" MATCHES "Hi from (Fred|Barney).*")
    message("${CMAKE_MATCH_1} says hello")
endif()
```

6.1.4. File System Tests

CMake also includes a set of tests which can be used to query the file system:

```
if(EXISTS pathToFileOrDir)
if(IS_DIRECTORY pathToDir)
if(IS_SYMLINK fileName)
if(IS_ABSOLUTE path)
if(file1 IS_NEWER_THAN file2)
```

Unlike most other `if` expressions, none of the above operators perform any variable/string substitution without `${}`, regardless of any quoting.

Each of the above operators should be self-explanatory, except for `IS_NEWER_THAN`. Unfortunately, `IS_NEWER_THAN` is an inaccurate name for what that operator does. It also returns true if both files have the *same* timestamp, not just if the timestamp of `file1` is newer than that of `file2`. This becomes especially important on file systems that only have timestamps with a resolution of one second, such as the HFS+ file system on macOS 10.12 and earlier. On such systems, it is very common to encounter scenarios where files have the same timestamp, even when those files are created by separate commands. Another less intuitive behavior is that it also returns true if *either* file is missing. Furthermore, if either file is not specified as an absolute path, the behavior is undefined. It will often therefore be necessary to use `IS_NEWER_THAN` in a negated way to obtain the desired condition.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

Consider a scenario where `secondFile` is generated from `firstFile`. If `firstFile` is updated or `secondFile` is missing, then `secondFile` needs to be recreated. If `firstFile` does not exist, it should be a fatal error. Such logic would need to be expressed like so:

```
set(firstFile  "/full/path/to/somewhere")
set(secondFile "/full/path/to/another/file")

if(NOT EXISTS ${firstFile})
    message(FATAL_ERROR "${firstFile} is missing")
elseif(NOT EXISTS ${secondFile} OR
       NOT ${secondFile} IS_NEWER_THAN ${firstFile})
    # ... commands to recreate secondFile
endif()
```

One might naively think that the condition could be expressed like this instead:

```
# WARNING: Very likely to be wrong
if(${firstFile} IS_NEWER_THAN ${secondFile})
    # ... commands to recreate secondFile
endif()
```

Although the words might express the desired condition, it doesn't do what it appears to because it also returns true if the two files have the same timestamp. If the operation to recreate `secondFile` is fast and the file system only has second timestamp resolution, it is very likely that `secondFile` would be recreated every time CMake is run. If build steps depend on `secondFile`, the build would also end up rebuilding those things after every CMake run.

6.1.5. Existence Tests

The last category of `if` expressions support testing whether or not various CMake entities exist. They can be particularly useful in larger, more complex projects where some parts might or might not be present or be enabled.

```
if(DEFINED name)
if(COMMAND name)
if(POLICY name)
if(TARGET name)
if(TEST name)   # Available from CMake 3.4 onward
```

Each of the above will return true if an entity of the specified `name` exists at the point where the `if` command is issued.

`DEFINED`

Returns true if a variable of the specified `name` exists. The value of the variable is irrelevant, only its existence is tested. The variable can be a regular CMake variable or it can be a cache variable. From CMake 3.14, it is possible to check for a cache variable only using the `CACHE{name}` form. All CMake versions also support testing for the existence of an environment variable using the `ENV{name}` form, even though this was only officially documented as supported from CMake 3.13.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

```
if(DEFINED SOMEVAR)         # Checks for a CMake variable (regular or cache)
if(DEFINED CACHE{SOMEVAR})  # Checks for a CMake cache variable
if(DEFINED ENV{SOMEVAR})    # Checks for an environment variable
```

COMMAND

Tests whether a CMake command, function or macro with the specified name exists. This can be useful for checking whether something is defined before trying to use it. For CMake-provided commands, prefer to test the CMake version instead, but for project-supplied functions and macros (see Chapter 8, *Functions And Macros*), this can be an appropriate check.

POLICY

Tests whether a particular policy is known to CMake. Policy names are usually of the form `CMPxxxx`, where `xxxx` is a four digit number. See Chapter 12, *Policies* for details on this topic.

TARGET

Returns true if a CMake target of the specified name has been defined by one of the commands `add_executable()`, `add_library()` or `add_custom_target()`. The target could have been defined in any directory, as long as it is known at the point where the `if` test is performed. This test is particularly useful in complex project hierarchies that pull in other external projects and where those projects may share common dependent subprojects (i.e. this sort of `if` test can be used to check if a target is already defined before trying to create it).

TEST

Returns true if a CMake test with the specified name has been previously defined by the `add_test()` command (covered in detail in Chapter 25, *Testing*).

Lastly, the following form returns true if `listVar` contains the specified `value`, where `value` follows the usual variable-or-string rules but `listVar` must be the name of a list variable.

```
if(value IN_LIST listVar)  # Available since CMake 3.3
```

6.1.6. Common Examples

A few uses of `if()` are so common, they deserve special mention. Many of these rely on predefined CMake variables for their logic, especially variables relating to the compiler and target platform. Unfortunately, it is common to see such expressions based on the wrong variables. For example, consider a project which has two C++ source files, one for building with Visual Studio compilers or those compatible with them (e.g. Intel) and another for building with all other compilers. Such logic is frequently implemented like so:

```
if(WIN32)
    set(platformImpl source_win.cpp)
else()
    set(platformImpl source_generic.cpp)
endif()
```


Exclusively for sbmain17@gmail.com Transaction: 0098315591

While this will likely work for the majority of projects, it doesn't actually express the right constraint. Consider, for example, a project built on Windows but using the MinGW compiler. For such cases, `source_generic.cpp` may be the more appropriate source file. The above could be more accurately implemented as follows:

```
if(MSVC)
    set(platformImpl source_msvc.cpp)
else()
    set(platformImpl source_generic.cpp)
endif()
```

Another example involves conditional behavior based on the CMake generator being used. In particular, CMake offers additional features when building with the Xcode generator which no other generators support. Projects sometimes make the assumption that building for macOS means the Xcode generator will be used, but this doesn't have to be the case (and often isn't). The following incorrect logic is sometimes used:

```
if(APPLE)
    # Some Xcode-specific settings here...
else()
    # Things for other platforms here...
endif()
```

Again, this may seem to do the right thing, but if a developer tries to use a different generator (e.g. Ninja or Unix Makefiles) on macOS, the logic fails. Testing the platform with the expression `APPLE` doesn't express the right condition, the CMake generator should be tested instead:

```
if(CMAKE_GENERATOR STREQUAL "Xcode")
    # Some Xcode-specific settings here...
else()
    # Things for other CMake generators here...
endif()
```

The above examples are both cases of testing the platform instead of the entity the constraint actually relates to. This is understandable, since the platform is one of the simplest things to understand and test, but using it instead of the more accurate constraint can unnecessarily limit the generator choices available to developers, or it may result in the wrong behavior entirely.

Another common example, this time used appropriately, is the conditional inclusion of a target based on whether or not a particular CMake option has been set.

```
option(BUILD_MYLIB "Enable building the MyLib target")
if(BUILD_MYLIB)
    add_library(MyLib src1.cpp src2.cpp)
endif()
```

Exclusively for sbmain17@gmail.com Transaction: 0098315591

More complex projects often use the above pattern to conditionally include subdirectories or perform a variety of other tasks based on a CMake option or cache variable. Developers can then turn that option on/off or set the variable to non-default values without having to edit the `CMakeLists.txt` file directly. This is especially useful for scripted builds driven by continuous integration systems, etc. which may want to enable or disable certain parts of the build.

6.2. Looping

Another common need in many CMake projects is to perform some action on a list of items or for a range of values. Alternatively, some action may need to be performed repeatedly until a particular condition is met. These needs are well covered by CMake, offering the traditional behavior with some additions to make working with CMake features a little easier.

6.2.1. foreach()

CMake provides the `foreach()` command to enable projects to iterate over a set of items or values. There are a few different forms of `foreach()`, the most basic of which is:

```
foreach(loopVar arg1 arg2 ...)
    # ...
endforeach()
```

In the above form, for each `argN` value, `loopVar` is set to that argument and the loop body is executed. No variable/string test is performed, the arguments are used exactly as the values are specified. Rather than listing out each item explicitly, the arguments can also be specified by one or more list variables using the more general form of the command:

```
foreach(loopVar IN [LISTS listVar1 ...] [ITEMS item1 ...])
    # ...
endforeach()
```

In this more general form, individual arguments can still be specified using the `ITEMS` keyword, but the `LISTS` keyword allows one or more list variables to be specified. Either `ITEMS` or `LISTS` (or both) must be provided when using this more general form. When both are provided, the `ITEMS` must appear after the `LISTS`. It is permitted for the `listVarN` list variables to hold an empty list. An example should help clarify this more general form's usage.

```
set(list1 A B)
set(list2)
set(foo WillNotBeShown)
foreach(loopVar IN LISTS list1 list2 ITEMS foo bar)
    message("Iteration for: ${loopVar}")
endforeach()
```

The output from the above would be:

Exclusively for sbmain17@gmail.com Transaction: 0098315591

```
Iteration for: A
Iteration for: B
Iteration for: foo
Iteration for: bar
```

CMake 3.17 added a more specialized form for looping over multiple lists at once:

```
foreach(loopVar... IN ZIP_LISTS listVar...)
    # ...
endforeach()
```

If only one `loopVar` is given, then the command will set variables of the form `loopVar_N` at each iteration, where `N` corresponds to the `listVarN` variable. Numbering starts from 0. If there is one `loopVar` for each `listVar`, then the command maps them one-to-one instead of creating `loopVar_N` variables. The following example demonstrates the two cases:

```
set(list0 A B)
set(list1 one two)

foreach(var0 var1 IN ZIP_LISTS list0 list1)
    message("Vars: ${var0} ${var1}")
endforeach()

foreach(var IN ZIP_LISTS list0 list1)
    message("Vars: ${var_0} ${var_1}")
endforeach()
```

Both `foreach()` loops will print the same output:

```
Vars: A one
Vars: B two
```

The lists to be "zipped" in this way do not have to be the same length. The associated iteration variable will be undefined when iteration moves past the end of the shorter list. Taking the value of an undefined variable results in an empty string. The next example demonstrates the behavior:

```
set(long  A B C)
set(short justOne)

foreach(varLong varShort IN ZIP_LISTS long short)
    message("Vars: ${varLong} ${varShort}")
endforeach()
```

```
Vars: A justOne
Vars: B
Vars: C
```

Exclusively for sbmain17@gmail.com Transaction: 0098315591

The `foreach()` command also supports the more C-like iteration over a range of numerical values:

```
foreach(loopVar RANGE start stop [step])
```

When using the `RANGE` form of `foreach()`, the loop is executed with `loopVar` set to each value in the range `start` to `stop` (inclusive). If the `step` option is provided, then this value is added to the previous one after each iteration and the loop stops when the result of that is greater than `stop`.

The `RANGE` form also accepts just one argument like so:

```
foreach(loopVar RANGE value)
```

This is equivalent to `foreach(loopVar RANGE 0 value)`, which means the loop body will execute `(value + 1)` times. This is unfortunate, since the more intuitive expectation is probably that the loop body executes `value` times. For this reason, it is likely to be clearer to avoid using this second `RANGE` form and explicitly specify both the `start` and `stop` values instead.

Similar to the situation for the `if()` and `endif()` commands, in very early versions of CMake (i.e. prior to 2.8.0), all forms of the `foreach()` command required that the `loopVar` also be specified as an argument to `endforeach()`. Again, this harms readability and offers little benefit, so specifying the `loopVar` with `endforeach()` is discouraged for new projects.

6.2.2. while()

The other looping command offered by CMake is `while()`:

```
while(condition)
    # ...
endwhile()
```

The `condition` is tested and if it evaluates to true (following the same rules as the expression in `if()` statements), then the loop body is executed. This is repeated until `condition` evaluates to false or the loop is exited early (see next section). Again, in CMake versions prior to 2.8.0, the `condition` had to be repeated in the `endwhile()` command, but this is no longer necessary and is actively discouraged for new projects.

6.2.3. Interrupting Loops

Both `while()` and `foreach()` loops support the ability to exit the loop early with `break()` or to skip to the start of the next iteration with `continue()`. These commands behave just like their similarly named C language counterparts and both operate only on the inner-most enclosing loop. The following example illustrates the behavior.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

```
foreach(outerVar IN ITEMS a b c)
    unset(s)

    foreach(innerVar IN ITEMS 1 2 3)
        # Stop inner loop once string s gets long
        list(APPEND s "${outerVar}${innerVar}")
        string(LENGTH "${s}" length)
        if(length GREATER 5)
            # End the innerVar foreach loop early
            break()
        endif()

        # Do no more processing if outerVar is "b"
        if(outerVar STREQUAL "b")
            # End current innerVar iteration and move on to next innerVar item
            continue()
        endif()

        message("Processing ${outerVar}-${innerVar}")
    endforeach()

    message("Accumulated list: ${s}")
endforeach()
```

The output from the above example would be:

```
Processing a-1
Processing a-2
Accumulated list: a1;a2;a3
Accumulated list: b1;b2;b3
Processing c-1
Processing c-2
Accumulated list: c1;c2;c3
```

6.3. Recommended Practices

Minimize opportunities for strings to be unintentionally interpreted as variables in `if()`, `foreach()` and `while()` commands. Avoid unary expressions with quotes, prefer to use a string comparison operation instead. Strongly prefer to set a minimum CMake version of at least 3.1 to disable the old behavior that allowed implicit conversion of quoted string values to variable names.

When regular expression matching in `if(xxx MATCHES regex)` commands and the group capture variables are needed, it is generally advisable to store the `CMAKE_MATCH_<n>` match results in ordinary variables as soon as possible. These variables will be overwritten by the next command that does any sort of regular expression operation.

Prefer to use looping commands which avoid ambiguous or misleading code. If using the RANGE form of `foreach()`, always specify both the start and end values. If iterating over items, consider whether using the `IN LISTS` or `IN ITEMS` forms communicate more clearly what is being done rather than a bare `foreach(loopVar item1 item2 ⋯)` form.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

Chapter 7. Using Subdirectories

For simple projects, keeping everything in one directory is fine, but most real world projects tend to split their files across multiple directories. It is common to find different file types or individual modules grouped under their own directories, or for files belonging to logical functional groupings to be in their own part of the project's directory hierarchy. While the directory structure may be driven by how developers think of the project, the way the project is structured also impacts the build system.

Two fundamental CMake commands in any multi-directory project are `add_subdirectory()` and `include()`. These commands bring content from another file or directory into the build, allowing the build logic to be distributed across the directory hierarchy rather than forcing everything to be defined at the top-most level. This offers a number of advantages:

- Build logic is *localized*, meaning that characteristics of the build can be defined in the directory where they have the most relevance.
- Builds can be composed of subcomponents which are defined independently from the top level project consuming them. This is especially important if a project makes use of things like git submodules or embeds third party source trees.
- Because directories can be self-contained, it becomes relatively trivial to turn parts of the build on or off simply by choosing whether or not to add in that directory.

`add_subdirectory()` and `include()` have quite different characteristics, so it is important to understand the strengths and weaknesses of both.

7.1. add_subdirectory()

The `add_subdirectory()` command allows a project to bring another directory into the build. That directory must have its own `CMakeLists.txt` file which will be processed at the point where `add_subdirectory()` is called and a corresponding directory will be created for it in the project's build tree.

```
add_subdirectory(sourceDir [binaryDir] [EXCLUDE_FROM_ALL])
```

The `sourceDir` does not have to be a subdirectory within the source tree, although it usually is. Any directory can be added, with `sourceDir` being specified as either an absolute or relative path, the latter being relative to the current *source* directory. Absolute paths are typically only needed when adding directories that are outside the main source tree.

Normally, the `binaryDir` does not need to be specified. When omitted, CMake creates a directory in the build tree with the same name as the `sourceDir`. If `sourceDir` contains any path components, these will be mirrored in the `binaryDir` created by CMake. Alternatively, the `binaryDir` can be explicitly specified as either an absolute or relative path, with the latter being evaluated relative to the current *binary* directory (discussed in more detail shortly). If `sourceDir` is a path outside the source tree, CMake requires the `binaryDir` to be specified since a corresponding relative path can no longer be constructed automatically.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

The optional `EXCLUDE_FROM_ALL` keyword is intended to control whether targets defined in the subdirectory being added should be included in the project's `ALL` target by default. Unfortunately, for some CMake versions and project generators, it doesn't always act as expected and can even result in broken builds.

7.1.1. Source And Binary Directory Variables

Sometimes a developer needs to know the location of the build directory corresponding to the current source directory, such as when copying files needed at run time or to perform a custom build task. With `add_subdirectory()`, both the source and the build trees' directory structures can be arbitrarily complex. There could even be multiple build trees being used with the same source tree. The developer therefore needs some assistance from CMake to determine the directories of interest. To that end, CMake provides a number of variables which keep track of the source and binary directories for the `CMakeLists.txt` file currently being processed. The following read-only variables are updated automatically as each file is processed by CMake. They always contain absolute paths.

`CMAKE_SOURCE_DIR`

The top-most directory of the *source* tree (i.e. where the top-most `CMakeLists.txt` file resides). This variable never changes its value.

`CMAKE_BINARY_DIR`

The top-most directory of the *build* tree. This variable never changes its value.

`CMAKE_CURRENT_SOURCE_DIR`

The directory of the `CMakeLists.txt` file currently being processed by CMake. It is updated each time a new file is processed as a result of an `add_subdirectory()` call and is restored back again when processing of that directory is complete.

`CMAKE_CURRENT_BINARY_DIR`

The build directory corresponding to the `CMakeLists.txt` file currently being processed by CMake. It changes for every call to `add_subdirectory()` and is restored again when `add_subdirectory()` returns.

An example should help demonstrate the behavior:

Top level CMakeLists.txt

```
cmake_minimum_required(VERSION 3.0)
project(MyApp)

message("top:   CMAKE_SOURCE_DIR         = ${CMAKE_SOURCE_DIR}")
message("top:   CMAKE_BINARY_DIR         = ${CMAKE_BINARY_DIR}")
message("top:   CMAKE_CURRENT_SOURCE_DIR = ${CMAKE_CURRENT_SOURCE_DIR}")
message("top:   CMAKE_CURRENT_BINARY_DIR = ${CMAKE_CURRENT_BINARY_DIR}")

add_subdirectory(mysub)

message("top:   CMAKE_CURRENT_SOURCE_DIR = ${CMAKE_CURRENT_SOURCE_DIR}")
message("top:   CMAKE_CURRENT_BINARY_DIR = ${CMAKE_CURRENT_BINARY_DIR}")
```

Exclusively for sbmain17@gmail.com Transaction: 0098315591

mysub/CMakeLists.txt

```
message("mysub: CMAKE_SOURCE_DIR         = ${CMAKE_SOURCE_DIR}")
message("mysub: CMAKE_BINARY_DIR         = ${CMAKE_BINARY_DIR}")
message("mysub: CMAKE_CURRENT_SOURCE_DIR = ${CMAKE_CURRENT_SOURCE_DIR}")
message("mysub: CMAKE_CURRENT_BINARY_DIR = ${CMAKE_CURRENT_BINARY_DIR}")
```

For the above example, if the top level `CMakeLists.txt` file was in the directory `/somewhere/src` and the build directory was `/somewhere/build`, the following output would be generated:

```
top:   CMAKE_SOURCE_DIR         = /somewhere/src
top:   CMAKE_BINARY_DIR         = /somewhere/build
top:   CMAKE_CURRENT_SOURCE_DIR = /somewhere/src
top:   CMAKE_CURRENT_BINARY_DIR = /somewhere/build
mysub: CMAKE_SOURCE_DIR         = /somewhere/src
mysub: CMAKE_BINARY_DIR         = /somewhere/build
mysub: CMAKE_CURRENT_SOURCE_DIR = /somewhere/src/mysub
mysub: CMAKE_CURRENT_BINARY_DIR = /somewhere/build/mysub
top:   CMAKE_CURRENT_SOURCE_DIR = /somewhere/src
top:   CMAKE_CURRENT_BINARY_DIR = /somewhere/build
```

7.1.2. Scope

In Chapter 5, *Variables*, the concept of *scope* was mentioned briefly. One of the effects of calling `add_subdirectory()` is that CMake creates a new scope for processing that directory's `CMakeLists.txt` file. That new scope acts like a child of the calling scope and there are a number of effects:

- All variables defined in the calling scope will be visible to the child scope and the child scope can read their values like any other variable.
- Any new variable created in the child scope will not be visible to the calling scope.
- Any change to a variable in the child scope is local to that child scope. Even if that variable existed in the calling scope, the calling scope's variable is left unchanged. The variable modified in the child scope acts like a new variable that is discarded when processing leaves the child scope.

Put another way, upon entry into the child scope, it receives a copy of all of the variables defined in the calling scope at that point in time. Any changes to variables in the child are performed on the child's copy, leaving the caller's variables unchanged. An example best illustrates the behavior:

CMakeLists.txt

```
set(myVar foo)
message("Parent (before): myVar    = ${myVar}")
message("Parent (before): childVar = ${childVar}")

add_subdirectory(subdir)

message("Parent (after):  myVar    = ${myVar}")
message("Parent (after):  childVar = ${childVar}")
```


Exclusively for sbmain17@gmail.com Transaction: 0098315591

subdir/CMakeLists.txt

```
message("Child  (before): myVar    = ${myVar}")
message("Child  (before): childVar = ${childVar}")

set(myVar bar)
set(childVar fuzz)

message("Child  (after):  myVar    = ${myVar}")
message("Child  (after):  childVar = ${childVar}")
```

This produces the following output:

```
Parent (before): myVar    = foo     ①
Parent (before): childVar =         ②
Child  (before): myVar    = foo     ③
Child  (before): childVar =         ④
Child  (after):  myVar    = bar     ⑤
Child  (after):  childVar = fuzz    ⑥
Parent (after):  myVar    = foo     ⑦
Parent (after):  childVar =         ⑧
```

① `myVar` is defined at the parent level.

② `childVar` is not defined at the parent level, so it evaluates to an empty string.

③ `myVar` is still visible in the child scope.

④ `childVar` is still undefined in the child scope before it is set.

⑤ `myVar` is modified in the child scope.

⑥ `childVar` is has been set in the child scope.

⑦ When processing returns to the parent scope, `myVar` still has the value from before the call to `add_subdirectory()`. The modification to `myVar` in the child scope is not visible to the parent.

⑧ `childVar` was defined in the child scope, so it is not visible to the parent and evaluates to an empty string.

The above behavior of scoping for variables highlights one of the important characteristics of `add_subdirectory()`. It allows the added directory to change whatever variables it wants without affecting variables in the calling scope. This helps keep the calling scope isolated from potentially unwanted changes.

There are times, however, where it is desirable for a variable change made in an added directory to be visible to the caller. For example, the directory may be responsible for collecting a set of source file names and passing it back up to the parent as a list of files. This is the purpose of the `PARENT_SCOPE` keyword in the `set()` command. When `PARENT_SCOPE` is used, the variable being set is the one in the parent scope, not the one in the current scope. Importantly, it does *not* mean set the variable in both the parent *and* the current scope. Modifying the previous example slightly, the effect of `PARENT_SCOPE` becomes clear:

Exclusively for sbmain17@gmail.com Transaction: 0098315591

CMakeLists.txt

```
set(myVar foo)
message("Parent (before): myVar = ${myVar}")
add_subdirectory(subdir)
message("Parent (after):  myVar = ${myVar}")
```

subdir/CMakeLists.txt

```
message("Child  (before): myVar = ${myVar}")
set(myVar bar PARENT_SCOPE)
message("Child  (after):  myVar = ${myVar}")
```

This produces the following output:

```
Parent (before): myVar = foo
Child  (before): myVar = foo
Child  (after):  myVar = foo    ①
Parent (after):  myVar = bar    ②
```

① The `myVar` in the child scope is not affected by the `set()` call because the `PARENT_SCOPE` keyword tells CMake to modify the parent's `myVar`, not the local one.

② The parent's `myVar` has been modified by the `set()` call in the child scope.

Because the use of `PARENT_SCOPE` prevents any local variable of the same name from being modified by the command, it can be less misleading if the local scope does not reuse the same variable name as one from the parent. In the above example, a clearer set of commands would be:

subdir/CMakeLists.txt

```
set(localVar bar)
set(myVar ${localVar} PARENT_SCOPE)
```

Obviously the above is a trivial example, but for real world projects, there may be many commands which contribute to building up the value of `localVar` before finally setting the parent's `myVar` variable.

It's not just variables that are affected by scope, policies and some properties also have similar behavior to variables in this regard. In the case of policies, each `add_subdirectory()` call creates a new scope in which policy changes can be made without affecting the policy settings of the parent. Similarly, there are directory properties which can be set in the child directory's `CMakeLists.txt` file which will have no effect on the parent's directory properties. Both of these are covered in more detail in their own respective chapters: Chapter 12, *Policies* and Chapter 9, *Properties*.

7.1.3. When To Call project()

A question that sometimes arises is whether to call `project()` in the `CMakeLists.txt` files of subdirectories. In most cases, it is not necessary or desirable to do so, but it is permitted. The only place where `project()` must be called is the top-most `CMakeLists.txt` file. Upon reading the top level

Exclusively for sbmain17@gmail.com Transaction: 0098315591

`CMakeLists.txt` file, CMake scans that file's contents looking for a call to `project()`. If no such call is found, CMake will issue a warning and insert an internal call to `project()` with the default C and C++ languages enabled. Projects should never rely on this mechanism, they should always explicitly call `project()` themselves. Note that it is not enough to call `project()` through a wrapper function or via a file read in via `add_subdirectory()` or `include()`, the top level `CMakeLists.txt` file must call `project()` *directly*.

Calling `project()` in subdirectories typically does no harm, but it may result in CMake having to generate extra files. For the most part, these extra `project()` calls and generated files are just noise, but in some cases they can be useful. When using a Visual Studio project generator, each `project()` command results in the creation of an associated solution file. Normally, the developer would load the solution file corresponding to the top-most `project()` call (that solution file will be at the top of the build directory). This top level solution file contains all the targets in the project. The solution files generated for any `project()` calls within subdirectories will contain a more trimmed down view, containing just the targets from that directory scope and below, plus any other targets from the rest of the build that they depend on. Developers can load these sub-solutions instead of the top level one for a more trimmed-down view of the project, allowing them to focus on a smaller subset of the set of targets. For very large projects with many targets, this can be especially useful.

The Xcode generator behaves in a similar way, creating an Xcode project for each `project()` call. These Xcode projects can be loaded for a similar trimmed down view, but unlike for Visual Studio generators, they do not include the logic for building targets from outside of that directory scope or below. The developer is responsible for ensuring that anything required from outside of that trimmed down view has been built already. In practice, this means the top level project likely needs to be loaded and built first before switching to the trimmed down Xcode project.

7.2. include()

The other method CMake provides for pulling in content from other directories is the `include()` command, which has the following two forms:

```
include(fileName [OPTIONAL] [RESULT_VARIABLE myVar] [NO_POLICY_SCOPE])
include(module   [OPTIONAL] [RESULT_VARIABLE myVar] [NO_POLICY_SCOPE])
```

The first form is somewhat analogous to `add_subdirectory()`, but there are important differences:

- `include()` expects the name of a file to read in, whereas `add_subdirectory()` expects a directory and will look for a `CMakeLists.txt` file within that directory. The file name passed to `include()` typically has the extension `.cmake`, but it can be anything.
- `include()` does not introduce a new variable scope, whereas `add_subdirectory()` does.
- Both commands introduce a new policy scope by default, but the `include()` command can be told not to do so with the `NO_POLICY_SCOPE` option (`add_subdirectory()` has no such option). See Chapter 12, *Policies* for further details on policy scope handling.
- The value of the `CMAKE_CURRENT_SOURCE_DIR` and `CMAKE_CURRENT_BINARY_DIR` variables do not change when processing the file named by `include()`, whereas they do change for `add_subdirectory()`. This will be discussed in more detail shortly.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

The second form of the `include()` command serves an entirely different purpose. It is used to load the named *module*, a topic covered in depth in Chapter 11, *Modules*. All but the first of the above points also hold true for this second form.

Since the value of `CMAKE_CURRENT_SOURCE_DIR` does not change when `include()` is called, it may seem difficult for the included file to work out the directory in which it resides. `CMAKE_CURRENT_SOURCE_DIR` will contain the location of the file from where `include()` was called, not the directory containing the included file. Furthermore, unlike `add_subdirectory()` where the `fileName` will always be `CMakeLists.txt`, the name of the file can be anything when using `include()`, so it can be difficult for the included file to determine its own name. To address situations like these, CMake provides an additional set of variables:

`CMAKE_CURRENT_LIST_DIR`

Analogous to `CMAKE_CURRENT_SOURCE_DIR` except it *will* be updated when processing the included file. This is the variable to use where the directory of the current file being processed is required, no matter how it has been added to the build. It will always hold an absolute path.

`CMAKE_CURRENT_LIST_FILE`

Always gives the name of the file currently being processed. It always holds an absolute path to the file, not just the file name.

`CMAKE_CURRENT_LIST_LINE`

Holds the line number of the file currently being processed. This variable is rarely needed, but may prove useful in some debugging scenarios.

It is important to note that the above three variables work for *any* file being processed by CMake, not just those pulled in by an `include()` command. They have the same values as described above even for a `CMakeLists.txt` file pulled in via `add_subdirectory()`, in which case `CMAKE_CURRENT_LIST_DIR` would have the same value as `CMAKE_CURRENT_SOURCE_DIR`. The following example demonstrates the behavior:

CMakeLists.txt

```
add_subdirectory(subdir)
message("")
include(subdir/CMakeLists.txt)
```

subdir/CMakeLists.txt

```
message("CMAKE_CURRENT_SOURCE_DIR = ${CMAKE_CURRENT_SOURCE_DIR}")
message("CMAKE_CURRENT_BINARY_DIR = ${CMAKE_CURRENT_BINARY_DIR}")
message("CMAKE_CURRENT_LIST_DIR   = ${CMAKE_CURRENT_LIST_DIR}")
message("CMAKE_CURRENT_LIST_FILE  = ${CMAKE_CURRENT_LIST_FILE}")
message("CMAKE_CURRENT_LIST_LINE  = ${CMAKE_CURRENT_LIST_LINE}")
```

This produces output like the following:

Exclusively for sbmain17@gmail.com Transaction: 0098315591

```
CMAKE_CURRENT_SOURCE_DIR = /somewhere/src/subdir
CMAKE_CURRENT_BINARY_DIR = /somewhere/build/subdir
CMAKE_CURRENT_LIST_DIR   = /somewhere/src/subdir
CMAKE_CURRENT_LIST_FILE  = /somewhere/src/subdir/CMakeLists.txt
CMAKE_CURRENT_LIST_LINE  = 5

CMAKE_CURRENT_SOURCE_DIR = /somewhere/src
CMAKE_CURRENT_BINARY_DIR = /somewhere/build
CMAKE_CURRENT_LIST_DIR   = /somewhere/src/subdir
CMAKE_CURRENT_LIST_FILE  = /somewhere/src/subdir/CMakeLists.txt
CMAKE_CURRENT_LIST_LINE  = 5
```

The above example also highlights another interesting characteristic of the `include()` command. It can be used to include content from a file which has already been included in the build previously. If different subdirectories of a large, complex project both want to make use of CMake code in some file in a common area of the project, they may both `include()` that file independently.

7.3. Project-relative Variables

As will be seen in later chapters, there are various scenarios where paths relative to a location in the source or build directory need to be constructed. Consider one such example where a project needs to construct a path to a file that resides in its top level source directory. From Section 7.1.1, "Source And Binary Directory Variables", `CMAKE_SOURCE_DIR` seems to be a natural fit, allowing a path like `${CMAKE_SOURCE_DIR}/someFile` to be used. But consider what happens if that project is later incorporated into another parent project by bringing it into the parent build via `add_subdirectory()`. It could be used as a git submodule or fetched on demand using one of the techniques discussed in Chapter 28, *External Content* and Chapter 29, *Project Organization*. What used to be the top of the original project's source tree is now a subdirectory within the parent project's source tree. `CMAKE_SOURCE_DIR` now points to the top of the *parent* project, so the file path will be pointing to the wrong directory. A similar trap exists for `CMAKE_BINARY_DIR`.

The above scenario is encountered surprisingly often in online tutorials and older projects, but it can be easily avoided. The `project()` command sets some variables that provide a much more robust way of defining paths relative to locations in the directory hierarchy. The following variables will be available after `project()` has been called at least once:

`PROJECT_SOURCE_DIR`

The source directory of the most recent call to `project()` in the current scope or any parent scope. The name of the project (i.e. the first argument given to the `project()` command) is not relevant.

`PROJECT_BINARY_DIR`

The build directory corresponding to the source directory defined by `PROJECT_SOURCE_DIR`.

`projectName_SOURCE_DIR`

The source directory of the most recent call to `project(projectName)` in the current scope or any parent scope. Unlike `PROJECT_SOURCE_DIR`, this is tied to a specific project name and therefore to a particular call to `project()`.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

projectName_BINARY_DIR

The build directory corresponding to the source directory defined by projectName_SOURCE_DIR.

The following example demonstrates how these variables can be used (the ⋯_BINARY_DIR variables follow a similar pattern to the ⋯_SOURCE_DIR variables shown).

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.0)
project(topLevel)

message("Top level:")
message("  PROJECT_SOURCE_DIR  = ${PROJECT_SOURCE_DIR}")
message("  topLevel_SOURCE_DIR = ${topLevel_SOURCE_DIR}")

add_subdirectory(child)
```

child/CMakeLists.txt

```
message("Child:")
message("  PROJECT_SOURCE_DIR (before) = ${PROJECT_SOURCE_DIR}")

project(child)

message("  PROJECT_SOURCE_DIR (after)  = ${PROJECT_SOURCE_DIR}")
message("  child_SOURCE_DIR            = ${child_SOURCE_DIR}")

add_subdirectory(grandchild)
```

child/grandchild/CMakeLists.txt

```
message("Grandchild:")
message("  PROJECT_SOURCE_DIR  = ${PROJECT_SOURCE_DIR}")
message("  child_SOURCE_DIR    = ${child_SOURCE_DIR}")
message("  topLevel_SOURCE_DIR = ${topLevel_SOURCE_DIR}")
```

Running cmake on the top level of this project hierarchy would give output similar to the following:

```
Top level:
  PROJECT_SOURCE_DIR  = /somewhere/src
  topLevel_SOURCE_DIR = /somewhere/src
Child:
  PROJECT_SOURCE_DIR (before) = /somewhere/src
  PROJECT_SOURCE_DIR (after)  = /somewhere/src/child
  child_SOURCE_DIR            = /somewhere/src/child
Grandchild:
  PROJECT_SOURCE_DIR  = /somewhere/src/child
  child_SOURCE_DIR    = /somewhere/src/child
  topLevel_SOURCE_DIR = /somewhere/src
```


Exclusively for sbmain17@gmail.com Transaction: 0098315591

The above example shows the versatility of the project-related variables. They can be used from any part of the directory hierarchy to robustly refer to any other directory in the project. For the scenario discussed at the start of this section, using `${PROJECT_SOURCE_DIR}/someFile` or perhaps `${projectName_SOURCE_DIR}/someFile` instead of `${CMAKE_SOURCE_DIR}/someFile` would ensure that the path to `someFile` would be correct, regardless of whether the project is being built stand-alone or being incorporated into a larger project hierarchy.

Some hierarchical build arrangements allow a project to be built either stand-alone or as part of a larger parent project (see Section 28.2, "FetchContent"). Some parts of the project might only make sense if it is the top of the build, such as setting up packaging support. A project can detect whether it is the top level by comparing the value of `CMAKE_SOURCE_DIR` with `CMAKE_CURRENT_SOURCE_DIR`. If they are the same, then the current directory scope must be the top level of the source tree.

```
if(CMAKE_CURRENT_SOURCE_DIR STREQUAL CMAKE_SOURCE_DIR)
    add_subdirectory(packaging)
endif()
```

The above technique is supported by all versions of CMake and is a very common pattern. With CMake 3.21 or later, a dedicated `PROJECT_IS_TOP_LEVEL` variable is provided which can achieve the same result, but is clearer in its intent:

```
# Requires CMake 3.21 or later
if(PROJECT_IS_TOP_LEVEL)
    add_subdirectory(packaging)
endif()
```

The value of `PROJECT_IS_TOP_LEVEL` will be true if the most recent call to `project()` in the current directory scope or above was in the top level `CMakeLists.txt` file. A similar variable, `<projectName>_IS_TOP_LEVEL`, is also defined by CMake 3.21 or later for every call to `project()`. It is created as a cache variable, so it can be read from any directory scope. `<projectName>` corresponds to the name given to the `project()` command of interest. This alternative variable is useful when there may be intervening calls to `project()` between the current scope and the scope of the project of interest.

7.4. Ending Processing Early

There can be occasions where a project may want to stop processing the remainder of the current file and return control back to the caller. The `return()` command can be used for exactly this purpose, but note that it cannot return a value to the caller. Its only effect is to end processing of the current scope. If not called from inside a function, `return()` ends processing of the current file regardless of whether it was brought in via `include()` or `add_subdirectory()`. The effect of calling `return()` inside a function is covered in Section 8.4, "Scope", including special attention for a common mistake that can result in returning from the current file unintentionally.

As noted in the previous section, different parts of a project may include the same file from multiple places. It can sometimes be desirable to check for this and only include the file once, returning early for subsequent inclusions to prevent reprocessing the file multiple times. This is

Exclusively for sbmain17@gmail.com Transaction: 0098315591

very similar to the situation for C/C++ headers and it is fairly common to see a similar form of include guard used:

```
if(DEFINED cool_stuff_include_guard)
    return()
endif()

set(cool_stuff_include_guard 1)
# ...
```

With CMake 3.10 or later, this can be expressed more succinctly and robustly with a dedicated command whose behavior is analogous to the `#pragma once` of C/C++:

```
include_guard()
```

Compared to manually writing out the `if-endif` code, this is more robust because it handles the name of the guard variable internally. The command also accepts an optional keyword argument `DIRECTORY` or `GLOBAL` to specify a different scope within which to check for the file having been processed previously, but these keywords are unlikely to be needed in most situations. With neither argument specified, variable scope is assumed and the effect is exactly equivalent to the `if-endif` code above. `GLOBAL` ensures the command ends processing of the file if it has been processed before anywhere else in the project (i.e. variable scope is ignored). `DIRECTORY` checks for previous processing only within the current directory scope and below.

7.5. Recommended Practices

The best choice between using `add_subdirectory()` or `include()` to bring another directory into the build is not always obvious. On the one hand, `add_subdirectory()` is simpler and does a better job of keeping directories relatively self contained because it creates its own scope. On the other, some CMake commands have restrictions which only allow them to operate on things defined within the current file scope, so `include()` works better for those cases. Section 29.5.1, “Target Sources” discusses some aspects of this topic.

As a general guide, most simple projects are probably better off preferring to use `add_subdirectory()` over `include()`. It promotes cleaner definition of the project and allows the `CMakeLists.txt` for a given directory to focus more on just what that directory needs to define. Following this strategy will promote better locality of information throughout the project and will also tend to introduce complexity only where it is needed and where it brings useful benefits. It’s not that `include()` itself is any more complicated than `add_subdirectory()`, but the use of `include()` tends to result in paths to files needing to be more explicitly spelled out, since what CMake considers the current source directory is not that of the included file. Many of the restrictions associated with calling certain commands from different directories have been removed in more recent CMake versions too, which further strengthens the argument to prefer `add_subdirectory()`.

Irrespective of whether using `add_subdirectory()`, `include()` or a combination of both, the `CMAKE_CURRENT_LIST_DIR` variable is generally going to be a better choice than `CMAKE_CURRENT_SOURCE_DIR`. By establishing the habit of using `CMAKE_CURRENT_LIST_DIR` early, it is much

Exclusively for sbmain17@gmail.com Transaction: 0098315591

easier to switch between `add_subdirectory()` and `include()` as a project grows in complexity and to move entire directories to restructure a project.

Where possible, avoid using the `CMAKE_SOURCE_DIR` and `CMAKE_BINARY_DIR` variables, as these typically break the ability of the project to be incorporated into a larger project hierarchy. In the vast majority of cases, `PROJECT_SOURCE_DIR` and `PROJECT_BINARY_DIR`, or their project-specific equivalents `projectName_SOURCE_DIR` and `projectName_BINARY_DIR` are more appropriate variables to use.

If the project requires CMake 3.10 or later, prefer to use the `include_guard()` command without arguments instead of an explicit `if-endif` block in cases where multiple inclusion of a file must be prevented.

Avoid the practice of arbitrarily calling `project()` in the `CMakeLists.txt` of every subdirectory. Only consider putting a `project()` command in a subdirectory's `CMakeLists.txt` file if that subdirectory can be treated as a more or less standalone project. Unless the whole build has a very high number of targets, there is little need to call `project()` anywhere other than in the top level `CMakeLists.txt` file.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

Chapter 8. Functions And Macros

Looking back on the material covered in this book so far, CMake's syntax is already starting to look a lot like a programming language in its own right. It supports variables, if-then-else logic, looping and inclusion of other files to be processed. It should be no surprise to learn that CMake also supports the common programming concepts of functions and macros too. Much like their role in other programming languages, functions and macros are the primary mechanism for projects and developers to extend CMake's functionality and to encapsulate repetitive tasks in a natural way. They allow the developer to define reusable blocks of CMake code which can be called just like regular built-in CMake commands. They are also a cornerstone of CMake's own module system (covered separately in Chapter 11, *Modules*).

8.1. The Basics

Functions and macros in CMake have very similar characteristics to their same-named counterparts in C/C++. Functions introduce a new scope and the function arguments become variables accessible inside the function body. Macros, on the other hand, effectively paste their body into the point of the call and the macro arguments are substituted as simple string replacements. These behaviors mirror the way functions and `#define` macros work in C/C++. A CMake function or macro is defined as follows:

```
function(name [arg1 [arg2 [...]]])
   # Function body (i.e. commands) ...
endfunction()

macro(name [arg1 [arg2 [...]]])
   # Macro body (i.e. commands) ...
endmacro()
```

Once defined, the function or macro is called in exactly the same way as any other CMake command. The function or macro's body is then executed at the point of the call. For example:

```
function(print_me)
    message("Hello from inside a function")
    message("All done")
endfunction()

# Called like so:
print_me()
```

As shown above, the `name` argument defines the name used to call the function or macro and it should only contain letters, numbers and underscores. The name will be treated case-insensitively, so upper/lowercase conventions are more a matter of style (the CMake documentation follows the convention that command names are all lowercase with words separated by underscores). Very early versions of CMake required the `name` to be repeated as an argument to `endfunction()` or `endmacro()`, but new projects should avoid this as it only adds unnecessary clutter.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

8.2. Argument Handling Essentials

The argument handling of functions and macros is the same except for one very important difference. For functions, each argument is a CMake variable and has all the usual behaviors of a CMake variable. For example, they can be tested in `if()` statements as variables. In comparison, macro arguments are string replacements, so whatever was used as the argument to the macro call is essentially pasted into wherever that argument appears in the macro body. If a macro argument is used in an `if()` statement, it would be treated as a string rather than a variable. The following example and its output demonstrate the difference:

```
function(func arg)
    if(DEFINED arg)
        message("Function arg is a defined variable")
    else()
        message("Function arg is NOT a defined variable")
    endif()
endfunction()

macro(macr arg)
    if(DEFINED arg)
        message("Macro arg is a defined variable")
    else()
        message("Macro arg is NOT a defined variable")
    endif()
endmacro()

func(foobar)
macr(foobar)
```

```
Function arg is a defined variable
Macro arg is NOT a defined variable
```

Aside from that difference, functions and macros both support the same features when it comes to argument processing. Each argument in the function definition serves as a case-sensitive label for the argument it represents. For functions, that label acts like a variable, whereas for macros it acts like a string substitution. The value of that argument can be accessed in the function or macro body using the usual variable notation, even though macro arguments are not technically variables.

```
function(func myArg)
    message("myArg = ${myArg}")
endfunction()

macro(macr myArg)
    message("myArg = ${myArg}")
endmacro()

func(foobar)
macr(foobar)
```

Exclusively for sbmain17@gmail.com Transaction: 0098315591

Both the call to `func()` and the call to `macr()` print the same thing:

```
myArg = foobar
```

In addition to the named arguments, functions and macros come with a set of automatically defined variables (or variable-like names in the case of macros) which allow processing of arguments in addition to or instead of the named ones:

ARGC

This will be set to the total number of arguments passed to the function. It counts the named arguments plus any additional unnamed arguments that were given.

ARGV

This is a list containing each of the arguments passed to the function, including both the named arguments and any additional unnamed arguments that were given.

ARGN

Like `ARGV`, except this only contains arguments beyond the named ones (i.e. the optional, unnamed arguments).

In addition to the above, each individual argument can be referenced with a name of the form `ARGVx` where `x` is the number of the argument (e.g. `ARGV0`, `ARGV1`, etc.). This includes the named arguments, so the first named argument could also be referenced via `ARGV0`, etc. Note that it should be considered undefined behavior to use `ARGVx` with `x >= ARGC`.

Typical situations where the `ARG…` names are used include supporting optional arguments and implementing a command which can take an arbitrary number of items to be processed. Consider a function that defines an executable target, links that target to some library and defines a test case for it. Such a function is frequently encountered when writing test cases (a topic covered in Chapter 25, *Testing*). Rather than repeating the steps for every test case, the function allows the steps to be defined once and then each test case becomes a simple one-line definition.

```
# Use a named argument for the target and treat all other
# (unnamed) arguments as the source files for the test case
function(add_mytest targetName)
    add_executable(${targetName} ${ARGN})

    target_link_libraries(${targetName} PRIVATE foobar)

    add_test(NAME    ${targetName}
             COMMAND ${targetName})
endfunction()

# Define some test cases using the above function
add_mytest(smallTest small.cpp)
add_mytest(bigTest   big.cpp algo.cpp net.cpp)
```

The above example shows the usefulness of `ARGN` in particular. It allows a function or macro to take a varying number of arguments, yet still specify a set of named arguments which must be provided.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

There is, however, a specific case to be aware of which can result in unexpected behavior. Because macros treat their arguments as string substitutions rather than as variables, if they use `ARGN` in a place where a variable name is expected, the variable it will refer to will be in the scope from which the macro is called, not the `ARGN` from the macro's own arguments. The following example highlights the situation:

```
# WARNING: This macro is misleading
macro(dangerous)
    # Which ARGN?
    foreach(arg IN LISTS ARGN)
        message("Argument: ${arg}")
    endforeach()
endmacro()

function(func)
    dangerous(1 2)
endfunction()

func(3)
```

The output from the above would be:

```
Argument: 3
```

When using the `LISTS` keyword with `foreach()`, a variable name has to be given, but the `ARGN` provided for a macro is not a variable name. When the macro is called from inside another function, the macro ends up using the `ARGN` *variable* from that enclosing function, not the `ARGN` from the macro itself. The situation becomes clear when pasting the contents of the macro body directly into the function where it is called (which is effectively what CMake will do with it):

```
function(func)
    # Now it is clear, ARGN here will use the arguments from func
    foreach(arg IN LISTS ARGN)
        message("Argument: ${arg}")
    endforeach()
endfunction()
```

In such cases, consider making the macro a function instead, or if it must remain a macro then avoid treating arguments as variables. For the above example, the implementation of `dangerous()` could be changed to use `foreach(arg IN ITEMS ${ARGN})` instead, but see Section 8.8, "Problems With Argument Handling" for some potential caveats.

8.3. Keyword Arguments

The previous section illustrated how the `ARG…` variables can be used to handle a varying set of arguments. That functionality is sufficient for the simple case where only one set of varying or optional arguments is needed, but if multiple optional or variable sets of arguments must be supported, the processing becomes quite tedious. Furthermore, the basic argument handling

Exclusively for sbmain17@gmail.com Transaction: 0098315591

described above is quite rigid compared to many of CMake's own built-in commands which support keyword-based arguments and flexible argument ordering.

Consider the `target_link_libraries()` command:

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

The `targetName` is required as the first argument, but after that, callers can provide any number of `PRIVATE`, `PUBLIC` or `INTERFACE` sections in any order, with each section permitted to contain any number of items. User-defined functions and macros can support a similar level of flexibility by using the `cmake_parse_arguments()` command, of which there are two forms. The first form is supported by all CMake versions and works for both functions and macros:

```
# Needed only for CMake 3.4 and earlier
include(CMakeParseArguments)

cmake_parse_arguments(
    prefix
    valuelessKeywords singleValueKeywords multiValueKeywords
    argsToParse...
)
```

The `cmake_parse_arguments()` command used to be provided by the `CMakeParseArguments` module, but it became a built-in command in CMake 3.5. The `include(CMakeParseArguments)` line will do nothing in CMake 3.5 and later, while for earlier versions of CMake it will define the `cmake_parse_arguments()` command (see Chapter 11, *Modules* for more on this sort of usage of `include()`).

The second form was introduced in CMake 3.7 and can only be used in functions, not macros:

```
# Available with CMake 3.7 or later, do not use in macros
cmake_parse_arguments(
    PARSE_ARGV startIndex
    prefix
    valuelessKeywords singleValueKeywords multiValueKeywords
)
```

Both forms of the command are similar, differing only in the way they take in the set of arguments to parse. With the first form, `argsToParse` will typically be given as `${ARGN}` without quotes. This provides all arguments given to the enclosing function or macro beyond the named arguments, except for a few specific corner cases that don't apply in most situations (see Section 8.8, "Problems With Argument Handling").

In the second form, the `PARSE_ARGV` option tells `cmake_parse_arguments()` to read the arguments directly from the set of `${ARGVx}` variables, with `x` ranging from `startIndex` to `(ARGC - 1)`. Because it reads variables directly, it does not support being used inside macros. As already explained in

Exclusively for sbmain17@gmail.com Transaction: 0098315591

Section 8.2, "Argument Handling Essentials", macros use string replacement rather than variables for its arguments. The main advantage of the second form is that for functions, it robustly handles the corner cases that the first form does not. If there are no named arguments for the enclosing function, then passing ${ARGV} or ${ARGN} to the first form is equivalent to giving PARSE_ARGV 0 to the second form when none of the corner cases apply.

The rest of the behavior of the two forms of the command is the same. Each of the ⋯Keywords is a list of keyword names to search for during parsing. Because they are a list, they need to be surrounded by quotes to ensure they are handled correctly. The valuelessKeywords define standalone keyword arguments which act like boolean switches. The keyword being present means one thing, its absence another. The singleValueKeywords each require exactly one additional argument after the keyword when they are used, whereas multiValueKeywords require zero or more additional arguments after the keyword. While not required, the prevailing convention is for keywords to be all uppercase, with words separated by underscores if required. Note that keywords should not be too long or they can be cumbersome to use.

When cmake_parse_arguments() returns, variables may be defined whose names consist of the specified prefix, an underscore and the name of the keyword they are associated with. For example, with a prefix of ARG, the variable corresponding to a keyword named FOO would be ARG_FOO. For each of the valuelessKeywords, the corresponding variable will be defined with the value TRUE if the keyword is present or FALSE if it is not. For each of the singleValueKeywords and multiValueKeywords, the corresponding variable will only be defined if that keyword is present and a value is provided after the keyword. An example best illustrates how the three different keyword types are defined and handled:

```
function(func)
    # Define the supported set of keywords
    set(prefix       ARG)
    set(noValues     ENABLE_NET COOL_STUFF)
    set(singleValues TARGET)
    set(multiValues  SOURCES IMAGES)

    # Process the arguments passed in
    include(CMakeParseArguments)
    cmake_parse_arguments(
        ${prefix}
        "${noValues}" "${singleValues}" "${multiValues}"
        ${ARGN}
    )

    # Log details for each supported keyword
    message("Option summary:")
    foreach(arg IN LISTS noValues)
        if(${prefix}_${arg})
            message("  ${arg} enabled")
        else()
            message("  ${arg} disabled")
        endif()
    endforeach()
```

Exclusively for sbmain17@gmail.com Transaction: 0098315591

```
    foreach(arg IN LISTS singleValues multiValues)
        # Single argument values will print as a string
        # Multiple argument values will print as a list
        message("  ${arg} = ${${prefix}_${arg}}")
    endforeach()
endfunction()

# Examples of calling with different combinations
# of keyword arguments
func(SOURCES foo.cpp bar.cpp
     TARGET MyApp
     ENABLE_NET
)
func(COOL_STUFF
     TARGET dummy
     IMAGES here.png there.png gone.png
)
```

The corresponding output would look like this:

```
Option summary:
  ENABLE_NET enabled
  COOL_STUFF disabled
  TARGET = MyApp
  SOURCES = foo.cpp;bar.cpp
  IMAGES =
Option summary:
  ENABLE_NET disabled
  COOL_STUFF enabled
  TARGET = dummy
  SOURCES =
  IMAGES = here.png;there.png;gone.png
```

The call to `cmake_parse_arguments()` in the above example could also have been written using the second form like so:

```
cmake_parse_arguments(
    PARSE_ARGV 0
    ${prefix}
    "${noValues}" "${singleValues}" "${multiValues}"
)
```

Arguments can be given to a command such that there are leftover arguments not associated with any keyword. The `cmake_parse_arguments()` command provides all leftover arguments as a list in the variable `<prefix>_UNPARSED_ARGUMENTS`. An advantage of the `PARSE_ARGV` form is that if any unparsed arguments are themselves a list, their embedded semicolons will be escaped. This preserves the original structure of the arguments, unlike the other form of the command which doesn't. The following reduced example demonstrates this more clearly:

Exclusively for sbmain17@gmail.com Transaction: 0098315591

```
function(demoArgs)
    set(noValues     "")
    set(singleValues SPECIAL)
    set(multiValues  EXTRAS)
    cmake_parse_arguments(
        PARSE_ARGV 0
        ARG
        "${noValues}" "${singleValues}" "${multiValues}"
    )

    message("Left-over args: ${ARG_UNPARSED_ARGUMENTS}")
    foreach(arg IN LISTS ARG_UNPARSED_ARGUMENTS)
        message("${arg}")
    endforeach()
endfunction()

demoArgs(burger fries "cheese;tomato" SPECIAL secretSauce)
```

```
Left-over args: burger;fries;cheese\;tomato
burger
fries
cheese;tomato
```

Inside the `demoArgs()` function, the call to `cmake_parse_arguments()` will define the variable `ARG_SPECIAL` with the value `secretSauce`. The `burger`, `fries` and `cheese;tomato` arguments do not correspond to any recognized keywords, so they are treated as leftover arguments. As the above output shows, the original `cheese;tomato` list is preserved because the `PARSE_ARGV` form was used. This important point is revisited in Section 8.8.2, "Forwarding Command Arguments".

In the above example, the `SPECIAL` keyword expects a single argument to follow it. If the call had omitted the value, `cmake_parse_arguments()` would not raise an error. With CMake 3.14 or earlier, the project would not be able to detect this situation, but with later versions, it can. With CMake 3.15 or later, the `<prefix>_KEYWORDS_MISSING_VALUES` variable will be populated with a list containing all single- or multi-value keywords that were present but which did not have any value following them. This can be demonstrated by modifying the previous example:

```
function(demoArgs)
    set(noValues     "")
    set(singleValues SPECIAL)
    set(multiValues  ORDINARY EXTRAS)
    cmake_parse_arguments(
        PARSE_ARGV 0
        ARG
        "${noValues}" "${singleValues}" "${multiValues}"
    )
    message("Keywords missing values: ${ARG_KEYWORDS_MISSING_VALUES}")
endfunction()

demoArgs(burger fries SPECIAL ORDINARY EXTRAS high low)
```

Exclusively for sbmain17@gmail.com Transaction: 0098315591

In the above, SPECIAL and ORDINARY are each immediately followed by another keyword, so they have no values associated with them. Both can or should have values, so they will both be present in the ARG_KEYWORDS_MISSING_VALUES variable populated by cmake_parse_arguments(). In the case of SPECIAL, it is probably an error, but for ORDINARY, it may still be valid since multi-value keywords can legitimately have no values. Projects should therefore be careful how they make use of <prefix>_KEYWORDS_MISSING_VALUES.

The cmake_parse_arguments() command provides considerable flexibility. While the first form of the command usually takes ${ARGN} as the set of arguments to parse, other arguments can be given. One can take advantage of this to do things like multi-level argument parsing:

```
function(libWithTest)
    # First level of arguments
    set(groups LIB TEST)
    cmake_parse_arguments(GRP "" "" "${groups}" ${ARGN})

    # Second level of arguments
    set(args SOURCES PRIVATE_LIBS PUBLIC_LIBS)
    cmake_parse_arguments(LIB  "" "TARGET" "${args}" ${GRP_LIB})
    cmake_parse_arguments(TEST "" "TARGET" "${args}" ${GRP_TEST})

    add_library(${LIB_TARGET} ${LIB_SOURCES})
    target_link_libraries(${LIB_TARGET}
        PUBLIC  ${LIB_PUBLIC_LIBS}
        PRIVATE ${LIB_PRIVATE_LIBS}
    )

    add_executable(${TEST_TARGET} ${TEST_SOURCES})
    target_link_libraries(${TEST_TARGET}
        PUBLIC  ${TEST_PUBLIC_LIBS}
        PRIVATE ${LIB_TARGET} ${TEST_PRIVATE_LIBS}
    )
endfunction()

libWithTest(
    LIB
        TARGET Algo
        SOURCES algo.cpp algo.h
        PUBLIC_LIBS SomeMathHelpers
    TEST
        TARGET AlgoTest
        SOURCES algoTest.cpp
        PRIVATE_LIBS gtest_main
)
```

In the above example, the first level of arguments parsed by cmake_parse_arguments() is the usual ${ARGN}. The only keywords at this first level are the two multi-value keywords LIB and TEST. These define which target the sub-options following it should be applied to. The second level of parsing is fed either ${GRP_LIB} or ${GRP_TEST} as the set of arguments to parse rather than ${ARGN}. There is no conflict as a result of sub-options appearing more than once in the original set of ARGN arguments, since each target's sub-options are parsed separately.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

Compared to basic argument handling using named arguments or using the `ARG⋯` variables, the advantages of `cmake_parse_arguments()` are numerous:

- Being keyword-based, the calling site has improved readability, since the arguments essentially become self-documenting. Other developers reading the call site usually won't need to look at the function implementation or its documentation to understand what each of the arguments mean.
- The caller gets to choose the order in which the arguments are given.
- The caller can simply omit those arguments which don't need to be provided.
- Since each of the supported keywords has to be passed to `cmake_parse_arguments()` and it is typically called near the top of the function, it is generally very clear what arguments the function supports.
- Since parsing of the keyword based arguments is handled by the `cmake_parse_arguments()` command rather than from an ad hoc, manually coded parser, argument parsing bugs are virtually eliminated.

While these capabilities are quite powerful, the command still has some limitations. Built-in commands are able to support keywords being repeated. For example, commands like `target_link_libraries()` allow the `PRIVATE`, `PUBLIC` and `INTERFACE` keywords to be used more than once in the same command. The `cmake_parse_arguments()` command does not support this to the same extent. It will only return the values associated with the last occurrence of a keyword and discard the earlier ones. A keyword can only be repeated if using a multi-level set of keywords and the keyword only appears once in any given set of arguments being processed.

8.4. Scope

A fundamental difference between functions and macros is that functions introduce a new variable scope, whereas macros do not. Functions receive a *copy* of all variables from the calling scope. Variables defined or modified inside a function have no effect on variables of the same name outside of the function. As far as variables are concerned, the function is essentially its own self-contained sandbox, unlike macros which share the same variable scope as their caller and can therefore modify the caller's variables directly. Note that functions do not introduce a new *policy* scope (see Section 12.3, "Recommended Practices" for further discussion of this).

Unlike their C/C++ counterparts, CMake functions and macros do not support returning a value directly. Furthermore, since functions introduce their own variable scope, it may seem that there is no easy way to pass information back to the caller, but this is not the case. The same approach as was discussed for `add_subdirectory()` in Section 7.1.2, "Scope" can be used for functions too. The `set()` command's `PARENT_SCOPE` keyword can be used to modify a variable in the caller's scope rather than a local variable within the function. While this isn't the same as returning a value from the function, it does allow a value (or multiple values) to be passed back to the caller.

A common approach is to allow a variable name to be passed in as a function argument so that the caller is still in control of the name of variables where function results are set. This is the approach used by `cmake_parse_arguments()`, with its `prefix` argument determining the prefix of all the variable names it sets in the caller's scope. The following example and its output demonstrate how to implement the technique:

Exclusively for sbmain17@gmail.com Transaction: 0098315591

```
function(func resultVar1 resultVar2)
    set(${resultVar1} "First result" PARENT_SCOPE)
    set(${resultVar2} "Second result" PARENT_SCOPE)
endfunction()

func(myVar otherVar)

message("myVar: ${myVar}")
message("otherVar: ${otherVar}")
```

```
myVar: First result
otherVar: Second result
```

Another alternative is for a function to document the variables it sets rather than allowing the caller to specify the variable names. This is less desirable, since it reduces the flexibility of the function and opens up opportunities for variable name clashes. Where possible, it is better to use the above method to give the caller the control over variable names being set or modified.

Macros can be handled the same way as functions, specifying the names of variables to be set by passing them in as arguments. The only difference is that the `PARENT_SCOPE` keyword should not be used within the macro since it already modifies the variables in the caller's scope. In fact, about the only reason one would use a macro instead of a function is if many variables need to be set in the calling scope. A macro will affect the calling scope with every `set()` call, whereas a function only affects the calling scope when `PARENT_SCOPE` is explicitly given to `set()`.

In Section 7.4, "Ending Processing Early", the `return()` statement was discussed as a way to end processing early within a file or function. As explained above, `return()` does not return a value, it only returns processing to the parent *scope*. If `return()` is called within a function, processing returns immediately to the caller, i.e. the rest of the function is skipped. The behavior of `return()` within a macro, on the other hand, is very different. Because a macro does not introduce a new scope, the behavior of the `return()` statement is dependent on where the macro is called. Recall that a macro effectively pastes its commands at the call site. That being the case, any `return()` statement from a macro will actually be returning from the scope of whatever called the macro, not from the macro itself. Consider the following example:

```
macro(inner)
    message("From inner")
    return()  # Usually dangerous within a macro
    message("Never printed")
endmacro()

function(outer)
    message("From outer before calling inner")
    inner()
    message("Also never printed")
endfunction()

outer()
```


Exclusively for sbmain17@gmail.com Transaction: 0098315591

The output from the above would be:

```
From outer before calling inner
From inner
```

To highlight why the second message in the function body is never printed, paste the contents of the macro body into where it is called:

```
function(outer)
    message("From outer before calling inner")

    # === Pasted macro body ===
    message("From inner")
    return()
    message("Never printed")
    # === End of macro body ===

    message("Also never printed")
endfunction()

outer()
```

It is now much clearer why the `return()` statement causes processing to leave the function, even though it was originally called from inside the macro. This highlights the danger of using `return()` within macros. Because macros do not create their own scope, the result of a `return()` statement is often not what was expected.

8.5. Overriding Commands

When `function()` or `macro()` is called to define a new command, if a command already exists with that name, the undocumented CMake behavior is to make the old command available using the same name except with an underscore prepended. This applies whether the old name is for a builtin command or a custom function or macro. Developers who are aware of this behavior are sometimes tempted to exploit it to try to create a wrapper around an existing command like so:

```
function(someFunc)
    # Do something...
endfunction()

# Later in the project...
function(someFunc)
    if(...)
        # Override the behavior with something else...
    else()
        # WARNING: Intended to call the original command, but it is not safe
        _someFunc()
    endif()
endfunction()
```

Exclusively for sbmain17@gmail.com Transaction: 0098315591

If the command is only ever overridden like this once, it appears to work, but if it is overridden again, then the original command is no longer accessible. The prepending of one underscore to "save" the previous command only applies to the current name, it is not applied recursively to all previous overrides. This has the potential to lead to infinite recursion, as the following contrived example demonstrates:

```
function(printme)
    message("Hello from first")
endfunction()

function(printme)
    message("Hello from second")
    _printme()
endfunction()

function(printme)
    message("Hello from third")
    _printme()
endfunction()

printme()
```

One would naively expect the output to be as follows:

```
Hello from third
Hello from second
Hello from first
```

But instead, the first implementation is never called because the second one ends up calling itself in an infinite loop. When CMake processes the above, here's what occurs:

1. The first implementation of `printme` is created and made available as a command of that name. No command by that name previously existed, so no further action is required.
2. The second implementation of `printme` is encountered. CMake finds an existing command by that name, so it defines the name `_printme` to point to the old command and sets `printme` to point to the new definition.
3. The third implementation of `printme` is encountered. Again, CMake finds an existing command by that name, so it *redefines* the name `_printme` to point to the old command (which is the second implementation) and sets `printme` to point to the new definition.

When `printme()` is called, execution enters the third implementation, which calls `_printme()`. This enters the second implementation which also calls `_printme()`, but `_printme()` points back at the second implementation again and infinite recursion results. Execution never reaches the first implementation.

In general, it is fine to override a function or macro as long as it does not try to call the previous implementation like in the above discussion. Projects should simply assume that the new implementation replaces the old one, with the old one considered to be no longer available.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

8.6. Special Variables For Functions

CMake 3.17 added support for a number of variables to assist with debugging and implementing functions. The following variables will be available during execution of a function:

`CMAKE_CURRENT_FUNCTION`

Holds the name of the function currently being executed.

`CMAKE_CURRENT_FUNCTION_LIST_FILE`

Contains the full path to the file that defined the function currently being executed.

`CMAKE_CURRENT_FUNCTION_LIST_DIR`

Holds the absolute directory containing the file that defined the function currently being executed.

`CMAKE_CURRENT_FUNCTION_LIST_LINE`

Holds the line number at which the currently executing function was defined within the file that defined it.

The `CMAKE_CURRENT_FUNCTION_LIST_DIR` variable is particularly useful when a function needs to refer to a file that is an internal implementation detail of the function. The value of `CMAKE_CURRENT_LIST_DIR` would contain the directory of the file where the function is *called*, whereas `CMAKE_CURRENT_FUNCTION_LIST_DIR` holds the directory where the function is *defined*. To see how this can be used, consider the following example. It demonstrates a common pattern where a function uses the `configure_file()` command to copy a file from the same directory as the file defining the function (see Section 19.2, "Copying Files" for further discussion):

```
function(writeSomeFile toWhere)
    configure_file(${CMAKE_CURRENT_FUNCTION_LIST_DIR}/template.cpp.in ${toWhere} @ONLY)
endfunction()
```

Before CMake 3.17, the above would typically be implemented something like the following instead (CMake's own modules used this technique prior to CMake 3.17):

```
set(__writeSomeFile_DIR ${CMAKE_CURRENT_LIST_DIR})

function(writeSomeFile toWhere)
    configure_file(${__writeSomeFile_DIR}/template.cpp.in ${toWhere} @ONLY)
endfunction()
```

This second example relies on the `__writeSomeFile_DIR` variable remaining visible at the point of the call to the function. Normally, that should be a reasonable assumption, but because functions have global scope visibility, projects can technically define a function in one place and call it in an unrelated variable scope. While that is technically legal, it is not a recommended practice. Extra care must also be taken with this technique when include guards are used in files that define functions (see Section 7.4, "Ending Processing Early").

Exclusively for sbmain17@gmail.com Transaction: 0098315591

The CMAKE_CURRENT_FUNCTION… variables are only updated for functions, they are not modified inside macros. When executing code for a macro, these variables will hold whatever values they had when the macro was called.

8.7. Other Ways Of Invoking CMake Code

Functions and macros provide powerful ways of defining code to be executed at some later time. They are an essential part of re-using common logic for similar or repetitive tasks. Nevertheless, there are situations where projects may want to define CMake code to be executed in a way that functions and macros alone cannot capture.

CMake 3.18 added the cmake_language() command which can be used to invoke arbitrary CMake code directly without having to define a function or macro. This functionality isn't designed to replace functions or macros, but rather to complement them by enabling more concise code and the ability to express logic in ways that were not previously possible. The two sub-commands provided by CMake 3.18 are CALL and EVAL CODE:

```
cmake_language(CALL command [args...])
cmake_language(EVAL CODE code...)
```

The CALL sub-command invokes a single CMake command, with arguments if required. It provides the ability to parameterize the command to be invoked without having to hard-code all available choices. Certain built-in commands cannot be invoked this way, specifically those commands that start or end a block like if(), endif(), foreach(), endforeach() and so on.

The following example demonstrates how a generic wrapper can be defined around a set of functions that include a version number in their name:

```
function(qt_generate_moc)
    set(cmd qt${QT_DEFAULT_MAJOR_VERSION}_generate_moc)

    cmake_language(CALL ${cmd} ${ARGV})
endfunction()
```

The above example assumes the QT_DEFAULT_MAJOR_VERSION variable has been set previously. As future Qt major versions are released, the above would continue to work as long as the appropriate versioned command was still provided. The alternative would be to implement an ever-expanding set of if() tests for each version individually.

The CALL sub-command is fairly limited in its usefulness. The EVAL CODE sub-command is much more powerful, as it supports executing any valid CMake script. One advantage of this is that it does not interfere with variables that get updated inside a function invocation, such as ARGV, CMAKE_CURRENT_FUNCTION and so on. The following example takes advantage of this behavior to implement a form of call tracing for functions:

Exclusively for sbmain17@gmail.com Transaction: 0098315591

```
set(myProjTraceCall [=[
    message("Called ${CMAKE_CURRENT_FUNCTION}")
    set(__x 0)
    while(__x LESS ${ARGC})
        message("  ARGV${__x} = ${ARGV${__x}}")
        math(EXPR __x "${__x} + 1")
    endwhile()
    unset(__x)
]=])

function(func)
    cmake_language(EVAL CODE "${myProjTraceCall}")
    # ...
endfunction()

func(one two three)
```

```
Called func
  ARGV0 = one
  ARGV1 = two
  ARGV2 = three
```

Note how the code stored in `myProjTraceCall` makes use of the various `ARG*` variables and also the `CMAKE_CURRENT_FUNCTION` variable. Bracket syntax `[=[` and `]=]` is used to prevent the evaluation of these variables when `myProjTraceCall` is set. The variables will be evaluated only when `cmake_language()` is called, so they will reflect the details of the enclosing function. Because of this delayed evaluation, the tracing code won't work as expected inside a macro, so only use it from inside a function.

See Section 8.8.2, "Forwarding Command Arguments" for another particularly interesting example of the `EVAL CODE` sub-command.

CMake 3.19 added the `DEFER` set of sub-commands. These allow a command to be queued for execution at a later time and to manage the set of currently queued commands. Creating a deferred command is accomplished with the following form:

```
cmake_language(DEFER
    [DIRECTORY dir]
    [ID id | ID_VAR outVar]
    CALL command [args...]   ①
)
```

① Evaluation of variables within `command` and `args` doesn't follow the usual behavior of most other CMake commands. See Section 8.8.3, "Special Cases For Argument Expansion" for a discussion of the important differences.

The `command` and its arguments will be queued for execution at the end of the current directory scope. The `DIRECTORY` option can be given to specify a different directory scope instead. In that case, the `dir` directory must already be known to CMake and it must not already have finished being processed. In practice, this means it must be either the current directory or one of the parent directory scopes.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

```
cmake_language(DEFER
    CALL message "End of current scope processing"
)
cmake_language(DEFER
    DIRECTORY ${CMAKE_SOURCE_DIR}
    CALL message "End of top level processing"
)
```

Each queued command has an identifier associated with it. Multiple commands can be associated with the same identifier to allow them to be manipulated as a group (see further below). Normally, the project would let CMake automatically assign a new identifier when queueing a new deferred command. The `ID_VAR` option can be used to capture the assigned identifier, which can then be re-used in later calls with the `ID` option to add more commands to the same identifier.

```
cmake_language(DEFER
    ID_VAR deferredId
    CALL message "First deferred command"
)
cmake_language(DEFER
    ID ${deferredId}
    CALL message "Second deferred command"
)
```

Other `DEFER` sub-commands can query and cancel deferred commands based on identifiers:

```
cmake_language(DEFER [DIRECTORY dir] GET_CALL_IDS outVar)
cmake_language(DEFER [DIRECTORY dir] GET_CALL id outVar)
cmake_language(DEFER [DIRECTORY dir] CANCEL_CALL ids...)
```

The `GET_CALL_IDS` form returns a list of the identifiers for all commands currently queued for the specified directory scope, or the current directory scope if no `DIRECTORY` option is given. The `GET_CALL` form returns the first command and its arguments associated with the specified `id`. It is not possible to retrieve the second or later commands for a given identifier, nor a count of how many commands are associated with an identifier. The `CANCEL_CALL` form will discard all deferred commands associated with any of the specified identifiers.

At this point, it would be natural to start thinking of different ways one might put the `DEFER` functionality to use. Before doing so, consider the following observations:

- Special rules apply to variable expansion within deferred commands and their arguments (see Section 8.8.3, “Special Cases For Argument Expansion”). These can lead to subtle problems that can be difficult to trace.
- Deferred commands make it harder for developers to follow the flow of execution. This is especially true when deferred commands are created inside functions or macros and their creation isn’t made obvious.
- In deferring commands, the project may be making assumptions about what can happen between the deferral and the execution of the commands. It may be quite difficult to guarantee

Exclusively for sbmain17@gmail.com Transaction: 0098315591

that these assumptions remain valid, especially for commands deferred to parent scopes or where deferred commands are created inside a function or macro that could be called from anywhere.

- Deferred commands can be a sign of a project's CMake API trying to do too much in one function or macro.

Given the above, where there is a choice, prefer other techniques or refactoring over deferring commands. As an example, a function might wrap a command that creates a target, then call other commands that use properties of that target before returning (properties are covered in the next chapter). By encapsulating all of this in a single function, the caller has no opportunity to modify target properties before they are used. Rather than deferring the commands that use the target so that the caller can modify the target properties, consider breaking up the function so that it doesn't have so many responsibilities. Requiring the target to be passed in instead of creating it would be one alternative solution for this particular example.

8.8. Problems With Argument Handling

CMake's implementation of command arguments contains a few subtle behaviors. For the most part, these behaviors don't lead to problems, but occasionally they can cause confusion or give rise to unexpected results. In order to appreciate why these behaviors exist and how to handle them safely, it helps to understand the way in which CMake constructs and passes arguments to commands.

Consider the following equivalent calls where `someCommand` could be any valid command:

```
someCommand(a b c)
someCommand(a    b    c)
```

Arguments are separated by spaces and consecutive spaces are treated as a single argument separator. Semi-colons act as argument separators too, so the following are also equivalent to the above:

```
someCommand(a b;c)
someCommand(a;;;;b;c)
```

Where an argument needs to contain embedded spaces or semicolons, quoting must be used:

```
someCommand(a "b b" c)
someCommand(a "b;b" c)
someCommand(a;"b;b";c)
```

All three of the above calls result in three arguments being passed to the command. The first call passes `b b` for the second argument, the other two calls pass `b;b` for the second argument.

Where spaces and semicolons differ is how they are handled when variable evaluation is involved and arguments are not quoted:

Exclusively for sbmain17@gmail.com Transaction: 0098315591

```
set(containsSpace     "b b")
set(containsSemiColon "b;b")

someCommand(a ${containsSpace} c)
someCommand(a ${containsSemiColon} c)
```

The first call to `someCommand()` results in *three* arguments being passed, whereas the second call results in *four* arguments. The embedded space in `containsSpace` does not act as an argument separator, but the embedded semicolon in `containsSemiColon` does. Spaces only act as argument separators before any variable evaluation is performed. The interaction between these two different behaviors can lead to some surprising results:

```
set(empty               "")
set(space               " ")
set(semicolon           ";")
set(semiSpace           "; ")
set(spaceSemi           " ;")
set(spaceSemiSpace      " ; ")
set(spaceSemiSemi       " ;;")
set(semiSemiSpace       ";; ")
set(spaceSemiSemiSpace  " ;; ")

someCommand(${empty})               # 0 args
someCommand(${space})               # 1 arg
someCommand(${semicolon})           # 0 args
someCommand(${semiSpace})           # 1 arg
someCommand(${spaceSemi})           # 1 arg
someCommand(${spaceSemiSpace})      # 2 args
someCommand(${spaceSemiSemi})       # 1 arg
someCommand(${semiSemiSpace})       # 1 arg
someCommand(${spaceSemiSemiSpace})  # 2 args
```

Some important observations should be noted from the above:

OBSERVATION 1

When they are the result of variable evaluations, spaces are never discarded and never act as argument separators.

OBSERVATION 2

One or more semicolons at the start or the end of an unquoted argument are discarded.

OBSERVATION 3

Consecutive semicolons not at the start or end of an unquoted argument are merged and act as a single argument separator.

Much of the confusion can be avoided by quoting arguments if they contain any variable evaluations. This eliminates any special interpretation of embedded spaces or semicolons. While not generally harmful, this isn't always desirable. As the next subsection will highlight, there are some situations which *require* arguments to be unquoted precisely because they rely on the above behavior.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

8.8.1. Parsing Arguments Robustly

Consider the `cmake_parse_arguments()` command discussed earlier in Section 8.3, "Keyword Arguments". The original form of this command is typically used like so:

```
function(func)
    set(noValues     ENABLE_A ENABLE_B)
    set(singleValues FORMAT ARCH)
    set(multiValues  SOURCES IMAGES)

    cmake_parse_arguments(
        ARG
        "${noValues}" "${singleValues}" "${multiValues}"
        ${ARGV}
    )
endfunction()
```

Note the quoting around the evaluation of `noValues`, `singleValues` and `multiValues`. When evaluated, each of these variables yields a string that contains a semicolon. For example, `${singleValues}` will evaluate to `FORMAT;ARCH`. The quotes are necessary to prevent those semicolons from acting as argument separators. The end result is that `cmake_parse_arguments()` will see `ARG` for its first argument, `ENABLE_A;ENABLE_B` as the second, `FORMAT;ARCH` as the third and `SOURCES;IMAGES` as the fourth.

The `${ARGV}` provided at the end of the call does not have quotes. This is specifically to take advantage of the fact that embedded semicolons will act as argument separators. The `cmake_parse_arguments()` command interprets the fifth and subsequent arguments it receives as the arguments to parse. By using an unquoted `${ARGV}`, `cmake_parse_arguments()` sees the same set of arguments as was passed in to `func()`.

The problem with the above is that using `${ARGV}` fails to preserve the original arguments in two specific cases. Consider the following calls:

```
func(a "" c)
func("a;b;c" "1;2;3")
```

For the first call, inside `func()` the evaluation of `${ARGV}` will be `a;;c`. As noted in OBSERVATION 3, however, the two semicolons will be merged and `cmake_parse_arguments()` will see only `a` and `c` as the arguments to be parsed. Empty arguments are silently dropped. For the second call, the evaluation of `${ARGV}` will be `a;b;c;1;2;3`. The original call to `func()` had `a;b;c` as the first argument and `1;2;3` as the second, but this gets flattened by the `${ARGV}` evaluation and the `cmake_parse_arguments()` command instead sees six individual arguments rather than two lists. Both of these problems can be solved by using the other form of the `cmake_parse_arguments()` command to avoid evaluating `${ARGV}` directly:

```
cmake_parse_arguments(
    PARSE_ARGV 0 ARG
    "${noValues}" "${singleValues}" "${multiValues}"
)
```

Exclusively for sbmain17@gmail.com Transaction: 0098315591

In practice, the `cmake_parse_arguments()` command is often used in situations where dropping empty arguments or flattening lists has no real impact. In these cases, either form of the `cmake_parse_arguments()` command can safely be called. Where the arguments need to be preserved exactly as passed in though, the `PARSE_ARGV` form should always be used.

8.8.2. Forwarding Command Arguments

A relatively common need is to create some sort of wrapper around an existing command. The project may wish to support some extra options or remove existing ones, or it may want to perform certain processing before or after the call. Preserving arguments and forwarding them on without changing their structure or losing information can be surprisingly difficult.

Consider the following example and its output, which picks up on one of the points in the previous subsection:

```
function(printArgs)
    message("ARGC = ${ARGC}\n"
            "ARGN = ${ARGN}"
    )
endfunction()

printArgs("a;b;c" "d;e;f")
```

```
ARGC = 2
ARGN = a;b;c;d;e;f
```

The arguments to `printArgs()` are quoted, so the function does see only two arguments. In forming the value for `${ARGN}` though, these two lists are joined with a semicolon and the result is a single list of six items. The original form of the arguments is lost as a result of this list flattening. Consider the consequences of this for a wrapper command when it attempts to forward arguments to the command being wrapped:

```
function(inner)
    message("inner:\n"
            "ARGC = ${ARGC}\n"
            "ARGN = ${ARGN}"
    )
endfunction()

function(outer)
    message("outer:\n"
            "ARGC = ${ARGC}\n"
            "ARGN = ${ARGN}"
    )
    inner(${ARGN})  # Naive forwarding, not robust
endfunction()

outer("a;b;c" "d;e;f")
```


Exclusively for sbmain17@gmail.com Transaction: 0098315591

```
outer:
ARGC = 2
ARGN = a;b;c;d;e;f
inner:
ARGC = 6
ARGN = a;b;c;d;e;f
```

The `outer()` function wants to wrap the `inner()` function and forward its arguments exactly, but as the above output shows, the number of arguments seen by `inner()` is different. The evaluation of `${ARGN}` as a way to pass arguments through to `inner()` triggers the list flattening behavior described previously. The original structure of the arguments is lost. The `PARSE_ARGV` form of the `cmake_parse_arguments()` command can be used to avoid this list flattening:

```
function(outer)
    cmake_parse_arguments(PARSE_ARGV 0 FWD "" "" "")
    inner(${FWD_UNPARSED_ARGUMENTS})
endfunction()
```

With no keywords to parse, all arguments given to `outer()` will be placed in `FWD_UNPARSED_ARGUMENTS`. As noted back in Section 8.3, "Keyword Arguments", when the `PARSE_ARGV` form of `cmake_parse_arguments()` populates `FWD_UNPARSED_ARGUMENTS`, it escapes any embedded semicolons from the original arguments. Therefore, when that variable is passed to `inner()`, the escaping preserves the structure of the original arguments and `inner()` will see the same arguments as `outer()`.

Unfortunately, the above technique still has a weakness. As a consequence of OBSERVATIONS 2 and 3, it does not preserve any empty arguments. To avoid dropping empty arguments, each argument needs to be listed individually and be quoted. The `cmake_language(EVAL CODE)` command available with CMake 3.18 or later provides the functionality needed:

```
function(outer)
    cmake_parse_arguments(PARSE_ARGV 0 FWD "" "" "")

    set(quotedArgs "")
    foreach(arg IN LISTS FWD_UNPARSED_ARGUMENTS)
        string(APPEND quotedArgs " [===[${arg}]===]")
    endforeach()

    cmake_language(EVAL CODE "inner(${quotedArgs})")
endfunction()
```

Note the use of the bracket form for quoting. This ensures that any arguments with embedded quotes will be handled robustly too.

The above implementation provides robust argument forwarding, but it requires a minimum CMake version of 3.18 or higher. For earlier versions, the `cmake_language()` command is not available. An equivalent capability can be implemented by writing out the command to be executed to a file and asking CMake to process that file via a call to `include()`, but this is very inefficient and not recommended as a general solution.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

8.8.3. Special Cases For Argument Expansion

While the above techniques work well in general, some built-in commands handle their arguments in special ways, causing them to diverge from the expected behavior. These exceptions fall into two main categories: `cmake_language()` and evaluating boolean expressions.

Variable Evaluation For `cmake_language()`

`cmake_language(CALL)` provides an alternative way of executing a command. The command to call can be provided by a variable instead of having to be hard-coded. In order to faithfully reproduce the normal argument expansion and quote handling behavior for the arguments given to the forwarded-to command, those arguments are required to be separated from the command itself in the call to `cmake_language(CALL)`. The following example demonstrates the limitation:

```
# ERROR: command must be its own argument
set(cmdWithArgs message STATUS "Hello world")
cmake_language(CALL ${cmdWithArgs})

# OK: Command can be a variable expansion, but it must evaluate to a single value.
# Arguments can also be a variable expansion and they can evaluate to a list.
set(cmd message)
set(args STATUS "Hello world")
cmake_language(CALL ${cmd} ${args})
```

The `cmake_language(DEFER CALL)` command has similar restrictions, but it has further differences. Variable evaluations are performed immediately for the command to be executed, but evaluations are deferred for the command arguments. The following example highlights this behavior:

```
set(cmd message)
set(args "before deferral")
cmake_language(DEFER CALL ${cmd} ${args})

set(cmd somethingElse)      # Doesn't affect the command
set(args "after deferral")  # But this does!
```

```
after deferral
```

The evaluation of `${cmd}` happens immediately, but `${args}` is not evaluated until the deferred command is called. At the end of the directory scope, `args` will have the value `after deferral`.

If evaluation of variables in command arguments needs to be performed immediately, one has to wrap the deferral within a call to `cmake_language(EVAL CODE)`. An example of this scenario is where the deferral is created inside a function or macro and the deferred command arguments need to incorporate information passed as arguments to the deferring function or macro:

```
function(endOfScopeMessage msg)
    cmake_language(EVAL CODE "cmake_language(DEFER CALL message [[${msg}]])")
endfunction()
```


Exclusively for sbmain17@gmail.com Transaction: 0098315591

Note how quoting with bracket syntax is used around the ${msg} evaluation to ensure spaces in the evaluated variable are handled correctly.

Boolean Expressions

Commands that treat their arguments as a boolean expression also have some special rules associated with quoting and argument expansion. The if() command best demonstrates this, but the rules also apply to while(). Consider the following example, which shows the subtle behavior of how unquoted arguments can be treated as either variable names or string values (see Section 6.1.1, "Basic Expressions" for a deeper discussion of this behavior):

```
cmake_minimum_required(VERSION 3.1)

set(someVar xxxx)
set(xxxx "some other value")

# Here, xxxx is unquoted, so it is treated as the name of
# a variable and its value is used. Result: prints NO
if(someVar STREQUAL xxxx)
    message(YES)
else()
    message(NO)
endif()

# Now use xxxx in quotes. This prevents it from being treated as a
# variable name and the value is used directly. Result: prints YES
if(someVar STREQUAL "xxxx")
    message(YES)
else()
    message(NO)
endif()
```

Attempting to reproduce the above using variables to provide arguments to the if() commands, one might try something like the following:

```
set(noQuotes   someVar STREQUAL xxxx)
set(withQuotes someVar STREQUAL [["xxxx"]])

if(${noQuotes})
    message(YES)
else()
    message(NO)
endif()

if(${withQuotes})  # Doesn't work as expected
    message(YES)
else()
    message(NO)
endif()
```

Exclusively for sbmain17@gmail.com Transaction: 0098315591

The value of `withQuotes` uses bracket syntax to make the quotes part of the value stored. The idea is to try to get the `if()` command to treat `xxxx` as a quoted argument, but it does not have the desired effect. The `if()` command checks for quotes *before* expansion, so in this case, the quotes get treated as part of the argument value. There is no way to get an argument to be treated as quoted when providing the arguments to `if()` through an expanded variable evaluation like in the above.

One other special case to be aware of is the way square brackets affect how semicolons are interpreted for lists, as discussed previously in Section 5.8.1, "Problems With Unbalanced Square Brackets". Semicolons between unbalanced square brackets are not interpreted as list separators when evaluating a variable. This does not extend to the way CMake assembles command arguments though, as demonstrated by the following example:

```
function(func)
    message("Number of arguments: ${ARGC}")

    math(EXPR lastIndex "${ARGC} - 1")

    foreach(n RANGE 0 ${lastIndex})
        message("ARGV${n} = ${ARGV${n}}")
    endforeach()

    foreach(arg IN LISTS ARGV)
        message("${arg}")
    endforeach()
endfunction()

func("a[a" "b]b" "c[c]c" "d[d" "eee")
```

This would produce the following output:

```
Number of arguments: 5
ARGV0 = a[a
ARGV1 = b]b
ARGV2 = c[c]c
ARGV3 = d[d
ARGV4 = eee
a[a;b]b
c[c]c
d[d;eee
```

The `func()` command *does* see the five original arguments, as demonstrated by the first `foreach()` loop. The evaluation of the `ARGV` variable by the second `foreach()` command is where the embedded unbalanced square brackets interfere with how the variable is interpreted as a list.

In practice, it is relatively uncommon to find situations where unbalanced square brackets may reasonably be present. Occasionally, balanced square brackets may be encountered, but as the `c[c]c` argument in the above example shows, they do not interfere with list interpretation.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

8.9. Recommended Practices

Functions and macros are a great way to re-use the same piece of CMake code throughout a project. In general, prefer to use functions rather than macros, since the use of a new variable scope within the function better isolates that function's effects on the calling scope. Macros should generally only be used where the contents of the macro body really do need to be executed within the scope of the caller. These situations should generally be relatively rare. To avoid unexpected behavior, also avoid calling `return()` from inside a macro.

Prefer to pass all values a function or macro needs as command arguments rather than relying on variables being set in the calling scope. This tends to make the implementation more robust to future changes and it is much clearer and easier to maintain.

For all but very trivial functions or macros, it is highly recommended to use the keyword-based argument handling provided by `cmake_parse_arguments()`. This leads to better usability and improved robustness of calling code (e.g. little chance of getting arguments mixed up). It also allows the function to be more easily extended in the future because there is no reliance on argument ordering or for all arguments to always be provided, even if not relevant.

Beware of dropping empty arguments and list flattening when parsing or forwarding command arguments. Where the project's minimum CMake version allows, prefer to use the `PARSE_ARGV` form of `cmake_parse_arguments()` inside functions. When forwarding arguments, use `cmake_language(EVAL CODE)` to quote each argument individually if preserving empty arguments and lists is required.

Prefer to avoid deferring commands via `cmake_language(DEFER)` if there are other alternatives. Deferred commands introduce fragility, hinder the ability to debug a project and can be a sign that CMake functions and macros should be refactored.

Rather than distributing functions and macros throughout the source tree, a common practice is to nominate a particular directory (usually just below the top level of the project) where various `XXX.cmake` files can be collected. That directory acts like a catalog of ready-to-use functionality, able to be conveniently accessed from anywhere in the project. Each of the files can provide functions, macros, variables and other features as appropriate. Using a `.cmake` file name suffix allows the `include()` command to find the files as modules, a topic covered in detail in Chapter 11, *Modules*. It also tends to allow IDE tools to recognize the file type and apply CMake syntax highlighting.

Do not define or call a function or macro with a name that starts with a single underscore. In particular, do not rely on the undocumented behavior whereby the old implementation of a command is made available by such a name when a function or macro redefines an existing command. Once a command has been overridden more than once, its original implementation is no longer accessible. This undocumented behavior may even be removed in a future version of CMake, so it should not be used. Along similar lines, do not override any builtin CMake command. Consider those to be off-limits so that projects will always be able to assume the builtin commands behave as per the official documentation and there will be no opportunity for the original command to become inaccessible.

Where the project's minimum CMake version is set to 3.17 or later, prefer to use `CMAKE_CURRENT_FUNCTION_LIST_DIR` to refer to any file or directory expected to exist at a location relative to the file in which a function is defined.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

Chapter 9. Properties

Properties affect just about all aspects of the build process, from how a source file is compiled into an object file, right through to the install location of built binaries in a packaged installer. They are always attached to a specific entity, whether that be a directory, target, source file, test case, cache variable or even the overall build process itself. Rather than holding a standalone value like a variable does, a property provides information specific to the entity it is attached to.

For those new to CMake, properties are sometimes confused with variables. Though both may initially seem similar in terms of function and features, properties serve a very different purpose. A variable is not attached to any particular entity and it is very common for projects to define and use their own variables. Compare this with properties which are typically well defined and documented by CMake and which always apply to a specific entity. A likely contributor to the confusion between the two is that a property's default value is sometimes provided by a variable. The names CMake uses for related properties and variables usually follow the same pattern, with the variable name being the property name with `CMAKE_` prepended.

9.1. General Property Commands

CMake provides a number of commands for manipulating properties. The most generic of these, `set_property()` and `get_property()`, allow setting and getting any property on any type of entity. These commands require the type of entity to be specified as command arguments along with some entity-specific information.

```
set_property(entitySpecific
    [APPEND | APPEND_STRING]
    PROPERTY propertyName values...
)
```

`entitySpecific` defines the entity whose property is being set. It must be *one* of the following:

```
GLOBAL
DIRECTORY [dir]
TARGET    targets...
SOURCE    sources...  # Additional options with CMake 3.18
INSTALL   files...
TEST      tests...
CACHE     vars...
```

The first word of each of the above defines the type of entity whose property is being set. `GLOBAL` means the build itself, so there is no specific entity name required. For `DIRECTORY`, if no `dir` is named, the current source directory is used. For all the other types of entities, any number of items of that type can be listed. The `SOURCE` entity type also supports some additional options when using CMake 3.18 or later, which are discussed in Section 9.5, "Source Properties".

The `PROPERTY` keyword marks all remaining arguments as defining the property name and its value(s). The `propertyName` would normally match one of the properties defined in the CMake

Exclusively for sbmain17@gmail.com Transaction: 0098315591

documentation, a number of which are discussed in later chapters. The meaning of the value(s) are property specific.

In addition to the properties defined by CMake, a project may also set its own custom properties. It is up to the project what such properties mean and how they affect the build. If choosing to do this, it would be wise for projects to use a project-specific prefix on the property name to avoid potential name clashes with properties defined by CMake or other third party packages.

```
set_property(TARGET MyApp1 MyApp2
    PROPERTY MYPROJ_CUSTOM_PROP val1 val2 val3
)
```

The above example defines a custom `MYPROJ_CUSTOM_PROP` property which will have the list `val1;val2;val3` as its value. It also demonstrates how to set a property for multiple targets at once.

The `APPEND` and `APPEND_STRING` keywords can be used to control how the named property is updated if it already has a value. With neither keyword specified, the value(s) given replace any previous value. The `APPEND` keyword changes the behavior to append the value(s) to the existing one, forming a list, whereas the `APPEND_STRING` keyword takes the existing value and appends the new value(s) by concatenating the two as strings rather than as a list (see also the special note for inherited properties further below). The following table demonstrates the differences.

Previous Value(s)	New Value(s)	No Keyword	APPEND	APPEND_STRING
foo	bar	bar	foo;bar	foobar
a;b	c;d	c;d	a;b;c;d	a;bc;d

The `get_property()` command follows a similar form:

```
get_property(resultVar entitySpecific
    PROPERTY propertyName
    [DEFINED | SET | BRIEF_DOCS | FULL_DOCS]
)
```

The `PROPERTY` keyword is always required and is always followed by the name of the property to retrieve. The result of the retrieval is stored in a variable whose name is given by `resultVar`. The `entitySpecific` part is similar to that for `set_property()` and must be *one* of the following:

```
GLOBAL
DIRECTORY [dir]
TARGET    target
SOURCE    source  # Additional options with CMake 3.18
INSTALL   file
TEST      test
CACHE     var
VARIABLE
```

As before, `GLOBAL` refers to the build as a whole and therefore requires no specific entity to be named. `DIRECTORY` can be used with or without specifying a particular directory, with the current

Exclusively for sbmain17@gmail.com Transaction: 0098315591

source directory being assumed if no directory is provided. For most of the other scopes, the particular entity within that scope must be named and the requested property attached to that entity will be retrieved. Again, the `SOURCE` entity type supports additional options when using CMake 3.18 or later and these are covered in Section 9.5, “Source Properties”.

The `VARIABLE` type is a bit different, with the variable name being specified as the `propertyName` rather than being attached to the `VARIABLE` keyword. This can seem somewhat unintuitive, but consider the situation if the variable was named as the entity along with the `VARIABLE` keyword, just like for the other entity type keywords. In that situation, there would be nothing to specify for the property name. It may help to think of `VARIABLE` as specifying the current *scope*, then the property of interest is the variable named by `propertyName`. When understood this way, `VARIABLE` is consistent with how the other entity types are handled.

If none of the optional keywords are given, the value of the named property is retrieved. This is the typical usage of the `get_property()` command. Note that in practice, the use of `VARIABLE` scope with `get_property()` is relatively uncommon. Variable values can be obtained directly with the `${}` syntax, which is both clearer and simpler than using `get_property()`.

The optional keywords can be used to retrieve details about the property other than just its value:

`DEFINED`

The result of the retrieval will be a boolean value indicating whether or not the named property has been defined. In the case of `VARIABLE` scope queries, the result will only be true if the named variable has been explicitly defined with the `define_property()` command (see below).

`SET`

The result of the retrieval will be a boolean value indicating whether or not the named property has been set. It differs from `DEFINED` in that it queries whether the named property has actually been set to some value (the value itself is irrelevant), whereas `DEFINED` is more about describing what the property means. `SET` is usually what projects need rather than `DEFINED` in most scenarios. Note also that a property can return true for `DEFINED` and false for `SET`, or vice versa.

`BRIEF_DOCS`

Retrieves the brief documentation string for the named property. If no brief documentation has been defined for the property, the result will be the string `NOTFOUND`.

`FULL_DOCS`

Retrieves the full documentation for the named property. If no full documentation has been defined for the property, the result will be the string `NOTFOUND`.

Of the optional keywords, all but `SET` have little value unless the project has explicitly called `define_property()` to populate the requested information for the particular entity:

```
define_property(entityType
    PROPERTY propertyName [INHERITED]
    BRIEF_DOCS briefDoc [moreBriefDocs...]
    FULL_DOCS fullDoc [moreFullDocs...]
)
```


Exclusively for sbmain17@gmail.com Transaction: 0098315591

The `define_property()` command is rarely used. It does not set the property's value, only its documentation and whether it inherits its value from elsewhere if it has not been set. The `entityType` must be one of `GLOBAL`, `DIRECTORY`, `TARGET`, `SOURCE`, `TEST`, `VARIABLE` or `CACHED_VARIABLE` and the `propertyName` specifies the property being defined. No entity is specified, although like for the `get_property()` command, in the case of `VARIABLE` the variable name is specified as `propertyName`. The brief docs should generally be kept to one relatively short line, while the full docs can be longer and span across multiple lines if required.

If the `INHERITED` option is used when defining a property, the `get_property()` command will chain up to the parent scope if that property is not set in the named scope. For example, if a `DIRECTORY` property is requested but is not set for the directory specified, the parent directory scope is queried recursively up the directory scope hierarchy until the property is found or the top level of the source tree is reached. If still not found at the top level directory, then the `GLOBAL` scope will be searched. Similarly, if a `TARGET`, `SOURCE` or `TEST` property is requested but is not set for the specified entity, the `DIRECTORY` scope will be searched (including recursively up the directory hierarchy and ultimately to the `GLOBAL` scope if necessary). No such chaining functionality is provided for `VARIABLE` or `CACHE`, since these already chain to the parent variable scope by design.

The inheriting behavior of `INHERITED` properties only applies to the `get_property()` command and its analogous `get_⋯` functions for specific property types (covered in the sections below). When calling `set_property()` with `APPEND` or `APPEND_STRING` options, only the immediate value of the property is considered (i.e. no inheriting occurs when working out the value to append to).

CMake has a large number of pre-defined properties of each type. Developers should consult the CMake reference documentation for the available properties and their intended purpose. In later chapters, many of these properties are discussed and their relationship to other CMake commands, variables and features are explored.

9.2. Global Properties

Global properties relate to the overall build as a whole. They are typically used for things like modifying how build tools are launched or other aspects of tool behavior, for defining aspects of how project files are structured and for providing some degree of build-level information.

In addition to the generic `set_property()` and `get_property()` commands, CMake also provides `get_cmake_property()` for querying global entities. It is more than just shorthand for `get_property()`, although it can be used simply to retrieve the value of any global property.

```
get_cmake_property(resultVar property)
```

Just like for `get_property()`, `resultVar` is the name of a variable in which the value of the requested property will be stored when the command returns. The `property` argument can be the name of any global property or one of the following pseudo properties:

`VARIABLES`

Return a list of all regular (i.e. non-cache) variables.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

`CACHE_VARIABLES`

Return a list of all cache variables.

`COMMANDS`

Return a list of all defined commands, functions and macros. Commands are pre-defined by CMake, whereas functions and macros can be defined either by CMake (typically through modules) or by projects themselves. Some of the returned names may correspond to undocumented or internal entities not intended for projects to use directly. The names may be returned with different upper/lower case than the way they were originally defined.

`MACROS`

Return a list of just the defined macros. This will be a subset of what the `COMMANDS` pseudo property would return, but note that the upper/lower case of the names can be different to what the `COMMANDS` pseudo property reports.

`COMPONENTS`

Return a list of all components defined by `install()` commands, which is covered in Chapter 26, *Installing*.

These read-only pseudo properties are technically not global properties (they cannot be retrieved using `get_property()`, for example), but they are notionally very similar. They can only be retrieved via `get_cmake_property()`.

9.3. Directory Properties

Directories also support their own set of properties. Logically, directory properties sit somewhere between global properties which apply everywhere and target properties which only affect individual targets. As such, directory properties mostly focus on setting defaults for target properties and overriding global properties or defaults for the current directory. A few read-only directory properties also provide a degree of introspection, holding information about how the build reached the directory, what things have been defined at that point, etc.

For convenience, CMake provides dedicated commands for setting and getting directory properties which are a little more concise than their generic counterparts:

```
set_directory_properties(PROPERTIES prop1 val1 [prop2 val2] ...)

get_directory_property(resultVar [DIRECTORY dir] property)
get_directory_property(resultVar [DIRECTORY dir] DEFINITION varName)
```

While being a little more concise, this directory-specific setter command lacks any `APPEND` or `APPEND_STRING` option. This means it can only be used to set or replace a property, it cannot be used to add to an existing property directly. A further restriction of this command compared to the more generic `set_property()` is that it always applies to the current directory. Projects may choose to use this more specific form where it is convenient and use the generic form elsewhere, or for consistency the more generic form may be used everywhere. Neither approach is more correct, it's more a matter of preference.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

The directory-specific getter command has two forms. The first form is used to get the value of a property from a particular directory or from the current directory if the `DIRECTORY` argument is not used. The second form retrieves the value of a *variable*, which may not seem all that useful, but it provides a means of obtaining a variable's value from a different directory scope other than the current one (when the `DIRECTORY` argument is used). In practice, this second form should rarely be needed and its use should be avoided for scenarios other than debugging the build or similar temporary tasks.

For either form of the `get_directory_property()` command, if the `DIRECTORY` argument is used, the named directory must have already been processed by CMake. It is not possible for CMake to know the properties of a directory scope it has not yet encountered.

9.4. Target Properties

Few things in CMake have such a strong and direct influence on how targets are built as target properties. They control and provide information about everything from the flags used to compile source files through to the type and location of the built binaries and intermediate files. Some target properties affect how targets are presented in the developer's IDE project, while others affect the tools used when compiling/linking. In short, target properties are where most of the details about how to actually turn source files into binaries are collected and applied.

A number of methods have evolved in CMake for manipulating target properties. In addition to the generic `set_property()` and `get_property()` commands, CMake also provides some target-specific equivalents for convenience:

```
set_target_properties(target1 [target2...]
    PROPERTIES
        propertyName1 value1
        [propertyName2 value2] ...
)
get_target_property(resultVar target propertyName)
```

As for the `set_directory_properties()` command, `set_target_properties()` lacks the full flexibility of `set_property()` but provides a simpler syntax for common cases. The `set_target_properties()` command does not support appending to existing property values and if a list value needs to be provided for a given property, the `set_target_properties()` command requires that value to be specified in string form, e.g. `"this;is;a;list"`.

The `get_target_property()` command is the simplified version of `get_property()`. It focuses purely on providing a simple way to obtain the value of a target property and is basically just a shorthand version of the generic command.

In addition to the generic and target-specific property getters and setters, CMake also has a number of other commands which modify target properties. In particular, the family of `target_…()` commands are a critical part of CMake and all but the most trivial of projects would typically use them. These commands define not only properties for a particular target, they also define how that information might be propagated to other targets that link to it. Chapter 15, *Compiler And Linker Essentials* covers those commands and how they relate to target properties in depth.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

9.5. Source Properties

CMake also supports properties on individual source files. These enable fine-grained manipulation of compiler flags on a file-by-file basis rather than for all of a target's sources. They also allow additional information about the source file to be provided to modify how CMake or build tools treat the file. For example, they may indicate whether the file is generated as part of the build, what compiler to use with it, options for non-compiler tools working with the file and so on.

Projects should rarely need to query or modify source file properties, but for those situations that require it, CMake provides dedicated setter and getter commands to make the task easier. These follow a similar pattern to the other property-specific setter and getter commands:

```
set_source_files_properties(file1 [file2...]
    PROPERTIES
        propertyName1 value1
        [propertyName2 value2] ...
)
get_source_file_property(resultVar sourceFile propertyName)
```

Again, no `APPEND` functionality is provided for the setter, while the getter is really just syntax shorthand for the generic `get_property()` command and offers no new functionality. The following example shows how to set a property on a source file, in this case to prevent it from being combined with other sources in a unity build (discussed in Section 30.1, "Unity Builds"):

```
add_executable(MyApp small.cpp big.cpp tall.cpp thin.cpp)

set_source_files_properties(big.cpp PROPERTIES SKIP_UNITY_BUILD_INCLUSION YES)
```

With CMake 3.17 and earlier, source properties are only visible to targets defined in the same directory scope. If the setting of a source property occurs in a different directory scope, the target will not see that property change and therefore the compilation, etc. of that source file will not be affected. With CMake 3.18 or later, additional options are available to specify the directory scope in which the source file properties should be searched or applied. The following shows the full set of options available for setting source file properties with CMake 3.18 or later:

```
set_property(SOURCE sources...
    [DIRECTORY dirs...]
    [TARGET_DIRECTORY targets...]
    [APPEND | APPEND_STRING]
    PROPERTY propertyName values...
)
set_source_files_properties(sources...
    [DIRECTORY dirs...]
    [TARGET_DIRECTORY targets...]
    PROPERTIES
        propertyName1 value1
        [propertyName2 value2] ...
)
```


Exclusively for sbmain17@gmail.com Transaction: 0098315591

The DIRECTORY option can be used to specify one or more directories in which the source properties should be set. Any targets created in those directories will be aware of the source properties. These directories must have already been added to the build by an earlier call to add_subdirectory(). Any relative paths will be treated as relative to the current source directory.

The TARGET_DIRECTORY option is similar, except it is followed by the names of targets. For each target listed, the directory in which that target was created (i.e. its source directory) will be treated as though it had been specified with the DIRECTORY option. Note that this means *all* targets defined in that directory will be aware of the source property, not just the target specified.

CMake 3.18 also added analogous options for retrieving source file properties:

```
get_property(resultVar SOURCE source
    [DIRECTORY dir | TARGET_DIRECTORY target]
    PROPERTY propertyName
    [DEFINED | SET | BRIEF_DOCS | FULL_DOCS]
)
get_source_file_property(resultVar source
    [DIRECTORY dir | TARGET_DIRECTORY target]
    propertyName
)
```

When retrieving source file properties, at most one target or directory can be listed to identify the directory scope from which to retrieve the property. The current source directory is assumed if neither DIRECTORY nor TARGET_DIRECTORY is given.

Whether DIRECTORY or TARGET_DIRECTORY options are specified or not, note that it is possible for a source file to be compiled into multiple targets. Therefore, in each of the directory scopes where the source properties are set, the properties should make sense for all targets using those files.

Developers should be aware of an implementation detail which may present a strong deterrent to their use in some situations. For some CMake generators (notably the Unix Makefiles generator), the dependencies between sources and source properties are stronger than one might expect. If source properties are used to modify the compiler flags for specific source files rather than for a whole target, changing the source's compiler flags will still result in all of the target's sources being rebuilt, not just the affected source file. This is a limitation of how the dependency details are handled in the Makefile, where testing whether each individual source's compiler flags have changed brings with it a prohibitively big performance hit. The relevant Makefile dependencies were implemented at the target level instead to avoid that problem.

A typical scenario where projects may be tempted to use source properties is to pass version details to just one or two sources as compiler definitions. As discussed in Section 20.2, "Source Code Access To Version Details", there are better alternatives to source properties which do not suffer from the sort of build performance problems mentioned above. Setting some source properties can also reduce build performance by preventing those sources from participating in unity builds (see Section 30.1, "Unity Builds").

The Xcode generator also has a limitation in its support for source properties which prevents it from handling configuration-specific property values. See Section 15.5, "Language-specific Compiler Flags" for a scenario of where this limitation can be important.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

9.6. Cache Variable Properties

Properties on cache variables are a little different in purpose to other property types. For the most part, cache variable properties are aimed more at how the cache variables are handled in the CMake GUI and the console-based `ccmake` tool rather than affecting the build in any tangible way. There are also no extra commands provided for manipulating them, so the generic `set_property()` and `get_property()` commands must be used with the `CACHE` keyword

In Section 5.3, "Cache Variables", a number of aspects of cache variables were discussed which are ultimately reflected in the cache variable properties.

- Each cache variable has a *type*, which must be one of `BOOL`, `FILEPATH`, `PATH`, `STRING` or `INTERNAL`. This type can be obtained using `get_property()` with the property name `TYPE`. The type affects how the CMake GUI and `ccmake` present that cache variable in the UI and what kind of widget is used for editing its value. Any variable with type `INTERNAL` will not be shown at all.
- A cache variable can be marked as advanced with the `mark_as_advanced()` command, which is really just setting the boolean `ADVANCED` cache variable property. The CMake GUI and the `ccmake` tool both provide an option to show or hide advanced cache variables, allowing the user to choose whether to focus on just the main basic variables or to see the full set of non-internal variables.
- The help string of a cache variable is typically set as part of a call to the `set()` command, but it can also be modified or read using the `HELPSTRING` cache variable property. This help string is used as the tooltip in the CMake GUI and as a one-line help tip in the `ccmake` tool.
- If a cache variable is of type `STRING`, then CMake GUI will look for a cache variable property named `STRINGS`. If not empty, it is expected to be a list of valid values for the variable and CMake GUI will then present that variable as a combo box of those values rather than an arbitrary text entry widget. In the case of `ccmake`, pressing enter on that cache variable will cycle through the values provided. Note that CMake does not enforce that the cache variable must be one of the values from the `STRINGS` property, it is only a convenience for the CMake GUI and `ccmake` tools. When CMake runs its configure step, it still treats the cache variable as an arbitrary string, so it is still possible to give the cache variable any value either at the `cmake` command line or via `set()` commands in the project.

9.7. Other Property Types

CMake also supports properties on individual tests and it provides the usual test-specific versions of the property setter and getter commands:

```
set_tests_properties(test1 [test2...]
    PROPERTIES
        propertyName1 value1
        [propertyName2 value2] ...
)
get_test_property(resultVar test propertyName)
```


Exclusively for sbmain17@gmail.com Transaction: 0098315591

Like their equivalent counterparts, these are just slightly more concise versions of the generic commands which lack `APPEND` functionality but may be more convenient in some circumstances. Tests are discussed in detail in Chapter 25, *Testing*.

The other type of property CMake supports is for installed files. These properties are specific to the type of packaging being used and are typically not needed by most projects.

9.8. Recommended Practices

Properties are a crucial part of CMake. A range of commands have the ability to set, modify or query the various types of properties, some of which have further implications for dependencies between projects.

- All but the special global pseudo properties can be fully manipulated using the generic `set_property()` command, making it predictable for developers and offering flexible `APPEND` functionality where needed. The property-specific setters may be more convenient in some situations, such as allowing multiple properties to be set at once, but their lack of `APPEND` functionality may steer some projects towards just using `set_property()`. Neither is right or wrong, although a common mistake is to use the property-specific commands to replace a property value instead of appending to it.
- For target properties, use of the various `target_⋯()` commands is strongly recommended over manipulating the associated target properties directly. These commands not only manipulate the properties on specific targets, they also set up dependency relationships between targets so that CMake can propagate some properties automatically. Chapter 15, *Compiler And Linker Essentials* discusses a range of topics which highlight the strong preference for the `target_⋯()` commands.
- Source properties offer a fine granularity on the level of control of compiler options, etc. These do, however, have the potential for undesirable negative impacts on the build behavior of a project. In particular, some CMake generators may rebuild more than should be necessary when compile options for only a few source files change. The Xcode generator also has limitations which prevent it from supporting configuration-specific source file properties. Projects should consider using other alternatives to source properties where available, such as the techniques given in Section 20.2, “Source Code Access To Version Details”.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

Chapter 10. Generator Expressions

When running CMake, developers tend to think of it as a single step which involves reading the project's `CMakeLists.txt` file and producing the relevant set of generator-specific project files (e.g. Visual Studio solution and project files, an Xcode project, Unix Makefiles or Ninja input files). There are, however, two quite distinct steps involved. When running CMake, the end of the output log typically looks something like this:

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

When CMake is invoked, it first reads in and processes the `CMakeLists.txt` file at the top of the source tree, including any other files it pulls in. An internal representation of the project is created as the commands, functions, etc. are executed. This is called the *configure* step. Most of the output to the console log is produced during this stage, including any content from `message()` commands. At the end of the configure step, the `-- Configuring done` message is printed to the log.

Once CMake has finished reading and processing the `CMakeLists.txt` file, it then performs the *generation* step. This is where the build tool's project files are created using the internal representation built up in the configure step. For the most part, developers tend to ignore the generation step and just think of it as the end result of configuration. The console log almost always shows the `-- Generating done` message immediately after the configure step completes, so this is understandable. But there are situations where understanding the separation into two distinct phases is particularly important.

Consider a project processed for a multi configuration CMake generator like Xcode, Visual Studio or Ninja Multi-Config. When the `CMakeLists.txt` files are being read, CMake doesn't know which configuration a target will be built for. It is a multi configuration setup, so there's more than one choice (e.g. Debug, Release, etc.). The developer selects the configuration at *build* time, well after CMake has finished. This would seem to present a problem if the `CMakeLists.txt` file wants to do something like copy a file to the same directory as the final executable for a given target, since the location of that directory depends on which configuration is being built. A placeholder is needed to tell CMake "For whichever configuration is being built, use the directory of the final executable".

This is a prime example of the functionality provided by generator expressions. They provide a way to encode some logic which is not evaluated at configure time, the evaluation is instead delayed until the generation phase when the project files are being written. They can be used to perform conditional logic, output strings providing information about various aspects of the build like directories, names of things, platform details and more. They can even be used to provide different content based on whether a build or an install is being performed.

Generator expressions cannot be used everywhere, but they are supported in many places. In the CMake reference documentation, if a particular command or property supports generator expressions, the documentation will mention it. The set of properties supporting generator expressions have expanded over time, with some CMake releases also expanding the set of supported expressions. Projects should confirm that for the minimum CMake version they require, the properties being modified do indeed support the generator expressions used.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

10.1. Simple Boolean Logic

A generator expression is specified using the syntax `$<…>` where the content between the angle brackets can take a few different forms. As will become clear shortly, an essential feature is the conditional inclusion of content. The most basic generator expressions for this are the following:

```
$<1:...>
$<0:...>
```

For `$<1:…>`, the result of the expression will be the … part, whereas for `$<0:…>`, the … part is ignored and the expression results in an empty string. These are basically the true and false conditional expressions, but unlike for variables, the concept of true and false only allows for these two specific values. Anything other than 0 or 1 for a conditional expression is rejected by CMake with a fatal error. Another generator expression can be used to make boolean expression evaluation more flexible and ensure content evaluates to 0 or 1:

```
$<BOOL:...>
```

This evaluates the … content in the same way that the `if()` command evaluates a boolean constant, so it understands all the usual special strings like `OFF`, `NO`, `FALSE` and so on. A very common pattern is to use it to wrap the evaluation of a variable that is expected to hold a boolean value, but which might not be restricted to 0 or 1 (see the table a little further below for examples).

Logical operations are also supported:

```
$<AND:expr[,expr...]>
$<OR:expr[,expr...]>
$<NOT:expr>
```

Each `expr` is expected to evaluate to either 1 or 0. The `AND` and `OR` expressions can take any number of comma-separated arguments and provide the corresponding logic result, while `NOT` accepts only a single expression and will yield the negation of its argument. Because `AND`, `OR` and `NOT` require that their expressions evaluate to only 0 or 1, consider wrapping those expressions in a `$<BOOL:…>` to force more tolerant logic of what is considered a true or false expression.

With CMake 3.8 and later, if-then-else logic can also be expressed very conveniently using a dedicated `$<IF:…>` expression:

```
$<IF:expr,val1,val0>
```

As usual, the `expr` must evaluate to 1 or 0. The result is `val1` if `expr` evaluates to 1 and `val0` if `expr` evaluates to 0. Before CMake 3.8, equivalent logic would have to be expressed in the following more verbose way that requires the expression to be given twice:

```
$<expr:val1>$<$<NOT:expr>:val0>
```

Exclusively for sbmain17@gmail.com Transaction: 0098315591

Generator expressions can be nested, allowing expressions of arbitrary complexity to be constructed. The above example shows a nested condition, but any part of a generator expression can be nested. The following examples demonstrate the features discussed so far:

Expression	Result
`$<1:foo>`	`foo`
`$<0:foo>`	
`$<true:foo>`	Error, not a 1 or 0
`$<$<BOOL:true>:foo>`	`foo`
`$<$<NOT:0>:foo>`	`foo`
`$<$<NOT:1>:foo>`	
`$<$<NOT:true>:foo>`	Error, NOT requires a 1 or 0
`$<$<AND:1,0>:foo>`	
`$<$<OR:1,0>:foo>`	`foo`
`$<1:$<$<BOOL:false>:foo>>`	
`$<IF:$<BOOL:${foo}>,yes,no>`	Result will be `yes` or `no` depending on `${foo}`

Just like for the `if()` command, CMake also provides support for testing strings, numbers and versions in generator expressions, although the syntax is slightly different. The following all evaluate to 1 if the respective condition is satisfied, or 0 otherwise.

```
$<STREQUAL:string1,string2>
$<EQUAL:number1,number2>
$<VERSION_EQUAL:version1,version2>
$<VERSION_GREATER:version1,version2>
$<VERSION_LESS:version1,version2>
```

Another very useful conditional expression is testing the build type:

```
$<CONFIG:arg>
```

This will evaluate to 1 if `arg` corresponds to the build type actually being built and 0 for all other build types. Common uses of this would be to provide compiler flags only for debug builds or to select different implementations for different build types. For example:

```
add_executable(MyApp src1.cpp src2.cpp)

# Before CMake 3.8
target_link_libraries(MyApp PRIVATE
    $<$<CONFIG:Debug>:CheckedAlgo>
    $<$<NOT:$<CONFIG:Debug>>:FastAlgo>
)

# CMake 3.8 or later allows a more concise form
target_link_libraries(MyApp PRIVATE $<IF:$<CONFIG:Debug>,CheckedAlgo,FastAlgo>)
```


Exclusively for sbmain17@gmail.com Transaction: 0098315591

The above would link the executable to the `CheckedAlgo` library for `Debug` builds and to the `FastAlgo` library for all other build types. The `$<CONFIG:⋯>` generator expression is the only way to robustly provide such functionality which works for all CMake project generators, including multi configuration generators like Xcode, Visual Studio or Ninja Multi-Config. This particular topic is covered in more detail in Section 14.2, "Common Errors".

CMake offers even more conditional tests based on things like platform and compiler details, CMake policy settings, etc. Developers should consult the CMake reference documentation for the full set of supported conditional expressions.

10.2. Target Details

Another common use of generator expressions is to provide information about targets. Any property of a target can be obtained with one of the following two forms:

```
$<TARGET_PROPERTY:target,property>
$<TARGET_PROPERTY:property>
```

The first form provides the value of the named property from the specified target, while the second form will retrieve the property from the target on which the generator expression is being used.

While `TARGET_PROPERTY` is a very flexible expression type, it is not always the best way to obtain information about a target. For example, CMake also provides other expressions which give details about the directory and file name of a target's built binary. These more direct expressions take care of extracting out parts of some properties or computing values based on raw properties. The most general of these is the `TARGET_FILE` set of generator expressions:

`TARGET_FILE`

This will yield the absolute path and file name of the target's binary, including any file prefix and suffix if relevant for the platform (e.g `.exe`, `.dylib`). For Unix-based platforms where shared libraries typically have version details in their file name, these will also be included.

`TARGET_FILE_NAME`

Same as `TARGET_FILE` but without the path (i.e. it provides just the file name part).

`TARGET_FILE_DIR`

Same as `TARGET_FILE` but without the file name. This is the most robust way to obtain the directory in which the final executable or library is built. Its value is different for different build configurations when using a multi configuration generator like Xcode, Visual Studio or Ninja Multi-Config.

The above three `TARGET_FILE` expressions are especially useful when defining custom build rules for copying files around in post build steps (see Section 18.2, "Adding Build Steps To An Existing Target"). In addition to the `TARGET_FILE` expressions, CMake also provides some library-specific expressions which have similar roles except they handle file name prefix and/or suffix details slightly differently. These expressions have names starting with `TARGET_LINKER_FILE` and `TARGET_SONAME_FILE` and tend not to be used as frequently as the `TARGET_FILE` expressions.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

CMake 3.15 added support for additional target-related generator expressions that extract the basename, prefix and suffix of target-related file names. Projects needing these more fine-grained details should consult the CMake documentation, but such a need should be rare.

Projects supporting the Windows platform can also obtain details about PDB files for a given target. Again, these would mostly find use in custom build tasks. Expressions starting with TARGET_PDB_FILE follow an analogous pattern as for TARGET_PROPERTY, providing path and file name details for the PDB file used for the target on which the generator expression is being used.

One other generator expression relating to targets deserves special mention. CMake allows a library target to be defined as an *object library*, meaning it isn't a library in the usual sense, it is just a collection of object files that CMake associates with a target but doesn't actually result in a final library file being created. When using CMake 3.11 or earlier, it is not possible to link to an object library. Instead, object libraries have to be added to targets in the same way that *sources* are added. CMake then includes those object files at the link stage just like the object files created by compiling that target's sources. This is done using the $<TARGET_OBJECTS:…> generator expression which lists the object files in a form suitable for add_executable() or add_library() to consume, as demonstrated by the following example:

```
# Define an object library
add_library(ObjLib OBJECT src1.cpp src2.cpp)

# Define two executables which each have their own source
# file as well as the object files from ObjLib
add_executable(App1 app1.cpp $<TARGET_OBJECTS:ObjLib>)
add_executable(App2 app2.cpp $<TARGET_OBJECTS:ObjLib>)
```

In the above example, no separate library is created for ObjLib, but the src1.cpp and src2.cpp source files are still only compiled once. This can be more convenient for some builds because it can avoid the build time cost of creating a static library or the run time cost of symbol resolution for a dynamic library, yet still avoid having to compile the same sources multiple times.

From CMake 3.12, it is possible to link directly to an object library instead of using $<TARGET_OBJECTS:…> as outlined above. There are limitations to such linking, details of which are discussed in Section 17.2, "Libraries".

10.3. General Information

Generator expressions can provide information about more than just targets. Information can be obtained about the compiler being used, the platform for which the target is being built, the name of the build configuration and more. These sort of expressions tend to find use in more advanced situations such as handling a custom compiler or to work around a problem specific to a particular compiler or toolchain. These expressions also invite misuse, since they may appear to provide a way to do things like construct paths to things which could otherwise have been obtained using more robust methods like using TARGET_FILE expressions or other CMake features. Developers should think carefully before relying on the more general information generator expressions as a way to solve a problem. That said, some of these expressions do have valid uses. Some of the more common ones are listed here as a starting point for further reading:

Exclusively for sbmain17@gmail.com Transaction: 0098315591

$<CONFIG>

Evaluates to the build type. Use this in preference to the CMAKE_BUILD_TYPE variable since that variable is not used on multi configuration project generators like Xcode, Visual Studio or Ninja Multi-Config. Earlier versions of CMake used the now deprecated $<CONFIGURATION> expression for this, but projects should now only use $<CONFIG>.

$<PLATFORM_ID>

Identifies the platform for which the target is being built. This can be useful in cross-compiling situations, especially where a build may support multiple platforms (e.g. device and simulator builds). This generator expression is closely related to the CMAKE_SYSTEM_NAME variable and projects should consider whether using that variable would be simpler in their specific situation.

$<C_COMPILER_VERSION>, $<CXX_COMPILER_VERSION>

In some situations, it may be useful to only add content if the compiler version is older or newer than some particular version. This is achievable with the help of the $<VERSION_xxx:…> generator expressions. For example, to produce the string OLDCXX if the C++ compiler version is less than 4.2.0, the following expression could be used:

```
$<$<VERSION_LESS:$<CXX_COMPILER_VERSION>,4.2.0>:OLDCXX>
```

Such expressions tend to be used only in situations where the type of compiler is known and a specific behavior of the compiler needs to be handled in some special way by the project. It can be a useful technique in specific situations, but it can reduce the portability of the project if it relies too heavily on such expressions.

10.4. Utility Expressions

Some generator expressions modify content or substitute special characters. Below are some of the ones that are more commonly used or are easily misunderstood.

$<COMMA>

There can be scenarios where a comma needs to be included in a generator expression, but doing so would interfere with the generator expression syntax itself. To work around such cases, $<COMMA> can be used instead to prevent parsing it as part of the expression syntax.

$<SEMICOLON>

Similar to the above, there can be cases where a semicolon embedded in a generator expression may be parsed by CMake as a command argument separator. By using $<SEMICOLON> instead, argument parsing won't see a raw semicolon character, so such argument splitting will not occur.

$<LOWER_CASE:…>, $<UPPER_CASE:…>

Any content can be converted to lower or upper case via these expressions. This can be especially useful as a step before performing a string comparison. For example:

```
$<STREQUAL:$<UPPER_CASE:${someVar}>,FOOBAR>
```

Exclusively for sbmain17@gmail.com Transaction: 0098315591

$<JOIN:list,⋯>

The effect of this generator expression is to replace each semicolon in `list` with the ⋯ content, effectively joining the list items with ⋯ between each one. Note that this generator expression should never be used without quoting the whole expression. This prevents the semicolons in `list` from acting as argument separators for the command in which the generator expression is being used (see Section 8.8, "Problems With Argument Handling" for a deeper discussion of this particular topic). Quoting is also needed to prevent any spaces in the ⋯ part from acting as argument separators. The following shows a very common example of this generator expression being used incorrectly:

```
set(dirs here there)  # dirs = here;there

# ERROR: space and ; treated as argument separators
set_target_properties(Foo PROPERTIES
    CUSTOM_INC -I$<JOIN:${dirs}, -I>
)

# OK: Whole generator expression is quoted
set_target_properties(Foo PROPERTIES
    CUSTOM_INC "-I$<JOIN:${dirs}, -I>"
)
```

$<GENEX_EVAL:⋯>, $<TARGET_GENEX_EVAL:target,⋯>

These two generator expressions were introduced in CMake 3.13. In certain more advanced scenarios, a situation can arise where the evaluation of a generator expression results in content that itself contains generator expressions. One example is when evaluating a target property using `$<TARGET_PROPERTY:⋯>` and the value of the retrieved property contains another generator expression. Normally, the retrieved property is not evaluated further to expand any generator expressions it contains, but expansion can be forced using `$<GENEX_EVAL:⋯>` or `$<TARGET_GENEX_EVAL:⋯>` like so:

```
# Evaluate in current context
$<GENEX_EVAL:$<TARGET_PROPERTY:MY_PROP>>

# Evaluate for a specific target "foo"
$<TARGET_GENEX_EVAL:foo,$<TARGET_PROPERTY:foo,MY_PROP>>
```

Projects should rarely need to use these two generator expressions. The above example demonstrates the primary motivation for why they were added to CMake, but that scenario should not typically arise in most projects.

10.5. Recommended Practices

Compared to other functionality, generator expressions are a more recently added CMake feature. Because of this, much of the material online and elsewhere about CMake tends not to use them. This is unfortunate, since generator expressions are typically more robust and provide more generality than older methods. There are some common examples where well-intentioned guidance leads to logic which only works for a subset of supported project generators or platforms,

Exclusively for sbmain17@gmail.com Transaction: 0098315591

but where the use of suitable generator expressions instead would result in no such limitations. This is particularly true in relation to project logic which tries to do different things for different build types. Therefore, developers should become familiar with the capabilities that generator expressions provide. Those expressions mentioned above are only a subset of what CMake supports, but they form a good foundation for covering the majority of situations most developers are likely to face.

Used judiciously, generator expressions can result in more succinct `CMakeLists.txt` files. For example, conditionally including a source file depending on the build type can be done relatively concisely, as the example given earlier for `$<CONFIG:…>` showed. Such uses reduce the amount of if-then-else logic, resulting in better readability as long as the generator expressions are not too complex. Generator expressions are also a perfect fit for handling content that changes depending on the target or the build type. No other mechanism in CMake offers the same degree of flexibility and generality for handling the multitude of factors which may contribute to the final content needed for a particular target property.

Conversely, it is easy to go overboard and to try to make everything a generator expression. This can lead to overly complex expressions which ultimately obscure the logic and which can be difficult to debug (Section 13.4, “Debugging Generator Expressions” provides some techniques to help with that). As always, developers should favor clarity over cleverness and this is especially true with generator expressions. Consider first whether CMake already provides a dedicated facility to achieve the same result. Various CMake modules provide more targeted functionality aimed at a particular third party package or for carrying out certain specific tasks. There are also a variety of variables and properties which could simplify or replace the need for generator expressions altogether. A few minutes consulting the CMake reference documentation can save many hours of unnecessary work constructing complex generator expressions which were not really needed.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

Chapter 11. Modules

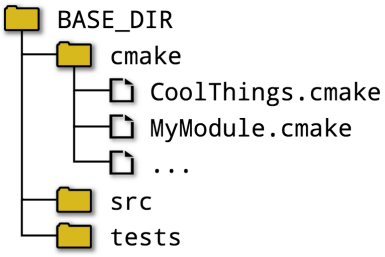
The preceding chapters have focused mostly on the core aspects of CMake. Variables, properties, flow control, generator expressions, functions, etc. are all part of what could be considered the CMake language. In contrast, modules are pre-built chunks of CMake code built on top of the core language features. They provide a rich set of functionality which projects can use to accomplish a wide variety of goals. Being written and packaged as ordinary CMake code and therefore being human readable, modules can also be a useful resource for learning more about how to get things done in CMake.

Modules are collected together and provided in a single directory as part of a CMake release. Projects employ modules in one of two ways, either directly or as part of finding an external package. The more direct method of employing modules uses the `include()` command to essentially inject the module code into the current scope. This works just like the behavior already discussed back in Section 7.2, "include()" except that only the base name of the module needs to be given to the `include()` command, not the full path or file extension. All of the options to `include()` work exactly as before.

```
include(module
    [OPTIONAL]
    [RESULT_VARIABLE myVar]
    [NO_POLICY_SCOPE]
)
```

When given a module name, the `include()` command will look in a well-defined set of locations for a file whose name is the name of the module (case-sensitive) with `.cmake` appended. For example, `include(FooBar)` would result in CMake looking for a file called `FooBar.cmake` and on case-sensitive systems like Linux, file names like `foobar.cmake` would not match.

When looking for a module's file, CMake first consults the variable `CMAKE_MODULE_PATH`. This is assumed to be a list of directories and CMake will search each of these in order. The first matching file will be used, or if no matching file is found or if `CMAKE_MODULE_PATH` is empty or undefined, CMake will then search in its own internal module directory. This search order allows projects to add their own modules seamlessly by adding directories to `CMAKE_MODULE_PATH`. A useful pattern is to collect together a project's module files in a single directory and add it to the `CMAKE_MODULE_PATH` somewhere near the beginning of the top level `CMakeLists.txt` file. The following directory structure shows such an arrangement:



Exclusively for sbmain17@gmail.com Transaction: 0098315591

The corresponding `CMakeLists.txt` file then only needs to add the `cmake` directory to the `CMAKE_MODULE_PATH` and it can then call `include()` using just the base file name when loading each of the modules.

CMakeLists.txt:

```
cmake_minimum_required(VERSION 3.0)
project(Example)

list(APPEND CMAKE_MODULE_PATH "${CMAKE_CURRENT_SOURCE_DIR}/cmake")

# Inject code from project-provided modules
include(CoolThings)
include(MyModule)
```

There is one exception to the search order used by CMake to find a module. If the file calling `include()` is itself inside CMake's own internal module directory, then the internal module directory will be searched first *before* consulting `CMAKE_MODULE_PATH`. This prevents project code from accidentally (or deliberately) replacing an official module with one of their own and changing the documented behavior.

The other way to employ modules is with the `find_package()` command. This is discussed in detail in Section 24.5, "Finding Packages", but for the moment, a simplified form of that command without any of the optional keywords demonstrates its basic usage:

```
find_package(PackageName)
```

When used in this way, the behavior is very similar to `include()` except CMake will search for a file called `FindPackageName.cmake` rather than `PackageName.cmake`. This is the method by which details about an external package are often brought into the build, including things like imported targets, variables defining locations of relevant files, libraries or programs, information about optional components, version details and so on. The set of options and features associated with `find_package()` is considerably richer than what is provided for `include()` and Chapter 24, *Finding Things* is dedicated to covering the topic in detail.

The remainder of this chapter introduces a number of interesting modules that are included as part of a CMake release. This is by no means a comprehensive set, but they do give a flavor of the sort of functionality that is available. Other modules are introduced in subsequent chapters where their functionality is closely related to the topic of discussion. The CMake documentation provides a complete list of all available modules, each with its own help section explaining what the module provides and how it can be used. Be forewarned though that the quality of the documentation does vary from module to module.

11.1. Checking Existence And Support

One of the more comprehensive areas covered by CMake's modules is checking for the existence of or support for various things. This family of modules all work in fundamentally the same way, writing a short amount of test code and then attempting to compile and possibly link and run the

Exclusively for sbmain17@gmail.com Transaction: 0098315591

resultant executable to confirm whether what is being tested in the code is supported. All of these modules have a name beginning with `Check`.

Some of the more fundamental `Check⋯` modules are those that compile and link a short test file into an executable and return a success/fail result. With CMake 3.19 or later, the `CheckSourceCompiles` module provides this capability. It defines the `check_source_compiles()` command:

```
include(CheckSourceCompiles)
check_source_compiles(lang code resultVar
    [FAIL_REGEX regexes...]
    [SRC_EXT extension]
)
```

The `lang` would be one of the languages CMake supports, such as `C`, `CXX`, `CUDA` and so on. With earlier CMake versions, separate per-language modules provide the same capabilities, but the set of supported languages is much smaller. These modules have names of the form `Check<LANG>SourceCompiles` and each one provides an associated command that performs the test:

```
include(CheckCSourceCompiles)
check_c_source_compiles(code resultVar
    [FAIL_REGEX regexes...]
)

include(CheckCXXSourceCompiles)
check_cxx_source_compiles(code resultVar
    [FAIL_REGEX regexes...]
)

include(CheckFortranSourceCompiles)
check_fortran_source_compiles(code resultVar
    [FAIL_REGEX regexes...]
    [SRC_EXT extension]
)
```

For all of these commands, the `code` argument is expected to be a string containing source code that should produce an executable for the corresponding language. The result of an attempt to compile and link the code is stored in `resultVar` as a cache variable, with true indicating success. False values could be an empty string, an error message, etc. depending on the situation. After the test has been performed once, subsequent CMake runs will use the cached result rather than performing the test again. This is the case even if the code being tested is changed. To force re-evaluation, the variable has to be manually removed from the cache.

If the `FAIL_REGEX` option is specified, then an additional criteria applies. If the output of the test compilation and linking matches any of the specified `regexes` (a list of regular expressions), the check will be deemed to have failed, even if the code compiles and links successfully.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

```
include(CheckCSourceCompiles)
check_c_source_compiles("int main() { int myVar; }"
    unusedNotDetected
    FAIL_REGEX "[Ww]arn"
)
if(unusedNotDetected)
    message("Unused variables do not generate warnings")
endif()
```

In the case of Fortran, the file extension can affect how compilers treat source files, so the file extension can be explicitly specified with the `SRC_EXT` option to obtain the expected behavior. There is no equivalent option for the C or C++ cases when using the older `Check<LANG>SourceCompiles` modules, but the newer `CheckSourceCompiles` module supports it for all languages.

A number of variables of the form `CMAKE_REQUIRED_⋯` can be set before calling any of the compilation test commands to influence how they compile the code:

`CMAKE_REQUIRED_FLAGS`

Additional flags to pass to the compiler command line after the contents of the relevant `CMAKE_<LANG>_FLAGS` and `CMAKE_<LANG>_FLAGS_<CONFIG>` variables (see Section 15.4, "Compiler And Linker Variables"). This must be a single string with multiple flags being separated by spaces, unlike all the other variables below which are CMake lists.

`CMAKE_REQUIRED_DEFINITIONS`

A CMake list of compiler definitions, each one specified in the form `-DFOO` or `-DFOO=bar`.

`CMAKE_REQUIRED_INCLUDES`

Specifies directories to search for headers. Multiple paths must be specified as a CMake list, with spaces being treated as part of a path.

`CMAKE_REQUIRED_LIBRARIES`

A CMake list of libraries to add to the linking stage. Do not prefix the library names with any `-l` option or similar, provide just the library name or the name of a CMake imported target (discussed in Chapter 17, *Target Types*).

`CMAKE_REQUIRED_LINK_OPTIONS`

A CMake list of options to be passed to the linker if building an executable, or to the archiver if building a static library. Support for this variable is only available with CMake 3.14 or later.

`CMAKE_REQUIRED_QUIET`

If this option is present, no status messages will be printed by the command.

These variables are used to construct arguments to the `try_compile()` call made internally to perform the check. The CMake documentation for `try_compile()` discusses additional variables which may have an effect on the checks, while other aspects of `try_compile()` behavior relating to toolchain selection and the type of target to build are covered in Section 22.5, "Compiler Checks".

Exclusively for sbmain17@gmail.com Transaction: 0098315591

In addition to checking whether code can be built, CMake also provides modules that test whether the built executable can be run successfully. Success is measured by the exit code of the executable created from the source provided, with 0 being treated as success and all other values indicating failure. With CMake 3.19 or later, a single module provides a command for all languages:

```
include(CheckSourceRuns)
check_source_runs(lang code resultVar
    [SRC_EXT extension]
)
```

Again, with earlier CMake versions, separate per-language modules provide the same capabilities, but with fewer supported languages:

```
# Supported by all CMake versions
include(CheckCSourceRuns)
check_c_source_runs(code resultVar)

include(CheckCXXSourceRuns)
check_cxx_source_runs(code resultVar)

# Requires CMake 3.14 or later
include(CheckFortranSourceRuns)
check_fortran_source_runs(code resultVar
    [SRC_EXT extension]
)
```

There is no `FAIL_REGEX` option for these commands, as success or failure is determined purely by the exit code of the test process. If the code cannot be built, this is also treated as a failure. All the same variables that affect how the code is built for `check_source_compiles()` or `check_<lang>_source_compiles()` also have the same effect for these modules' commands as well.

For builds that are cross-compiling to a different target platform, the `check_source_runs()` and `check_<lang>_source_runs()` commands behave quite differently. They may run the code under a simulator if the necessary details have been provided, which would likely slow down the CMake stage considerably. If simulator details have not been provided, the commands will instead expect a predetermined result to be provided through a set of variables and will not try to run anything. This fairly advanced topic is covered in CMake's documentation for the `try_run()` command, which is what the module commands use internally to perform the checks.

Certain categories of checks are so common that CMake provides dedicated modules for them. These remove much of the boilerplate of defining the test code and allow projects to specify a minimal amount of information for the check. These are typically just wrappers around the commands provided by one of the modules mentioned above, so the same set of variables used for customizing how the test code is built still apply. These more specialized modules check compiler flags, pre-processor symbols, functions, variables, header files and more.

Just as for the modules above, CMake 3.19 or later provides a single module and command for all supported languages. For earlier CMake versions, a set of per-language modules must be used instead.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

```
# Requires CMake 3.19 or later
include(CheckCompilerFlag)
check_compiler_flag(lang flag resultVar)
```

```
# Supported by all CMake versions
include(CheckCCompilerFlag)
check_c_compiler_flag(flag resultVar)

include(CheckCXXCompilerFlag)
check_cxx_compiler_flag(flag resultVar)

# Requires CMake 3.3 or later
include(CheckFortranCompilerFlag)
check_fortran_compiler_flag(flag resultVar)
```

The flag-checking commands update the `CMAKE_REQUIRED_DEFINITIONS` variable internally to include `flag` in a call to `check_source_compiles()` with a trivial test file. An internal set of failure regular expressions is also passed as the `FAIL_REGEX` option, testing whether the flag results in a diagnostic message being issued or not. The result of the call will be a true value if no matching diagnostic message is issued. Note that this means any flag that results in a compiler warning but successful compilation will still be deemed to have failed the check. Also be aware that these commands assume that any flags already present in the relevant `CMAKE_<LANG>_FLAGS` variables (see Section 15.4, "Compiler And Linker Variables") do not themselves generate any compiler warnings. If they do, then the logic for each of these flag-testing commands will be defeated and the result of all such checks will be failure.

CMake 3.18 also introduced the `CheckLinkerFlag` module. It provides the analogous command `check_linker_flag()`, which is mostly just a convenience wrapper around `check_source_compiles()`. As such, it supports all the same variables as previously discussed, except that it takes over handling of the `CMAKE_REQUIRED_LINK_OPTIONS` variable.

```
include(CheckLinkerFlag)
check_linker_flag(language flag resultVar)
```

The specified `flag` is not passed directly to the linker. The linker is invoked via the compiler, which internally adds extra language-specific flags, libraries, etc. needed to successfully link for the specified `language`. A raw linker flag will usually not work, it typically needs some sort of prefix like `-Wl,⋯` or `-Xlinker` to tell the compiler to pass it through to the linker. This prefix is compiler-specific, but the special prefix `LINKER:` can be used and CMake will substitute the correct compiler-specific prefix automatically. See Section 15.1.2, "Linker Flags" and the `target_link_options()` command in Section 15.1.3, "Target Property Commands" for related discussions.

```
include(CheckLinkerFlag)
check_linker_flag(CXX LINKER:-stats LINKER_STATS_SUPPORTED)
```

Two other notable modules are `CheckSymbolExists` and `CheckCXXSymbolExists`. The former provides a command which builds a test C executable and the latter does the same as a C++ executable. Both

Exclusively for sbmain17@gmail.com Transaction: 0098315591

check whether a particular symbol exists as either a pre-processor symbol (i.e. something that can be tested via an `#ifdef` statement), a function or a variable.

```
include(CheckSymbolExists)
check_symbol_exists(symbol headers resultVar)

include(CheckCXXSymbolExists)
check_cxx_symbol_exists(symbol headers resultVar)
```

For each of the items specified in `headers` (a CMake list if more than one header needs to be given), a corresponding `#include` will be added to the test source code. In most cases, the symbol being checked will be defined by one of these headers. The result of the test is stored in the `resultVar` cache variable in the usual way.

In the case of functions and variables, the symbol needs to resolve to something that is part of the test executable. If the function or variable is provided by a library, that library must be linked as part of the test, which can be done using the `CMAKE_REQUIRED_LIBRARIES` variable.

```
include(CheckSymbolExists)
check_symbol_exists(sprintf stdio.h HAVE_SPRINTF)

include(CheckCXXSymbolExists)
set(CMAKE_REQUIRED_LIBRARIES SomeCxxSDK)
check_cxx_symbol_exists(SomeCxxInitFunc somecxxsdk.h HAVE_SOMECXXSDK)
```

There are limitations on the sort of functions and variables that can be checked by these commands. Only those symbols that satisfy the naming requirements for a preprocessor symbol can be used. The implications are stronger for `check_cxx_symbol_exists()`, since it means only non-template functions or variables in the global namespace can be checked because any scoping (`::`) or template markers (`<>`) would not be valid for a preprocessor symbol. It is also impossible to distinguish between different overloads of the same function, so these cannot be checked either.

There are other modules which aim to provide functionality that is similar to or a subset of that covered by `CheckSymbolExists`. These other modules are either from earlier versions of CMake or are for a language other than C or C++. The `CheckFunctionExists` module is already documented as being deprecated and the `CheckVariableExists` module offers nothing that `CheckSymbolExists` doesn't already provide. The `CheckFortranFunctionExists` module may be useful for those projects working with Fortran, but note that there is no `CheckFortranVariableExists` module. Fortran projects may want to use `CheckFortranSourceCompiles` for consistency instead.

More detailed checks are provided by other modules. For example, struct members can be tested with `CheckStructHasMember`, specific C or C++ function prototypes can be tested with `CheckPrototypeDefinition` and the size of non-user types can be tested with `CheckTypeSize`. Other higher level checks are also possible, as provided by `CheckLanguage`, `CheckLibraryExists` and the various `CheckIncludeFile⋯` modules. Further check modules continue to be added to CMake as it evolves, so consult the CMake module documentation to see the full set of available functionality.

In situations where multiple checks are being made or where the effects of performing the checks need to be isolated from each other or from the rest of the current scope, it can be cumbersome to

Exclusively for sbmain17@gmail.com Transaction: 0098315591

manually save and restore the state before and after the checks. In particular, the various CMAKE_REQUIRED_⋯ variables often need to be saved and restored. To help with this, CMake provides the CMakePushCheckState module which defines the following three commands:

```
include(CMakePushCheckState)
cmake_push_check_state([RESET])
cmake_pop_check_state()
cmake_reset_check_state()
```

These commands allow the various CMAKE_REQUIRED_⋯ variables to be treated as a set and to have their state pushed and popped onto/from a virtual stack. Each time cmake_push_check_state() is called, it effectively begins a new virtual variable scope for just the CMAKE_REQUIRED_⋯ variables (and also the CMAKE_EXTRA_INCLUDE_FILES variable which is only used by the CheckTypeSize module). cmake_pop_check_state() is the opposite, it discards the current values of the CMAKE_REQUIRED_⋯ variables and restores them to the previous stack level's values. The cmake_reset_check_state() command is a convenience for clearing all the CMAKE_REQUIRED_⋯ variables and the RESET option to cmake_push_check_state() is also just a convenience for clearing the variables as part of the push. Note, however, that a bug existed prior to CMake 3.10 which resulted in the RESET option being ignored, so for projects that need to work with versions before 3.10, it is better to use a separate call to cmake_reset_check_state() instead.

```
# Start with a known state we can modify and undo later
include(CMakePushCheckState)
cmake_push_check_state()
cmake_reset_check_state()

set(CMAKE_REQUIRED_FLAGS -Wall)
include(CheckSymbolExists)
check_symbol_exists(FOO_VERSION foo/version.h HAVE_FOO)

if(HAVE_FOO)
  # Preserve -Wall and add more things for extra checks
  cmake_push_check_state()
  set(CMAKE_REQUIRED_INCLUDES foo/inc.h foo/more.h)
  set(CMAKE_REQUIRED_DEFINES -DFOOBXX=1)
  check_symbol_exists(FOOBAR "" HAVE_FOOBAR)
  check_symbol_exists(FOOBAZ "" HAVE_FOOBAZ)
  check_symbol_exists(FOOBOO "" HAVE_FOOBOO)
  cmake_pop_check_state()
  # Now back to just -Wall
endif()

# Clear CMAKE_REQUIRED_... variables for this last check
cmake_reset_check_state()
check_symbol_exists(__TIME__ "" HAVE_PPTIME)

# Restore all CMAKE_REQUIRED_... variables to their
# original values from the top of this example
cmake_pop_check_state()
```

Exclusively for sbmain17@gmail.com Transaction: 0098315591

11.2. Other Modules

CMake has excellent built-in support for some languages, especially C and C++. It also includes a number of modules which provide support for languages in a more extensible and configurable way. These modules allow aspects of some languages or language-related packages to be made available to projects by defining relevant commands, variables and properties. Many of these modules are provided as part of the support for `find_package()` calls (see Section 24.5, "Finding Packages"), while others are intended to be used more directly via `include()` to bring things into the current scope. The following module list should give a taste of the sort of language support available:

- `CSharpUtilities`
- `FindCUDA` (but note this has been superseded by support for CUDA as a first class language in its own right in recent CMake versions)
- `FindJava`, `FindJNI`, `UseJava`
- `FindLua`
- `FindMatlab`
- `FindPerl`, `FindPerlLibs`
- `FindPython`
- `FindPHP4`
- `FindRuby`
- `FindSWIG`, `UseSWIG`
- `FindTCL`
- `FortranCInterface`

In addition, modules are also provided for interacting with external data and projects, a topic covered in depth in Chapter 28, *External Content*. A number of modules are also provided to facilitate various aspects of testing and packaging. These have a close relationship with the CTest and CPack tools distributed as part of the CMake suite and are covered in depth in Chapter 25, *Testing* and Chapter 27, *Packaging*. Debugging assistance is also provided by the `CMakePrintHelpers` module (see Section 13.2, "Print Helpers").

11.3. Recommended Practices

CMake's collection of modules provides a wealth of functionality built on top of the core CMake language. A project can easily extend the set of available functionality by adding their own custom modules under a particular directory and then adding that path to the `CMAKE_MODULE_PATH` variable. The use of `CMAKE_MODULE_PATH` should be preferred over hard-coding absolute or relative paths across complex directory structures in `include()` calls, since this will encourage generic CMake logic to be decoupled from the places where that logic may be applied. This in turn makes it easier to relocate CMake modules to different directories as a project evolves, or to re-use the logic across different projects. Indeed, it is not unusual for an organization to build up its own collection of modules, perhaps even storing them in their own separate repository. By setting `CMAKE_MODULE_PATH`

Exclusively for sbmain17@gmail.com Transaction: 0098315591

appropriately in each project, those reusable CMake building blocks are then made available for use as widely as needed.

Over time, a developer will typically be exposed to an increasing number of interesting scenarios for which a CMake module may provide useful shortcuts or ready-made solutions. Sometimes a quick scan of the available modules can yield an unexpected hidden gem, or a new module may offer a better maintained implementation of something a project has been implementing in an inferior way up to that point. CMake's modules have the benefit of a potentially large pool of developers and projects using them across a diverse set of platforms and situations, so they may offer a more compelling alternative to projects doing their own manual logic in many cases. The quality does, however, vary from one module to another. Some modules began their life quite early on in CMake's existence and these can sometimes become less useful if not kept up to date with changes to CMake or to the areas those modules relate to. This can be particularly true of `Find⋯` modules which may not track newer versions of the packages they are finding as closely as one might like. On the other hand, modules are just ordinary CMake code, so anyone can inspect them, learn from them, improve or update them without having to learn much beyond what is needed for basic CMake use in a project. In fact, they are an excellent starting point for developers wishing to get involved with working on CMake itself.

The abundance of different `Check⋯` modules provided by CMake can be a mixed blessing. Developers can be tempted to get too over-zealous with checking all manner of things, which can result in slowing down the configure stage for sometimes questionable gains. It is common to see the commands related to these checks dominate the profiling results of a CMake run (see Section 13.5, "Profiling CMake Calls"). Consider whether the benefits outweigh the costs in terms of time to implement and maintain the checks and the complexity of the project. Sometimes a few judicious checks are sufficient for covering the most useful cases, or to catch a subtle problem that might otherwise cause hard to trace problems later. Furthermore, if using any of the `Check⋯` modules, aim to isolate the checking logic from the scope in which it may be invoked. Use of the `CMakePushCheckState` module is highly recommended, but avoid using the `RESET` option to `cmake_push_check_state()` if support for CMake versions before 3.10 is important.

When the minimum CMake version can be set to 3.20 or later, avoid using the relatively popular but now deprecated `TestBigEndian` module. That module was deprecated in CMake 3.20 in favor of a new `CMAKE_<LANG>_BYTE_ORDER` variable, which was also introduced in the same CMake release. Projects that are using `TestBigEndian` should transition to the new variable where possible.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

Chapter 12. Policies

CMake has evolved over a long period, introducing new functionality, fixing bugs and changing the behavior of certain features to address shortcomings or introduce improvements. While the introduction of new capabilities is unlikely to cause problems for existing projects built with CMake, any change in behavior has the potential to break projects if they are relying on the old behavior. For this reason, the CMake developers are careful to ensure that changes are implemented in such a way as to preserve backward compatibility and to provide a straightforward, controlled migration path for projects to be updated to the new behavior.

This control over whether old or new behavior should be used is done through CMake's policy mechanisms. In general, policies are not something that developers are exposed to all that often, mostly just when CMake issues a warning about the project relying on an older version's behavior. When developers move to a more recent CMake release, the newer CMake version will sometimes issue such warnings to highlight how the project should be updated to use a new behavior.

12.1. Policy Control

CMake's policy functionality is closely tied to the `cmake_minimum_required()` command, which was introduced back in Chapter 3, *A Minimal Project*. Not only does this command specify the minimum CMake version a project requires, it also sets CMake's behavior to match that of the version given. Thus, when a project starts with `cmake_minimum_required(VERSION 3.2)`, it says that at least CMake 3.2 is needed and also that the project expects CMake to behave like the 3.2 release. This gives projects confidence that developers should be able to update to any newer version of CMake at their convenience and the project will still build as it did before.

Sometimes, however, a project may want more fine-grained control than what the `cmake_minimum_required()` command provides. Consider the following scenarios:

- A project wants to set a low minimum CMake version, but it also wants to take advantage of newer behavior if it is available.
- A part of the project is not able to be modified (e.g. it might come from an external read-only code repository) and it relies on old behavior which has been changed in newer CMake versions. The rest of the project, however, wants to move to the new behavior.
- A project relies heavily on some old behavior which would require a non-trivial amount of work to update. Some parts of the project want to make use of recent CMake features, but the old behavior for that particular change needs to be preserved until time can be set aside to update the project.

These are some common examples where the high level control provided by the `cmake_minimum_required()` command alone is not enough. More specific control over policies is enabled through the `cmake_policy()` command, which has a number of forms acting at different degrees of granularity. The form acting at the coarsest level is a close relative to `cmake_minimum_required()`:

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


Exclusively for sbmain17@gmail.com Transaction: 0098315591

In this form, the command changes CMake's behavior to match that of the specified version. The `cmake_minimum_required()` command implicitly calls this form to set CMake's behavior. The two are largely interchangeable except for the top of the project where a call to `cmake_minimum_required()` is mandatory to enforce a minimum CMake version. Apart from the start of the top level `CMakeLists.txt` file, using `cmake_policy()` generally communicates the intent more clearly when a project needs to enforce a particular version's behavior for a section of the project, as demonstrated by the following example:

```
cmake_minimum_required(VERSION 3.7)
project(WithLegacy)

# Uses recent CMake features
add_subdirectory(modernDir)

# Imported from another project, relies on old behavior
cmake_policy(VERSION 2.8.11)
add_subdirectory(legacyDir)
```

CMake 3.12 extends this capability in a backward-compatible way by optionally allowing the project to specify a version *range* rather than a single version to either `cmake_minimum_required()` or `cmake_policy(VERSION)`. The range is specified using three dots `...` between the minimum and maximum version with no spaces. The range indicates that the CMake version in use must be at least the minimum and the behavior to use should be the lesser of the specified maximum and the running CMake version. This allows the project to effectively say "I need at least CMake X, but I am safe to use with policies from up to CMake Y". The following example shows two ways for a project to require only CMake 3.7, but still support the newer behavior for all policies up to CMake 3.12 if the running CMake version supports them:

```
cmake_minimum_required(VERSION 3.7...3.12)
cmake_policy(VERSION 3.7...3.12)
```

CMake versions before 3.12 would effectively see just a single version number and would ignore the `...3.12` part, whereas 3.12 and later would understand it to mean a range.

CMake also provides the ability to control each behavior change individually with the `SET` form:

```
cmake_policy(SET CMPxxxx NEW)
cmake_policy(SET CMPxxxx OLD)
```

Each individual behavior change is given its own policy number of the form `CMPxxxx`, where `xxxx` is always four digits. By specifying `NEW` or `OLD`, a project tells CMake to use the new or old behavior for that particular policy. The CMake documentation provides the full list of policies, along with an explanation of the OLD and NEW behavior of each one.

As an example, before version 3.0, CMake allowed a project to call `get_target_property()` with the name of a target that didn't exist. In such a case, the value of the property was returned as `-NOTFOUND` rather than issuing an error, but in all likelihood, the project probably contained incorrect logic. Therefore, from version 3.0 onward, CMake halts with an error if such a situation is encountered.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

In the event that a project was relying on the old behavior, it could continue to do so using policy `CMP0045` like so:

```
# Allow non-existent target with get_target_property()
cmake_policy(SET CMP0045 OLD)

# Would halt with an error without the above policy change
get_target_property(outVar doesNotExist COMPILE_DEFINITIONS)
```

The need for setting a policy to `NEW` is less common. One situation is where a project wants to set a low minimum CMake version, but still take advantage of later features if a later version is used. For example, in CMake 3.2, policy `CMP0055` was introduced to provide strict checking on usage of the `break()` command. If the project still wanted to support being built with earlier CMake versions, then the additional checks would have to be explicitly enabled when run with later CMake versions.

```
cmake_minimum_required(VERSION 3.0)
project(PolicyExample)

if(CMAKE_VERSION VERSION_GREATER 3.1)
    # Enable stronger checking of break() command usage
    cmake_policy(SET CMP0055 NEW)
endif()
```

Testing the `CMAKE_VERSION` variable is one way of determining whether a policy is available, but the `if()` command provides a more direct way using the `if(POLICY…)` form. The above could alternatively be implemented like so:

```
cmake_minimum_required(VERSION 3.0)
project(PolicyExample)

# Only set the policy if the version of CMake being used
# knows about that policy number
if(POLICY CMP0055)
    cmake_policy(SET CMP0055 NEW)
endif()
```

It is also possible to get the current state of a particular policy. The main situation where the current policy setting may need to be read is in a module file, which may be one provided by CMake itself or by the project. It would be unusual, however, for a project module to change its behavior based on a policy setting.

```
cmake_policy(GET CMPxxxx outVar)
```

The value stored in `outVar` will be `OLD`, `NEW` or empty. The `cmake_minimum_required(VERSION…)` and `cmake_policy(VERSION…)` commands reset the state of all policies. Those policies introduced at the specified CMake version or earlier are reset to `NEW`. Those policies that were added after the specified version will effectively be reset to empty.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

If CMake detects that the project is doing something that either relies on the old behavior, conflicts with the new behavior or whose behavior is ambiguous, it may warn if the relevant policy is unset. These warnings are the most common way developers are exposed to CMake's policy functionality. They are designed to be noisy but informative, encouraging developers to update the project to the new behavior. In some cases, a deprecation warning may be issued even if the policy has been explicitly set, but this is typically only for a policy that has already been documented as deprecated for a long time (many releases).

Sometimes the policy warnings cannot be addressed immediately, but the warnings could be undesirable. The preferred way to handle this is to explicitly set the policy to the desired behavior (`OLD` or `NEW`), which stops the warning. This isn't always possible though, such as when a deeper part of the project issues its own call to `cmake_minimum_required(VERSION…)` or `cmake_policy(VERSION…)`, thereby resetting the policy states. As a temporary way to work around such situations, CMake provides the `CMAKE_POLICY_DEFAULT_CMPxxxx` and `CMAKE_POLICY_WARNING_CMPxxxx` variables where `xxxx` is the usual four-digit policy number. These are not intended to be set by the project, but rather by the developer as a cache variable temporarily to enable/disable a warning or to check whether the project issues warnings with a particular policy enabled. Ultimately, the long term solution is to address the underlying problem highlighted by the warning. Nevertheless, it may occasionally be appropriate for a project to set one of these variables to silence a warning known to not be harmful.

12.2. Policy Scope

Sometimes a policy setting only needs to be applied to a specific section of a file. Rather than requiring a project to manually save the existing value of any policies it wants to change temporarily, CMake provides a policy stack which can be used to simplify this process:

```
cmake_policy(PUSH)
cmake_policy(POP)
```

The existing state of all policies can be saved with a `PUSH` operation and the current state discarded with a corresponding `POP`. Every `PUSH` is required to eventually have a matching `POP`. In between, the project can modify the settings of any policies it needs to without having to explicitly save each one first. Again, module files are one of the more common places where the policy stack might be manipulated like this. A simple example might be a module file which sets a few policies temporarily like so:

```
# Save existing policy state
cmake_policy(PUSH)

# Set some policies to OLD to preserve a few old behaviors
cmake_policy(SET CMP0060 OLD)  # Library path linking behavior
cmake_policy(SET CMP0021 OLD)  # Tolerate relative INCLUDE_DIRECTORIES

# Do various processing here...

# Restore earlier policy settings
cmake_policy(POP)
```

Exclusively for sbmain17@gmail.com Transaction: 0098315591

Some commands implicitly push a new policy state onto the stack and pop it again at a well defined point later. One example is the `add_subdirectory()` command which pushes a new policy scope onto the stack upon entering the specified subdirectory and pops it again when the command returns. The `include()` command does a similar thing, pushing a new policy scope before starting to process the specified file and popping it again when processing of that file is completed. The `find_package()` command also does a similar thing to `include()`, pushing and popping upon starting and finishing processing of its associated `FindXXX.cmake` module file respectively.

The `include()` and `find_package()` commands also support a `NO_POLICY_SCOPE` option which prevents the automatic push-pop of the policy stack (`add_subdirectory()` has no such option). In very early versions of CMake, `include()` and `find_package()` did not automatically push and pop an entry on the policy stack. The `NO_POLICY_SCOPE` option was added as a way for projects using later CMake versions to revert back to the old behavior for specific parts of the project, but its use is generally discouraged and should be unnecessary for new projects.

12.3. Recommended Practices

Where possible, projects should prefer to work with policies at the CMake version level rather than manipulating specific policies. Setting policies to match a particular CMake release's behavior makes the project easier to understand and update, whereas changes to individual policies can be harder to trace through multiple directory levels, especially because of their interaction with version-level policy changes where they are always reset.

When choosing how to specify the CMake version to conform to, the choice between `cmake_minimum_required(VERSION)` and `cmake_policy(VERSION)` would usually fall to the latter. The two main exceptions to this are at the start of the project's top level `CMakeLists.txt` file and at the top of a module file that could be re-used across multiple projects. For the latter case, it is preferable to use `cmake_minimum_required(VERSION)` because the projects using the module may enforce their own minimum CMake version, but the module may have specific minimum version requirements of its own. Aside from these cases, `cmake_policy(VERSION)` usually expresses the intent more clearly, but both commands will effectively achieve the same thing from a policy perspective.

In cases where a project does need to manipulate a specific policy, it should check whether the policy is available using `if(POLICY…)` rather than testing the `CMAKE_VERSION` variable. This leads to greater consistency of the code. Compare the following two ways of setting policy behavior and note how the check and the enforcement use a consistent approach:

```
# Version-level policy enforcement
if(NOT CMAKE_VERSION VERSION_LESS 3.4)
    cmake_policy(VERSION 3.4)
endif()

# Individual policy-level enforcement
if(POLICY CMP0055)
    cmake_policy(SET CMP0055 NEW)
endif()
```

If a project needs to manipulate multiple individual policies locally, surround that section with calls to `cmake_policy(PUSH)` and `cmake_policy(POP)` to ensure that the rest of the scope is isolated from the

Exclusively for sbmain17@gmail.com Transaction: 0098315591

changes. Pay special attention to any possible `return()` statements that exit that section of code and ensure no push is left without a corresponding pop. Note also that `add_subdirectory()`, `include()` and `find_package()` all push and pop an entry on the policy stack automatically, so no explicit push and pop is needed unless policy settings need to be changed locally for a small section of the file being pulled in. Projects should avoid the `NO_POLICY_SCOPE` keyword of these commands, as it is intended only for addressing a change in behavior of very early CMake versions and its use is rarely appropriate for new projects.

Aim to avoid modifying policy settings inside a function. Since functions do not introduce a new policy scope, a policy change can affect the caller if the change is not properly isolated using the appropriate push-pop logic. Furthermore, the policy settings for the function implementation are taken from the scope in which the function was *defined*, not the one from which it is called. Therefore, prefer to adjust any policy settings in the scope that defines the function rather than within the function itself.

As a last resort, the `CMAKE_POLICY_DEFAULT_CMPxxxx` and `CMAKE_POLICY_WARNING_CMPxxxx` variables may allow a developer or project to work around some specific policy-related situations. Developers may use these to temporarily change a specific policy setting's default or to prevent warnings about a particular policy. Projects should generally avoid setting these variables so that developers have control locally, but in certain situations, they can be used to ensure the behavior or warning about a particular policy persists even through calls to `cmake_minimum_required()` or `cmake_policy(VERSION)`. Where possible, projects should instead try to update to the newer behavior rather than relying on these variables.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

Chapter 13. Debugging And Diagnostics

When a build is behaving well, users tend not to pay much attention to the output generated by CMake. For developers working on a project, however, diagnostic output and debugging capabilities are essential. CMake has always provided basic printing functionality, but enhancements added in versions 3.15 to 3.18 significantly extended the available capabilities.

13.1. Log Messages

CMake has always supported logging arbitrary text using the `message()` command, which was introduced briefly back in Section 5.6, "Printing Variable Values". The more general form of that command is:

```
message([mode] msg1 [msg2]...)
```

If more than one `msg` is specified, they will be joined into a single string with no separators. To preserve spaces, semicolons or newlines, surround the message with quotes (see Section 8.8, "Problems With Argument Handling" for a detailed explanation of why).

The message output can be affected by the optional `mode` argument, `cmake` command-line options and the value of a few variables at the time of the call. The next few subsections cover these in detail.

13.1.1. Log Levels

The `message()` command accepts an optional `mode` keyword which provides information about the type of message being provided. It affects how and where the message is output, whether it is output at all and in some cases can halt further processing for that CMake run. Recognized `mode` values in order of importance are:

`FATAL_ERROR`

Denotes a hard error. Processing will stop immediately after the message is printed and the log will also normally record the location of the fatal `message()` command.

`SEND_ERROR`

Like `FATAL_ERROR` except processing will continue until the configure stage completes, but generation will not be performed. This can be quite confusing for users, so projects should avoid this mode and prefer to use `FATAL_ERROR` instead.

`WARNING`

Denotes a warning. The log will also normally record the location of the `message()` command raising the warning. Processing will continue.

`AUTHOR_WARNING`

Like `WARNING`, but only shown if developer warnings are enabled (use the `-Wno-dev` option on the `cmake` command line to disable them). Projects do not often use this particular type of message, they are usually generated by CMake itself.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

DEPRECATION

Special category used to log a deprecation message. If the `CMAKE_ERROR_DEPRECATED` variable is set to true, the message will be treated as an error. If the `CMAKE_WARN_DEPRECATED` variable is set to true, the message will be treated as a warning. If neither variable is set, the message will be shown for CMake 3.5 or later and hidden for earlier versions.

NOTICE

This keyword is only recognized with CMake 3.15 or later, but for all versions, this is the default log level when no mode keyword is provided. The keyword was added for consistency and to allow projects to be clearer about the meaning of such messages. Avoid using this log level if the message doesn't require any action from the user (see further below).

STATUS

Concise status information, generally expected to be a single line. Prefer to use this message mode rather than `NOTICE` for purely informational messages.

VERBOSE

(CMake 3.15 or later only) More detailed information that wouldn't normally be of interest, but could be helpful to project users when seeking a deeper understanding of what is happening.

DEBUG

(CMake 3.15 or later only) Not intended for project users, but rather for developers working on the project itself. These may record internal implementation details that would not be of interest to those simply wanting to build the project.

TRACE

(CMake 3.15 or later only) Very low level details, used almost exclusively for temporary messages during project development.

Messages of `STATUS` through to `TRACE` level will be printed to `stdout`, whereas `NOTICE` and above are printed to `stderr`. This can result in messages of different log levels sometimes appearing out of order in the output. Furthermore, messages on `stderr` usually imply a problem or something that the user should investigate, so `NOTICE` is generally a poor choice for purely informational messages that don't require follow-up. Use `STATUS` or below for such messages.

Messages of `STATUS` through to `TRACE` may also have two hyphens and a space automatically prepended. The CMake GUI application and the `ccmake` tool do not prepend this prefix, whereas the current version of the `cmake` tool will. Future CMake versions may drop this prefix completely, so do not rely on it being present. No such prefix is prepended for messages of `NOTICE` level and above.

CMake 3.15 also added the ability to set a minimum logging level with the `--loglevel=…` command-line option. This option was renamed to `--log-level` in CMake 3.16 for consistency reasons, but `--loglevel` is still accepted for backward compatibility. The option specifies the desired log level and only messages of that level or higher will be shown. When no `--log-level` option is given, only messages of `STATUS` level or higher will be recorded.

```
cmake --log-level=VERBOSE ...
```

Exclusively for sbmain17@gmail.com Transaction: 0098315591

CMake 3.17 added the ability to specify the default log level with the `CMAKE_MESSAGE_LOG_LEVEL` variable. It is overridden by the `--log-level` command line option if both are present. The cache variable is intended for developer use only, projects should not try to read or modify it.

13.1.2. Message Indenting

When a project logs a non-trivial amount of output, adding some structure can help the user better understand which parts of the project each message relates to. One way to do that is to make use of the `CMAKE_MESSAGE_INDENT` variable. When using CMake 3.16 or later, the contents of this variable at the time of the call to `message()` will be concatenated and prepended to the message for log levels of `NOTICE` and below. If the message contains embedded newlines, the indentation contents will be prepended to each line of the output.

```
set(CMAKE_MESSAGE_INDENT aa bb)  # Don't do this, see below
message("First line\nSecond line")
```

```
aabbFirst line
aabbSecond line
```

While the above example demonstrates how the feature works, it has problems. The general expectation is that the list elements in `CMAKE_MESSAGE_INDENT` will only contain whitespace, typically two spaces each. This isn't a requirement, but deviating from it will likely be annoying for users. Projects should also never `set()` the variable, they should only append to it, typically by calling `list(APPEND)`. This avoids any assumption about the contents of the variable and always preserves the existing indenting. This is especially important for the output of hierarchical projects (discussed in detail in Section 28.2, "FetchContent") or when using indenting within function calls. The following example demonstrates these guidelines and how they can be applied in practice.

```
function(funcA)
    list(APPEND CMAKE_MESSAGE_INDENT "  ")
    message("${CMAKE_CURRENT_FUNCTION}")
endfunction()

function(funcB)
    list(APPEND CMAKE_MESSAGE_INDENT "  ")
    message("${CMAKE_CURRENT_FUNCTION}")
    funcA()
endfunction()

function(funcC)
    list(APPEND CMAKE_MESSAGE_INDENT "  ")
    message("${CMAKE_CURRENT_FUNCTION}")
    funcB()
endfunction()

message("Top level")
funcA()
funcB()
funcC()
```


Exclusively for sbmain17@gmail.com Transaction: 0098315591

```
Top level
  funcA
  funcB
    funcA
  funcC
    funcB
      funcA
```

Note how the indenting of output from both `funcA()` and `funcB()` varies depending on the call stack. Another feature of the example is that because functions introduce their own variable scope, it is not necessary to pop the indent off the end of the list before returning. The caller has its own separate copy of the `CMAKE_MESSAGE_INDENT` variable, so from its perspective, the value of the variable doesn't change as a result of the function call.

Projects can add support for indenting even if their minimum CMake version is less than 3.16. Older CMake versions will simply ignore the indenting and output will be unchanged.

13.1.3. Message Contexts

CMake 3.17 extended the support for message metadata even further. In the same way that `CMAKE_MESSAGE_INDENT` can be used to provide indenting, the `CMAKE_MESSAGE_CONTEXT` variable can be used to provide information about the context in which each message is generated. This can be used to record things like the project name or some logical part within the project, for example. Users can then instruct CMake to print context information with each message by including the `--log-context` option on the `cmake` command line.

When the `--log-context` option is given and `CMAKE_MESSAGE_CONTEXT` is not empty, a prefix is generated for each line of output from a call to `message()`. This prefix will be the concatenation of the items in `CMAKE_MESSAGE_CONTEXT`, with each item separated by a dot. The result of that will be enclosed in square brackets and a space added to the end of the prefix. For messages logged at `STATUS` level or below, the context follows after any leading hyphens that may be added by `cmake`.

An example best demonstrates the usage of this feature:

CMakeLists.txt:

```
cmake_minimum_required(VERSION 3.17)
list(APPEND CMAKE_MESSAGE_CONTEXT Coolio)
project(Coolio)

message("Adding features\nHere we go:")

add_subdirectory(networking)
add_subdirectory(graphics)

message("All done")
```

Exclusively for sbmain17@gmail.com Transaction: 0098315591

networking/CMakeLists.txt:

```
list(APPEND CMAKE_MESSAGE_CONTEXT net)
message("Doing something")
```

graphics/CMakeLists.txt:

```
list(APPEND CMAKE_MESSAGE_CONTEXT graphics)
message("Doing something else")
```

Running `cmake --log-context` on the above would result in output like the following:

```
-- [Coolio] The C compiler identification is GNU 9.3.0
-- [Coolio] The CXX compiler identification is GNU 9.3.0
-- [Coolio] Detecting C compiler ABI info
-- [Coolio] Detecting C compiler ABI info - done
-- [Coolio] Check for working C compiler: /.../cc - skipped
-- [Coolio] Detecting C compile features
-- [Coolio] Detecting C compile features - done
-- [Coolio] Detecting CXX compiler ABI info
-- [Coolio] Detecting CXX compiler ABI info - done
-- [Coolio] Check for working CXX compiler: /.../c++ - skipped
-- [Coolio] Detecting CXX compile features
-- [Coolio] Detecting CXX compile features - done
[Coolio] Adding features
[Coolio] Here we go:
[Coolio.net] Doing something
[Coolio.graphics] Doing something else
[Coolio] All done
-- Configuring done
-- Generating done
-- Build files have been written to: /...
```

Compiler feature checks are triggered by the call to `project()`, so `Coolio` is appended to `CMAKE_MESSAGE_CONTEXT` before that call to ensure that the check output includes context information. The final few lines at the end of the output which record the completion of the different stages of the run will always have no context.

The general expectation is that when the output of context information is enabled, that output is going to undergo further post-processing of some sort rather than being shown directly to the user. For example, an IDE tool might use this to gain understanding of the structure of the output and offer filtering capabilities to the user. The IDE might support the user expanding and collapsing or showing and hiding parts of the `cmake` output according to their interest.

Another use case would be scripted builds. These might use Unix commands like `tee` and `awk` to save an annotated log to a file, but still show the output without context details to allow it to be monitored in real time. The saved file can then be searched later to find specific lines of interest. The following is one way of accomplishing this within a `bash` shell on a Unix-like system (it also strips off any leading hyphen prefix along with the context information):

Exclusively for sbmain17@gmail.com Transaction: 0098315591

```
cmake . --log-context |& \
        tee out.log | \
        awk 'sub("^((-- )?(\\[[^\\]]*\\] ))?", "")'
```

One could then extract just the messages associated with networking from the saved log file using a tool like `grep`:

```
grep -E '^(-- )?\[.*Coolio\.net\] ' out.log
```

CMake places certain restrictions on what a project may use as context names. Valid context names (i.e. each item in the `CMAKE_MESSAGE_CONTEXT` list) are those that could be used as the name of a CMake variable. For the most part, this essentially means letters, numbers and underscores. In addition, CMake considers context names that begin with `cmake_` or a leading underscore to be reserved for its own use.

Message contexts are particularly effective when `message()` calls specify appropriate log levels. For example, projects may provide more detailed information using a `VERBOSE` log level, but only fairly minimal output at `STATUS` level or higher. That would make the default output uncluttered, but using a `--log-level` of `VERBOSE` would provide extra detail when needed. Users could then focus in on just the details they want by searching for the message context(s) of interest.

13.1.4. Check Messages

Another useful feature available with CMake 3.17 or later is support for messages that record the status of some form of check. The syntax is essentially the same as the main form of the `message()` command, but the meaning of the first argument is different:

```
message(checkState msg1 [msg2]...)
```

The `checkState` argument is expected to be one of the following values:

`CHECK_START`

Signifies the start of the check. The message should be short, ideally not more than a few words. It will be repeated as part of the pass or fail message at the end of the check.

`CHECK_PASS`

The check completed successfully.

`CHECK_FAIL`

The check completed with a failure.

Start, pass and fail messages are always output with a `STATUS` log level. The meaning of success or failure is up to the project and failure does not necessarily imply an error. For example, the project may want to check for a number of related things and stop upon the first successful one it finds.

Upon completion of the check (pass or fail), the `message()` command will repeat the message from the most recent `CHECK_START` and then "forget" that message. Output from nested checks then work

Exclusively for sbmain17@gmail.com Transaction: 0098315591

intuitively with minimal effort. When combined with appropriate indenting using `CMAKE_MESSAGE_INDENT`, the readability and consistency of the output is particularly good.

```
# Functions just to demonstrate pass/fail behavior
function(checkSomething resultVar)
    set(${resultVar} YES PARENT_SCOPE)
endfunction()

function(checkSomethingElse resultVar)
    set(${resultVar} NO PARENT_SCOPE)
endfunction()

# Outer check starts here
message(CHECK_START "Checking things")
list(APPEND CMAKE_MESSAGE_INDENT "  ")

# Inner check 1
message(CHECK_START "Checking support for something")
checkSomething(successVar1)
if(successVar1)
    message(CHECK_PASS "supported")
else()
    message(CHECK_FAIL "not supported")
endif()

# Inner check 2
message(CHECK_START "Checking support for something else")
checkSomethingElse(successVar2)
if(successVar2)
    message(CHECK_PASS "supported")
else()
    message(CHECK_FAIL "not supported")
endif()

# Outer check finishes here
list(POP_BACK CMAKE_MESSAGE_INDENT)
if(successVar1 OR successVar2)
    message(CHECK_PASS "ok")
else()
    message(CHECK_FAIL "failed")
endif()
```

Output from the above would contain lines like the following:

```
-- Checking things
--   Checking support for something
--   Checking support for something - supported
--   Checking support for something else
--   Checking support for something else - not supported
-- Checking things - ok
```


Exclusively for sbmain17@gmail.com Transaction: 0098315591

13.2. Print Helpers

The `CMakePrintHelpers` module provides two macros which make printing the values of properties and variables more convenient during development. They are not intended for permanent use, but are more aimed at helping developers quickly and easily log information temporarily to help investigate problems in the project.

```
cmake_print_properties(
    [TARGETS target1 [target2...]]
    [SOURCES source1 [source2...]]
    [DIRECTORIES dir1 [dir2...]]
    [TESTS test1 [test2...]]
    [CACHE_ENTRIES var1 [var2...]]
    PROPERTIES property1 [property2...]
)
```

This command essentially combines `get_property()` with `message()` into a single call. Exactly one of the property types must be specified and each of the named properties will be printed for each entity listed. It is particularly convenient when logging information for multiple entities and/or properties. For example:

```
add_executable(MyApp main.c)
add_executable(MyAlias ALIAS MyApp)
add_library(MyLib STATIC src.cpp)

include(CMakePrintHelpers)
cmake_print_properties(TARGETS MyApp MyLib MyAlias
    PROPERTIES TYPE ALIASED_TARGET
)
```

The output of the above would be:

```
Properties for TARGET MyApp:
  MyApp.TYPE = "EXECUTABLE"
  MyApp.ALIASED_TARGET = <NOTFOUND>
Properties for TARGET MyLib:
  MyLib.TYPE = "STATIC_LIBRARY"
  MyLib.ALIASED_TARGET = <NOTFOUND>
Properties for TARGET MyAlias:
  MyAlias.TYPE = "EXECUTABLE"
  MyAlias.ALIASED_TARGET = "MyApp"
```

The module also provides a similar function for logging the value of one or more variables:

```
cmake_print_variables(var1 [var2...])
```

This works for all variables regardless of whether they have been explicitly set by the project, are automatically set by CMake or have not been set at all.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

```
set(foo "My variable")
unset(bar)

include(CMakePrintHelpers)
cmake_print_variables(foo bar CMAKE_VERSION)
```

The output for the above would be something similar to the following:

```
foo="My variable" ; bar="" ; CMAKE_VERSION="3.8.2"
```

13.3. Tracing Variable Access

Another mechanism provided for debugging variable use is the `variable_watch()` command. This is intended for more complex projects where it may not be clear how a variable ended up with a particular value. When a variable is watched, all attempts to read or modify it are logged.

```
variable_watch(myVar [command])
```

For the vast majority of cases, listing the variable to be watched without the optional `command` is sufficient, as it logs all accesses to the nominated variable. For a more customized degree of control, a command can be given which will be executed every time the variable is read or modified. The command is expected to be the name of a CMake function or macro, which will receive the following arguments:

- The name of the variable.
- The type of access.
- The variable's value.
- The name of the current list file.
- The list file stack.

In practice, specifying a command with `variable_watch()` would be very uncommon. The default message is usually enough to help diagnose the situations where `variable_watch()` is typically used. The default message also contains more detail in the call stack than is available in the last argument passed to a custom watcher `command`.

13.4. Debugging Generator Expressions

Generator expressions can quickly become complicated and it can be difficult to confirm their correctness. Because they are only evaluated at generation time, their results are not available during the configure stage and therefore cannot be printed using the `message()` command.

One way to debug the value of a generator expression is to use the `file(GENERATE)` command, which is covered in Section 19.3, "Reading And Writing Files Directly". The generator expression can be written to a temporary file and inspected when CMake finishes execution. For example:

Exclusively for sbmain17@gmail.com Transaction: 0098315591

```
add_executable(someTarget ...)
target_include_directories(someTarget ...)

set(incDirs $<TARGET_PROPERTY:someTarget,INCLUDE_DIRECTORIES>)
set(genex "-I$<JOIN:${incDirs}, -I>")

file(GENERATE OUTPUT genex.txt CONTENT "${genex}\n")
```

Another approach is to create a temporary custom build target whose command prints the value of the generator expression (see Section 18.1, “Custom Targets”). Building that target then prints the result of the expression.

```
add_custom_target(printGenex
    COMMENT "Result of generator expression:"
    COMMAND ${CMAKE_COMMAND} -E echo "${genex}"
    VERBATIM
)
```

Building that target and some representative output might look like this:

```
cmake --build . --target printGenex
[1/1] Result of generator expression:
-I/some/path -I/some/other/path
```

This technique is especially useful for configuration-specific generator expressions and when using multi-config generators like Xcode, Visual Studio and Ninja Multi-Config:

```
set(genex "$<IF:$<CONFIG:Debug>,is debug,not debug>")

add_custom_target(printGenex
    COMMENT "Result of generator expression:"
    COMMAND ${CMAKE_COMMAND} -E echo "${genex}"
    VERBATIM
)
```

For multi-configuration generators, the configuration can be specified with the build command:

```
cmake --build . --target printGenex --config Release
[1/1] Result of generator expression:
not debug
```

```
cmake --build . --target printGenex --config Debug
[1/1] Result of generator expression:
is debug
```

Exclusively for sbmain17@gmail.com Transaction: 0098315591

13.5. Profiling CMake Calls

CMake 3.18 added the ability to profile CMake's own processing of a project. For large, complicated projects where the configure stage takes a long time, this can provide valuable insights into where time is being spent. When profiling is enabled, every CMake command invocation is recorded in the profiling output.

Both of the following command line options need to be provided to enable profiling:

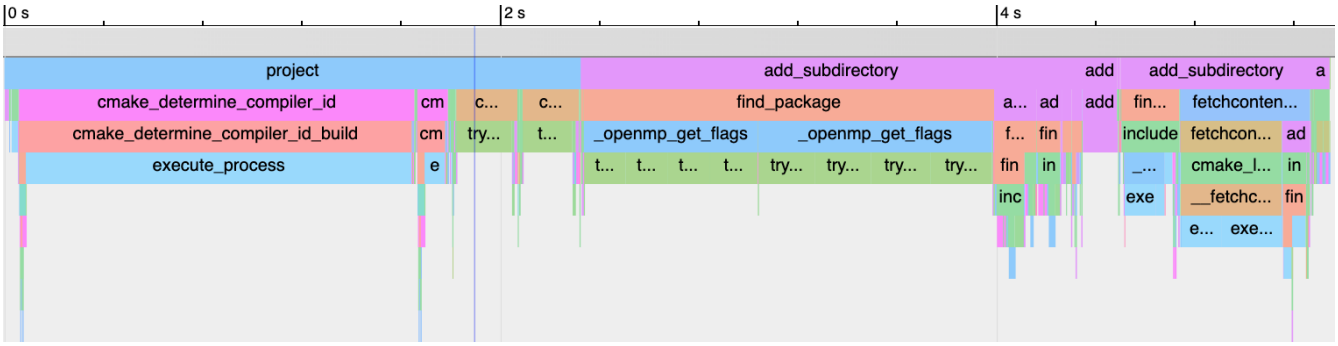
`--profiling-output=fileName`

The profiling data will be written to the specified `fileName`.

`--profiling-format=fmt`

This specifies the format of the profiling data. The only supported value for `fmt` is `google-trace`, but future CMake versions may expand this to include other formats.

For the `google-trace` format, the output file can be loaded directly into a Chrome web browser (navigate to the URL `about:tracing`) or some IDEs (e.g. Qt Creator). Using a `.json` extension for the output file name may make it easier to find and load into tools that understand the `google-trace` format.



The profiling results often show calls like `try_compile()` and `execute_process()` as consuming the majority of the execution time. Rather than focusing on those two calls specifically, inspect the call stacks that led up to those commands. There may be opportunities for reducing how often these two commands are called by avoiding unnecessary or overly pessimistic logic higher in the call stack.

13.6. Recommended Practices

A common problem with many projects is that they log an excessive amount of output during the configure step. This tends to train users to ignore the output, which in turn means that important messages and warnings are easily missed. When the output is fairly minimal and a warning does occur, users tend to take note and investigate the cause. Therefore, aim to minimize the amount of output at the `STATUS` log level, saving more detailed output for a log level of `VERBOSE` or lower. If supporting CMake versions older than 3.15 where log levels below `STATUS` are not available, consider putting detailed logging behind a project-specific cache option, which should be off by default.

For log messages intended to remain as part of the build, aim to always specify a log level as the first argument to the `message()` command. If the message is of a general informational nature,

Exclusively for sbmain17@gmail.com Transaction: 0098315591

prefer to use `STATUS` rather than no keyword at all so that message output does not appear out of order in the build log. Temporary debugging messages frequently omit specifying a log level for convenience, but if they are likely to remain part of the project for any length of time, it is better that they too specify a log level.

For non-trivial projects, consider adding message context information to allow users to filter log output and focus on just those messages that are of interest to them. Never discard existing contents of the `CMAKE_MESSAGE_CONTEXT` variable, always use `list(APPEND)` when starting a new message context. If the message context should end before the end of the current variable scope, use `list(POP_BACK)`. Make no assumptions about what the variable contains other than that this append / pop back pattern can be used. Consider appending the project name as a message context immediately before the first `project()` call in the top level `CMakeLists.txt` file so that compiler feature checks also have a message context.

In a similar manner, also consider using the `CMAKE_MESSAGE_INDENT` variable to provide some logical structure to the message output. Prefer to append two spaces for an indent. While other indents are permitted, following this convention will make the output more consistent, especially in hierarchical projects that make use of external dependencies. Use `list(APPEND)` to add to the existing indent, never replace or discard the existing contents of the `CMAKE_MESSAGE_INDENT` variable. If required, `list(POP_BACK)` can be used to reduce the indent again before the end of the current variable scope.

Both the `CMAKE_MESSAGE_CONTEXT` and `CMAKE_MESSAGE_INDENT` variables can be populated by the project regardless of the minimum supported CMake version. When an earlier CMake version that doesn't know about these variables is used, it will simply ignore them and the output will be unaffected. Therefore, consider using these features even if the project needs to support earlier CMake versions. Note that the `list(POP_BACK)` command requires CMake 3.15 or later, so if the project needs to support versions earlier that that, it must use alternative commands to achieve the same effect where that is needed. In most cases though, a new message context or indentation level will apply through to the end of the current variable scope, in which case popping the last value from the end of the list variable won't be necessary.

Consider using the `CHECK_START`, `CHECK_PASS` and `CHECK_FAIL` form of the `message()` command to record details of checks. This reduces duplication of messages and provides improved readability. It is especially effective when used in conjunction with the indenting support provided by the `CMAKE_MESSAGE_INDENT` variable.

If the configure stage of a project takes a long time to complete, consider running `cmake` with the `--profiling-output` and `--profiling-format` options to investigate where the time is being spent. Available with CMake 3.18 or later, these options enable the generation of command-level profiling information which can be viewed with tools like the Chrome web browser or IDEs like Qt Creator.

Section 24.6, "Debugging find_...() Calls" also discusses further debugging features added in CMake 3.18. Those features relate to finding files, packages and other things, which is covered in detail in Chapter 24, *Finding Things*.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

Part II: Builds In Depth

In the preceding chapters, the most fundamental aspects of CMake were progressively introduced. Core language features, key concepts and important building blocks were presented, providing a solid foundation for a deeper exploration of CMake's functionality.

In this part of the book, the build products become the focus. Chapters cover the toolchain and build configuration, different types of targets, carrying out custom tasks and handling platform-specific features. Understanding these areas well can be the difference between a fragile, complex project and a robust, easy to maintain one.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

Chapter 14. Build Type

This chapter and the next cover two closely related topics. The build type (also known as the build configuration or build scheme in some IDE tools) is a high level control which selects different sets of compiler and linker behavior. Manipulation of the build type is the subject of this chapter, while the next chapter presents more specific details of controlling compiler and linker options. Together, these chapters cover material every CMake developer will typically use for all but the most trivial projects.

14.1. Build Type Basics

The build type has the potential to affect almost everything about the build in one way or another. While it primarily has a direct effect on the compiler and linker behavior, it also has an effect on the directory structure used for a project. This can in turn influence how a developer sets up their own local development environment, so the effects of the build type can be quite far-reaching.

Developers commonly think of builds as being one of two arrangements: debug or release. For a debug build, compiler flags are used to enable the recording of information that debuggers can use to associate machine instructions with the source code. Optimizations are frequently disabled in such builds so that the mapping from machine instruction to source code location is direct and easy to follow when stepping through program execution. A release build, on the other hand, generally has full optimizations enabled and no debug information generated.

These are examples of what CMake refers to as the *build type*. While projects are able to define whatever build types they want, the default build types provided by CMake are usually sufficient for most projects:

Debug

With no optimizations and full debug information, this is commonly used during development and debugging, as it typically gives the fastest build times and the best interactive debugging experience.

Release

This build type typically provides full optimizations for speed and no debug information, although some platforms may still generate debug symbols in certain circumstances. It is generally the build type used when building software for final production releases.

RelWithDebInfo

This is somewhat of a compromise of the previous two. It aims to give performance close to a `Release` build, but still allow some level of debugging. Most optimizations for speed are typically applied, but most debug functionality is also enabled. This build type is therefore most useful when the performance of a `Debug` build is not acceptable even for a debugging session. Note that the default settings for `RelWithDebInfo` will disable assertions.

MinSizeRel

This build type is typically only used for constrained resource environments such as embedded devices. The code is optimized for size rather than speed and no debug information is created.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

Each build type results in a different set of compiler and linker flags. It may also change other behaviors, such as altering which source files get compiled or what libraries to link to. These details are covered in the next few sections, but before launching into those discussions, it is essential to understand how to select the build type and how to avoid some common problems.

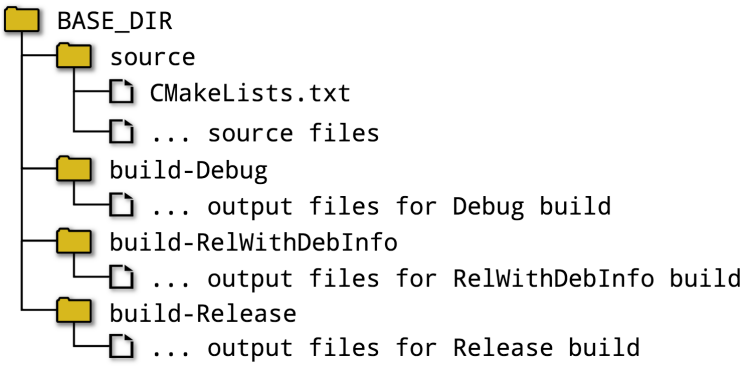
14.1.1. Single Configuration Generators

Back in Section 2.3, "Generating Project Files", the different types of project generators were introduced. Some, like Makefiles and Ninja, support only a single build type per build directory. For these generators, the build type is chosen by setting the `CMAKE_BUILD_TYPE` cache variable. For example, to configure and then build a project with Ninja, one might use commands like this:

```
cmake -G Ninja -DCMAKE_BUILD_TYPE:STRING=Debug ../source
cmake --build .
```

The `CMAKE_BUILD_TYPE` cache variable can also be changed in the CMake GUI application instead of from the command line, but the end effect is the same. With CMake 3.22 or later, if the `CMAKE_BUILD_TYPE` cache variable is not set, it will be initialized from the `CMAKE_BUILD_TYPE` environment variable (if defined).

Rather than switching between different build types in the same build directory, an alternative strategy is to set up separate build directories for each build type, all still using the same sources. Such a directory structure might look something like this:



If frequently switching between build types, this arrangement avoids having to constantly recompile the same sources just because compiler flags change. It also allows a single configuration generator to effectively act like a multi configuration generator. IDE environments like Qt Creator support switching between build directories just as easily as Xcode or Visual Studio allow switching between build schemes or configurations.

14.1.2. Multiple Configuration Generators

Some generators, notably Xcode and Visual Studio, support multiple configurations in a single build directory. From CMake 3.17, the Ninja Multi-Config generator is also available. These multi-config generators ignore the `CMAKE_BUILD_TYPE` cache variable and instead require the developer to choose the build type within the IDE or with a command line option at build time. Configuring and building such projects would typically look something like this:

Exclusively for sbmain17@gmail.com Transaction: 0098315591

```
cmake -G Xcode ../source
cmake --build . --config Debug
```

When building within the Xcode IDE, the build type is controlled by the build scheme, while within the Visual Studio IDE, the current solution configuration controls the build type. Both environments keep separate directories for the different build types, so switching between builds doesn't cause constant rebuilds. In effect, the same thing is being done as the multiple build directory arrangement described above for single configuration generators, it's just that the IDE is handling the directory structure on the developer's behalf.

For command-line builds, the Ninja Multi-Config generator has a little more flexibility compared to the other multi-config generators. The `CMAKE_DEFAULT_BUILD_TYPE` cache variable can be used to change the default configuration to use when no configuration is specified on the build command line. The Xcode and Visual Studio generators have their own fixed logic for determining the default configuration in this scenario. The Ninja Multi-Config generator also supports advanced features that allow custom commands to execute as one configuration, but other targets to be built with one or more other configurations. Most projects would not typically need or benefit from these more advanced features, but the CMake documentation for the Ninja Multi-Config generator provides the essential details, with examples.

14.2. Common Errors

Note how for single configuration generators, the build type is specified at *configure* time, whereas for multi configuration generators, the build type is specified at *build* time. This distinction is critical, as it means the build type is not always known when CMake is processing a project's `CMakeLists.txt` file. Consider the following piece of CMake code, which unfortunately is rather common, but demonstrates an incorrect pattern:

```
# WARNING: Do not do this!
if(CMAKE_BUILD_TYPE STREQUAL "Debug")
    # Do something only for debug builds
endif()
```

The above would work fine for Makefile-based generators and Ninja, but not for Xcode, Visual Studio or Ninja Multi-Config. In practice, just about any logic based on `CMAKE_BUILD_TYPE` within a project is questionable unless it is protected by a check to confirm a single configuration generator is being used. For multi configuration generators, this variable is likely to be empty, but even if it isn't, its value should be considered unreliable because the build will ignore it. Rather than referring to `CMAKE_BUILD_TYPE` in the `CMakeLists.txt` file, projects should instead use other more robust alternative techniques, such as generator expressions based on `$<CONFIG:…>`.

When scripting builds, a common deficiency is to assume a particular generator is used or to not properly account for differences between single and multi configuration generators. Developers should ideally be able to change the generator in one place and the rest of the script should still function correctly. Conveniently, single configuration generators will ignore any build-time specification and multi configuration generators will ignore the `CMAKE_BUILD_TYPE` variable, so by specifying both, a script can account for both cases. For example:

Exclusively for sbmain17@gmail.com Transaction: 0098315591

```
mkdir build
cd build
cmake -G Ninja -DCMAKE_BUILD_TYPE=Release ../source
cmake --build . --config Release
```

With the above example, a developer could simply change the generator name given to the `-G` parameter and the rest of the script would work unchanged.

Not explicitly setting the `CMAKE_BUILD_TYPE` for single configuration generators is also common, but usually not what the developer intended. A behavior unique to single configuration generators is that if `CMAKE_BUILD_TYPE` is not set, the build type will usually be empty. This can lead to the misunderstanding that an empty build type is equivalent to Debug, but this is not so. An empty build type is its own unique, nameless build type. In such cases, no configuration-specific compiler or linker flags are used, which often results in invoking the compiler and linker with minimal flags. The behavior is then determined by the compiler's and linker's own defaults. While this may often be similar to the Debug build type's behavior, it is by no means guaranteed.

Using the Visual Studio compilers with a single configuration generator is somewhat of a special case. For that toolchain, there are different runtime libraries for debug and non-debug builds. An empty build type would make it unclear which runtime should be used. To avoid this ambiguity, the build type will default to Debug for this combination.

14.3. Custom Build Types

Sometimes a project may want to limit the set of build types to a subset of the defaults, or it may want to add other custom build types with a special set of compiler and linker flags. A good example of the latter is adding a build type for profiling or code coverage, both of which require specific compiler and linker settings.

There are two main places where a developer may see the set of build types. When using IDE environments for multi configuration generators like Xcode and Visual Studio, the IDE provides a drop-down list or similar from which the developer selects the configuration they wish to build. For single configuration generators like Makefiles or Ninja, the build type is entered directly for the `CMAKE_BUILD_TYPE` cache variable, but the CMake GUI application can be made to present a combo box of valid choices instead of a simple text edit field. The mechanisms behind these two cases are different, so they must be handled separately.

The set of build types known to multi configuration generators is controlled by the `CMAKE_CONFIGURATION_TYPES` cache variable, or more accurately, by the value of this variable at the end of processing the top level `CMakeLists.txt` file. The first encountered `project()` command populates the cache variable if it has not already been defined. With CMake 3.22 or later, a `CMAKE_CONFIGURATION_TYPES` environment variable can provide the default value. If that environment variable isn't set or an earlier CMake version is used, the default value will be (possibly a subset of) the four standard configurations mentioned in Section 14.1, "Build Type Basics" (`Debug`, `Release`, `RelWithDebInfo` and `MinSizeRel`).

Projects may modify the `CMAKE_CONFIGURATION_TYPES` variable after the first `project()` command. Custom build types can be defined by adding them to `CMAKE_CONFIGURATION_TYPES` and unwanted build

Exclusively for sbmain17@gmail.com Transaction: 0098315591

types can be removed from that list. Note that only the non-cache variable should be modified, as changing the cache variable may discard changes made by the developer.

Care needs to be taken to avoid setting CMAKE_CONFIGURATION_TYPES if it is not already defined. Prior to CMake 3.9, a very common approach for determining whether a multi configuration generator was being used was to check if CMAKE_CONFIGURATION_TYPES was non-empty. Even parts of CMake itself did this prior to 3.11. While this method is usually accurate, it is not unusual to see projects unilaterally set CMAKE_CONFIGURATION_TYPES even if using a single configuration generator. This can lead to wrong decisions being made regarding the type of generator in use. To address this, CMake 3.9 added a new GENERATOR_IS_MULTI_CONFIG global property which is set to true when a multi configuration generator is being used, providing a definitive way to obtain that information instead of relying on inferring it from CMAKE_CONFIGURATION_TYPES. Even so, checking CMAKE_CONFIGURATION_TYPES is still such a prevalent pattern that projects should continue to only modify it if it exists and never create it themselves. It should also be noted that prior to CMake 3.11, adding custom build types to CMAKE_CONFIGURATION_TYPES was not safe. Certain parts of CMake only accounted for the default build types, but even so, projects may still be able to usefully define custom build types with earlier CMake versions, depending on how they are going to be used. That said, for better robustness, it is recommended that at least CMake 3.11 be used if custom build types are going to be defined.

Another aspect of this issue is that developers may add their own types to the CMAKE_CONFIGURATION_TYPES cache variable and/or remove those they are not interested in. Projects should therefore not make any assumptions about what configuration types are or are not defined.

Taking the above points into account, the following pattern shows the preferred way for projects to add their own custom build types for multi configuration generators:

```
cmake_minimum_required(3.11)
project(Foo)

get_property(isMultiConfig GLOBAL
    PROPERTY GENERATOR_IS_MULTI_CONFIG
)
if(isMultiConfig)
    if(NOT "Profile" IN_LIST CMAKE_CONFIGURATION_TYPES)
        list(APPEND CMAKE_CONFIGURATION_TYPES Profile)
    endif()
endif()
# Set relevant Profile-specific flag variables as needed...
```

For single configuration generators, there is only one build type and it is specified by the CMAKE_BUILD_TYPE cache variable, which is a string. In the CMake GUI, this is normally presented as a text edit field, so the developer can edit it to contain whatever arbitrary content they wish. As discussed back in Section 9.6, “Cache Variable Properties”, however, cache variables can have their STRINGS property defined to hold a set of valid values. The CMake GUI application will then present that variable as a combo box containing the valid values instead of as a text edit field.

```
set_property(CACHE CMAKE_BUILD_TYPE PROPERTY
    STRINGS Debug Release Profile
)
```

Exclusively for sbmain17@gmail.com Transaction: 0098315591

Properties can only be changed from within the project's `CMakeLists.txt` files, so they can safely set the `STRINGS` property without having to worry about preserving any developer changes. Note, however, that setting the `STRINGS` property of a cache variable does not guarantee that the cache variable will hold one of the defined values, it only controls how the variable is presented in the CMake GUI application. Developers can still set `CMAKE_BUILD_TYPE` to any value at the `cmake` command line or edit the `CMakeCache.txt` file manually. In order to rigorously require the variable to have one of the defined values, a project must explicitly perform that test itself.

```
set(allowedBuildTypes Debug Release Profile)

# WARNING: This logic is not sufficient
if(NOT CMAKE_BUILD_TYPE IN_LIST allowedBuildTypes)
    message(FATAL_ERROR "${CMAKE_BUILD_TYPE} is not a known build type")
endif()
```

The default value for `CMAKE_BUILD_TYPE` is an empty string, so the above would cause a fatal error for both single and multi configuration generators unless the developer explicitly set it. This is undesirable, especially for multi configuration generators which don't even use the `CMAKE_BUILD_TYPE` variable's value. This can be handled by having the project provide a default value if `CMAKE_BUILD_TYPE` hasn't been set. Furthermore, the techniques for multi and single configuration generators can and should be combined to give robust behavior across all generator types. The end result would look something like this:

```
cmake_minimum_required(3.11)
project(Foo)

get_property(isMultiConfig GLOBAL
    PROPERTY GENERATOR_IS_MULTI_CONFIG
)
if(isMultiConfig)
    if(NOT "Profile" IN_LIST CMAKE_CONFIGURATION_TYPES)
        list(APPEND CMAKE_CONFIGURATION_TYPES Profile)
    endif()
else()
    set(allowedBuildTypes Debug Release Profile)
    set_property(CACHE CMAKE_BUILD_TYPE PROPERTY
        STRINGS "${allowedBuildTypes}"
    )
    if(NOT CMAKE_BUILD_TYPE)
        set(CMAKE_BUILD_TYPE Debug CACHE STRING "" FORCE)
    elseif(NOT CMAKE_BUILD_TYPE IN_LIST allowedBuildTypes)
        message(FATAL_ERROR "Unknown build type: ${CMAKE_BUILD_TYPE}")
    endif()
endif()

# Set relevant Profile-specific flag variables as needed...
```

All the techniques discussed above merely allow a custom build type to be selected, they don't define anything about that build type. Fundamentally, when a build type is selected, it specifies which configuration-specific variables CMake should use. It also affects any generator expressions

Exclusively for sbmain17@gmail.com Transaction: 0098315591

whose logic depends on the current configuration (i.e. $<CONFIG> and $<CONFIG:⋯>). These variables and generator expressions are discussed in detail in the next chapter, but for now, the following two families of variables are of primary interest:

- CMAKE_<LANG>_FLAGS_<CONFIG>
- CMAKE_<TARGETTYPE>_LINKER_FLAGS_<CONFIG>

These can be used to add additional compiler and linker flags over and above the default set provided by the same-named variables without the _<CONFIG> suffix. For example, flags for a custom Profile build type could be defined as follows:

```
set(CMAKE_C_FLAGS_PROFILE             "-p -g -O2" CACHE STRING "")
set(CMAKE_CXX_FLAGS_PROFILE           "-p -g -O2" CACHE STRING "")
set(CMAKE_EXE_LINKER_FLAGS_PROFILE    "-p -g -O2" CACHE STRING "")
set(CMAKE_SHARED_LINKER_FLAGS_PROFILE "-p -g -O2" CACHE STRING "")
set(CMAKE_STATIC_LINKER_FLAGS_PROFILE ""          CACHE STRING "")
set(CMAKE_MODULE_LINKER_FLAGS_PROFILE "-p -g -O2" CACHE STRING "")
```

The above assumes a GCC-compatible compiler to keep the example simple and turns on profiling as well as enabling debugging symbols and most optimizations. An alternative is to base the compiler and linker flags on one of the other build types and add the extra flags needed. This can be done as long as it comes after the project() command, since that command populates the default compiler and linker flag variables. For profiling, the RelWithDebInfo default build type is a good one to choose as the base configuration since it enables both debugging and most optimizations:

```
set(CMAKE_C_FLAGS_PROFILE
    "${CMAKE_C_FLAGS_RELWITHDEBINFO} -p"
    CACHE STRING ""
)
set(CMAKE_CXX_FLAGS_PROFILE
    "${CMAKE_CXX_FLAGS_RELWITHDEBINFO} -p"
    CACHE STRING ""
)
set(CMAKE_EXE_LINKER_FLAGS_PROFILE
    "${CMAKE_EXE_LINKER_FLAGS_RELWITHDEBINFO} -p"
    CACHE STRING ""
)
set(CMAKE_SHARED_LINKER_FLAGS_PROFILE
    "${CMAKE_SHARED_LINKER_FLAGS_RELWITHDEBINFO} -p"
    CACHE STRING ""
)
set(CMAKE_STATIC_LINKER_FLAGS_PROFILE
    "${CMAKE_STATIC_LINKER_FLAGS_RELWITHDEBINFO}"
    CACHE STRING ""
)
set(CMAKE_MODULE_LINKER_FLAGS_PROFILE
    "${CMAKE_MODULE_LINKER_FLAGS_RELWITHDEBINFO} -p"
    CACHE STRING ""
)
```

Exclusively for sbmain17@gmail.com Transaction: 0098315591

Each custom configuration should have the associated compiler and linker flag variables defined. For some multi configuration generator types, CMake will check that the required variables exist and will fail with an error if they are not set.

Another variable which may sometimes be defined for a custom build type is `CMAKE_<CONFIG>_POSTFIX`. It is used to initialize the `<CONFIG>_POSTFIX` property of each library target, with its value being appended to the file name of such targets when built for the specified configuration. This allows libraries from multiple build types to be put in the same directory without overwriting each other. `CMAKE_DEBUG_POSTFIX` is often set to values like `d` or `_debug`, especially for Visual Studio builds where different runtime DLLs must be used for Debug and non-Debug builds, so packages may need to include libraries for both build types. In the case of the custom `Profile` build type defined above, an example might be:

```
set(CMAKE_PROFILE_POSTFIX _profile)
```

If creating packages that contain multiple build types, setting `CMAKE_<CONFIG>_POSTFIX` for each build type is highly recommended. By convention, the postfix for Release builds is typically empty. Note though that the `<CONFIG>_POSTFIX` target property is ignored on Apple platforms.

For historical reasons, the items passed to the `target_link_libraries()` command can be prefixed with the `debug` or `optimized` keywords to indicate that the named item should only be linked in for debug or non-debug builds respectively. A build type is considered to be a `debug` build if it is listed in the `DEBUG_CONFIGURATIONS` global property, otherwise it is considered to be `optimized`. For custom build types, they should have their name added to this global property if they should be treated as a `debug` build in this scenario. As an example, if a project defines its own custom build type called `StrictChecker` and that build type should be considered a non-optimized debug build type, it can (and should) make this clear like so:

```
set_property(GLOBAL APPEND PROPERTY
    DEBUG_CONFIGURATIONS StrictChecker
)
```

New projects should normally prefer to use generator expressions instead of the `debug` and `optimized` keywords with the `target_link_libraries()` command. The next chapter discusses this area in more detail.

14.4. Recommended Practices

Developers should not assume a particular CMake generator is being used to build their project. Another developer on the same project may prefer to use a different generator because it integrates better with their IDE tool, or a future version of CMake may add support for a new generator type which might bring other benefits. Certain build tools may contain bugs which a project may later be affected by, so it can be useful to have alternative generators to fall back on until such bugs are fixed. Expanding a project's set of supported platforms can also be hindered if a particular CMake generator has been assumed.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

When using single configuration generators like Makefiles or Ninja, consider using multiple build directories, one for each build type of interest. This allows switching between build types without forcing a complete recompile each time. This provides similar behavior to that inherently offered by multi configuration generators and can be a useful way to enable IDE tools like Qt Creator to simulate multi configuration functionality.

For single configuration generators, consider setting `CMAKE_BUILD_TYPE` to a better default value if it is empty. While an empty build type is technically valid, it is also often misunderstood by developers to mean a Debug build rather than its own distinct build type. Furthermore, avoid creating logic based on `CMAKE_BUILD_TYPE` unless it is first confirmed that a single configuration generator is being used. Even then, such logic is likely to be fragile and could probably be expressed with more generality and robustness using generator expressions instead.

Only consider modifying the `CMAKE_CONFIGURATION_TYPES` variable if it is known that a multi configuration generator is being used or if the variable already exists. If adding a custom build type or removing one of the default build types, do not modify the cache variable but instead change the regular variable of the same name (it will take precedence over the cache variable). Also prefer to add and remove individual items rather than completely replacing the list. Both of these measures will help avoid interfering with changes made to the cache variable by the developer.

If requiring CMake 3.9 or later, use the `GENERATOR_IS_MULTI_CONFIG` global property to definitively query the generator type instead of relying on the existence of `CMAKE_CONFIGURATION_TYPES` to perform a less robust check.

A common but incorrect practice is to query the `LOCATION` target property to work out a target's output file name. A related error is to assume a particular build output directory structure in custom commands (see Chapter 18, *Custom Tasks*). These methods do not work for all build types, since `LOCATION` is not known at configure time for multi configuration generators and the build output directory structure is typically different across the various CMake generator types. Generator expressions like `$<TARGET_FILE:…>` should be used instead, as they robustly provide the required path for all generators, whether they be single or multi configuration.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

Chapter 15. Compiler And Linker Essentials

The previous chapter discussed the build type and how it relates to selecting a particular set of compiler and linker behavior. This chapter discusses the fundamentals of how that compiler and linker behavior is controlled. The material presented here covers some of the most important topics and techniques with which every CMake developer should become familiar.

Before proceeding, note that as CMake has evolved, the available methods for controlling the compiler and linker behavior have also improved. The focus has shifted from a more build-global view to one where the requirements of each individual target can be controlled, along with how those requirements should or should not be applied to other targets that depend on it. This is an important shift in thinking, as it affects how a project can most effectively define the way targets should be built. CMake's more mature features can be used to control behavior at a coarse level at the expense of losing the ability to define relationships between targets. The more recent target-focused features should be preferred instead, since they greatly improve the robustness of the build and offer much more precise control over compiler and linker behavior. The newer features also tend to be more consistent in their behavior and the way they are meant to be used.

15.1. Target Properties

Within CMake's property system, the target properties form the primary mechanism by which compiler and linker flags are controlled. Some properties provide the ability to specify any arbitrary flag, whereas others focus on a specific capability so they can abstract away platform or compiler differences. This chapter focuses on the more commonly used and general purpose properties, with later chapters covering a number of the more specific ones.

Before proceeding, it should be noted that the target properties discussed in the following sections are not usually modified directly. CMake provides dedicated commands which are generally more convenient and more robust than direct property manipulation. Nevertheless, understanding the underlying properties involved can help developers understand some of the features and restrictions of those commands.

15.1.1. Compiler Flags

The most fundamental target properties for controlling compiler flags are the following, each of which hold a list of items:

`INCLUDE_DIRECTORIES`

This is a list of directories to be used as header search paths, all of which must be absolute paths. CMake will add a compiler flag for each path with an appropriate prefix prepended (typically `-I` or `/I`). When a target is created, the initial value of this target property is taken from the directory property of the same name.

`COMPILE_DEFINITIONS`

This holds a list of definitions to be set on the compile command line. A definition has the form `VAR` or `VAR=VALUE`, which CMake will convert to the appropriate form for the compiler being used (typically `-DVAR⋯` or `/DVAR⋯`). When a target is created, the initial value of this target property

Exclusively for sbmain17@gmail.com Transaction: 0098315591

will be empty. There is a directory property of the same name, but it is *not* used to provide an initial value for this target property. Rather, the directory and target properties are *combined* in the final compiler command line.

`COMPILE_OPTIONS`

Any compiler flags that are neither header search paths nor symbol definitions are provided in this property. When a target is created, the initial value of this target property is taken from the directory property of the same name. Note that this property is also subject to de-duplication (see Section 15.3, "De-duplicating Options" for further details).

> An older and now deprecated target property with the name `COMPILE_FLAGS` used to serve a similar purpose as `COMPILE_OPTIONS`. The `COMPILE_FLAGS` property is treated as a single string that is included directly on the compiler command line. As a result, it may require manual escaping, whereas `COMPILE_OPTIONS` is a list and CMake performs any required escaping or quoting automatically.

The `INCLUDE_DIRECTORIES` and `COMPILE_DEFINITIONS` properties are really just conveniences, taking care of the compiler specific flags for the most common things projects often want to set. All remaining compiler specific flags are then provided in the `COMPILE_OPTIONS` property.

Each of the three target properties above has a related target property with the same name, but with `INTERFACE_` prepended. These interface properties do exactly the same thing, except instead of applying to the target itself, they apply to any other target which links directly to it. In other words, they are used to specify compiler flags which *consuming* targets should inherit. For this reason, they are often referred to as *usage requirements*, in contrast to the non-`INTERFACE` properties which are sometimes called *build requirements*. Two special library types `IMPORTED` and `INTERFACE` are discussed later in Chapter 17, *Target Types*. These special library types support only the `INTERFACE_`⋯ target properties and not the non-`INTERFACE_`⋯ properties.

Unlike their non-interface counterparts, none of the above `INTERFACE_`⋯ properties are initialized from directory properties. They instead all start out empty, since only the project has knowledge of what header search paths, defines and compiler flags should propagate to consuming targets.

With the exception of `COMPILE_FLAGS`, all of the above target properties support generator expressions. Generator expressions are particularly useful for the `COMPILE_OPTIONS` property, since it enables adding a particular flag only if some condition is met, such as only for one particular compiler.

If compiler flags need to be manipulated at the individual source file level, target properties are not granular enough. For such cases, CMake provides the `COMPILE_DEFINITIONS`, `COMPILE_FLAGS` and `COMPILE_OPTIONS` source file properties (the `COMPILE_OPTIONS` source file property was only added in CMake 3.11). These are each analogous to their same-named target properties except that they apply only to the individual source file on which they are set. Note that their support for generator expressions has lagged behind that of the target properties, with the `COMPILE_DEFINITIONS` source file property gaining generator expression support in CMake 3.8 and the others in 3.11. Furthermore, the Xcode project file format does not support configuration specific source file properties at all, so if targeting Apple platforms, `$<CONFIG>` or `$<CONFIG:⋯>` should not be used in source file properties. Also keep in mind the warnings discussed back in Section 9.5, "Source Properties" regarding implementation details leading to performance issues when source file properties are used.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

15.1.2. Linker Flags

The target properties associated with linker flags have similarities to those for compiler flags, but some were only added in more recent CMake versions. CMake 3.13 in particular added a number of improvements for linker control. Also note that only some of the linker-related properties have an associated interface property and that not all properties support generator expressions.

`LINK_LIBRARIES`

This target property holds a list of all libraries the target should link to directly. It is initially empty when the target is created and it supports generator expressions. An associated interface property `INTERFACE_LINK_LIBRARIES` is supported. Each library listed can be one of the following:

- A path to a library, usually specified as an absolute path.
- Just the library name without a path, usually also without any platform-specific file name prefix (e.g. `lib`) or suffix (e.g. `.a`, `.so`, `.dll`).
- The name of a CMake library target. CMake will convert this to a path to the built library when generating the linker command, including supplying any prefix or suffix to the file name as appropriate for the platform. Because CMake handles all the various platform differences and paths on the project's behalf, using a CMake target name is generally the preferred method.

CMake will use the appropriate linker flags to link each item listed in the `LINK_LIBRARIES` property. In some circumstances, linker flags may also be present in this property, but the other target properties below are generally preferred for holding such options.

`LINK_OPTIONS`

Support for this property was added in CMake 3.13. It holds a list of flags to be passed to the linker for targets that are executables, shared libraries or module libraries. It is ignored for targets being built as a static library. This property is intended for general linker flags, not those flags which specify other libraries to link to. When a target is created, the initial value of this target property is taken from the directory property of the same name. Generator expressions are supported and the property is also subject to de-duplication (see Section 15.3, "De-duplicating Options" for further details).

An associated interface property `INTERFACE_LINK_OPTIONS` is also supported. Note that the contents of this interface property will be applied to consuming targets even if the target on which `INTERFACE_LINK_OPTIONS` is set is a static library. This is because the interface property is specifying linker flags that the *consumer* should use, so the type of the library being consumed is not a factor.

`LINK_FLAGS`

This property serves a similar purpose to `LINK_OPTIONS`, but there are a number of differences. The first key difference is that it holds a single string that will be placed directly on the linker command line rather than a list of linker flags. Another difference is that it does not support generator expressions. Furthermore, there is no associated interface property and it is initialized to an empty value when the target is created. In general, `LINK_OPTIONS` is more robust and offers a broader set of features, so only use `LINK_FLAGS` if CMake versions earlier than 3.13 must be supported.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

STATIC_LIBRARY_OPTIONS

This is the counterpart to LINK_OPTIONS. It only has meaning for targets being built as a static library and will be used for the librarian or archiver tool. It holds a list of options, generator expressions are supported and it is subject to de-duplication (see Section 15.3, "De-duplicating Options"). Like LINK_OPTIONS, support for STATIC_LIBRARY_OPTIONS was only added in CMake 3.13, but note that there is no associated interface property. A target cannot dictate librarian/archiver flags of its consumers, only linker flags (see the comments regarding INTERFACE_LINK_OPTIONS above).

STATIC_LIBRARY_FLAGS

This is the counterpart to LINK_FLAGS and should only be used if CMake versions earlier than 3.13 must be supported. It is a single string rather than a list and it does not support generator expressions. There is no associated interface property.

In some older projects, one may occasionally encounter a target property named LINK_INTERFACE_LIBRARIES, which is an older version of INTERFACE_LINK_LIBRARIES. This older property has been deprecated since CMake 2.8.12, but policy CMP0022 can be used to give the old property precedence if needed. New projects should prefer to use INTERFACE_LINK_LIBRARIES instead.

The LINK_FLAGS and STATIC_LIBRARY_FLAGS properties do not support generator expressions. They do, however, have related configuration-specific properties:

- LINK_FLAGS_<CONFIG>
- STATIC_LIBRARY_FLAGS_<CONFIG>

These flags will be used in addition to the non-configuration-specific flags when the <CONFIG> matches the configuration being built. These should only be used if the project must support CMake versions earlier than 3.13. For 3.13 or later, prefer to use LINK_OPTIONS and STATIC_LIBRARY_OPTIONS and express configuration-specific content using generator expressions.

One of the difficulties of passing flags to the linker is that the linker is usually invoked via the compiler front end, but each compiler has its own syntax for how to pass through linker options. For example, invoking the ld linker via gcc requires specifying linker flags using the form -Wl,⋯, whereas clang expects the form -Xlinker ⋯. With CMake 3.13 or later, this difference can be handled automatically by adding a LINKER: prefix to each linker flag in the LINK_OPTIONS and INTERFACE_LINK_OPTIONS properties. This will result in the linker flag being transformed into the required form for the compiler front end being used. For example:

```
set_target_properties(Foo PROPERTIES
    LINK_OPTIONS LINKER:-stats
)
```

Using the gcc compiler, this would add -Wl,-stats, whereas with clang it would add -Xlinker -stats. See Section 15.3, "De-duplicating Options" for further related discussion of the LINKER: prefix.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

15.1.3. Target Property Commands

As mentioned earlier, the above target properties are not normally manipulated directly. CMake provides dedicated commands for modifying them in a more convenient and robust manner which also encourage clear specification of dependencies and transitive behavior between targets. Back in Section 4.3, "Linking Targets", the `target_link_libraries()` command was presented, along with an explanation of how inter-target dependencies are expressed using `PRIVATE`, `PUBLIC` and `INTERFACE` specifications. That earlier discussion focused on the dependency relationships between targets, but following the above discussion of target properties, the exact effects of those keywords can now be made more precise.

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

`PRIVATE`

Items listed after `PRIVATE` only affect the behavior of `targetName` itself. The items are added to the `LINK_LIBRARIES` target property.

`INTERFACE`

This is the complement to `PRIVATE`, with items following the `INTERFACE` keyword being added to the target's `INTERFACE_LINK_LIBRARIES` property. Any target that links to `targetName` will have these items applied to them as though the items were listed in their own `LINK_LIBRARIES` property.

`PUBLIC`

This is equivalent to combining the effects of `PRIVATE` and `INTERFACE`.

Most of the time, developers will probably find the explanation in Section 4.3, "Linking Targets" more intuitive, but the above more precise description can help explain the behavior in more complex projects where properties may be manipulated in unusual ways. The above description also happens to map very closely to the behavior of the other `target_…()` commands which manipulate compiler and linker flags. In fact, they all follow the same pattern and apply the `PRIVATE`, `PUBLIC` and `INTERFACE` keywords in the same way.

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

The `target_link_options()` command was added in CMake 3.13. Instead of specifying linker options in `target_link_libraries()`, it allows projects to more clearly and accurately communicate that linker options are being added. It also avoids populating the `LINK_LIBRARIES` property with linker flags and instead populates the relevant target properties set aside for such flags. The `target_link_options()` command populates the `LINK_OPTIONS` target property with the `PRIVATE` items and the `INTERFACE_LINK_OPTIONS` target property with the `INTERFACE` items. As one would expect, `PUBLIC`

Exclusively for sbmain17@gmail.com Transaction: 0098315591

items are added to both target properties. Since these properties support generator expressions, so does the `target_link_options()` command.

Normally, each time `target_link_options()` is called, the specified items are appended to the relevant target properties. This makes it easy to add multiple options in a natural, progressive manner. If required, the `BEFORE` keyword can be used to prepend the listed options to existing contents of the target properties instead.

This command can be used even if `targetName` is a static library, but note that `PRIVATE` and `PUBLIC` items will populate `LINK_OPTIONS`, not `STATIC_LIBRARY_OPTIONS`. In order to populate `STATIC_LIBRARY_OPTIONS`, there is currently no alternative other than to modify the target property directly with `set_property()` or `set_target_properties()`. Using `target_link_options()` to add `INTERFACE` items to a static library target is still useful, since the contents of `INTERFACE_LINK_OPTIONS` are applied to the *consuming* target.

Since `target_link_options()` adds items to the `LINK_OPTIONS` and `INTERFACE_LINK_OPTIONS` properties, the command also supports items being prefixed with `LINKER:` to handle the compiler front end differences. The example from the previous section can therefore be better implemented as:

```
target_link_options(Foo PRIVATE LINKER:-stats)
```

A number of commands are available for managing a target's compiler-related properties.

```
target_include_directories(targetName [AFTER|BEFORE] [SYSTEM]
     <PRIVATE|PUBLIC|INTERFACE> dir1 [dir2 ...]
    [<PRIVATE|PUBLIC|INTERFACE> dir3 [dir4 ...]]
    ...
)
```

The `target_include_directories()` command adds header search paths to the `INCLUDE_DIRECTORIES` and `INTERFACE_INCLUDE_DIRECTORIES` target properties. Directories following a `PRIVATE` keyword are added to the `INCLUDE_DIRECTORIES` target property, while directories following an `INTERFACE` keyword are added to the `INTERFACE_INCLUDE_DIRECTORIES` target property. Directories following a `PUBLIC` keyword are added to both. The `BEFORE` keyword has the same effect as for `target_link_options()`, causing the specified directories to be prepended to the relevant properties instead of appending. CMake 3.20 added support for the `AFTER` keyword for symmetry, but it doesn't need to be used since appending is the default behavior already.

If the `SYSTEM` keyword is specified, the compiler will treat the listed directories as system include paths on some platforms. The effects of this can include skipping certain compiler warnings or changing how the file dependencies are handled. It can also affect the order in which header paths are searched for some compilers. Developers are sometimes tempted to use `SYSTEM` to silence warnings coming from headers rather than addressing those warnings directly. If such headers are part of the project, `SYSTEM` is not typically an appropriate option to use. In general, `SYSTEM` is intended for paths outside of the project, but even then it should rarely be needed.

It is also worth noting that paths specified by an *imported* target's `INTERFACE_INCLUDE_DIRECTORIES` property will be treated by consuming targets as though they were `SYSTEM` paths by default. This is

Exclusively for sbmain17@gmail.com Transaction: 0098315591

because imported targets are assumed to be coming from outside the project and therefore their associated headers should be treated in a similar way to other system-provided headers. The project can override this behavior by setting the *consuming* target's `NO_SYSTEM_FROM_IMPORTED` property to true, which will prevent all of the imported targets it consumes from being treated as `SYSTEM`. Imported targets are covered in detail in Chapter 17, *Target Types*.

The `target_include_directories()` command offers another advantage over manipulating the target properties directly. Projects can specify relative directories too, not just absolute directories. Relative paths will be automatically converted to absolute paths where needed (with one exception discussed below), with paths being treated as relative to the current source directory.

Since the `target_include_directories()` command is basically just populating the relevant target properties, all the usual features of those properties apply. In particular, generator expressions can be used, a feature which becomes much more important when installing targets and creating packages. The `$<BUILD_INTERFACE:…>` and `$<INSTALL_INTERFACE:…>` generator expressions allow different paths to be specified for building and installing. For installed targets, relative paths are normally used and they would be interpreted as relative to the base install location rather than the source directory. Section 26.2.1, "Interface Properties" covers this aspect of specifying header search paths in more detail. For building targets, the expansion of the `$<BUILD_INTERFACE:…>` generator expression takes place after the check for relative paths, so such expressions must evaluate to an absolute path or CMake will issue an error.

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

The `target_compile_definitions()` command is quite straightforward, with each item having the form `VAR` or `VAR=VALUE`. `PRIVATE` items populate the `COMPILE_DEFINITIONS` target property, while `INTERFACE` items populate the `INTERFACE_COMPILE_DEFINITIONS` target property. `PUBLIC` items populate both target properties. Generator expressions can be used, but there would usually be no need to handle build and install situations differently.

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

The `target_compile_options()` command is also quite straightforward. Each item is treated as a compiler option, with `PRIVATE` items populating the `COMPILE_OPTIONS` target property and `INTERFACE` items populating the `INTERFACE_COMPILE_OPTIONS` target property. As usual, `PUBLIC` items populate both target properties. For all cases, each item is appended to existing target property values, but the `BEFORE` keyword can be used to prepend instead. Generator expressions are supported in all cases and there would usually be no need to handle build and install situations differently.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

15.2. Directory Properties And Commands

With CMake 3.0 and later, target properties are strongly preferred for specifying compiler and linker flags due to their ability to define how they interact with targets that link to one another. In earlier versions of CMake, target properties were much less prominent and properties were often specified at the directory level instead. These directory properties and the commands typically used to manipulate them lack the consistency shown by their target-based equivalents, which is another reason they should generally be avoided by projects where possible. That said, since many online tutorials and examples still use them, developers should at least be aware of the directory level properties and commands.

```
include_directories([AFTER | BEFORE] [SYSTEM] dir1 [dir2...])
```

Simplistically, the `include_directories()` command adds header search paths to targets created in the current directory scope and below. By default, paths are appended to the existing list of directories, but that default can be changed by setting the `CMAKE_INCLUDE_DIRECTORIES_BEFORE` variable to `ON`. It can also be controlled on a per call basis with the `BEFORE` and `AFTER` options to explicitly direct how the paths for that call should be handled. Projects should be wary about setting `CMAKE_INCLUDE_DIRECTORIES_BEFORE`, as most developers will likely assume that the default behavior of directories being appended will apply. The `SYSTEM` keyword has the same effect as for the `target_include_directories()` command.

The paths provided to `include_directories()` can be relative or absolute. Relative paths are converted to absolute paths automatically and are treated as relative to the current source directory. Paths may also contain generator expressions.

The details of what `include_directories()` actually does is more complex than the simplistic explanation above. Primarily, there are two main effects of calling `include_directories()`:

- The listed paths are added to the `INCLUDE_DIRECTORIES` directory property of the current `CMakeLists.txt` file. This means all targets created in the current directory and below will have the directories added to their `INCLUDE_DIRECTORIES` target property.
- Any target created in the current `CMakeLists.txt` file (or more accurately, the current directory scope) will also have the paths added to their `INCLUDE_DIRECTORIES` target property, even if those targets were created before the call to `include_directories()`. This applies strictly only to the targets created in the current `CMakeLists.txt` file or other files pulled in via `include()`, but not to any targets created in parent or child directory scopes.

It is the second of the above points which tends to surprise many developers. To avoid creating situations which may lead to such confusion, if the `include_directories()` command must be used, prefer to call it early in a `CMakeLists.txt` file before any targets have been created or any subdirectories have been pulled in with `include()` or `add_subdirectory()`.

```
add_definitions(-DSomeSymbol /DFoo=Value ...)
remove_definitions(-DSomeSymbol /DFoo=Value ...)
```

Exclusively for sbmain17@gmail.com Transaction: 0098315591

The `add_definitions()` and `remove_definitions()` commands add and remove entries in the `COMPILE_DEFINITIONS` directory property. Each entry should begin with either `-D` or `/D`, the two most prevalent flag formats used by the vast majority of compilers. This flag prefix is stripped off by CMake before the define is stored in the `COMPILE_DEFINITIONS` directory property, so it doesn't matter which prefix is used, regardless of the compiler or platform on which the project is built.

Just as for `include_directories()`, these two commands affect all targets created in the current `CMakeLists.txt` file, even those created before the `add_definitions()` or `remove_definitions()` call. Targets created in child directory scopes will only be affected if created *after* the call. This is a direct consequence of how the `COMPILE_DEFINITIONS` directory property is used by CMake.

Although not recommended, it is also possible to specify compiler flags other than definitions with these commands. If CMake does not recognize a particular item as looking like a compiler define, that item will instead be added unmodified to the `COMPILE_OPTIONS` directory property. This behavior is present for historical reasons, but new projects should avoid this behavior (see the `add_compile_options()` command a little further below for an alternative).

Since the underlying directory properties support generator expressions, so do these two commands, with some caveats. Generator expressions should only be used for the value part of a definition, not for the name part (i.e. only after the "=" in a `-DVAR=VALUE` item or not at all for a `-DVAR` item). This relates to how CMake parses each item to check if it is a compiler definition or not. Note also that these commands only modify directory properties, they do not affect the `COMPILE_DEFINITIONS` target property.

The `add_definitions()` command has a number of shortcomings. The requirement to prefix each item with `-D` or `/D` to have it treated as a definition is not consistent with other CMake behavior. The fact that omitting the prefix makes the command treat the item as a generic option instead is also counter-intuitive given the command's name. Furthermore, the restriction on generator expressions only being supported for the `VALUE` part of a `KEY=VALUE` definition is also a direct consequence of the prefix requirement. In recognition of this, CMake 3.12 introduced the `add_compile_definitions()` command as a replacement for `add_definitions()`:

```
add_compile_definitions(SomeSymbol Foo=Value ...)
```

The new command handles only compile definitions, it does not require any prefix on each item and generator expressions can be used without the `VALUE`-only restriction. The new command's name and treatment of the definition items is consistent with the analogous `target_compile_definitions()` command. `add_compile_definitions()` still affects all targets created in the same directory scope regardless of whether those targets are created before or after `add_compile_definitions()` is called, as this is a characteristic of the underlying `COMPILE_DEFINITIONS` directory property the command manipulates, not of the command itself.

```
add_compile_options(opt1 [opt2 ...])
```

The `add_compile_options()` command is used to provide arbitrary compiler options. Unlike the `include_directories()`, `add_definitions()`, `remove_definitions()` and `add_compile_definitions()` commands, its behavior is very straightforward and predictable. Each option given to

Exclusively for sbmain17@gmail.com Transaction: 0098315591

add_compile_options() is added to the COMPILE_OPTIONS directory property. Every target subsequently created in the current directory scope and below will then inherit those options in their own COMPILE_OPTIONS target property. Any targets created before the call are not affected. This behavior is much closer to what developers would intuitively expect compared to the other directory property commands. Furthermore, generator expressions are supported by the underlying directory and target properties, so the add_compile_options() command also supports them.

```
link_libraries(item1 [item2 ...]
    [ [debug | optimized | general] item] ...
)
link_directories( [ BEFORE | AFTER ] dir1 [dir2 ...])
```

In early CMake versions, these two commands were the primary way to tell CMake to link libraries into other targets. They affect all targets created in the current directory scope and below after the commands are called, but any existing targets remain unaffected (i.e. similar to the behavior of add_compile_options()). The items specified in the link_libraries() command can be CMake targets, library names, full paths to libraries or even linker flags.

Loosely speaking, an item can be made to apply to just the Debug build type by preceding it with the keyword debug, or to all build types except Debug by preceding it with the keyword optimized. An item can be preceded by the keyword general to indicate that it applies to all build types, but since general is the default anyway, there is little benefit to doing so. All three keywords only affect the single item following it, not all items up to the next keyword. The use of these keywords is strongly discouraged, since generator expressions provide much better control over when an item should be added. To account for custom build types, a build type is considered to be a debug configuration if it is listed in the DEBUG_CONFIGURATIONS global property.

The directories added by link_directories() only have an effect when CMake is given a bare library name to link to. CMake adds the supplied paths to the linker command line and leaves the linker to find such libraries on its own. The directories given should be absolute paths, although relative paths were permitted prior to CMake 3.13 (see policy CMP0081 which controls whether CMake halts with an error if a relative path is encountered). The BEFORE and AFTER keywords were added in CMake 3.13 and have a similar effect as they do for include_directories(), including the default behavior being equivalent to AFTER if neither keyword is present.

For robustness reasons, when using link_libraries(), prefer to provide a full path or the name of a CMake target. No linker search directory is necessary for either of those cases and the exact location of the library will be given to the linker. Furthermore, once a linker search directory has been added by link_directories(), projects have no convenient way to remove that search path if they need to. Adding linker search directories should generally be avoided and is usually not necessary.

CMake 3.13 also introduced the add_link_options() command. It is analogous to the target_link_options() command, acting instead on a directory property rather than on target properties.

```
add_link_options(item1 [item2...])
```

Exclusively for sbmain17@gmail.com Transaction: 0098315591

This command appends items to the `LINK_OPTIONS` directory property, which is used to initialize the same-named target property of all targets subsequently created in the current directory scope and below. As with other directory level commands, `add_link_options()` should generally be avoided in favor of the target level command.

15.3. De-duplicating Options

When CMake constructs the final compiler and linker command lines, it performs a de-duplication step on the command line options. This can greatly reduce the length of command lines, which has benefits for both the implementation and for developers trying to understand the final set of command line options used. In some cases though, de-duplication will be undesirable. For example, an option might need to be repeated with different second arguments, such as passing multiple linker options with Clang:

```
# This won't work as expected
target_link_options(SomeTarget PRIVATE
    -Xlinker -z
    -Xlinker defs
)
```

After de-duplication, the second `-Xlinker` would be removed, resulting in the incorrect set of command line options `-Xlinker -z defs`. A similar case exists for the compiler:

```
# This won't work as expected either
target_compile_options(SomeTarget PRIVATE
    -Xassembler --keep
    -Xassembler --no_esc
)
```

CMake provides the `SHELL:` prefix as a way of preventing groups of options from being split up by de-duplication. It is supported since CMake 3.12 for compiler options and 3.13 for linker options. To force two or more options to be treated as a group that should not be split, they should be prefixed by `SHELL:` and given as a single quoted string with options separated by spaces.

```
target_link_options(SomeTarget PRIVATE
    "SHELL:-Xlinker -z"
    "SHELL:-Xlinker defs"
)
target_compile_options(SomeTarget PRIVATE
    "SHELL:-Xassembler --keep"
    "SHELL:-Xassembler --no_esc"
)
```

The presence of duplicated options isn't always obvious. Sometimes they are generated by CMake as part of translating what the project specifies. Consider the following more generic way of adding the above linker options, which is intended to work for both Clang and gcc:

Exclusively for sbmain17@gmail.com Transaction: 0098315591

```
# This won't work as expected for some compiler front ends
target_link_options(SomeTarget PRIVATE "LINKER:-z,defs")
```

When using Clang, the `LINKER:-z,defs` expression expands to the same linker options as in the above example where `-Xlinker` appears twice. To prevent de-duplication from removing the second `-Xlinker` option in this situation, `LINKER:` and `SHELL:` need to be combined:

```
target_link_options(SomeTarget PRIVATE "LINKER:SHELL:-z defs")
```

The `SHELL:`, `LINKER:` and `LINKER:SHELL:` prefixes are handled at the target property level. This means they can be used with any of the commands that manipulate target properties. `LINK_OPTIONS` and `INTERFACE_LINK_OPTIONS` support all of the prefixes. `COMPILE_OPTIONS`, `INTERFACE_COMPILE_OPTIONS` and `STATIC_LIBRARY_OPTIONS` only support `SHELL:`. Since all of these target properties are initialized from directory properties of the same names, those directory properties can also use the prefixes.

15.4. Compiler And Linker Variables

Properties are the main way that projects should seek to influence compiler and linker flags. End users cannot manipulate properties directly, so the project is in full control of how the properties are set. There are situations, however, where the user will want to add their own compiler or linker flags. They may wish to add more warning options, turn on special compiler features such as sanitizers or debugging switches, and so on. For these situations, variables are more appropriate.

CMake provides a set of variables that specify compiler and linker flags to be merged with those provided by the various directory, target and source file properties. They are normally cache variables to allow the user to easily view and modify them, but they can also be set as regular CMake variables within the project's `CMakeLists.txt` files (something projects should aim to avoid). CMake gives the cache variables suitable default values the first time it runs in a build directory.

The primary variables directly affecting the compiler flags have the following form:

- `CMAKE_<LANG>_FLAGS`
- `CMAKE_<LANG>_FLAGS_<CONFIG>`

`<LANG>` corresponds to the language being compiled, with typical values being `C`, `CXX`, `Fortran`, `Swift` and so on. The `<CONFIG>` part is an uppercase string corresponding to one of the build types, such as `DEBUG`, `RELEASE`, `RELWITHDEBINFO` or `MINSIZEREL`. The first variable will be applied to all build types, including single configuration generators with an empty `CMAKE_BUILD_TYPE`. The second variable is only applied to the build type specified by `<CONFIG>`. Thus, a C++ file being built with a Debug configuration would have compiler flags from both `CMAKE_CXX_FLAGS` and `CMAKE_CXX_FLAGS_DEBUG`.

The first `project()` command encountered will create cache variables for these if they don't already exist (this is a bit of a simplification, a more complete explanation is given in Chapter 22, *Toolchains And Cross Compiling*). Therefore, after the first time CMake has been run, their values are easy to check in the CMake GUI application. As an example, for one particular compiler, the following variables for the C++ language are defined by default:

Exclusively for sbmain17@gmail.com Transaction: 0098315591

CMAKE_CXX_FLAGS	
CMAKE_CXX_FLAGS_DEBUG	-g -O0
CMAKE_CXX_FLAGS_RELEASE	-O3 -DNDEBUG
CMAKE_CXX_FLAGS_RELWITHDEBINFO	-O2 -g -DNDEBUG
CMAKE_CXX_FLAGS_MINSIZEREL	-Os -DNDEBUG

The handling of linker flags is similar. They are controlled by the following family of variables:

- CMAKE_<TARGETTYPE>_LINKER_FLAGS
- CMAKE_<TARGETTYPE>_LINKER_FLAGS_<CONFIG>

These are specific to a particular type of target, each of which was introduced back in Chapter 4, *Building Simple Targets*. The <TARGETTYPE> part of the variable name must be one of the following:

EXE

Targets created with add_executable().

SHARED

Targets created with add_library(name SHARED …) or equivalent, such as omitting the SHARED keyword but with the BUILD_SHARED_LIBS variable set to true.

STATIC

Targets created with add_library(name STATIC …) or equivalent, such as omitting the STATIC keyword but with the BUILD_SHARED_LIBS variable set to false or not defined.

MODULE

Targets created with add_library(name MODULE …).

Just like for the compiler flags, the CMAKE_<TARGETTYPE>_LINKER_FLAGS are used when linking any build configuration, whereas the CMAKE_<TARGETTYPE>_LINKER_FLAGS_<CONFIG> flags are only added for the corresponding CONFIG. It is not unusual for some or all of the linker flags to be empty strings on some platforms.

CMake tutorials and example code frequently use the above variables to control the compiler and linker flags. This was fairly common practice in the pre CMake 3.0 era, but with the focus shifting to a target-centric model with CMake 3.0 and later, such examples are no longer a good model to follow. They often lead to a number of very common mistakes, with some of the more prevalent ones presented below.

15.4.1. Compiler And Linker Variables Are Single Strings, Not Lists

If multiple compiler flags need to be set, they need to be specified as a single string, not as a list. CMake will not properly handle flag variables if their contents contain semicolons, which is what a list would be turned into if specified by the project.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

```
# Wrong, list used instead of a string
set(CMAKE_CXX_FLAGS -Wall -Werror)

# Correct, but see later sections for why appending would be preferred
set(CMAKE_CXX_FLAGS "-Wall -Werror")

# Appending to existing flags the correct way (two methods)
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -Wall -Werror")
string(APPEND CMAKE_CXX_FLAGS " -Wall -Werror")
```

15.4.2. Distinguish Between Cache And Non-cache Variables

All of the variables mentioned above are *cache* variables. Non-cache variables of the same name can be defined and they will override the cache variables for the current directory scope and its children (i.e. those created by `add_subdirectory()`). Problems can arise, however, when a project tries to force updating the cache variable instead of a local variable. Code like the following tends to make projects harder to work with and can lead to developers feeling like they are fighting the project when they want to change flags for their own build through the CMake GUI application or similar:

```
# Case 1: Only has an effect if the variable isn't already in the cache
set(CMAKE_CXX_FLAGS "-Wall -Werror" CACHE STRING "C++ flags")

# Case 2: Using FORCE to always update the cache variable, but this overwrites
#         any changes a developer might make to the cache
set(CMAKE_CXX_FLAGS "-Wall -Werror" CACHE STRING "C++ flags" FORCE)

# Case 3: FORCE + append = recipe for disaster (see discussion below)
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -Wall -Werror" CACHE STRING "C++ flags" FORCE)
```

The first case above highlights a common oversight made by developers new to CMake. Without the `FORCE` keyword, the `set()` command only updates a cache variable if it is not already defined. The first run of CMake may therefore appear to do what the developer intended (if placed before any `project()` command), but if the line is ever changed to specify something else for the flags, that change won't be applied to an existing build because the variable will already be in the cache at that point.

The usual reaction to discovering this is to then use `FORCE` to ensure the cache variable is always updated, as shown in the second case, but this then creates another problem. The cache is a primary means for developers to change variables locally without having to edit project files. If a project uses `FORCE` to unilaterally set cache variables in this manner, any change made by the developer to that cache variable will be lost.

The third case is even more problematic because every time CMake is run, the flags will be appended again, leading to an ever growing and repeating set of flags. Using `FORCE` to update the cache like this for compiler and linker flags is rarely a good idea.

Rather than simply removing the `FORCE` keyword, the correct behavior is to set a non-cache variable rather than the cache variable. It is then safe to append flags to the current value because the cache

Exclusively for sbmain17@gmail.com Transaction: 0098315591

variable is left untouched. Every CMake run will start with the same set of flags from the cache variable, regardless of how often CMake is invoked. Any changes the developer chooses to make to the cache variable will also be preserved.

```
# Preserves the cache variable contents, appends new flags safely
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -Wall -Werror")
```

15.4.3. Prefer Appending Over Replacing Flags

As touched on above, developers are sometimes tempted to unilaterally set compiler flags in their `CMakeLists.txt` files like so:

```
# Not ideal, discards any developer settings from cache
set(CMAKE_CXX_FLAGS "-Wall -Werror")
```

Because this discards any value set by the cache variable, developers lose their ability to easily inject their own flags. This forces developers to go hunting through the project to find and modify the offending lines. For a complex project with many subdirectories, this can be tedious. Where possible, projects should prefer to append flags to the existing value.

One reasonable exception to this guideline may be if a project is required to enforce a mandated set of compiler or linker flags. In such cases, a workable compromise may be to set the variable values in the top level `CMakeLists.txt` file as early as possible, ideally at the very top just after the `cmake_minimum_required()` command (or even better, in the toolchain file if one is being used - see Chapter 22, *Toolchains And Cross Compiling* for further details). Keep in mind though that over time, the project may itself become a child of another project, at which point it would no longer be the top level of the build and the suitability of this compromise may be reduced.

15.4.4. Understand When Variable Values Are Used

One of the more obscure aspects of the compiler and linker flag variables is the point in the build process at which their value actually gets used. One might reasonably expect the following code to behave as noted in the inline comments:

```
# Save the original set of flags so we can restore them later
set(oldCxxFlags "${CMAKE_CXX_FLAGS}")

# This library has stringent build requirements, so
# enforce them just for it alone.
# WARNING: This doesn't do what it may appear to do!
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -Wall -Werror")
add_library(StrictReq STATIC ...)

# Less strict requirements from here, so restore the
# original set of compiler flags
set(CMAKE_CXX_FLAGS "${oldCxxFlags}")
add_library(RelaxedReq STATIC ...)
```


Exclusively for sbmain17@gmail.com Transaction: 0098315591

It may be surprising to learn that with the arrangement above, the `StrictReq` library will *not* be built with `-Wall -Werror` flags. Intuitively, one may expect that the variable's value at the time of the call to `add_library()` is what CMake uses, but in fact it is the variable's value at the end of processing for that directory scope that gets used. In other words, what matters is the value the variable holds at the end of the `CMakeLists.txt` file for that directory. This can lead to unexpected results in a variety of situations for the unaware.

One of the main ways developers get caught out by this behavior is to treat the compiler and linker variables as though they apply immediately to any targets that are created. Another related trap is when an `include()` is used after targets have been created and the included file(s) modify the compiler or linker variables. This would also alter the compiler and linker flags for the already defined targets in the current directory scope. This delayed nature of compiler and linker variables makes them fragile to work with. Ideally, a project would only modify them early in the top level `CMakeLists.txt` file, if at all, so as to minimize opportunities for misuse and developer surprise.

15.5. Language-specific Compiler Flags

There are limitations to be aware of when it comes to setting compiler flags that should only apply for specific languages. It is possible to set language-specific compiler flags for a particular target using generator expressions, as the following example shows (for simplicity, the compiler is assumed to support the `-fno-exceptions` option):

```
target_compile_options(Foo PRIVATE
    $<$<COMPILE_LANGUAGE:CXX>:-fno-exceptions>
)
```

Unfortunately, this will not work as expected for Visual Studio or for Xcode. Those generators' implementations do not support setting different flags for different languages at the target level. Instead, they evaluate generator expressions with the target language assumed to be C++ if the target has any C++ sources, or as C otherwise. This applies not just to compile options, but also to compile definitions and include directories. This limitation is a result of a compromise needed to avoid considerably degrading the build performance. If projects are willing to accept slower builds, compiler flags can be applied using source file properties instead. For example:

```
add_executable(Foo src1.c src2.cpp)

set_property(SOURCE src2.cpp APPEND PROPERTY
    COMPILE_OPTIONS -fno-exceptions
)
```

Source file properties also have their own limitations, as discussed back in Section 9.5, "Source Properties". In particular, note that the Xcode generator has restrictions which prevent it from supporting configuration-specific source file properties, so generator expressions like `$<CONFIG>` would need to be avoided.

A better way to work around these limitations is to split out the different languages to their own separate targets rather than combining them in the same target. The compiler flags can then be

Exclusively for sbmain17@gmail.com Transaction: 0098315591

applied to the whole target, which will work for all CMake generators and will not degrade build performance.

```
add_library(Foo_c src1.c)
add_executable(Foo src2.cpp)

target_link_libraries(Foo PRIVATE Foo_c)
target_compile_options(Foo PRIVATE -fno-exceptions)
```

A less ideal workaround is to use the `CMAKE_<LANG>_FLAGS` variables which are handled correctly by Visual Studio and Xcode, but they will apply indiscriminately to all targets in a directory scope and should preferably be left alone for developers to manipulate.

15.6. Recommended Practices

This chapter has covered areas of CMake which have undergone some of the most significant improvements since earlier versions. The reader should expect to encounter plenty of examples and tutorials online and elsewhere which still recommend patterns and approaches employing the older methods using variables and directory property commands. It should be understood that the `target_⋯()` commands should be the preferred approach in the CMake 3.0+ era.

Projects should seek to define all dependencies between targets with the `target_link_libraries()` command. This clearly expresses the nature of the relationships between targets and communicates unambiguously to all of a project's developers how targets are related. The `target_link_libraries()` command should be preferred over `link_libraries()` or manipulating target or directory properties directly. Similarly, the other `target_⋯()` commands offer a cleaner, more consistent and more robust way to manipulate compiler and linker flags than variables, directory property commands or direct manipulation of properties.

CMake 3.13 introduced a number of new commands and properties related to linker options, some of which were added for consistency reasons or to address specific use cases. Projects should generally avoid the new `add_link_options()` directory level command and prefer to use the new `target_link_options()` command instead. CMake 3.13 also introduced a new target level command `target_link_directories()`, which is a complement to the existing directory level `link_directories()` command. Both of these link directory commands should be avoided for robustness reasons. Projects are advised to link to target names or use full paths to libraries where those libraries are not in directories expected to be on the default linker search path.

The following general guide may be useful in deciding which methods are more appropriate in given situations:

- Where possible, prefer to use the `target_⋯()` commands to describe relationships between targets and to modify compiler and linker behavior.
- In general, prefer to avoid the directory property commands. While they can be convenient in a few specific circumstances, consistent use of the `target_⋯()` commands instead will establish clear patterns that all developers in a project can follow. If directory property commands must be used, do so as early in the `CMakeLists.txt` file as possible to avoid some of the less intuitive behavior described in the preceding sections.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

- Avoid direct manipulation of the target and directory properties that affect compiler and linker behavior. Understand what the properties do and how the different commands manipulate them, but prefer to use the more specialized target and directory specific commands where possible. Querying the target properties can, however, be useful from time to time when investigating unexpected compiler or linker command line flags.
- Prefer to avoid modifying the various `CMAKE_⋯_FLAGS` variables and their configuration specific counterparts. Consider these to be reserved for the developer who may wish to change them locally at will. If changes need to be applied on a whole-of-project basis, consider using a few strategic directory property commands at the top level of the project instead, but consider whether such settings really should be unilaterally applied. A partial exception to this is in toolchain files where initial defaults may be defined (see Chapter 22, *Toolchains And Cross Compiling* for a detailed discussion of this area).

Developers should become familiar with the concepts of `PRIVATE`, `PUBLIC` and `INTERFACE` relationships. They are a critical part of the `target_⋯()` command set and they become even more important for the install and packaging stages of a project. Think of `PRIVATE` as meaning for the target itself, `INTERFACE` for things that link against the target and `PUBLIC` as meaning both behaviors combined. While it may be tempting to just mark everything as `PUBLIC`, this may unnecessarily expose dependencies out beyond targets they need to. Build times can be impacted and private dependencies can be forced onto other targets which should not have to know about them. This in turn has a strong impact on other areas such as symbol visibility (discussed in detail in Section 21.5, "Symbol Visibility"). Therefore, prefer to start with a dependency as `PRIVATE` and only make it `PUBLIC` when it is clear that the dependency is needed by those linking to the target.

The `INTERFACE` keyword tends to be used mostly for imported or interface library targets, or occasionally for adding missing dependencies to a target defined in a part of the project which the developer may not be allowed to change. Examples of this include sub-parts of the project that were written for older CMake versions and therefore don't use the `target_⋯()` commands, or external libraries with imported targets that omit some important flags needed by targets linking to them. CMake 3.13 removed the restriction that `target_link_libraries()` could not be called to operate on a target defined in a different directory scope. For all other `target_⋯()` commands, there was no such restriction previously and so they can always be used to extend the interface properties of targets defined elsewhere in the project. Section 29.5.1, "Target Sources" revisits this topic, demonstrating how these capabilities can also be used to promote a more modular project structure.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

Chapter 16. Language Requirements

With the ongoing evolution of the C and C++ languages, developers are increasingly required to understand the compiler and linker flags that enable support for the C and/or C++ version their code uses. Different compilers use different flags, but even when using the same compiler and linker, flags can be used to select different implementations of the standard library.

In the days where C++11 support was relatively new, CMake had no direct support for choosing which standard to use, so projects were left to work out the required flags on their own. In CMake 3.1, features were introduced to allow the C and C++ standard to be selected in a consistent and convenient way, abstracting away the various compiler and linker differences. This support has been extended in subsequent versions and from CMake 3.6 covers most common compilers (CMake 3.2 added most of the compiler support, 3.6 added the Intel compiler).

Two main methods are provided by CMake for specifying language requirements. The first is to set the language standard directly and the second is to allow projects to specify the language features they need and let CMake select the appropriate language standard. While the functionality has largely been driven by the C and C++ languages, other languages and pseudo-languages such as CUDA and Objective C/C++ are also supported.

16.1. Setting The Language Standard Directly

The simplest way for a project to control the language standards used by a build is to set them directly. Using this approach, developers do not need to know or specify the individual language features used by the code, they just need to set a single number indicating the standard the code assumes is supported. Not only is this easy to understand and use, it also has the advantage that it is relatively straightforward to ensure that the same standard is used throughout a project. This becomes important at the link stage where a consistent standard library should be used across all the linked libraries and object files. It can also be important at compile time if headers define things differently depending on what language standard is being used.

As is the usual pattern with CMake, target properties control which standard will be used when building that target's sources and when linking the final executable or shared library. For a given language, there are three target properties related to specifying the standard. In the following, `<LANG>` will most commonly be either `C` or `CXX`, but `CUDA`, `OBJC` and `OBJCXX` are also supported with more recent CMake versions.

`<LANG>_STANDARD`

Specifies the language standard the project wants to use for the specified target.

- `C_STANDARD` supports the values 90, 99 and 11, with CMake 3.21 adding 17 and 23.
- `CXX_STANDARD` supports the values 98, 11 and 14. The value 17 is also supported since CMake 3.8, 20 since CMake 3.12 and 23 since CMake 3.20.
- `CUDA_STANDARD` is a CUDA-specific version of what `CXX_STANDARD` would normally control. If `CUDA_STANDARD` is not defined, it effectively falls back to `CXX_STANDARD`. Supported values for `CUDA_STANDARD` are 98 and 11 since CMake 3.8, 14 since CMake 3.9, 03, 17 and 20 since CMake 3.18 and 23 since CMake 3.22. Some of these values are recognized as valid by earlier CMake

Exclusively for sbmain17@gmail.com Transaction: 0098315591

versions than the ones just mentioned, but no compilers actually supported the values prior to the minimum versions stated.

- From CMake 3.16, OBJC_STANDARD and OBJCXX_STANDARD follow a similar pattern. If they are not defined, then C_STANDARD and CXX_STANDARD are used as fallbacks respectively. In practice, it would be unusual to set OBJC_STANDARD or OBJCXX_STANDARD, since the fallback to C_STANDARD and CXX_STANDARD would typically be desirable to ensure that C/C++ code uses language standards consistent with the Objective C/C++ code.

One would reasonably presume that later CMake versions would add support for other language standards as they evolve over time. When a target is created, the initial value of this <LANG>_STANDARD property is taken from the CMAKE_<LANG>_STANDARD variable.

<LANG>_STANDARD_REQUIRED

While the <LANG>_STANDARD property specifies the language standard the project wants, <LANG>_STANDARD_REQUIRED determines whether that language standard is treated as a minimum requirement or as just a "use if available" guideline. One might intuitively expect that <LANG>_STANDARD would be a requirement by default, but for better or worse, the <LANG>_STANDARD_REQUIRED properties are OFF by default. When OFF, if the requested standard is not supported by the compiler, CMake will decay the request to an earlier standard rather than halting with an error. This decaying behavior is often unexpected for new developers and in practice can be a cause of confusion. Thus, for most projects, when specifying a <LANG>_STANDARD property, its corresponding <LANG>_STANDARD_REQUIRED property will almost always need to be set to true as well to ensure the particular requested standard is treated as a firm requirement. When a target is created, the initial value of this property is taken from the CMAKE_<LANG>_STANDARD_REQUIRED variable.

<LANG>_EXTENSIONS

Many compilers support their own extensions to the language standard. A compiler and/or linker flag is usually provided to enable or disable those extensions. The <LANG>_EXTENSIONS target property controls whether those extensions are enabled for that particular target. See below for how this property is initialized.

For many compilers, the same flag is used to control both the language standard and whether extensions are enabled. With CMake 3.21 or earlier, if a project sets the <LANG>_EXTENSIONS property without also setting the <LANG>_STANDARD property, <LANG>_EXTENSIONS may effectively be ignored. CMake 3.22 introduced policy CMP0128 which affects this behavior. When CMP0128 is set to NEW, <LANG>_EXTENSIONS will be honored regardless of whether <LANG>_STANDARD is set or not.

CMP0128 also affects how <LANG>_EXTENSIONS is initialized when a target is created. With CMP0128 set to OLD or not set, the initial value for the <LANG>_EXTENSIONS property is taken from the CMAKE_<LANG>_EXTENSIONS variable. If that variable isn't set, the default value will be true. When CMP0128 is set to NEW, the initial value of <LANG>_EXTENSIONS will also be taken from the CMAKE_<LANG>_EXTENSIONS variable if it is defined, but will fall back to the value of a separate, read-only CMAKE_<LANG>_EXTENSIONS_DEFAULT variable otherwise. CMake determines the compiler's default during its compiler checks and will set CMAKE_<LANG>_EXTENSIONS_DEFAULT accordingly to reflect that behavior.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

CMP0128 has one more potential effect. When this policy is set to NEW, flags related to setting the language standard or extensions will only be added if the compiler's default behavior doesn't already provide those settings. This may change the flags seen on the compiler command line, but shouldn't change the actual behavior unless the project is adding its own language standard flags directly (which is strongly discouraged).

In practice, projects would more typically set the variables that provide the defaults for the above target properties rather than setting the target properties directly. This ensures that all targets in a project are built in a consistent manner with compatible settings. It is also recommended that projects set all three rather than just some of them. The defaults for <LANG>_STANDARD_REQUIRED and <LANG>_EXTENSIONS have proven to be relatively unintuitive for many developers. By explicitly setting them, a project makes clear the standard behavior it expects.

```
# Require C++11 and disable extensions for all targets
set(CMAKE_CXX_STANDARD          11)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_CXX_EXTENSIONS        OFF)
```

When using GCC or Clang, the above would typically add the -std=c++11 flag. For Visual Studio compilers before VS2015 Update 3, no flags would be added since the compiler either supports C++11 by default or it has no support for C++11 at all. From Visual Studio 15 Update 3, the compiler supports specifying a C++ standard of C++14 or later, with C++14 being the default if not set.

In comparison, the following example requests a later C++ version and enables compiler extensions, resulting in a GCC/Clang compiler flag like -std=gnu++14 instead. Visual Studio compilers again may support the requested standard by default or not, depending on compiler version. If the compiler in use does not support the requested C++ standard, CMake will configure the compiler to use the most recent C++ standard it supports.

```
# Use C++14 if available and allow compiler extensions for all targets
set(CMAKE_CXX_STANDARD          14)
set(CMAKE_CXX_STANDARD_REQUIRED OFF)
set(CMAKE_CXX_EXTENSIONS        ON)
```

The following example shows how to set the C standard details for a specific target only:

```
# Build target Foo with C99, no compiler extensions
set_target_properties(Foo PROPERTIES
    C_STANDARD          99
    C_STANDARD_REQUIRED ON
    C_EXTENSIONS        OFF
)
```

Note that <LANG>_STANDARD specifies a minimum standard, not necessarily an exact requirement. CMake may select a more recent standard due to compile feature requirements (discussed next).

Exclusively for sbmain17@gmail.com Transaction: 0098315591

16.2. Setting The Language Standard By Feature Requirements

Directly setting the language standard for a target or for a whole project is the simplest way to manage standard requirements. It is the most suitable approach when the project's developers know which language version provides the features used by the project's code. It is particularly convenient when a large number of language features are being used, since each feature does not have to be explicitly specified. In some cases, however, developers may prefer to state which language features their code uses and leave CMake to select the appropriate language standard. This has the advantage that, unlike specifying the standard directly, compile feature requirements can be part of a target's interface and therefore can be enforced on other targets linking to it.

Compile feature requirements are controlled by the target properties `COMPILE_FEATURES` and `INTERFACE_COMPILE_FEATURES`, but these properties are typically populated using the `target_compile_features()` command rather than being manipulated directly. This command follows a very similar form to the various other `target_⋯()` commands provided by CMake:

```
target_compile_features(targetName
     <PRIVATE|PUBLIC|INTERFACE> feature1 [feature2 ...]
    [<PRIVATE|PUBLIC|INTERFACE> feature3 [feature4 ...]] ...
)
```

The `PRIVATE`, `PUBLIC` and `INTERFACE` keywords have their usual meanings, controlling how the listed features should be applied. `PRIVATE` features populate the `COMPILE_FEATURES` property, which is applied to the target itself. Those features specified with the `INTERFACE` keyword populate the `INTERFACE_COMPILE_FEATURES` property, which is applied to any target that links to `targetName`. Features specified as `PUBLIC` will be added to both properties and will therefore be applied to both the target itself and to any other target which links to it.

Each feature must be one of the features supported by the underlying compiler. CMake provides two lists of known features: `CMAKE_<LANG>_KNOWN_FEATURES` which contains all known features for the language and `CMAKE_<LANG>_COMPILE_FEATURES` which contains only those features supported by the compiler. If a requested feature is not supported by the compiler, CMake will report an error. Developers may find the CMake documentation for the `CMAKE_<LANG>_KNOWN_FEATURES` variables to be a particularly useful resource, since it not only lists the features understood by that particular version of CMake, it also contains references to standard documents relating to each feature. Note that not all functionality provided by a particular language version can be explicitly specified using compile features. For example, new C++ STL types, functions, etc. have no associated feature.

From CMake 3.8, a per language meta-feature is available to indicate a particular language standard rather than a specific compile feature. These meta-features take the form `<lang>_std_<value>` and when listed as a required compile feature, CMake will ensure compiler flags are used which enable that language standard. For example, to add a compile feature which ensures that a target and anything that links against it has C++14 support enabled, the following could be used:

Exclusively for sbmain17@gmail.com Transaction: 0098315591

```
target_compile_features(targetName PUBLIC cxx_std_14)
```

If a project needs to support CMake versions earlier than 3.8, then the above meta-feature will not be available. In such cases, each compile feature would have to be listed out individually, which can be impractical and would likely be incomplete. This tends to limit the usefulness of compile features in general, with projects frequently choosing to set the language standard through the target properties described in the previous section instead.

In situations where a target has both its `<LANG>_STANDARD` property set and compile features specified (directly or transitively as a result of `INTERFACE` features from something it links to), CMake will enforce the stronger standard requirement. In the following example, `Foo` would be built with C++14, `Bar` with C++17 and `Guff` with C++14:

```
set_target_properties(Foo PROPERTIES CXX_STANDARD 11)
target_compile_features(Foo PUBLIC cxx_std_14)

set_target_properties(Bar PROPERTIES CXX_STANDARD 17)
target_compile_features(Bar PRIVATE cxx_std_11)

set_target_properties(Guff PROPERTIES CXX_STANDARD 11)
target_link_libraries(Guff PRIVATE Foo)
```

Note that this may mean a more recent language standard could be used than what the project expected, which in some cases can result in compilation errors. For example, C++17 removed `std::auto_ptr`, so if code expects to be compiled with an older language standard and still uses `std::auto_ptr`, it could fail to compile if the toolchain strictly enforces this removal.

The `CUDA`, `OBJC` and `OBJCXX` languages are a little unusual in that they are based off C or C++. The `OBJC` and `OBJCXX` languages do not as yet have their own separate set of compile features, but the compile features for the corresponding base language can be used instead. In CMake 3.16 or earlier, the `CUDA` language also did not have its own separate set of compile features and relied on the C++ compile features. CMake 3.17 added dedicated compile feature support for `CUDA`.

16.2.1. Detection And Use Of Optional Language Features

Some projects have the ability to handle a particular language feature being supported or not. They may provide a fall back implementation, for example, or only define certain function overloads if they are supported by the compiler. A project may support some compile features being optional, such as keywords intended to guide the developer or provide an increased ability for the compiler to catch common mistakes. C++ keywords such as `final` and `override` are common examples of this.

CMake provides a number of ways to handle the above scenarios. One approach is to use generator expressions to conditionally set compiler defines or include directories based on the availability of a particular compile feature. These can be a little verbose, but they offer a great deal of flexibility and support very precise handling of feature-based functionality. Consider the following example:

Exclusively for sbmain17@gmail.com Transaction: 0098315591

```
add_library(Foo ...)

# Make override a feature requirement only if available
target_compile_features(Foo PUBLIC
    $<$<COMPILE_FEATURES:cxx_override>:cxx_override>
)

# Define the foo_OVERRIDE symbol so it provides the
# override keyword if available or empty otherwise
target_compile_definitions(Foo PUBLIC
    $<$<COMPILE_FEATURES:cxx_override>:-Dfoo_OVERRIDE=override>
    $<$<NOT:$<COMPILE_FEATURES:cxx_override>>:-Dfoo_OVERRIDE>
)
```

The above would allow code such as the following to compile for any C++ compiler, regardless of whether or not it supported the `override` keyword:

```
class MyClass : public Base
{
public:
    void func() foo_OVERRIDE;
    ...
};
```

Other features can also have a similar conditionally defined symbol used in much the same way. C++ keywords like `final`, `constexpr` and `noexcept` can potentially be used if available or omitted if not supported by the compiler and still produce valid code. Other keywords such as `nullptr` and `static_assert` have alternative implementations which can be used if the keyword is not supported.

Specifying generator expressions for each feature to cover the supported and unsupported cases can be both tedious and fragile. Techniques such as using the `WriteCompilerDetectionHeader` module (which is deprecated as of CMake 3.20) can help reduce the drawbacks, but in the long run, switching behavior at the standard level rather than on individual features is likely to be the better approach.

16.3. Recommended Practices

Projects should avoid setting compiler and linker flags directly to control the language standard used. The required flags vary from compiler to compiler, so it is more robust, more maintainable and more convenient to use the features CMake provides and allow it to populate the flags appropriately. The `CMakeLists.txt` file will also more clearly express the intent, since human readable variables and properties are used instead of often cryptic raw compiler and linker flags.

The simplest method for controlling language standard requirements is to use the `CMAKE_<LANG>_STANDARD`, `CMAKE_<LANG>_STANDARD_REQUIRED` and `CMAKE_<LANG>_EXTENSIONS` variables. These can be used to set the language standard behavior for the entire project, ensuring consistent usage across all targets. These variables should ideally be set just after the first `project()` command in the top level `CMakeLists.txt` file. Projects should always set all three variables together to make clear how the language standard requirements should be enforced and whether compiler extensions are

Exclusively for sbmain17@gmail.com Transaction: 0098315591

permitted. Omitting CMAKE_<LANG>_STANDARD_REQUIRED or CMAKE_<LANG>_EXTENSIONS can lead to unexpected behavior, as the defaults may not be what some developers intuitively expect.

If the language standard only needs to be enforced for some targets and not others, the <LANG>_STANDARD, <LANG>_STANDARD_REQUIRED and <LANG>_EXTENSIONS target properties can be set on individual targets rather than for the whole project. These properties behave as though they were PRIVATE, meaning they only specify requirements on that target and not on anything linking to it. This places more of a burden on the project to ensure that all targets have correctly specified language standard details. In practice, it is usually easier and more robust to use the variables to set language requirements project-wide rather than use per target properties. Prefer using the variables unless the project needs different language standard behavior for different targets.

If using CMake 3.8 or later, compile features can be used to specify the desired language standard on a per-target basis. The target_compile_features() command makes this easy and clearly specifies whether such requirements are PRIVATE, PUBLIC or INTERFACE. The main advantage of specifying a language requirement this way is that it can be enforced transitively on other targets via PUBLIC and INTERFACE relationships. These requirements are also preserved when targets are exported and installed (see Chapter 26, *Installing*). Note, however, that only the equivalent of the <LANG>_STANDARD and <LANG>_STANDARD_REQUIRED target property behaviors are provided, so the <LANG>_EXTENSIONS target property or CMAKE_<LANG>_EXTENSIONS variable should still be used to control whether compiler extensions are allowed.

If policy CMP0128 is not set to NEW, the <LANG>_EXTENSIONS properties and their associated variables often only take effect if the corresponding <LANG>_STANDARD is also set. This is due to how compilers frequently combine the two into a single flag. Therefore, unless using CMake 3.22 or later with policy CMP0128 set to NEW, it is difficult to escape having to specify <LANG>_STANDARD, even when compile features are used.

Specifying individual compile features provides fine-grained control over the language requirements at a per-target level. In practice, it is difficult for developers to ensure that all features used by a target are explicitly specified, so there will always be the question of whether the language requirements are properly defined. They can also easily become out of date as code development continues over time. Most projects will probably find specifying language requirements this way to be tedious and fragile, so they should only be used if the situation clearly warrants it.

In the early days of C++11 and C++14, working with compile features was potentially useful because compiler support was often lagging behind. For later language releases, support by the mainstream compilers has arrived much faster. Subsequently, CMake stopped providing fine-grained features for language standards beyond C++14. For C++17 and later, only the high level meta features like cxx_std_17 are provided.

For projects still needing to support older compilers, they can detect available compile features and provide implementations for whether a feature is available or not. The WriteCompilerDetectionHeader module provided by CMake was sometimes useful in helping projects transition to more modern compilers, but that module was deprecated in CMake 3.20. Therefore, it should no longer be used.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

Chapter 17. Target Types

CMake supports a wide variety of target types, not just the simple executables and libraries introduced back in Chapter 4, *Building Simple Targets*. Different target types can be defined that act as a reference to other entities rather than being built themselves. They can be used to collect together transitive properties and dependencies without actually producing their own binaries, or they can even be a kind of library that is simply a collection of object files rather than a traditional static or shared library. Many things can be abstracted away as a target to hide the complexities of platform differences, locations in the filesystem, file names and so on. This chapter covers all of these various target types and discusses their uses.

Another category of target is the utility or custom target. These can be used to execute arbitrary commands and define custom build rules, allowing projects to implement just about any sort of behavior needed. They have their own dedicated commands and unique behaviors and are covered in depth in the next chapter.

17.1. Executables

The `add_executable()` command has more than just the form introduced back in Chapter 4, *Building Simple Targets*. Two other forms also exist which can be used to define executable targets that reference other things. The full set of supported forms are:

```
add_executable(targetName
    [WIN32] [MACOSX_BUNDLE]
    [EXCLUDE_FROM_ALL]
    source1 [source2 ...]
)
add_executable(targetName IMPORTED [GLOBAL])
add_executable(aliasName ALIAS targetName)
```

The `IMPORTED` form can be used to create a CMake target for an existing executable rather than one built by the project. By creating a target to represent the executable, other parts of the project can treat it just like it would any other executable target that the project built itself (with some restrictions). The most significant benefit is that it can be used in contexts where CMake automatically replaces a target name with its location on disk, such as when executing commands for tests or custom tasks (both covered in later chapters). One of the few differences compared to a regular target is that imported targets cannot be installed, a topic covered in Chapter 26, *Installing*.

When defining an imported executable target, certain target properties need to be set before it can be useful. Most of the relevant properties for any imported target have names beginning with `IMPORTED`, but for executables, `IMPORTED_LOCATION` and `IMPORTED_LOCATION_<CONFIG>` are the most important. When the location of the imported executable is needed, CMake will first look at the configuration-specific property and only if that is not set will it look at the more generic `IMPORTED_LOCATION` property. Typically, the location doesn't need to be configuration-specific, so it is very common for only `IMPORTED_LOCATION` to be set.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

When defined without the GLOBAL keyword, an imported target will only be visible in the current directory scope and below, but adding GLOBAL makes the target visible everywhere. In contrast, regular executable targets built by the project are always global. The reasons for this and some of the associated implications of reduced target visibility are covered in Section 17.3, "Promoting Imported Targets" further below.

An ALIAS target is a read-only way to refer to another target within CMake. It can be used to read properties of the target it aliases and it may be used in custom commands and test commands just like the aliased target (see Section 18.1, "Custom Targets" and Section 25.1, "Defining And Executing A Simple Test" respectively). An alias does not create a new build target with the alias name. There are limitations to defining and using aliases:

- Aliases cannot be installed or exported (both covered in Chapter 26, *Installing*).
- An alias of an alias is not supported.
- Prior to CMake 3.11, imported targets could not be aliased at all.
- From CMake 3.11, imported targets with global visibility can be aliased.
- From CMake 3.18, an alias of a non-global imported target can be created and that alias will also be non-global. The alias cannot later be promoted to global visibility, even if the imported target it aliases is promoted (see Section 17.3, "Promoting Imported Targets").

17.2. Libraries

The add_library() command also has a number of different forms. The details for libraries are more involved than for executables, which is a consequence of the variety of roles that libraries can take in a project.

17.2.1. Basic Library Types

The basic form introduced back in Chapter 4, *Building Simple Targets* can be used to define the common types of libraries most developers are familiar with:

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

If no STATIC, SHARED or MODULE keyword is given, the library will be either STATIC or SHARED. The choice is determined by the value of the BUILD_SHARED_LIBS variable (see Section 21.1, "Build Basics").

17.2.2. Object Libraries

The add_library() command can also be used to define *object libraries*. These are a collection of object files that are not combined into a single archive or shared library:

Exclusively for sbmain17@gmail.com Transaction: 0098315591

```
add_library(targetName OBJECT
    [EXCLUDE_FROM_ALL]
    source1 [source2 ...]
)
```

With CMake 3.11 or earlier, object libraries cannot be linked like other library types (i.e. they cannot be used with `target_link_libraries()`). They require using a generator expression of the form `$<TARGET_OBJECTS:objLib>` as part of the list of sources of another executable or library target. Because they cannot be linked, they therefore do not provide transitive dependencies to the targets they are added to as objects/sources. This can make them less convenient than the other library types, since header search paths, compiler defines, etc. have to be manually carried across to the targets they are added to.

CMake 3.12 introduced features that make object libraries behave more like other types of libraries, but with some caveats. From CMake 3.12, object libraries *can* be used with `target_link_libraries()`, either as the target being added to (i.e. the first argument to the command) or as one of the libraries being added. But because they add object files rather than actual libraries, their transitive nature is more restricted to prevent object files from being added multiple times to consuming targets. A simplistic explanation is that object files are only added to a target that links *directly* to the object library, not transitively beyond that. The object library's usage requirements *do*, however, propagate transitively exactly like an ordinary library would.

The propagation of an object library's own link library dependencies initially contained an implementation bug which was not fixed until CMake 3.14.0. Projects should set their minimum CMake version to 3.14 or later if they intend to link to object libraries.

Some developers may find object libraries more natural if coming from a background where non-CMake projects defined their targets based on sources or object files rather than a related set of static libraries. In general, however, where there is a choice, static libraries will typically be the more convenient choice in CMake projects. Before relying on the expanded features available for object libraries in CMake 3.12 and later, consider whether an ordinary static library is more appropriate and ultimately easier to use.

17.2.3. Imported Libraries

Just like executables, libraries may also be defined as imported targets. These are heavily used by config files created during packaging or by Find module implementations (covered in Chapter 24, *Finding Things* and Chapter 26, *Installing*), but have limited use outside of those contexts. They don't define a library to be built by the project, rather they act as a reference to a library that is provided externally (e.g. it already exists on the system, is built by some process outside of the current CMake project or is provided by the package that a config file is part of).

```
add_library(targetName
    (STATIC | SHARED | MODULE | OBJECT | UNKNOWN)
    IMPORTED [GLOBAL]
)
```

Exclusively for sbmain17@gmail.com Transaction: 0098315591

The library type must be given immediately after the `targetName`. If the type of library that the new target will refer to is known, it should be specified as such. This will allow CMake to treat the imported target just like a regular library target of the named type in various situations. The type can only be set to `OBJECT` with CMake 3.9 or later (imported object libraries were not supported before that version). If the library type is not known, the `UNKNOWN` type should be given, in which case CMake will simply use the full path to the library without further interpretation in places like linker command lines. This will mean fewer checks and in the case of Windows builds, no handling of DLL import libraries.

Except for `OBJECT` libraries, the location on the filesystem that the imported target represents needs to be specified by the `IMPORTED_LOCATION` and/or `IMPORTED_LOCATION_<CONFIG>` properties (i.e. the same as for imported executables). In the case of Windows platforms, two properties should be set: `IMPORTED_LOCATION` should hold the location of the DLL and `IMPORTED_IMPLIB` should hold the location of the associated import library, which usually has a `.lib` file extension (the `…_<CONFIG>` variants of these properties can also be set and will take precedence). For object libraries, instead of the above location properties, the `IMPORTED_OBJECTS` property must be set to a list of object files that the imported target represents.

Imported libraries also support a number of other target properties, most of which can typically be left alone or are automatically set by CMake. Developers who need to manually write config packages should refer to the CMake reference documentation to understand the other `IMPORTED_…` target properties which may be relevant to their situation. Most projects will rely on CMake generating such files for them though, so the need to do this should be fairly uncommon.

By default, imported libraries are defined as local targets, meaning they are only visible in the current directory scope and below. The `GLOBAL` keyword can be given to make them have global visibility instead, just like other regular targets. A library may initially be created without the `GLOBAL` keyword but later promoted to global visibility, a topic covered in detail in Section 17.3, “Promoting Imported Targets” further below.

```
# Windows-specific example of imported library
add_library(MyWindowsLib SHARED IMPORTED)

set_target_properties(MyWindowsLib PROPERTIES
    IMPORTED_LOCATION /some/path/bin/foo.dll
    IMPORTED_IMPLIB   /some/path/lib/foo.lib
)
```

```
# Assume FOO_LIB holds the location of the library but its type is unknown
add_library(MysteryLib UNKNOWN IMPORTED)

set_target_properties(MysteryLib PROPERTIES
    IMPORTED_LOCATION ${FOO_LIB}
)
```


Exclusively for sbmain17@gmail.com Transaction: 0098315591

```
# Imported object library, Windows example shown
add_library(MyObjLib OBJECT IMPORTED)

set_property(TARGET MyObjLib PROPERTY
    # These .obj files would be .o on most other platforms
    IMPORTED_OBJECTS /some/path/obj1.obj
                     /some/path/obj2.obj
)

# Regular executable target using imported object library.
# Platform differences assumed to be handled by MyObjLib.
add_executable(MyExe $<TARGET_OBJECTS:MyObjLib>)
```

17.2.4. Interface Libraries

Another form of the `add_library()` command allows interface libraries to be defined. These do not usually represent a physical library, instead they primarily serve to collect usage requirements and dependencies to be applied to anything that links to them. A popular example of their use is for header-only libraries where there is no physical library that needs to be linked, but header search paths, compiler definitions, etc. need to be carried forward to anything using the headers.

```
add_library(targetName INTERFACE)
```

All the various `target_…()` commands can be used with their `INTERFACE` keywords to define the usage requirements the interface library will carry. One can also set the relevant `INTERFACE_…` properties directly with `set_property()` or `set_target_properties()`, but the `target_…()` commands are safer and easier to use.

```
add_library(MyHeaderOnlyToolkit INTERFACE)
target_include_directories(MyHeaderOnlyToolkit
    INTERFACE /some/path/include
)
target_compile_definitions(MyHeaderOnlyToolkit
    INTERFACE COOL_FEATURE=1
              $<$<COMPILE_FEATURES:cxx_std_11>:HAVE_CXX11>
)

add_executable(MyApp ...)
target_link_libraries(MyApp PRIVATE MyHeaderOnlyToolkit)
```

In the above example, the `MyApp` target links against the `MyHeaderOnlyToolkit` interface library. When the `MyApp` sources are compiled, they will have `/some/path/include` as a header search path and will also have a compiler definition `COOL_FEATURE=1` provided on the compiler command line. If the `MyApp` target is being built with C++11 support enabled, it will also have the symbol `HAVE_CXX11` defined. The headers in `MyHeaderOnlyToolkit` can then use this symbol to determine what things they declare and define rather than relying on the `__cplusplus` symbol provided by the C++ standard, the value of which is often unreliable for a range of compilers.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

Ordinarily, an interface library would not have any sources, but in some cases it can make sense. Header-only libraries are one such example. The headers are likely to be of interest to developers and they may want the headers to show up in their IDE. Since the headers are not part of any other target, they normally wouldn't appear. By adding them to the interface library as sources, IDEs usually have enough information to be able to show the headers under one or more targets.

A special sub-case of the above example is when one or more headers in a header-only library are generated as part of the build (a topic covered in Section 18.3, "Commands That Generate Files"). With CMake 3.18 or earlier, if nothing in the project actually uses the interface library, the project needs to create a separate custom target to ensure that the headers will be generated. Another limitation is that sources cannot be added to the interface library in the `add_library()` call, a separate call to `target_sources()` has to be used instead (see Section 29.5.1, "Target Sources"). The resultant code would take the following form:

```
# Defines how to generate the header
add_custom_command(OUTPUT someHeader.h COMMAND ...)

# Required for CMake <= 3.18 to ensure header is generated
add_custom_target(GenerateSomeHeader ALL
    DEPENDS ${CMAKE_CURRENT_BINARY_DIR}/someHeader.h
)

# CMake <= 3.18 doesn't allow sources to be added to an
# INTERFACE target directly in the add_library() call
add_library(MyHeaderOnly INTERFACE)
target_sources(MyHeaderOnly
    INTERFACE
        ${CMAKE_CURRENT_BINARY_DIR}/someHeader.h
)
```

From CMake 3.19, both of the above-mentioned restrictions have been removed. Sources can be listed directly in the `add_library()` call for interface libraries and they will be treated as private sources. This means that unlike the example above, the headers will not be added to targets that link to the interface library. Instead, they will remain associated only with the interface library itself and therefore will only show up in IDEs under that library instead of in all of its consumers. CMake will also create a build system target for the interface library if any sources are added to it. Bringing that build system target up to date like any other target will then ensure that its generated sources are created. The end result is both more concise and produces a better result in IDEs:

```
add_custom_command(OUTPUT someHeader.h COMMAND ...)

# Requires CMake 3.19 or later
add_library(MyHeaderOnly INTERFACE
    ${CMAKE_CURRENT_BINARY_DIR}/someHeader.h
)
```

Another use of interface libraries is to provide a convenience for linking in a larger set of libraries, possibly encapsulating logic that selects which libraries should be in the set. For example:

Exclusively for sbmain17@gmail.com Transaction: 0098315591

```
# Regular library targets
add_library(Algo_Fast ...)
add_library(Algo_Accurate ...)
add_library(Algo_Beta ...)

# Convenience interface library
add_library(Algo_All INTERFACE)
target_link_libraries(Algo_All INTERFACE
    Algo_Fast
    Algo_Accurate
    $<$<BOOL:${ENABLE_ALGO_BETA}>:Algo_Beta>
)

# Other targets link to the interface library
# instead of each of the real libraries
add_executable(MyApp ...)
target_link_libraries(MyApp PRIVATE Algo_All)
```

The above will only include `Algo_Beta` in the list of libraries to link if the CMake option variable `ENABLE_ALGO_BETA` is true. Other targets then simply link to `Algo_All` and the conditional linking of `Algo_Beta` is handled by the interface library. This is an example of using an interface library to abstract away details of what is actually going to be linked, defined, etc. so that the targets linking against them don't have to implement those details for themselves. This can be exploited to do things like abstract away completely different library structures on different platforms, switch library implementations based on some condition (variables, generator expressions, etc.), provide an old library target name where the library structure has been refactored (e.g. split up into separate libraries) and so on.

17.2.5. Interface Imported Libraries

```
add_library(targetName INTERFACE IMPORTED [GLOBAL])
```

While the use cases for `INTERFACE` libraries are generally well understood, the addition of the `IMPORTED` keyword to yield an `INTERFACE IMPORTED` library can sometimes be a cause of confusion. This combination usually arises when an `INTERFACE` library is exported or installed for use outside of the project. It still serves the purpose of an `INTERFACE` library when consumed by another project, but the `IMPORTED` part is added to indicate the library came from somewhere else. The effect of this is to restrict the default visibility of the library to the current directory scope instead of global. With one exception discussed below, adding the `GLOBAL` keyword to yield the keyword combination `INTERFACE IMPORTED GLOBAL` results in a library with little practical difference compared to `INTERFACE` alone. An `INTERFACE IMPORTED` library is not required to (and indeed is prohibited from) setting an `IMPORTED_LOCATION`.

Before CMake 3.11, none of the `target_⋯()` commands could be used to set `INTERFACE_⋯` properties on any kind of `IMPORTED` library. These properties *could*, however, be set using `set_property()` or `set_target_properties()`. CMake 3.11 removed the restriction on using `target_⋯()` commands to set these properties, so whereas `INTERFACE IMPORTED` used to be very similar to plain `IMPORTED` libraries, with CMake 3.11 they are now much closer to plain `INTERFACE` libraries in terms of their set of restrictions.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

The following table summarizes what the various keyword combinations support:

Keywords	Visibility	Imported Location	Set Interface Properties	Installable
`INTERFACE`	Global	Prohibited	Any method	Yes
`IMPORTED`	Local	Required	Restricted*	No
`IMPORTED GLOBAL`	Global	Required	Restricted*	No
`INTERFACE IMPORTED`	Local	Prohibited	Restricted*	No
`INTERFACE IMPORTED GLOBAL`	Global	Prohibited	Restricted*	No

* The various `target_…()` commands can be used to set `INTERFACE_…` properties with CMake 3.11 or later. The `set_property()` or `set_target_properties()` commands can be used with any CMake version.

One could be forgiven for thinking that the number of different interface and imported library combinations is overly complicated and confusing. For most developers, however, imported targets are generally created for them behind the scenes and they appear to act more or less like regular targets. Of all the combinations in the above table, only plain `INTERFACE` targets would typically be defined by a project directly. Chapter 26, *Installing* covers much of the motivation and mechanics of the other combinations.

17.2.6. Library Aliases

The last form of the `add_library()` command is for defining a library alias:

```
add_library(aliasName ALIAS otherTarget)
```

A library alias is mostly analogous to an executable alias. It acts as a read-only way to refer to another library, but does not create a new build target. Library aliases cannot be installed and they cannot be defined as an alias of another alias. Before CMake 3.11, a library alias could not be created for imported targets, but as with other changes made for imported targets in CMake 3.11, this restriction was relaxed and it has become possible to create aliases for globally visible imported targets. CMake 3.18 relaxed that restriction further to allow non-global aliases to be created for non-global imported targets.

There is a particularly common use of library aliases that relates to an important feature introduced in CMake 3.0. For each library that will be installed or packaged, a common pattern is to also create a matching library alias with a name of the form `projNamespace::targetName`. All such aliases within a project would typically share the same `projNamespace`. For example:

```
# Any sort of real library (SHARED, STATIC, MODULE or possibly OBJECT)
add_library(MyRealThings SHARED src1.cpp ...)
add_library(OtherThings STATIC srcA.cpp ...)

# Aliases to the above with special names
add_library(BagOfBeans::MyRealThings ALIAS MyRealThings)
add_library(BagOfBeans::OtherThings  ALIAS OtherThings)
```


Exclusively for sbmain17@gmail.com Transaction: 0098315591

Within the project itself, other targets would link to either the real targets or the namespaced targets (both have the same effect). The motivation for the aliases comes from when the project is installed and something else links to the imported targets created by the installed/packaged config files. Those config files would define imported libraries with the namespaced names rather than the bare original names (see Section 26.3, "Installing Exports"). The consuming project would then link against the namespaced names. For example:

```
# Pull in imported targets from an installed package
find_package(BagOfBeans REQUIRED)    ①

# Define an executable that links to the imported library from the installed package
add_executable(EatLunch main.cpp ...)
target_link_libraries(EatLunch PRIVATE
    BagOfBeans::MyRealThings
)
```

① The `find_package()` command is discussed in Chapter 24, *Finding Things*.

If at some point the above project wanted to incorporate the `BagOfBeans` project directly into its own build instead of finding an installed package, it could do so without changing its linking relationship because the `BagOfBeans` project provided an alias for the namespaced name:

```
# Add BagOfBeans directly to this project, making all of its targets directly available
add_subdirectory(BagOfBeans)

# Same definition of linking relationship still works
add_executable(EatLunch main.cpp ...)
target_link_libraries(EatLunch PRIVATE
    BagOfBeans::MyRealThings
)
```

Another important aspect of names having a double-colon (`::`) is that CMake will always treat them as the name of an alias or imported target. Any attempt to use such a name for a different target type will result in an error. Perhaps more usefully though, when the target name is used as part of a `target_link_library()` call, if CMake doesn't know of a target by that name, it will issue an error at generation time. Compare this to an ordinary name which CMake will treat as a library assumed to be provided by the system if it doesn't know of a target by that name. This can lead to the error only becoming apparent much later at build time.

```
add_executable(Main main.cpp)
add_library(Bar STATIC ...)
add_library(Foo::Bar ALIAS Bar)

# Typo in name being linked to, CMake will assume a library called "Bart"
# will be provided by the system at link time and won't issue an error.
target_link_libraries(Main PRIVATE Bart)

# Typo in name being linked to, CMake flags an error at generation time
# because a namespaced name must be a CMake target.
target_link_libraries(Main PRIVATE Foo::Bart)
```

Exclusively for sbmain17@gmail.com Transaction: 0098315591

It is therefore more robust to link to namespaced names where they are available. Projects are strongly encouraged to define namespaced aliases at least for all targets that are intended to be installed/packaged. Such namespaced aliases can even be used within the project itself, not just by other projects consuming it as a pre-built package or child project. Section 26.3, "Installing Exports" also discusses how to change the name used for the `targetName` part of a namespaced name, which can allow an original target like `MyProj_Algo` to have a namespaced name like `MyProj::Algo` instead of the more verbose and repetitive `MyProj::MyProj_Algo`.

17.3. Promoting Imported Targets

When defined without the `GLOBAL` keyword, imported targets are only visible in the directory scope in which they are created or below. This behavior stems from their main intended use, which is as part of a Find module or package config file. Anything defined by a Find module or package config file is generally expected to have local visibility, so they shouldn't generally add globally visible targets. This allows different parts of a project hierarchy to pull in the same packages and modules with different settings, yet not interfere with each other.

Nevertheless, there are situations where imported targets need to be created with global visibility, such as to ensure that the same version or instance of a particular package is used consistently throughout the whole project. Adding the `GLOBAL` keyword when creating the imported library achieves this, but the project may not be in control of the command that does the creation. To provide projects with a way to address this situation, CMake 3.11 introduced the ability to promote an imported target to global visibility by setting the target's `IMPORTED_GLOBAL` property to true. Note that this is a one-way transition, it is not possible to demote a global target back to local visibility.

```
# Imported library created with local visibility. This could be in an external file
# brought in by an include() call rather than in the same file as the lines further below.
add_library(BuiltElsewhere STATIC IMPORTED)
set_target_properties(BuiltElsewhere PROPERTIES
    IMPORTED_LOCATION /path/to/libSomething.a
)

# Promote the imported target to global visibility
set_target_properties(BuiltElsewhere PROPERTIES
    IMPORTED_GLOBAL TRUE
)
```

It is important to note that an imported target can only be promoted if it is defined in exactly the same scope as the promotion. An imported target defined in a parent or child scope cannot be promoted. The `include()` command does not introduce a new directory scope and neither does a `find_package()` call, so imported targets defined by files brought into the build that way *can* be promoted. In fact, this is the main use case for which the ability to promote imported targets was created.

Promoting an imported target does not promote any aliases already pointing to that target. An alias to an imported target always has the visibility that the imported target had when the alias was created. Aliases do not support promotion to global visibility.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

```
add_library(Original STATIC IMPORTED)

# Local alias (requires CMake 3.18 or later)
add_library(LocalAlias ALIAS Original)

# Promote imported target to global visibility,
# but LocalAlias remains with local visibility
set_target_properties(Original PROPERTIES
    IMPORTED_GLOBAL TRUE
)

# Global alias (requires CMake 3.11 or later)
add_library(GlobalAlias ALIAS Original)
```

In practice, aliases to imported targets should rarely be needed. `INTERFACE IMPORTED` libraries can largely achieve the same thing and they work for a wider range of CMake versions. `INTERFACE IMPORTED` libraries don't support reading the underlying properties of the real library target, but they do carry all the linking and transitive properties.

```
add_library(Original STATIC IMPORTED)
add_library(OtherName INTERFACE IMPORTED Original)
target_link_libraries(OtherName INTERFACE Original)
```

An added advantage of `INTERFACE IMPORTED` libraries is that they can be promoted to global visibility if required, whereas aliases cannot.

17.4. Recommended Practices

Version 3.0 of CMake brought with it a significant change to the recommended way projects should manage dependencies and requirements between targets. Instead of specifying most things through variables which then had to be managed manually by the project, or by directory level commands that would apply to all targets in a directory and below without much discrimination, each target gained the ability to carry all the necessary information in its own properties. This shift in focus to a target-centric model has also led to a family of pseudo target types that facilitate expressing inter-target relationships more flexibly and accurately.

Developers should become familiar with interface libraries in particular. They open up a range of techniques for capturing and expressing relationships without needing to create or refer to a physical file. They can be useful for representing the details of header-only libraries, collections of resources and many other scenarios and should be strongly preferred over trying to achieve the same result with variables or directory-level commands alone.

Imported targets are encountered frequently once projects start using externally built packages or they refer to tools from the file system that are found through Find modules. Developers should be comfortable using imported targets, but understanding all the ins and outs of how they are defined is not usually necessary unless writing Find modules or manually creating config files for a package. Some specific cases are discussed in Chapter 26, *Installing* where developers may come up against certain limitations of imported targets, but such scenarios are not very common.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

A number of older CMake modules used to provide only variables to refer to imported entities. Starting with CMake 3.0, these modules are progressively being updated to also provide imported targets where appropriate. For those situations where a project needs to refer to an external tool or library, prefer to do so through an imported target if one is available. These typically do a better job of abstracting away things like platform differences, option-dependent tool selection and so on, but more importantly the usage requirements are then robustly handled by CMake. If there is a choice between using an imported library or a variable to refer to the same thing, prefer to use the imported library wherever possible.

Prefer defining static libraries over object libraries. Static libraries are simpler, have more complete and robust support from earlier CMake versions and they are well understood by most developers. Object libraries have their uses, but they are also less flexible than static libraries. In particular, object libraries cannot be linked at all prior to CMake 3.12 and not robustly before CMake 3.14. Without such linking, they don't support transitive dependencies, which forces projects to manually apply the dependencies themselves. This increases the opportunity for errors and omissions. It also reduces the encapsulation that a library target would normally provide. Even the name itself can cause some confusion among developers, since an object library is not a true library, but rather just a set of uncombined object files, yet developers still sometimes expect it to behave like a real library. The changes with CMake 3.12 blur that distinction, but the remaining differences still leave room for unexpected results, as evidenced by the number of queries relating to object libraries and their transitive behavior on the CMake mailing list and issue tracker.

When it comes to naming targets, don't use target names that are too generic. Globally visible target names must be unique and names may clash with targets from other projects when used in a larger hierarchical arrangement. In addition, consider adding an alias `namespace::…` target for each target that is not private to the project (i.e. every target that may end up being installed or packaged). This allows consuming projects to link to the namespaced target name instead of the real target name, a technique which enables consuming projects to switch between building the child project themselves or using a pre-built installed project relatively easily. While this may initially seem like extra work for not much gain, it is emerging as an expected standard practice among the CMake community, especially for those projects that take a non-trivial amount of time to build. This pattern is discussed further in Section 26.3, "Installing Exports".

Inevitably, at some point it may become desirable to rename or refactor a library, but there may be external projects which expect the existing library targets to be available to link to. In these situations, use an interface target to provide an old name for a renamed target so that those external projects can continue to build and be updated at their convenience. When splitting up a library, define an interface library with the old target name and have it define link dependencies to the new split out libraries. For example:

```
# Old library previously defined like this:
add_library(DeepCompute SHARED ...)
```

Now change `DeepCompute` to an `INTERFACE` library that links to the new refactored libraries to preserve backward compatibility:

Exclusively for sbmain17@gmail.com Transaction: 0098315591

```
# Now refactored into two separate libraries
add_library(ComputeAlgoA SHARED ...)
add_library(ComputeAlgoB SHARED ...)

# Forwarding interface library keeps old projects working
add_library(DeepCompute INTERFACE)
target_link_libraries(DeepCompute INTERFACE
    ComputeAlgoA
    ComputeAlgoB
)
```

Exclusively for sbmain17@gmail.com Transaction: 0098315591

Chapter 18. Custom Tasks

No build tool can ever hope to implement every feature that will ever be needed by any given project. At some point, developers will need to carry out a task that falls outside the directly supported functionality. For example, a special tool may need to be run to produce source files or to post-process a target after it has been built. Files may need to be copied, verified or a hash value computed. Build artifacts may need to be archived or a notification service contacted. These and other tasks don't always fit into a predictable pattern that allows them to be easily provided as a general build system capability.

CMake supports such tasks through custom commands and custom targets. These allow any command or set of commands to be executed at build time to perform whatever arbitrary tasks a project requires. CMake also supports executing tasks at configure time, enabling various techniques that rely on tasks being completed before the build stage or even before processing later parts of the current `CMakeLists.txt` file.

18.1. Custom Targets

Library and executable targets are not the only kinds of targets CMake supports. Projects can also define their own custom targets that perform arbitrary tasks defined as a sequence of commands to be executed at build time. These custom targets are defined using the `add_custom_target()` command:

```
add_custom_target(targetName
    [ALL]
    [command1 [args1...]]
    [COMMAND command2 [args2...]]
    [DEPENDS depends1...]
    [BYPRODUCTS [files...]]
    [WORKING_DIRECTORY dir]
    [COMMENT comment]
    [VERBATIM]
    [USES_TERMINAL]      # Requires CMake 3.2 or later
    [JOB_POOL poolName]  # Requires CMake 3.15 or later
    [SOURCES source1 [source2...]]
)
```

A new target with the specified `targetName` will be available to the build and it will always be considered out of date. The `ALL` option makes the *all* target depend on this new custom target (the various generators name the *all* target slightly differently, but it is generally something like `all`, `ALL` or similar). If the `ALL` option is not provided, then the target is only built if it is explicitly requested or if building some other target that depends on it.

When the custom target is built, the specified command(s) will be executed in the order given, with each command able to have any number of arguments. For improved readability, arguments can be split across multiple lines. The first command does not need to have the `COMMAND` keyword preceding it, but for clarity it is recommended to always include the `COMMAND` keyword even for the first command. This is especially true when specifying multiple commands, since it makes each command use a consistent form.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

Commands can be defined to do anything that could be performed on the host platform. Typical commands involve running a script or a system-provided executable, but they can also run executable targets created as part of the build. If another executable target name is listed as the command to execute, CMake will automatically substitute the built location of that other target's executable. This works regardless of the platform or CMake generator being used, thereby freeing the project from having to work out the various platform and generator differences that lead to a range of different output directory structures, file names, etc. If another target needs to be used as an argument to one of the commands, CMake will not automatically perform the same substitution, but it is trivial to obtain an equivalent substitution with the `TARGET_FILE` generator expression. Projects should take advantage of these features to let CMake provide locations of targets rather than hard-coding paths manually, as this allows the project to robustly support all platforms and generator types with minimal effort. The following example shows how to define a custom target which uses two other targets as part of the command and argument list:

```
add_executable(Hasher hasher.cpp)
add_library(MyLib api.cpp)

add_custom_target(CreateHash
    COMMAND Hasher $<TARGET_FILE:MyLib>
)
```

When a target is used as the command to execute, CMake automatically creates a dependency on that executable target to ensure it is built before the custom target. Similarly, if a target is referred to in one of the following generator expressions anywhere in the command or its arguments, a dependency will also be automatically created on that target:

- `$<TARGET_FILE:⋯>`
- `$<TARGET_LINKER_FILE:⋯>`
- `$<TARGET_SONAME_FILE:⋯>`
- `$<TARGET_PDB_FILE:⋯>`

For CMake 3.18 and earlier, other `$<TARGET_xxx:⋯>` generator expressions will also lead to a dependency being added automatically. With CMake 3.19 or later, the behavior depends on policy CMP0112. See the CMake documentation of that policy for further details. To create a dependency on a target not mentioned in such generator expressions, use the `add_dependencies()` command to define that relationship.

If a dependency exists on a *file* rather than a target, the `DEPENDS` keyword can be used to specify that relationship as part of the `add_custom_target()` call directly. Note that `DEPENDS` should not be used for target dependencies, only file dependencies. The `DEPENDS` keyword is especially useful when the file being listed is generated by some other custom command (see Section 18.3, "Commands That Generate Files" further below), where CMake will set up the necessary dependencies to ensure the other custom commands execute before this custom target's commands. Always use an absolute path for `DEPENDS`, since relative paths can give unexpected results due to a legacy feature that allows path matching against multiple locations.

When multiple commands are provided, each one will be executed in the order listed. A project should not assume any particular shell behavior, however, as each command might run in its own

Exclusively for sbmain17@gmail.com Transaction: 0098315591

separate shell or without any shell environment at all. Custom commands should be defined as though they were being executed in isolation and without any shell features such as redirection, variable substitution, etc., with only command order being enforced. While some of these features may work on some platforms, they are not universally supported. Also, since no particular shell behavior is guaranteed, escaping within the executable names or their arguments may be handled differently on different platforms. To help reduce these differences, the `VERBATIM` option can be used to ensure that the only escaping done is that by CMake itself when parsing the `CMakeLists.txt` file. No further escaping is performed by the platform, so the developer can have confidence in how the command is ultimately constructed for execution. If there is any chance of escaping being relevant, use of the `VERBATIM` keyword is recommended.

The directory in which the commands are executed is the current binary directory by default. This can be changed with the `WORKING_DIRECTORY` option, which can be an absolute path or a relative path, the latter being relative to the current binary directory. This means that using `${CMAKE_CURRENT_BINARY_DIR}` as part of the working directory should not be necessary, since a relative path already implies it.

The `BYPRODUCTS` option can be used to list other files that are created as part of running the command(s). If the Ninja generator is being used, this option is required if another target depends on any of the files created as a by-product of running this set of custom commands. Files listed as `BYPRODUCTS` are marked as `GENERATED` (for all generator types, not just Ninja) which ensures the build tool knows how to correctly handle dependency details related to the by-product files. For cases where a custom target generates files as a by-product, consider whether `add_custom_command()` would be a more appropriate way to define the commands and the things it outputs (see Section 18.3, “Commands That Generate Files”).

With CMake 3.20 or later, a restricted set of generator expressions is supported for `BYPRODUCTS`. Any expression that refers to a target (e.g. `$<TARGET_FILE:…>`) may not be used.

If the commands produce no output on the console, it can sometimes be useful to specify a short message with the `COMMENT` option. The specified message is logged just before running the commands, so if the commands silently fail for some reason, the comment can be a useful marker to indicate where the build failed. Note, however, that for some generators, the comment will not be shown, so this cannot be considered a reliable mechanism, but it may still be useful for those generators that do support it. A universally supported alternative is presented in Section 18.5, “Platform Independent Commands” below.

`USES_TERMINAL` is another console-related option which instructs CMake to give the command direct access to the terminal, if possible. When using the Ninja generator, this has the effect of placing the command in the console pool. This may lead to better output buffering behavior in some situations, such as helping IDE environments capture and present the build output in a more timely manner. It can also be useful if interactive input is required for non-IDE builds. The `USES_TERMINAL` option is supported for CMake 3.2 and later.

To provide even more control over Ninja job pools, CMake 3.15 added support for the `JOB_POOL` option. Whereas `USES_TERMINAL` assigns the task to the console job pool, the `JOB_POOL` option allows the project to assign the task to any custom job pool. `USES_TERMINAL` and `JOB_POOL` cannot both be given. Job pools are a fairly advanced topic that few projects should need. Readers interested in exploring this topic should consult the CMake documentation, starting with the `JOB_POOLS` global property.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

The SOURCES option allows arbitrary files to be listed which will then be associated with the custom target. These files might be used by the commands or they could just be some additional files which are loosely associated with the target, such as documentation, etc. Listing a file with SOURCES has no effect on the build or the dependency relationships, it is purely for the benefit of associating those files with the target so that IDE projects can show them in an appropriate context. This feature is sometimes exploited by defining a dummy custom target and listing sources with no commands just to make them show up in IDE projects. While this works, it does have the disadvantage of creating a build target with no real meaning. Many projects deem this to be an acceptable trade-off, while some developers consider this undesirable or even an anti-pattern.

18.2. Adding Build Steps To An Existing Target

Custom commands sometimes do not require a new target to be defined, they may instead specify additional steps to be performed when building an existing target. This is where add_custom_command() should be used with the TARGET keyword as follows:

```
add_custom_command(TARGET targetName buildStage
    COMMAND command1 [args1...]
    [COMMAND command2 [args2...]]
    [WORKING_DIRECTORY dir]
    [BYPRODUCTS files...]
    [COMMENT comment]
    [VERBATIM]
    [USES_TERMINAL]      # Requires CMake 3.2 or later
    [JOB_POOL poolName]  # Requires CMake 3.15 or later
)
```

Most of the options are very similar to those for add_custom_target(), but instead of defining a new target, the above form attaches the commands to an existing target. That existing target can be an executable or library target, or it can even be a custom target (with some restrictions). The commands will be executed as part of building targetName, with the buildStage argument required to be one of the following:

PRE_BUILD

The commands should be run before any other rules for the specified target. Be aware that only the Visual Studio generator supports this option and only for Visual Studio 7 or later. All other CMake generators will treat this as PRE_LINK instead. Given the limited support for this option, projects should aim for a structure which does not require a PRE_BUILD custom command so as to avoid command ordering differences between generators.

PRE_LINK

The commands will be run after sources are compiled, but before they are linked. For static library targets, the commands will run before the library archiver tool. For custom targets, PRE_LINK is not supported.

POST_BUILD

The commands will be run after all other rules for the specified target. All target types and generators support this option, making it the preferred build stage whenever there is a choice.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

POST_BUILD tasks are relatively common, but PRE_LINK and PRE_BUILD are rarely needed since they can usually be avoided by using the OUTPUT form of add_custom_command() instead (see the next section).

Multiple calls to add_custom_command() can be made to append multiple sets of custom commands to a particular target. This can be useful, for example, to have some commands run from one working directory and other commands run from somewhere else.

```
add_executable(MyExe main.cpp)

add_custom_command(TARGET MyExe POST_BUILD
    COMMAND script1 $<TARGET_FILE:MyExe>
)

# Additional command which will run after the above from a different directory
add_custom_command(TARGET MyExe POST_BUILD
    COMMAND writeHash $<TARGET_FILE:MyExe>
    BYPRODUCTS ${CMAKE_BINARY_DIR}/verify/MyExe.md5
    WORKING_DIRECTORY ${CMAKE_BINARY_DIR}/verify
)
```

18.3. Commands That Generate Files

Defining commands as additional build steps for a target covers many common use cases. Sometimes, however, a project needs to create one or more files by running a command or series of commands and the generation of that file doesn't really belong to any existing target. This is where the OUTPUT form of add_custom_command() can be used. It implements all of the same options as the TARGET form as well as some additional options related to dependency handling and appending to a previous OUTPUT command set.

```
add_custom_command(OUTPUT output1 [output2...]
    COMMAND command1 [args1...]
    [COMMAND command2 [args2...]]
    [WORKING_DIRECTORY dir]
    [BYPRODUCTS files...]
    [COMMENT comment]
    [VERBATIM]
    [USES_TERMINAL]      # Requires CMake 3.2 or later
    [JOB_POOL poolName]  # Requires CMake 3.15 or later
    [APPEND]
    [DEPENDS [depends1...]
    [MAIN_DEPENDENCY depend]
    [IMPLICIT_DEPENDS <lang1> depend1 [<lang2> depend2...]]
    [DEPFILE depfile]
)
```

Instead of specifying a target and pre/post build stage, this form requires one or more output file names to be given after the OUTPUT keyword. CMake will then interpret the commands as a recipe for generating the named output files. If the output files are specified with no path or with a relative path, they are relative to the current binary directory. With CMake 3.20 or later, generator expressions that do not refer to a target can be used.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

On its own, this form won't result in the output files being built, since no target is defined. If, however, some other target defined in the same directory scope depends on any of the output files, CMake will automatically create dependency relationships that ensure the output files are generated before the target that needs them. A common error is to try to make a target defined in a different directory scope depend on the output of an `add_custom_command()`, but this is not supported. Furthermore, only one target should depend on any of the output files or else parallel builds may try to invoke the custom command multiple times simultaneously to satisfy the dependencies of multiple targets.

The target that depends on the output of the `add_custom_command()` can be an ordinary executable, a library target or it can even be a custom target. In fact, it is quite common for a custom target to be defined simply to provide a way for the developer to trigger the custom command. The following variation on the hashing example of the preceding section demonstrates the technique:

```
add_executable(MyExe main.cpp)

# Output file with relative path, generated in the build directory
add_custom_command(OUTPUT MyExe.md5
    COMMAND writeHash $<TARGET_FILE:MyExe>
)

# Absolute path needed for DEPENDS, otherwise relative to source directory
add_custom_target(ComputeHash
    DEPENDS ${CMAKE_CURRENT_BINARY_DIR}/MyExe.md5
)
```

When defined this way, building the *MyExe* target will not result in running the hashing step, unlike the earlier example which added the hashing command as a `POST_BUILD` step of the *MyExe* target. Instead, hashing will only be performed if the developer explicitly requests it as a build target. This allows optional steps to be defined and invoked when needed instead of always being run, which can be quite useful if the additional steps are time consuming or won't always be relevant.

Of course, `add_custom_command()` can also be used to generate files consumed by existing targets, such as generating source files. In the following example, an executable built by the project is used to generate a source file which is then compiled as part of another executable.

```
add_executable(Generator generator.cpp)

add_custom_command(OUTPUT onTheFly.cpp
    COMMAND Generator
)

add_executable(MyExe ${CMAKE_CURRENT_BINARY_DIR}/onTheFly.cpp)
```

CMake automatically recognizes that `MyExe` needs the source file generated by the custom command, which in turn requires the `Generator` executable. Asking for the `MyExe` target to be built will result in the `Generator` and the generated source file being built before building `MyExe`. Note, however, that this dependency relationship has limitations. Consider the following scenario:

Exclusively for sbmain17@gmail.com Transaction: 0098315591

- The `onTheFly.cpp` file initially does not exist.
- Build the `MyExe` target, which results in the following sequence:
  - The `Generator` target is brought up to date.
  - The custom command is executed to create `onTheFly.cpp`.
  - The `MyExe` target is built.
- Now modify the `generator.cpp` file.
- Build the `MyExe` target again, which this time results in the following sequence:
  - The `Generator` target is brought up to date. This will cause the `Generator` executable to be rebuilt because its source file was modified.
  - The custom command is NOT executed, since `onTheFly.cpp` already exists.
  - The `MyExe` target is NOT rebuilt because its source file remains unchanged.

One might intuitively expect that if the `Generator` target is rebuilt, then the custom command should also be re-run. The dependency CMake automatically creates does not enforce this, it creates a weaker dependency which does ensure `Generator` is brought up to date but the custom command is only run if the output file is missing altogether. In order to force the custom command to be re-run if the `Generator` target is rebuilt, an explicit dependency has to be specified rather than relying on the dependency CMake automatically creates.

Dependencies can be manually specified with the `DEPENDS` option. Items listed with `DEPENDS` can be CMake targets or files (compare this with the `DEPENDS` option for `add_custom_target()` which can only list files). If a target is listed, it will be brought up to date any time the custom command's output files are required to be brought up to date. Similarly, if a listed file is modified, the custom command will be executed if anything requires any of the custom command's output files. Furthermore, if any listed file is itself an output file of another custom command in the same directory scope, that other custom command will be executed first. As for `add_custom_target()`, always use an absolute path if listing a file for `DEPENDS` to avoid ambiguous legacy behavior.

While CMake's automatic dependencies may seem convenient, in practice the project will still typically need to list out all the required targets and files in a `DEPENDS` section to ensure that the full dependency relationships are adequately specified. It can be easy to omit the `DEPENDS` section by mistake, since the first build will run the custom command to create the missing output files and the build will appear to be behaving correctly. Subsequent builds will not re-run the custom command unless the output file is removed, even if any of the automatically detected dependency targets are rebuilt. This can be easy to miss, often going undetected for a long time in complex projects until a developer encounters the situation and tries to work out why something isn't being rebuilt when it was expected to be. Therefore, developers should expect that a `DEPENDS` section will typically be needed unless the custom command doesn't require anything created by the build or any of the project's source files.

Another common error is to not create a dependency on a file that is needed by the custom command, but which isn't listed as part of the command line to be executed. Such files need to appear in a `DEPENDS` section for the build to be considered robust.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

There are a few more dependency-related options supported by `add_custom_command()`. The `MAIN_DEPENDENCY` option is intended to identify a source file which should be considered the main dependency of the custom command. It has mostly the same effect as `DEPENDS` for the listed file, but some generators may apply additional logic such as where to place the custom command in an IDE project. An important distinction to note is that if a source file is listed as a `MAIN_DEPENDENCY`, then the custom command becomes a replacement for how that source file would normally be compiled. This can lead to some unexpected results. Consider the following example:

```
add_custom_command(OUTPUT transformed.cpp
    COMMAND transform
            ${CMAKE_CURRENT_SOURCE_DIR}/original.cpp
            transformed.cpp
    MAIN_DEPENDENCY
            ${CMAKE_CURRENT_SOURCE_DIR}/original.cpp
)
add_executable(Original    original.cpp)
add_executable(Transformed transformed.cpp)
```

The above would lead to a linker error for the `Original` target because `original.cpp` would not be compiled to an object file, so there would be no object files at all (and therefore no `main()` function). Instead, the build tool would treat `original.cpp` as an input file used to create `transformed.cpp`. The problem can be fixed by using `DEPENDS` instead of `MAIN_DEPENDENCY`, as this would preserve the same dependency relationship, but it would not result in the default compilation rule for the `original.cpp` source file being replaced.

The other two dependency-related options, `IMPLICIT_DEPENDS` and `DEPFILE`, are not universally supported by all project generators. `IMPLICIT_DEPENDS` directs CMake to invoke a C or C++ scanner to determine dependencies of the listed files. It is ignored for all but Makefile generators, so projects should generally avoid it if other alternatives are available for expressing the necessary dependencies. `DEPFILE` can be used to provide a `*.d` dependency file (which the project is responsible for generating), but up until CMake 3.19, only the Ninja generator supported it. From CMake 3.20, `DEPFILE` can also be used with Makefile generators, while CMake 3.21 added support for the Xcode and Visual Studio generators. While depfiles have their uses, they are more complex to work with and they shouldn't need to be manually managed for most typical projects. `IMPLICIT_DEPENDS` and `DEPFILE` cannot be used together.

CMake 3.20 also introduced another change related to `DEPFILE` which may affect projects that were using that functionality with earlier CMake versions. CMake 3.20 added policy `CMP0116` which, unlike most policies, can result in warnings even where the project is not invoking or relying on the `OLD` behavior. The warning draws attention to the changed handling of relative paths when `DEPFILE` is used with a call to `add_custom_command()` anywhere other than in the top level source directory. CMake cannot reliably check the contents of a depfile, since it can be and usually is updated at build time. Therefore, it conservatively issues a warning unless the project has been updated to ensure that policy `CMP0116` is set to `NEW`. Projects that were using `DEPFILE` with CMake 3.19 or earlier should be checked to ensure that absolute paths are used in accordance with the `CMP0116` policy requirements. They can then be updated to set the policy to avoid the warning. This can be done globally by adjusting the version range passed to the `cmake_minimum_required()` call, or locally around the call in question.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

The following example demonstrates how to adjust the policy setting locally without affecting the minimum CMake version requirement (see Chapter 12, *Policies* for further details):

```
# Give ourselves a local policy set we can safely modify
cmake_policy(PUSH)

# Use the NEW policy setting only if it is available
if(POLICY CMP0116)
    cmake_policy(SET CMP0116 NEW)
endif()

# We guarantee that depfile.d will not use relative paths
# for any dependency it specifies
add_custom_command(OUTPUT ...
    DEPFILE /some/absolute/path/to/depfile.d
    ...
)

# Restore the original policy settings
cmake_policy(POP)
```

The `OUTPUT` and `TARGET` forms also have slightly different behavior when it comes to appending more dependencies or commands to the same output file or target. For the `OUTPUT` form, the `APPEND` keyword must be specified and the first `OUTPUT` file listed must be the same for the first and subsequent calls to `add_custom_command()`. Only `COMMAND` and `DEPENDS` can be used for the second and subsequent calls for the same output file. The other options, such as `MAIN_DEPENDENCY`, `WORKING_DIRECTORY` and `COMMENT`, are ignored when the `APPEND` keyword is present. In contrast, for the `TARGET` form, no `APPEND` keyword is necessary for second and subsequent calls to `add_custom_command()` for the same target. The `COMMENT` and `WORKING_DIRECTORY` options can also be specified for each call and they will take effect for the commands being added in that call.

18.4. Configure Time Tasks

Both `add_custom_target()` and `add_custom_command()` define commands to be executed during the build stage. This is typically when custom commands should be run, but there are some situations where a custom task needs to be performed during the configure stage instead. Some examples of when this is needed include:

- Executing external commands to obtain information to be used during configuration. The command output is often captured directly into CMake variables for further processing.
- Writing or touching files which need to be updated any time CMake is re-run.
- Generation of `CMakeLists.txt` or other files which need to be included or processed as part of the current configure step.

CMake provides the `execute_process()` command for running tasks like these during the configure stage:

Exclusively for sbmain17@gmail.com Transaction: 0098315591

```
execute_process(
    COMMAND command1 [args1...]
    [COMMAND command2 [args2...]]
    [WORKING_DIRECTORY directory]
    [RESULT_VARIABLE resultVar]
    [RESULTS_VARIABLE resultsVar]
    [OUTPUT_VARIABLE outputVar]
    [ERROR_VARIABLE errorVar]
    [OUTPUT_STRIP_TRAILING_WHITESPACE]
    [ERROR_STRIP_TRAILING_WHITESPACE]
    [INPUT_FILE inFile]
    [OUTPUT_FILE outFile]
    [ERROR_FILE errorFile]
    [OUTPUT_QUIET]
    [ERROR_QUIET]
    [TIMEOUT seconds]

    # CMake 3.15 or later required:
    [COMMAND_ECHO STDOUT | STDERR | NONE]

    # CMake 3.18 or later required:
    [ECHO_OUTPUT_VARIABLE]
    [ECHO_ERROR_VARIABLE]

    # CMake 3.19 or later required:
    [COMMAND_ERROR_IS_FATAL ANY | LAST]
)
```

Similar to `add_custom_command()` and `add_custom_target()`, one or more `COMMAND` sections specify the tasks to be executed and the `WORKING_DIRECTORY` option can be used to control where those commands are run. The commands are passed to the operating system for execution as is with no intermediate shell environment. Therefore, features like input/output redirection and environment variables are not supported. The commands run immediately.

If multiple commands are given, they are executed in order, but instead of being fully independent from each other, the standard output from one command is piped to the input of the next. In the absence of any other options, the output of the *last* command is sent to the output of the CMake process itself but the standard error of *every* command is sent to the standard error stream of the CMake process.

The standard output and standard error streams can be captured and stored in variables instead of being sent to the default pipes. The output of the last command in the set of commands can be captured by specifying the name of a variable to store it in with the `OUTPUT_VARIABLE` option. Similarly, the standard error streams of all commands can be stored in the variable named by the `ERROR_VARIABLE` option. Passing the same variable name to both of these options will result in the standard output and standard error being merged just as they would be if outputting to a terminal, with the merged result being stored in the named variable. With CMake 3.18 or later, the `ECHO_OUTPUT_VARIABLE` and `ECHO_ERROR_VARIABLE` options can be added to echo the output and error streams while also capturing them to variables. This can be useful for long-running commands where seeing progress in the output helps to confirm that the command has not hung.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

If the OUTPUT_STRIP_TRAILING_WHITESPACE option is present, any trailing whitespace will be omitted from the content stored in the output variable. The ERROR_STRIP_TRAILING_WHITESPACE option does a similar thing for the content stored in the error variable. If using the output or error variables' contents for any sort of string comparison, a common problem is failing to account for trailing whitespace, so its removal is often desirable.

Instead of capturing the output and error streams in variables, they can be sent to files. The OUTPUT_FILE and ERROR_FILE options can be used to specify the names of files to send the streams to. Just like the variable-focused options, specifying the same file name for both results in a merged stream. In addition, a file can be specified for the input stream to the first command with the INPUT_FILE option. Note, however, that the OUTPUT_STRIP_TRAILING_WHITESPACE and ERROR_STRIP_TRAILING_WHITESPACE options have no effect on content sent to files. There is also no ability to echo the output or error streams when capturing to files.

The same stream cannot be captured in a variable and sent to a file at the same time. It is possible, however, to send different streams to different places, such as the output stream to a variable and the error stream to a file or vice versa. It is also possible to silently discard the content of a stream altogether with the OUTPUT_QUIET and ERROR_QUIET options. These options can be useful if just success or failure of a command is of interest.

Success or failure of the set of commands can be captured using the RESULT_VARIABLE option. The result of running the commands will be stored in the named variable as either an integer return code of the last command or a string containing some kind of error message. The if() command conveniently treats both non-empty error strings and integer values other than 0 as boolean true (unless a project is unlucky enough to have an error string that satisfies one of the special cases, see Section 6.1.1, "Basic Expressions"). Therefore, checking for the success of a call to execute_process() is generally relatively simple:

```
execute_process(
    COMMAND runSomeScript
    RESULT_VARIABLE result
)
if(result)
    message(FATAL_ERROR "runSomeScript failed: ${result}")
endif()
```

From CMake 3.10, if the result of each individual command is required rather than just the last one, the RESULTS_VARIABLE option can be used instead. This option stores the result of each command in the variable named by resultsVar as a list.

With CMake 3.19 or later, the COMMAND_ERROR_IS_FATAL option can be used as a more concise way of halting with an error if the command fails. It avoids the need to receive the result in a variable and perform an explicit check, since the check is performed by the execute_process() command directly. The option must be followed by either ANY or LAST. When more than one COMMAND is given, specifying ANY will cause the execute_process() command to fail with a fatal error if any of the commands fail. If LAST is given instead, then execute_process() only fails if the last COMMAND fails. If only one COMMAND is given, then ANY and LAST are equivalent.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

```
# Automatically halt with an error if either command fails
execute_process(
    COMMAND runSomeScript
    COMMAND runSomethingElse
    COMMAND_ERROR_IS_FATAL ANY
)
```

The `TIMEOUT` option can be used to handle commands which may run longer than expected or which might possibly never complete. This ensures the configure step doesn't block indefinitely and allows an unexpectedly long configure step to be treated as an error. Note, however, that the `TIMEOUT` option on its own won't cause CMake to halt and report an error. It is still necessary to either check the result using `RESULT_VARIABLE` and an `if()` test, or provide the `COMMAND_ERROR_IS_FATAL` option. Note that in the initial CMake 3.19.0 release, if a command timed out, it was not caught and treated as a fatal error by the `COMMAND_ERROR_IS_FATAL` option. That was fixed in CMake 3.19.2, so consider this to be the minimum CMake version if using this option. If using the `RESULT_VARIABLE` method, the result variable will hold an error string indicating the command was terminated due to timeout if it runs for too long, so printing it in the error message is useful.

CMake 3.15 added support for the `COMMAND_ECHO` option, which must be followed by one of `STDOUT`, `STDERR` or `NONE`. This controls where to echo each `COMMAND` (the command line itself, not the command's output), or in the case of `NONE`, prevents commands from being echoed. If the `COMMAND_ECHO` option is not present, the default behavior is determined by the `CMAKE_EXECUTE_PROCESS_COMMAND_ECHO` variable, which supports the same three values. If that variable isn't defined either or the CMake version is 3.14 or earlier, commands are not echoed.

When CMake executes the commands, the child process largely inherits the same environment as the main process. An important exception to this is that the first time CMake is run on a project, the `CC` and `CXX` environment variables of the child process are explicitly set to the C and C++ compilers being used by the main build (if the main project has enabled the C and C++ languages). For subsequent CMake runs, the `CC` and `CXX` environment variables are *not* substituted in this way, which can lead to unexpected results if the commands perform actions that rely on `CC` and/or `CXX` having the same values every time `execute_process()` is called. This undocumented behavior has existed since early versions of CMake, even as far back as the now deprecated `exec_program()` command which `execute_process()` replaced. It was added to facilitate child processes being able to configure and run sub-builds with the same compilers as the main project. In some cases, however, the child process might not want the compiler to be preserved, such as when the main build is cross-compiling but the child process should use the default host compilers. In such cases, projects can set a variable named `CMAKE_GENERATOR_NO_COMPILER_ENV` to a boolean true value and then CMake will not set `CC` and `CXX` for any `execute_process()` call, even the initial invocation.

18.5. Platform Independent Commands

The `add_custom_command()`, `add_custom_target()` and `execute_process()` commands provide projects with a great deal of freedom. Any task not already directly supported by CMake can be implemented using commands provided by the host operating system instead. These custom commands are inherently platform specific, which works against one of the main reasons many projects use CMake in the first place, i.e. to abstract away platform differences or to at least support a range of platforms with minimal effort.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

A large proportion of custom tasks are related to file system manipulation. Creating, deleting, renaming or moving files and directories form the bulk of these tasks, but the commands to do so vary between operating systems. As a result, projects often end up using `if-else` conditions to define the different platforms' versions of the same command, or worse, they only bother to implement the commands for some platforms. Many developers are not aware that CMake provides a command mode which abstracts away many of these platform specific tasks:

```
cmake -E cmd [args...]
```

The full set of supported commands can be listed using `cmake -E help`, but some of the more commonly used ones include:

- `compare_files`
- `copy`
- `copy_directory`
- `copy_if_different`
- `echo`
- `env`
- `make_directory`
- `md5sum`
- `remove` (deprecated from CMake 3.17)
- `remove_directory` (deprecated from CMake 3.17)
- `rename`
- `rm` (requires CMake 3.17 or later)
- `tar`
- `time`
- `touch`

Consider the example of a custom task to remove a particular directory and all its contents:

```
set(discardDir "${CMAKE_CURRENT_BINARY_DIR}/private")

# Naive platform specific implementation (not robust)
if(WIN32)
    add_custom_target(MyCleanup COMMAND rmdir /S /Q "${discardDir}")
elseif(UNIX)
    add_custom_target(MyCleanup COMMAND rm -rf "${discardDir}")
else()
    message(FATAL_ERROR "Unsupported platform")
endif()

# Platform independent equivalent
add_custom_target(MyCleanup COMMAND "${CMAKE_COMMAND}" -E rm -R "${discardDir}")
```


Exclusively for sbmain17@gmail.com Transaction: 0098315591

The platform-specific implementation shows how projects often try to implement this scenario, but the if-else conditions are testing the *target* platform rather than the *host* platform. In a cross compiling scenario, this may result in the wrong command being used. The platform independent version has no such weakness. It always selects the right command for the *host* platform.

The example also shows how to invoke the `cmake` command correctly. The `CMAKE_COMMAND` variable is populated by CMake and it contains the full path to the `cmake` executable being used in the main build. Using `CMAKE_COMMAND` in this way ensures that the same version of CMake is also used for the custom command. The `cmake` executable does not have to be on the current `PATH` and if multiple versions of CMake are installed, the correct version is always used, regardless of which one might otherwise have been selected based on the user's `PATH`. It also ensures the build uses the same CMake version during the build stage as was used in the configure stage, even if the user's `PATH` environment variable changes.

Earlier in this chapter, it was noted that the `COMMENT` option for `add_custom_target()` and `add_custom_command()` isn't always reliable. Instead of using `COMMENT`, projects can use the `-E echo` command to intersperse comments anywhere in a sequence of custom commands:

```
set(discardDir "${CMAKE_CURRENT_BINARY_DIR}/private")
add_custom_target(MyCleanup
    COMMAND ${CMAKE_COMMAND} -E echo "Removing ${discardDir}"
    COMMAND ${CMAKE_COMMAND} -E rm -R "${discardDir}"
    COMMAND ${CMAKE_COMMAND} -E echo "Recreating ${discardDir}"
    COMMAND ${CMAKE_COMMAND} -E make_directory "${discardDir}"
)
```

CMake's command mode is a very useful way of carrying out a range of common tasks in a platform independent way. Sometimes, however, more complex logic is required and such custom tasks are often implemented using platform specific shell scripts. An alternative is to use CMake itself as a scripting engine, providing a platform independent language in which to express arbitrary logic. The `-P` option to the `cmake` command puts CMake into script processing mode:

```
cmake [options] -P filename
```

The `filename` argument is the name of the CMake script file to execute. The usual `CMakeLists.txt` syntax is supported, but there is no configure or generate step and the `CMakeCache.txt` file is not updated. The script file is essentially processed as just a set of commands rather than as a project, so any commands which relate to build targets or project-level features are not supported. Nonetheless, script mode allows complex logic to be implemented and it comes with the advantage of not requiring any additional shell interpreter to be installed.

While script mode doesn't support command line options like ordinary shells or command interpreters, it does support passing in variables with `-D` options, just like ordinary `cmake` invocations. Since no `CMakeCache.txt` file is updated in script mode, `-D` options can be used freely without affecting the main build's cache. Such options must be placed before `-P`.

```
cmake -DOPTION_A=1 -DOPTION_B=foo -P myCustomScript.cmake
```

Exclusively for sbmain17@gmail.com Transaction: 0098315591

18.6. Combining The Different Approaches

The following example demonstrates many of the features discussed in this chapter:

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.0)
project(Example)

# This executable generates files in a directory passed as a command line argument
add_executable(GenerateFiles generateFiles.cpp)

# Custom target to run the above executable and archive its results
set(outDir "foo")
add_custom_target(Archiver
    COMMAND ${CMAKE_COMMAND} -E echo "Archiving files"
    COMMAND ${CMAKE_COMMAND} -E rm -R "${outDir}"
    COMMAND ${CMAKE_COMMAND} -E make_directory "${outDir}"
    COMMAND GenerateFiles "${outDir}"
    COMMAND ${CMAKE_COMMAND} "-DTAR_DIR=${outDir}"
            -P "${CMAKE_CURRENT_SOURCE_DIR}/archiver.cmake"
)
```

archiver.cmake

```
cmake_minimum_required(VERSION 3.0)
if(NOT TAR_DIR)
    message(FATAL_ERROR "TAR_DIR must be set")
endif()

# Create an archive of the directory
set(archive archive.tar)
execute_process(
    COMMAND ${CMAKE_COMMAND} -E tar cf ${archive} "${TAR_DIR}"
    RESULT_VARIABLE result
)
if(result)
    message(FATAL_ERROR "Archiving ${TAR_DIR} failed: ${result}")
endif()

# Compute MD5 checksum of the archive
execute_process(
    COMMAND ${CMAKE_COMMAND} -E md5sum ${archive}
    OUTPUT_VARIABLE md5output
    RESULT_VARIABLE result
)
if(result)
    message(FATAL_ERROR "Unable to compute md5 of archive: ${result}")
endif()

# Extract just the checksum from the output
string(REGEX MATCH "^ *[^ ]*" md5sum "${md5output}")
message("Archive MD5 checksum: ${md5sum}")
```


Exclusively for sbmain17@gmail.com Transaction: 0098315591

The above shows how custom tasks can be specified in different ways. An important aspect of the example is how it accomplishes non-trivial things without having to resort to platform specific commands or functionality.

By putting the archiving logic in the separate `archiver.cmake` file, it can also be used either within a project as shown above, or it can be invoked on its own through CMake's script mode. This can be useful from a development and testing perspective, or to provide a platform-independent way of archiving a directory for general use outside of any project.

18.7. Recommended Practices

When custom tasks need to be executed, it is preferable that they be done during the build stage rather than the configure stage. A fast configure stage is important because it can be invoked automatically when some files are modified (e.g. any `CMakeLists.txt` file in the project, any file included by a `CMakeLists.txt` file or any file listed as a source of a `configure_file()` command as discussed in the next chapter). For this reason, prefer to use `add_custom_target()` or `add_custom_command()` instead of `execute_process()` if there is a choice. If the task needs to run immediately or as part of a `cmake -P` script mode invocation, then `execute_process()` is appropriate.

It is relatively common to see platform specific commands used with `add_custom_command()`, `add_custom_target()` and `execute_process()`. Quite often, however, such commands can instead be expressed in a platform independent manner using CMake's command mode (`-E`). Where possible, the use of platform independent commands should be preferred. In addition, CMake can be used as a platform independent scripting language, processing a file as a sequence of CMake commands when invoked with the `-P` option. The use of CMake scripts instead of a platform specific shell or a separately installed script engine can reduce the complexity of the project and reduce the additional dependencies it requires in order to build. Specifically, consider whether CMake's script mode would be a better choice than using a Unix shell script or Windows batch file, or even a script for a language like Python, Perl etc. which may not be available by default on all platforms. The next chapter shows how to manipulate files directly with CMake instead of having to resort to such tools and methods.

When implementing custom tasks, try to avoid features that lack support in all situations:

- Prefer to use command mode `-E echo` rather than the `COMMENT` keyword with `add_custom_command()` and `add_custom_target()`.
- Try to avoid using `PRE_BUILD` with the `TARGET` form of `add_custom_command()`.
- Consider whether using `IMPLICIT_DEPENDS` or `DEPFILE` options with `add_custom_command()` is worth the generator-specific behavior.
- Avoid listing a source file as a `MAIN_DEPENDENCY` in `add_custom_command()` unless the intention is to replace the default build rule for that source file.

Pay special attention to dependencies for the inputs and outputs of custom tasks. Ensure that *all* files created by `add_custom_command()` are listed as `OUTPUT` files. When listing build targets as the command or arguments in a call to `add_custom_command()` or `add_custom_target()`, prefer to explicitly list them as `DEPENDS` items rather than relying on CMake's automatic target dependency handling. The weaker automatic dependencies may not enforce all the relationships that developers may

Exclusively for sbmain17@gmail.com Transaction: 0098315591

intuitively expect. If listing a file in `DEPENDS` for either `add_custom_target()` or `add_custom_command()`, always use an absolute path to avoid non-robust legacy path matching behavior.

When calling `execute_process()`, most of the time the success of the command should be tested by capturing the result using `RESULT_VARIABLE` and testing it with the `if()` command. This includes when a `TIMEOUT` option is being used, since `TIMEOUT` on its own will not generate an error, it will only ensure the command doesn't run longer than the nominated timeout period.

In certain types of projects, the difference between executing custom commands with optimized targets versus non-optimized targets may have a noticeable effect on build times. A common example of this is where custom commands are used to run code generators that are themselves built as part of the project. If code generation is a non-trivial process, it may be desirable to have the code generators build with the Release configuration even when building the rest of the project as Debug. The Ninja Multi-Config generator introduced with CMake 3.17 is the only generator which directly supports this workflow. Given the fairly new status of this generator, consider carefully before relying heavily on this capability. The interested reader should consult the CMake documentation for further details on this and other related advanced features of the Ninja Multi-Config generator.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

Chapter 19. Working With Files

Many projects need to manipulate files and directories as part of the build. While such manipulations range from trivial through to quite complex, the more common tasks include:

- Constructing paths or extracting components of a path.
- Obtaining a list of files from a directory.
- Copying files.
- Generating a file from string contents.
- Generating a file from another file's contents.
- Reading in the contents of a file.
- Computing a checksum or hash of a file.

CMake provides a variety of features related to working with files and directories. In some cases, there can be multiple ways of achieving the same thing, so it is useful to be aware of the different choices and understand how to use them effectively. A number of these features are frequently misused, some due to such misuse being prevalent in online tutorials and examples, leading to the belief that it is the right way to do things. Some of the more problematic anti-patterns are discussed in this chapter.

Much of CMake's file-related functionality is provided by the `file()` command, with a few other commands offering alternatives better suited to certain situations or providing related helper capabilities. CMake's command mode, which was introduced in the previous chapter, also provides a variety of file-related features which overlap with much of what `file()` provides, but it covers a complimentary set of scenarios to `file()` rather than being an alternative in most cases.

19.1. Manipulating Paths

One of the most basic parts of file handling is manipulating file names and paths. Projects often need to extract file names, file suffixes, etc. from full paths, or convert between absolute and relative paths. With CMake 3.19 and earlier, this functionality is spread across two commands, `get_filename_component()` and `file()`. The two have some overlap and there are inconsistencies. CMake 3.20 introduced a new command, `cmake_path()` which supersedes most of the path-handling capabilities of those two commands. It provides a more consistent, more predictable interface.

19.1.1. cmake_path()

The official documentation for `cmake_path()` is fairly comprehensive. It covers the concepts used, presents the various sub-commands in logical, task-based groups and provides numerous examples. The reader is encouraged to study the material presented therein for a deeper understanding of the command. Only some key concepts and sub-commands for more common tasks are described here.

The `cmake_path()` command never accesses the underlying file system. It only operates on paths syntactically, so it knows nothing about symbolic links or the existence of a path. It uses a clearly

Exclusively for sbmain17@gmail.com Transaction: 0098315591

defined path structure where forward slashes are always used as directory separators, regardless of the host or target platform. A drive letter or mapped drive name (e.g. `C:` or `//myserver`) is supported only if the host platform supports it.

The following example path and table demonstrate the terminology used:

```
C:/one/two/start.middle.end
```

Path component	Example
ROOT_NAME	C:
ROOT_DIRECTORY	/
ROOT_PATH	C:/
FILENAME	start.middle.end
EXTENSION	.middle.end
STEM	start
RELATIVE_PART	one/two/start.middle.end
PARENT_PATH	C:/one/two

The `GET` sub-command can be used to retrieve any of the above path components (replace `<COMP>` with one of the items from the first column in the above table):

```
cmake_path(GET pathVar <COMP> [LAST_ONLY] outVar)
```

`LAST_ONLY` can only be given if `<COMP>` is either `EXTENSION` or `STEM`. By default, an `EXTENSION` starts at the left-most dot (`.`) character of the `FILENAME`, but `LAST_ONLY` changes this to use the right-most dot character instead. The `STEM` is the `FILENAME` without the `EXTENSION`, with the `LAST_ONLY` keyword changing the meaning of `EXTENSION` in this case as well.

```
set(path "a.b.c")

cmake_path(GET path EXTENSION result)           # .b.c
cmake_path(GET path EXTENSION LAST_ONLY result)  # .c

cmake_path(GET path STEM result)                # a
cmake_path(GET path STEM LAST_ONLY result)      # a.b
```

As demonstrated in the above example, note that `pathVar` is the name of a variable holding a path, it is not a string:

```
# WRONG: Cannot use a string for the path
cmake_path(GET "/some/path/example" FILENAME result)

# Correct, but can be improved (see below)
set(path "/some/path/example")
cmake_path(GET path FILENAME result)
```


Exclusively for sbmain17@gmail.com Transaction: 0098315591

While using `set()` to create a path variable is permitted, `cmake_path()` has a dedicated `SET` sub-command which is more robust and has additional features:

```
cmake_path(SET pathVar [NORMALIZE] input)
```

The main advantage of using `cmake_path(SET)` over `set()` is that the former automatically converts a native path to a cmake-style path, as expected by other `cmake_path()` sub-commands. Another advantage is the ability to normalize the path. The CMake documentation describes the formal rules for what the `NORMALIZE` keyword implies, but essentially it means to simplify a path by resolving things like `.` and `..`, collapsing multiple consecutive path separators (`/`) down to a single separator and a few other special cases. Note that because `cmake_path()` never accesses the file system, it does not resolve symbolic links as part of normalization.

```
cmake_path(SET path NORMALIZE "/some//path/xxx/../example")

# The path variable now holds the value: /some/path/example
```

One can also query a path to see if it has a particular path component:

```
cmake_path(<OP> pathVar outVar)
```

Valid values for `<OP>` include all the same `<COMP>` values for `GET`, except prefixed with `HAS_` (e.g. `HAS_EXTENSION`, `HAS_RELATIVE_PART`). In addition, `IS_ABSOLUTE` and `IS_RELATIVE` are also supported values for `<OP>`, but they have some less obvious behavioral aspects to be aware of. `IS_ABSOLUTE` technically means that the path unambiguously refers to a location without needing to refer to some relative point. The consequences of this are mostly relevant for Windows hosts and paths, since a path has to start at the root `/` and also have a drive letter to be considered absolute. On non-Windows host platforms, paths having a drive letter are considered malformed. The same path can therefore yield different results on different host platforms. This is especially dangerous if cross-compiling on a Windows host and testing paths intended for a non-Windows target platform.

The following table shows the result of `cmake_path(IS_ABSOLUTE pathVar result)` for different cases:

pathVar	Windows host	Other host platforms
`C:/somewhere`	Absolute	Undefined
`C:somewhere`	Relative	Undefined
`/somewhere`	Relative	Absolute

On all platforms, the result of `IS_RELATIVE` is always the opposite of `IS_ABSOLUTE`, so it exhibits the same platform-dependent differences.

There are sub-commands for converting to absolute or relative paths:

Exclusively for sbmain17@gmail.com Transaction: 0098315591

```
cmake_path(RELATIVE_PATH pathVar
    [BASE_DIRECTORY baseDir]
    [OUTPUT_VARIABLE outVar]
)
cmake_path(ABSOLUTE_PATH pathVar [NORMALIZE]
    [BASE_DIRECTORY baseDir]
    [OUTPUT_VARIABLE outVar]
)
```

Relative paths are considered relative to the specified `baseDir`, if given, or `CMAKE_CURRENT_SOURCE_DIR` otherwise. If no `outVar` is given, the `pathVar` is modified in-place. The `NORMALIZE` keyword has the usual effect of normalizing the resultant path. One can also explicitly normalize a path with another sub-command:

```
cmake_path(NORMAL_PATH pathVar [OUTPUT_VARIABLE outVar])
```

Across all of CMake's file handling, most of the time a project can use forward slashes for directory separators on all platforms and CMake will do the right thing, converting to native paths as necessary on the project's behalf. Occasionally, however, a project may need to explicitly convert between CMake and native paths. One such example is when working with custom commands and needing to pass a path to a script which requires native paths. For these situations, the `NATIVE_PATH` sub-command can be used:

```
cmake_path(NATIVE_PATH pathVar [NORMALIZE] outVar)
```

There are other `cmake_path()` sub-commands, but the ones presented above cover the most common use cases.

19.1.2. Older Commands

For projects that must support CMake 3.19 or earlier, the `cmake_path()` command cannot be used. The same capabilities are more or less available with the much older `get_filename_component()` and `file()` commands though, albeit in less consistent syntactic forms. These two commands also potentially access the file system as part of resolving paths, unlike `cmake_path()` which doesn't.

With CMake 3.19 and earlier, the primary method for performing path-related operations is the `get_filename_component()` command. It has three different forms. The first form allows for the extraction of the different parts of a path or file name, similar to the functionality provided by `cmake_path(GET)`:

```
get_filename_component(outVar input component [CACHE])
```

The result of the call is stored in the variable named by `outVar`. The component to extract from `input` is specified by `component`, which must be one of the following:

Exclusively for sbmain17@gmail.com Transaction: 0098315591

DIRECTORY

Extract the path part of `input` without the file name. Prior to CMake 2.8.12, this option used to be `PATH`, which is still accepted as a synonym for `DIRECTORY` to preserve compatibility with older versions.

NAME

Extract the full file name, including any extension. This essentially just discards the directory part of `input`.

NAME_WE

Extract the base file name only. This is like `NAME` except only the part of the file name up to but not including the first "." is extracted.

NAME_WLE

Similar to `NAME_WE` except the name up to the last "." is extracted. This option is only available with CMake 3.14 or later.

EXT

This is the complement to `NAME_WE`. It extracts just the extension part of the file name from the first "." onward. This can be thought of as the longest extension in the file name.

LAST_EXT

Similar to `EXT` except the shortest extension is returned (i.e. the part of the file name from the last "." onward). This option is only available with CMake 3.14 or later.

The `CACHE` keyword is optional. If present, the result is stored as a cache variable rather than a regular variable. Typically, it is not desirable to store the result in the cache, so the `CACHE` keyword is not often required.

```
set(input /some/path/foo.bar.txt)

get_filename_component(path1          ${input} DIRECTORY)    # /some/path
get_filename_component(path2          ${input} PATH)         # /some/path
get_filename_component(fullName       ${input} NAME)         # foo.bar.txt
get_filename_component(baseNameShort  ${input} NAME_WE)      # foo
get_filename_component(baseNameLong   ${input} NAME_WLE)     # foo.bar
get_filename_component(extensionLong  ${input} EXT)          # .bar.txt
get_filename_component(extensionShort ${input} LAST_EXT)     # .txt
```

The second form of `get_filename_component()` is used to obtain an absolute path:

```
get_filename_component(outVar input component
    [BASE_DIR baseDir] [CACHE]
)
```

In this form, `input` can be a relative path or it can be an absolute path. If `BASE_DIR` is given, relative paths are interpreted as being relative to `baseDir` instead of the current source directory (i.e. `CMAKE_CURRENT_SOURCE_DIR`). `BASE_DIR` will be ignored if `input` is already an absolute path. Unlike

Exclusively for sbmain17@gmail.com Transaction: 0098315591

cmake_path(), this command can access the file system and resolve symbolic links. The component argument controls how symbolic links are handled for the path stored in outVar:

ABSOLUTE

Compute the absolute path of input without resolving symbolic links.

REALPATH

Compute the absolute path of input with symbolic links resolved.

The file() command provides the inverse operation, converting an absolute path to relative:

```
file(RELATIVE_PATH outVar relativeToDir input)
```

CMake 3.19 also added a file() sub-command which is essentially equivalent to the REALPATH operation of get_filename_component():

```
file(REAL_PATH input outVar
    [BASE_DIRECTORY baseDir]
    [EXPAND_TILDE]    # Requires CMake 3.21 or later
)
```

With CMake 3.21 or later, when the EXPAND_TILDE keyword is given and input starts with a tilde (~), the tilde will be replaced by the path to the user's home directory. This mimics the behavior of most Unix shells.

Unfortunately, the file(REAL_PATH) sub-command introduced a number of inconsistencies:

- The order of the input and outVar arguments is different to the complementary RELATIVE_PATH operation.
- The file() command uses REAL_PATH (note the underscore), whereas get_filename_component() uses REALPATH.
- The optional base directory keyword is named BASE_DIRECTORY for file(REAL_PATH), whereas it is named BASE_DIR for the get_filename_component() command.

The above inconsistencies can make the use of these commands somewhat more error-prone, so extra care is needed.

The following example demonstrates the usage of these file() sub-commands:

```
set(basePath   /base)
set(fooBarPath /base/foo/bar)
set(otherPath  /other/place)

file(RELATIVE_PATH fooBar ${basePath} ${fooBarPath})
file(RELATIVE_PATH other  ${basePath} ${otherPath})

file(REAL_PATH ${other} otherReal BASE_DIRECTORY ${basePath})
```


Exclusively for sbmain17@gmail.com Transaction: 0098315591

At the end of the above code block, the variables have the following values:

```
fooBar    = foo/bar
other     = ../other/place
otherReal = /other/place
```

The third form of the `get_filename_component()` command is a convenience for extracting parts of a full command line (there is no equivalent `cmake_path()` command for this):

```
get_filename_component(progVar input PROGRAM
    [PROGRAM_ARGS argVar] [CACHE]
)
```

With this form, `input` is assumed to be a command line which may contain arguments. CMake will extract the full path to the executable which would be invoked by the specified command line, resolving the executable's location using the `PATH` environment variable if necessary and store the result in `progVar`. If `PROGRAM_ARGS` is given, the set of command line arguments are also stored as a list in the variable named by `argVar`. The `CACHE` keyword has the same meaning as the other forms of `get_filename_component()`.

The `file()` command offers two more forms which help transform paths between platform native and CMake formats:

```
file(TO_NATIVE_PATH input outVar)
file(TO_CMAKE_PATH  input outVar)
```

The `TO_NATIVE_PATH` form converts `input` into a native path for the host platform. This amounts to ensuring the correct directory separator is used (backslash on Windows, forward slash everywhere else). The `TO_CMAKE_PATH` form converts all directory separators in `input` to forward slashes. This is the representation used by CMake for paths on all platforms. The input can also be a list of paths specified in a form compatible with the platform's `PATH` environment variable. All colon separators are replaced with semi-colons, thereby converting a `PATH`-like input into a CMake list of paths.

```
# Unix example
set(customPath /usr/local/bin:/usr/bin:/bin)

file(TO_CMAKE_PATH ${customPath} outVar)
# outVar = /usr/local/bin;/usr/bin;/bin
```

19.2. Copying Files

The need to copy a file during the configure stage or during the build itself is relatively common. Because copying a file is generally a familiar task to most users, it is natural for new CMake developers to implement file copying in terms of the same methods they already know. Unfortunately, this often results in the use of platform-specific shell commands with `add_custom_target()` and `add_custom_command()`, sometimes also with dependency problems that

Exclusively for sbmain17@gmail.com Transaction: 0098315591

require developers to run CMake multiple times and/or manually build targets in a particular sequence.

In almost all cases, CMake offers better alternatives to such platform-specific approaches. In this section, a number of techniques for copying files are presented. Some are aimed at meeting a particular need, while others are intended to be more generic and can be used in a variety of situations. All methods presented work exactly the same way on all platforms.

One of the most useful commands for copying files at configure time is, unfortunately, one of the less intuitively named. The `configure_file()` command allows a single file to be copied from one location to another, optionally performing CMake variable substitution along the way. The copy is performed immediately, so it is a configure-time operation. A slightly reduced form of the command is as follows:

```
configure_file(source destination
    [COPYONLY | @ONLY] [ESCAPE_QUOTES]
    # See below for availability
    [NO_SOURCE_PERMISSIONS | USE_SOURCE_PERMISSIONS |
     FILE_PERMISSIONS permissions...]
)
```

The `source` must be an existing file and can be an absolute or relative path, with the latter being relative to the current source directory (i.e. `CMAKE_CURRENT_SOURCE_DIR`). The `destination` can be an existing directory or the file name to copy to. Using a directory name is risky, since if there is no directory by the specified name, a file of that name will be created instead. The `destination` can include a path, which can be absolute or relative. If the `destination` is not an absolute path, it is interpreted as being relative to the current binary directory (i.e. `CMAKE_CURRENT_BINARY_DIR`). If any part of the destination path does not exist, CMake will attempt to create the missing directories as part of the call. Note that it is not unusual to see projects include `CMAKE_CURRENT_SOURCE_DIR` or `CMAKE_CURRENT_BINARY_DIR` as part of the path with the `source` and `destination` respectively, but this just adds unnecessary clutter and should be avoided.

By default, the destination file will have the same permissions as the source file. With CMake 3.19 or later, the `NO_SOURCE_PERMISSIONS` option can be given and the destination file will be readable by everyone, writable by the user only and not executable. With CMake 3.20 or later, `USE_SOURCE_PERMISSIONS` or `FILE_PERMISSIONS` can be used. The former is already the default behavior, but it can be specified to clearly indicate the intent. `FILE_PERMISSIONS` gives full control over the permissions assigned to the destination. These three permission-related keywords are also supported by the `file(COPY)` and `file(GENERATE)` commands. Examples of how permissions can be specified are included in the discussion of those commands further below.

If the `source` file is modified, the build will consider the `destination` to be out of date and will re-run `cmake` automatically. If the configure and generation time is non-trivial and the `source` file is being modified frequently, this can be a source of frustration for developers. For this reason, `configure_file()` is best used only for files that don't need to be changed all that often.

When performing the copy, `configure_file()` has the ability to substitute CMake variables. Without the `COPYONLY` or `@ONLY` options, anything in the `source` file that looks like a use of a CMake variable (i.e. has the form `${someVar}`) will be replaced by the value of that variable. If no variable exists with that

Exclusively for sbmain17@gmail.com Transaction: 0098315591

name, an empty string is substituted. Strings of the form `@someVar@` are also substituted in the same way. The following shows a number of substitution examples:

CMakeLists.txt

```
set(FOO "String with spaces")
configure_file(various.txt.in various.txt)
```

various.txt.in

```
CMake version: ${CMAKE_VERSION}
Substitution works inside quotes too: "${FOO}"
No substitution without the $ and {}: FOO
Empty ${} specifier gets removed
Escaping has no effect: \${FOO}
@-syntax also supported: @FOO@
```

various.txt

```
CMake version: 3.7.0
Substitution works inside quotes too: "String with spaces"
No substitution without the $ and {}: FOO
Empty  specifier gets removed
Escaping has no effect: \String with spaces
@-syntax also supported: String with spaces
```

The `ESCAPE_QUOTES` keyword can be used to cause any substituted quotes to be preceded with a backslash.

CMakeLists.txt

```
set(BAR "Some \"quoted\" value")
configure_file(quoting.txt.in quoting.txt)
configure_file(quoting.txt.in quoting_escaped.txt ESCAPE_QUOTES)
```

quoting.txt.in

```
A: @BAR@
B: "@BAR@"
```

quoting.txt

```
A: Some "quoted" value
B: "Some "quoted" value"
```

quoting_escaped.txt

```
A: Some \"quoted\" value
B: "Some \"quoted\" value"
```

Exclusively for sbmain17@gmail.com Transaction: 0098315591

As the above example shows, the `ESCAPE_QUOTES` option causes escaping of all quotes regardless of their context. Therefore, a degree of care must be taken when the file being copied is sensitive to spaces and quoting in any substitutions which may be performed.

Some file types need to have the `${someVar}` form preserved without substitution. A classic example of this is copying a Unix shell script where `${someVar}` is a common way to refer to a shell variable. In such cases, substitution can be limited to only the `@someVar@` form with the `@ONLY` keyword:

CMakeLists.txt

```
set(USER_FILE whoami.txt)
configure_file(whoami.sh.in whoami.sh @ONLY)
```

whoami.sh.in

```
#!/bin/sh
echo ${USER} > "@USER_FILE@"
```

whoami.sh

```
#!/bin/sh
echo ${USER} > "whoami.txt"
```

Substitution can also be disabled entirely with the `COPYONLY` keyword. If it is known that substitution is not needed, specifying `COPYONLY` is good practice, since it prevents unnecessary processing and any unexpected substitutions.

When using `configure_file()` and substituting file names or paths, a common mistake is to mishandle spaces and quoting. The source file may need to surround a substituted variable with quotes if it needs to be treated as a single path or file name. This is why the source file in the above example used `"@USER_FILE@"` rather than `@USER_FILE@` as the filename to write the output to.

Substitution of CMake variables in `${someVar}` or `@someVar@` form can also be performed on strings, not just files. The `string(CONFIGURE)` command provides equivalent functionality and options. It can be useful when the content to be copied requires more complex steps than a simple substitution:

```
string(CONFIGURE input outVar [@ONLY] [ESCAPE_QUOTES])
```

The `configure_file()` command uses files for input and output. The `string(CONFIGURE)` sub-command uses strings for input and output. CMake 3.18 added a third method that supports using a string for input and a file for output:

```
file(CONFIGURE
    OUTPUT  outFile
    CONTENT inputString
    [@ONLY] [ESCAPE_QUOTES]
    # ... Other rarely used options
)
```


Exclusively for sbmain17@gmail.com Transaction: 0098315591

The OUTPUT and CONTENT specify the output file and input string respectively. If the outFile is a relative path, it is assumed to be relative to the current binary directory. The remaining options have exactly the same meaning as they do for the configure_file() command.

Where no substitution is needed, another alternative is to use one of the relevant file() sub-commands. The most flexible and mature, available with all CMake versions, are the closely related COPY and INSTALL forms, both of which support the same set of options:

```
file(<COPY|INSTALL> fileOrDir1 [fileOrDir2...]
    DESTINATION dir
    [NO_SOURCE_PERMISSIONS | USE_SOURCE_PERMISSIONS |
     [FILE_PERMISSIONS permissions...]
     [DIRECTORY_PERMISSIONS permissions...]]
    [FOLLOW_SYMLINK_CHAIN]   # Requires CMake 3.15 or later
    [FILES_MATCHING]
    [ [PATTERN pattern | REGEX regex] [EXCLUDE]
      [PERMISSIONS permissions...] ] [...]
)
```

Multiple files or even entire directory hierarchies can be copied to a chosen directory, even preserving symlinks if present. Any source files or directories specified without an absolute path are treated as being relative to the current source directory. Similarly, if the destination directory is not absolute, it will be interpreted as being relative to the current binary directory. The destination directory structure is created as necessary.

If a source is a directory name, it will be copied into the destination. To copy the directory's contents into the destination instead, append a forward slash (/) to the source directory like so:

```
file(COPY base/srcDir  DESTINATION destDir)  # --> destDir/srcDir
file(COPY base/srcDir/ DESTINATION destDir)  # --> destDir
```

By default, the COPY form will result in all files and directories keeping the same permissions as the source from which they are copied, whereas the INSTALL form will not preserve the original permissions. The NO_SOURCE_PERMISSIONS and USE_SOURCE_PERMISSIONS options can be used to override these defaults, or the permissions can be explicitly specified with the FILE_PERMISSIONS and DIRECTORY_PERMISSIONS options. The permission values are based on those supported by Unix systems:

OWNER_READ	OWNER_WRITE	OWNER_EXECUTE
GROUP_READ	GROUP_WRITE	GROUP_EXECUTE
WORLD_READ	WORLD_WRITE	WORLD_EXECUTE
SETUID	SETGID	

If a particular permission is not understood on a given platform, it is simply ignored. Multiple permissions can be (and usually are) listed together. For example, a Unix shell script might be copied to the current binary directory as follows:

Exclusively for sbmain17@gmail.com Transaction: 0098315591

```
file(COPY whoami.sh
    DESTINATION .
    FILE_PERMISSIONS
        OWNER_READ OWNER_WRITE OWNER_EXECUTE
        GROUP_READ GROUP_EXECUTE
        WORLD_READ WORLD_EXECUTE
)
```

The `COPY` and `INSTALL` signatures both also preserve the timestamps of the files and directories being copied. Furthermore, if the source is already present at the destination with the same timestamp, the copy for that file is deemed as already having been done and will be skipped. The only other difference between `COPY` and `INSTALL` apart from the default permissions is that the `INSTALL` form prints status messages for each copied item, whereas `COPY` does not. This difference is because the `INSTALL` form is typically used as part of CMake scripts run in script mode for installing files, where common behavior is to print the name of each file installed.

With CMake 3.15 or later, the `FOLLOW_SYMLINK_CHAIN` keyword is supported. When this option is present, symlinks in the list of files to copy/install will be copied recursively, with symlinks being preserved. The recursion stops with the final non-symlink file being copied as normal. When copying or installing symlinks like this, all paths are stripped off, so this functionality really only suits situations where symlinks point to things in the same directory.

Consider a fairly standard set of library symlinks on Linux, such as the following:

```
libMyStuff.so.2.4.3
libMyStuff.so.2 --> libMyStuff.so.2.4.3
libMyStuff.so   --> libMyStuff.so.2
```

If `libMyStuff.so` was given to the `file(COPY)` or `file(INSTALL)` command and the `FOLLOW_SYMLINK_CHAIN` option was present, all three of the above would be copied/installed and the relative symlinks would all be preserved exactly as shown. Note that the symlinks are only followed in one direction, there is no logic for finding things that link *to* a file listed. So for the above example, if only `libMyStuff.so.2` was listed in the `file()` command, the `libMyStuff.so` symlink would not be discovered and therefore it wouldn't be copied/installed.

Both `COPY` and `INSTALL` support applying certain logic to files that match or do not match a particular wildcard pattern or regular expression. This can be used to limit which files are copied and to override the permissions just for the matched files. Multiple patterns and regular expressions can be given in the one `file()` command. The use is best demonstrated by example.

The following copies all header (`.h`) and script (`.sh`) files from `someDir`, except headers whose file name ends with `_private.h`. Headers are given the same permissions as the file they are copied from, whereas scripts are given owner read, write and execute permissions. The directory structure is preserved.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

```
file(COPY someDir
    DESTINATION .
    FILES_MATCHING
        REGEX .*_private\\.h EXCLUDE
        PATTERN *.h
        PATTERN *.sh
            PERMISSIONS OWNER_READ OWNER_WRITE OWNER_EXECUTE
)
```

If the whole source should be copied but permissions need to be overridden for just a subset of matched files, the `FILES_MATCHING` keyword can be omitted and the patterns and regular expressions are used purely to apply permission overrides.

```
file(COPY someDir
    DESTINATION .
    # Make Unix shell scripts executable by everyone
    PATTERN *.sh PERMISSIONS
        OWNER_READ OWNER_WRITE OWNER_EXECUTE
        GROUP_READ GROUP_EXECUTE
        WORLD_READ WORLD_EXECUTE
    # Ensure only owner can read/write private key files
    REGEX _dsa\$|_rsa\$ PERMISSIONS
        OWNER_READ OWNER_WRITE
)
```

For very simple file copying operations, CMake 3.21 or later provides an alternative sub-command that some developers may find easier to use:

```
file(COPY_FILE source destination
    [RESULT result]
    [ONLY_IF_DIFFERENT]
)
```

If `ONLY_IF_DIFFERENT` is given, then the timestamp of the `destination` will not be updated if it already has the same contents as the `source` file. It will typically be advisable to include this option to avoid triggering unnecessary rebuilds of anything that depends on the `destination`.

If the `RESULT` keyword is given, the outcome of the command is stored in the named variable. This allows processing to continue and recover from an error. The value stored in `outVar` will be 0 upon success or an error message otherwise. Without the `RESULT` keyword, processing will halt if any error is encountered.

CMake offers further methods for copying files and directories. Whereas `configure_file()` and `file()` are used mainly at configure time or in a CMake script at install time, CMake's command mode is often used for copying files and directories at build time. Command mode is the preferred way to copy content as part of `add_custom_target()` and `add_custom_command()` rules, since it provides platform independence (see Section 18.5, "Platform Independent Commands"). There are three commands related to copying, the first of which is used to copy individual files:

Exclusively for sbmain17@gmail.com Transaction: 0098315591

```
cmake -E copy file1 [file2...] destination
```

If only one source file is provided, then `destination` is treated as the name of the file to copy to, unless it names an existing directory. When the destination is an existing directory, the source file will be copied into it. This behavior is consistent with that of most operating systems' native copy commands, but it also means that the behavior is dependent on the state of the file system before the copy operation. Therefore, it is more robust to explicitly specify the target file name when copying a single file unless it is guaranteed that the destination is a directory that will already exist.

As a convenience, if `destination` includes a path (relative or absolute), CMake will try to create the destination path as needed when copying only a single source file. This means that when copying individual files, the `copy` command does not require an earlier step to ensure the destination directory exists. If more than one source file is listed, `destination` must refer to an existing directory. Once again, CMake's command mode can be used to ensure this using `make_directory` which creates the named directory if it does not already exist, including any parent directories as needed. The following shows how to safely put these command mode commands together:

```
add_custom_target(CopyOne
    COMMAND ${CMAKE_COMMAND} -E copy a.txt output/textfiles/a.txt
)
add_custom_target(CopyTwo
    COMMAND ${CMAKE_COMMAND} -E make_directory output/textfiles
    COMMAND ${CMAKE_COMMAND} -E copy a.txt b.txt output/textfiles
)
```

The `copy` command will always copy the source to the destination, even if the destination is already identical to the source. This results in the target timestamps always being updated, which can sometimes be undesirable. If the timestamps should not be updated if the files already match, then the `copy_if_different` command may be more appropriate:

```
cmake -E copy_if_different file1 [file2...] destination
```

This functions exactly like the `copy` command except if a source file already exists at the destination and is the same as the source, no copy is performed and the timestamp of the target is left alone. It is also possible to copy entire directories rather than individual files:

```
cmake -E copy_directory dir1 [dir2...] destination
```

Unlike the file-related copy commands, the destination directory is created if required, including any intermediate path. Note also that `copy_directory` copies the *contents* of the source directories into the destination, not the source directories themselves. For example, suppose a directory `myDir` contains a file `someFile.txt` and the following command was issued:

```
cmake -E copy_directory myDir targetDir
```


Exclusively for sbmain17@gmail.com Transaction: 0098315591

The result would be that `targetDir` would contain the file `someFile.txt`, not `myDir/someFile.txt`.

Generally speaking, `configure_file()` and `file()` are best suited to copying files at configure time, whereas CMake's command mode is the preferred way to copy at build time. While it is possible to use command mode in conjunction with `execute_process()` to copy files at configure time, there is little reason to do so, since `configure_file()` and `file()` are both more direct and have the added benefit that they stop on any error automatically.

19.3. Reading And Writing Files Directly

CMake offers more than just the ability to copy files, it also provides a number of commands for reading and writing file contents. The `file()` command provides the bulk of the functionality, with the simplest being the forms that write directly to a file:

```
file(WRITE fileName content)
file(APPEND fileName content)
```

Both of these commands will write the specified `content` to the named file. The only difference between the two is that if `fileName` already exists, `APPEND` will append to the existing contents, whereas `WRITE` will discard the existing contents before writing. The `content` is just like any other function argument and can be the contents of a variable or a string.

```
set(msg "Hello world")
file(WRITE hello.txt ${msg})
file(APPEND hello.txt " from CMake")
```

The above would result in the file `hello.txt` containing the single line `Hello world from CMake`. Note that newlines are not automatically added, so the text from the `APPEND` line continues directly after the `WRITE` line's text without a break. To have a newline written, it must be included in the content passed to the `file()` command. One way is to use a quoted value spread across multiple lines:

```
file(WRITE multi.txt "First line
Second line
")
```

If using CMake 3.0 or later, the bracket syntax introduced back in Section 5.1, "Variable Basics" can sometimes be more convenient, since it prevents any variable substitution of the content.

```
file(WRITE multi.txt [[
First line
Second line
]])

file(WRITE userCheck.sh [=[
#!/bin/bash
[[ -n "${USER}" ]] && echo "Have USER"
]=])
```

Exclusively for sbmain17@gmail.com Transaction: 0098315591

In the above, the contents to be written to `multi.txt` consist only of simple text with no special characters, so the simplest bracket syntax where = characters can be omitted is sufficient, leaving just a pair of square brackets to mark the start and end of the content. Note how the behavior to ignore the first newline immediately after the opening bracket makes the command more readable.

The contents for `userCheck.sh` are much more interesting and highlight the features of bracket syntax. Without bracket syntax, CMake would see the `${USER}` part and treat it as a CMake variable substitution, but because bracket syntax performs no such substitution, it is left as is. For the same reason, the various quote characters in the content are also not interpreted as anything other than part of the content. They do not need to be escaped to prevent them being interpreted as the start or end of an argument. Furthermore, note how the embedded contents contain a pair of square brackets. This is the sort of situation the variable number of = signs in the start and end markers is meant to handle, allowing the markers to be chosen so that they do not match anything in the content they surround. When writing out multiple lines to a file and when no substitution should be performed, bracket syntax is often the most convenient way to specify the content to be written.

Sometimes a project may need to write a file whose contents depend on the build type. A naive approach would be to assume the `CMAKE_BUILD_TYPE` variable could be used as a substitution, but this does not work for multi configuration generators like Xcode, Visual Studio or Ninja Multi-Config. Instead, the `file(GENERATE…)` command can be used:

```
file(GENERATE
    OUTPUT outFile
    INPUT inFile | CONTENT content
    [CONDITION expression]
    # Requires CMake 3.19 or later:
    [TARGET target]
    # Requires CMake 3.20 or later:
    [NO_SOURCE_PERMISSIONS | USE_SOURCE_PERMISSIONS |
     [FILE_PERMISSIONS permissions...]
)
```

This works somewhat like `file(WRITE…)` except that it writes out one file for each build type supported for the current CMake generator. Either of the `INPUT` or `CONTENT` options must be present, but not both. They define the content to be written to the specified output file.

`outFile`, `inFile` and `content` all support generator expressions, which is how the file names and contents are customized for each build type. A build type can be skipped by using the `CONDITION` option. The `expression` must evaluate to either 0 or 1 after any generator expressions have been expanded. The output file is not generated if it evaluates to 0.

If a generator expression in any of the arguments needs a target for it to be evaluated, but the target is not part of the expression (e.g. `$<TARGET_PROPERTY:propName>`), the `TARGET` option must be provided so it can be resolved. The `TARGET` option is only supported with CMake 3.19 or later.

With CMake 3.20 or later, the permissions for the `outFile` can be specified in the same way as for `configure_file()` or `file(COPY)`. The `NO_SOURCE_PERMISSIONS` and `USE_SOURCE_PERMISSIONS` keywords are only valid if using the `INPUT` option to specify an input file, but `FILE_PERMISSIONS` can be used with either `INPUT` or `CONTENT`.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

The following examples show how to make use of generator expressions to customize the contents and file names depending on the build type.

```
# Generate unique files for all but Release
file(GENERATE
    OUTPUT ${CMAKE_CURRENT_BINARY_DIR}/outFile-$<CONFIG>.txt
    INPUT  ${CMAKE_CURRENT_SOURCE_DIR}/input.txt.in
    CONDITION $<NOT:$<CONFIG:Release>>
)
```

```
# Embedded content, bracket syntax does not prevent the use of generator expressions
file(GENERATE
    OUTPUT ${CMAKE_CURRENT_BINARY_DIR}/details-$<CONFIG>.txt
    CONTENT [[
Built as "$<CONFIG>" for platform "$<PLATFORM_ID>".
Defines: $<TARGET_PROPERTY:COMPILE_DEFINITIONS>
]]
    TARGET SomeTarget
)
```

In the first case above, any generator expressions in the `input.txt.in` file will be evaluated when writing the output file. This is somewhat analogous to the way `configure_file()` substitutes CMake variables, except this time the substitution is for generator expressions. The second case demonstrates how bracket syntax can be a particularly convenient way of defining file contents inline, even when generator expressions and quoting are involved.

Usually, the output file would be different for each build type. In some situations, however, it may be desirable for the output file to always be the same, such as where the file contents do not depend on the build type but rather on some other generator expressions. To support such use cases, CMake allows the output file to be the same for different build types, but only if the generated file contents are also identical for those build types. CMake disallows multiple `file(GENERATE…)` commands trying to generate the same output file.

Like for `file(COPY…)`, the `file(GENERATE…)` command will only modify the output file if the contents actually change. Therefore, the output file's timestamp will also only be updated if the contents differ. This is useful when the generated file is used as an input in a build target, such as a generated header file, since it can prevent unnecessary rebuilds.

There are some important differences in the way `file(GENERATE…)` behaves compared to most other CMake commands. Because it evaluates generator expressions, it cannot write out the files immediately. Instead, the files are written as part of the generation phase, which occurs after all of the `CMakeLists.txt` files have been processed. This means that the generated files won't exist when the `file(GENERATE…)` command returns, so the files cannot be used as inputs to something else during the configure phase. In particular, since the generated files won't exist until the end of the configure phase, they cannot be copied or read with `configure_file()`, `file(COPY…)`, etc. They can, however, still be used as inputs for the *build* phase, such as generated sources or headers.

Another point to note is that before CMake 3.10, `file(GENERATE…)` handled relative paths differently compared to usual CMake conventions. The behavior of relative paths was left unspecified and

Exclusively for sbmain17@gmail.com Transaction: 0098315591

usually ended up being relative to the working directory of when `cmake` was invoked. This was unreliable and inconsistent, so in CMake 3.10 the behavior was changed to make `INPUT` act as relative to the current source directory and `OUTPUT` relative to the current binary directory, just like most other CMake commands that handle paths. Projects should consider relative paths unsafe to use with `file(GENERATE…)` unless the minimum CMake version is set to 3.10 or later.

The `file()` command can not only copy or create files, it can also be used to read in a file's contents:

```
file(READ fileName outVar [OFFSET offset] [LIMIT byteCount] [HEX])
```

Without any of the optional keywords, this command reads all the contents of `fileName` and stores them as a single string in `outVar`. The `OFFSET` option can be used to read only from the *offset* specified, counted in bytes from the beginning of the file. The maximum number of bytes to read can also be limited with the `LIMIT` option. If the `HEX` option is given, the contents will be converted to a hexadecimal representation, which can be useful for files containing binary data rather than text.

If it is more desirable to break up the file contents line-by-line, the `STRINGS` sub-command may be more convenient. Instead of storing the entire file's contents as a single string, it stores them as a list, with each line being one list item.

```
file(STRINGS fileName outVar
    [LENGTH_MAXIMUM maxBytesPerLine]
    [LENGTH_MINIMUM minBytesPerLine]
    [LIMIT_INPUT maxReadBytes]
    [LIMIT_OUTPUT maxStoredBytes]
    [LIMIT_COUNT maxStoredLines]
    [REGEX regex]
    # ... other less commonly used options not shown
)
```

The `LENGTH_MAXIMUM` and `LENGTH_MINIMUM` options can be used to exclude strings longer or shorter than a certain number of bytes respectively. The total number of bytes read can be limited using `LIMIT_INPUT`, while the total number of bytes stored can be limited using `LIMIT_OUTPUT`. Perhaps more likely to be useful, however, is the `LIMIT_COUNT` option which limits the total number of *lines* stored rather than the number of bytes.

The `REGEX` option is a useful way to extract only specific lines from a file. For example, the following obtains a list with all lines in `myStory.txt` that contain either `PKG_VERSION` or `MODULE_VERSION`.

```
file(STRINGS myStory.txt versionLines
     REGEX "(PKG|MODULE)_VERSION"
)
```

It can also be combined with `LIMIT_COUNT` to obtain just the first match. The following example shows how to combine `file()` and `string()` to extract a portion of the first line matching a regular expression.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

```
set(regex "^ *FOO_VERSION *= *([^ ]+) *$")
file(STRINGS config.txt fooVersion
     REGEX "${regex}"
)
string(REGEX REPLACE "${regex}" "\\1" fooVersion "${fooVersion}")
```

If `config.txt` contained a line like this:

```
FOO_VERSION = 2.3.5
```

Then the value stored in `fooVersion` would be `2.3.5`.

19.4. File System Manipulation

In addition to reading and writing files, CMake also supports other common file system operations.

```
file(MAKE_DIRECTORY dirs...)
file(REMOVE files...)
file(REMOVE_RECURSE filesOrDirs...)
file(RENAME source destination
    # CMake 3.21 or later required for these options
    [RESULT outVar]
    [NO_REPLACE]
)
```

The `MAKE_DIRECTORY` sub-command will ensure the listed directories exist. Intermediate paths are created as necessary and no error is reported if a directory already exists.

The `REMOVE` sub-command can be used to delete files. If any of the listed files do not exist, the `file()` command does not report an error. Attempting to delete a directory with the `REMOVE` sub-command will have no effect. To delete directories and all of their contents, use the `REMOVE_RECURSE` sub-command instead.

The `RENAME` sub-command renames a file or directory. The `source` and `destination` must be the same type, i.e. both files or both directories. It is not permitted to specify a file as the `source` and an existing directory for the `destination`. To move a file into a directory, the file name must be specified as part of the `destination`. Furthermore, any path part of the `destination` must already exist — the `RENAME` form will not create intermediate directories.

Ordinarily, `file(RENAME)` will halt if any error is encountered. With CMake 3.21 or later, if the `RESULT` keyword is given, the outcome of the command is stored in the named variable instead. This allows processing to continue and recover from an error. The value stored in `outVar` will be 0 upon success or an error message otherwise. Note that if `destination` already exists, it is not considered an error unless `NO_REPLACE` is given (CMake 3.21 or later required). Without `NO_REPLACE`, the `destination` will be silently replaced by the `source`.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

```
# Requires CMake 3.21 or later
file(RENAME someFile toSomethingElse
    RESULT result
    NO_REPLACE
)
if(result)
    message(WARNING "File rename failed, taking action")
    # ... handle failure to rename the file
endif()
```

CMake's command mode also supports a very similar set of capabilities which can be used at build time rather than configure time:

```
cmake -E make_directory dirs...
cmake -E remove [-f] files...         # deprecated from CMake 3.17
cmake -E remove_directory dir         # deprecated from CMake 3.17
cmake -E rm [-rRf] filesOrDirs...
cmake -E rename source destination
```

These commands largely behave in a comparable way to their `file()`-based counterparts, with only slight variations. The `remove_directory` command can strictly only be used with a single directory, whereas `file(REMOVE_RECURSE…)` can remove multiple items and both files and directories can be listed. The `remove` command accepts an optional `-f` flag which is intended to change the behavior when an attempt is made to remove a file that does not exist. The documented behavior of this flag is that without `-f`, a non-zero exit code is returned, whereas with `-f`, a zero exit code will be returned. This is intended to mimic aspects of the behavior of the Unix `rm -f` command. Unfortunately, due to a long-standing bug in the implementation, this isn't the actual behavior and the exit code of `cmake -E remove` should be considered unreliable, with or without the `-f` flag. The `rm` command was added in CMake 3.17 as a replacement for both the `remove` and `remove_directory` commands. It fixes the exit code bug and is more closely aligned with the Unix `rm` command.

CMake 3.14 added two new `file()` sub-commands which enable the project to query and manipulate file system links:

```
file(READ_SYMLINK linkName outVar)
file(CREATE_LINK pointedTo linkName
     [RESULT outVar]
     [COPY_ON_ERROR]
     [SYMBOLIC]
)
```

The `READ_SYMLINK` sub-command gives the path of what `linkName` points to. Note that symlinks often use relative paths and the value stored in `outVar` will just be the raw relative path for such cases.

The `CREATE_LINK` command allows the project to create hard or symbolic links. Hard links are created by default, but the `SYMBOLIC` option can be given to create a symbolic link instead. In most situations, `SYMBOLIC` would be recommended since it supports more scenarios (e.g. linking across different file systems). The `RESULT` keyword can be used to name a variable in which to store the

Exclusively for sbmain17@gmail.com Transaction: 0098315591

result of the operation. The value stored will be 0 upon success or an error message otherwise. Without the `RESULT` option, a failure will cause CMake to halt with a fatal error. The `COPY_ON_ERROR` option provides a fallback for when an attempt to create a link fails, downgrading the operation to copy `pointedTo` to `linkName`. It exists primarily to allow the command to be used where creation of links is not supported, such as a hard link to a path on a different drive or device.

All CMake versions allow basic creation of symbolic links using CMake's command mode too (hard links cannot be created via this method):

```
cmake -E create_symlink pointedTo linkName
```

CMake 3.14 also added the ability to query the size of a file:

```
file(SIZE fileName outVar)
```

The specified `fileName` must exist and importantly it must also be readable.

CMake 3.19 added `file()` sub-commands for setting file and directory permissions:

```
file(CHMOD | CHMOD_RECURSE
    files... directories...
    [PERMISSIONS permissions...]
    [FILE_PERMISSIONS permissions...]
    [DIRECTORY_PERMISSIONS permissions...]
)
```

The `CHMOD` and `CHMOD_RECURSE` sub-commands are identical in their behavior except that the latter will also descend recursively into subdirectories. The supported values for `permissions` are the same as those supported by the `file(COPY)` sub-command. `FILE_PERMISSIONS` or `DIRECTORY_PERMISSIONS` will apply only to their respective type of entity and they will override `PERMISSIONS` for that entity type. One can specify just one of the two more specific types to operate on just that type of entity. The following shows how to take advantage of this to set permissions just for directories and leave file permissions unmodified:

```
file(CHMOD_RECURSE ${someFilesAndDirs}
    DIRECTORY_PERMISSIONS
        OWNER_READ OWNER_WRITE OWNER_EXECUTE
        GROUP_READ GROUP_EXECUTE
        WORLD_READ WORLD_EXECUTE
)
```

19.5. File Globbing

CMake also supports listing the contents of one or more directories with either a recursive or non-recursive form of globbing:

Exclusively for sbmain17@gmail.com Transaction: 0098315591

```
file(GLOB outVar
    [LIST_DIRECTORIES true|false]
    [RELATIVE path]
    [CONFIGURE_DEPENDS]   # Requires CMake 3.12 or later
    expressions...
)
```

```
file(GLOB_RECURSE outVar
    [LIST_DIRECTORIES true|false]
    [RELATIVE path]
    [FOLLOW_SYMLINKS]
    [CONFIGURE_DEPENDS]   # Requires CMake 3.12 or later
    expressions...
)
```

These commands find all files whose names match any of the provided `expressions`, which can be thought of as simplified regular expressions. It may be easier to think of them as ordinary wildcards with the addition of character subset selection. For `GLOB_RECURSE`, they can also include path components.

Some examples should help clarify basic use:

`*.txt`	All files whose name ends with `.txt`.
`foo?.txt`	Files like `foo2.txt`, `fooB.txt`, etc.
`bar[0-9].txt`	Matches all files of the form `barX.txt` where `X` is a single digit.
`/images/*.png`	For `GLOB_RECURSE`, this will match only those files with a `.png` extension and that are in a subdirectory called `images`. This can be a useful way of finding files in a well-structured directory hierarchy.

For `GLOB`, both files and directories matching the expression are stored in `outVar`. For `GLOB_RECURSE`, on the other hand, directory names are not included by default but this can be controlled with the `LIST_DIRECTORIES` option. Furthermore, for `GLOB_RECURSE`, symlinks to directories are normally reported as entries in `outVar` rather than descending into them, but the `FOLLOW_SYMLINKS` option directs CMake to descend into the directory instead of listing it.

The set of file names returned will be full absolute paths by default, regardless of the expressions used. The `RELATIVE` option can be used to change this behavior such that the reported paths are relative to a specific directory.

```
set(base /usr/share)

file(GLOB_RECURSE images
    RELATIVE ${base}
    ${base}/*/*.png
)
```


Exclusively for sbmain17@gmail.com Transaction: 0098315591

The above will find all images below `/usr/share` and include the path to those images, except with the `/usr/share` part stripped off. Note the `/*/` in the expression to allow any directory below the base point to be matched.

Developers should be aware that the `file(GLOB…)` commands are not as fast as, say, the Unix `find` shell command. Therefore, run time can be non-trivial if using it to search parts of the file system that contain many files.

> The `file(GLOB)` and `file(GLOB_RECURSE)` commands are some of the most misused parts of CMake. They should not be used to collect a set of files to be used as sources, headers or any other set of files that act as inputs to the build. One of the reasons this should be avoided is that if files are added or removed, CMake is not automatically re-run, so the build is unaware of the change. This becomes particularly problematic if developers are using a version control system and are switching between branches, etc. where the set of files might change, but not in a way which causes CMake to re-run. The `CONFIGURE_DEPENDS` option added in CMake 3.12 tries to address this deficiency, but it comes with performance penalties and is not guaranteed to be supported by all project generators. The use of this option should be avoided.
>
> Unfortunately, it is very common to see tutorials and examples use `file(GLOB)` and `file(GLOB_RECURSE)` to collect the set of sources to pass to commands like `add_executable()` and `add_library()`. This is explicitly discouraged by the CMake documentation for precisely the above reasons. Such a practice also ignores the possibility that some files might only be intended for specific platforms. For projects with many files spread across multiple directories, there are better ways to collect the set of source files which do not suffer from these deficiencies. Section 29.5.1, “Target Sources” presents alternative strategies which not only avoid these problems, they also encourage a more modular, self-contained directory structure.

19.6. Downloading And Uploading

The `file()` command has a number of other forms which carry out different tasks. A surprisingly powerful pair of sub-commands gives projects the ability to download files from and upload files to an arbitrary URL.

```
file(DOWNLOAD url fileName [options...])
file(UPLOAD   fileName url [options...])
```

The `DOWNLOAD` form downloads a file from the specified `url` and saves it to `fileName`. If a relative `fileName` is given, it is interpreted as being relative to the current binary directory. CMake 3.19 and later allow the `fileName` to be omitted, in which case the file is downloaded but discarded. This can be used to check that a URL exists without having to save the file anywhere (not recommended for files that are expected to be large).

The `UPLOAD` form performs the complementary operation, uploading the named file to the specified url. For uploads, a relative path is interpreted as being relative to the current source directory.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

The following options can be used, most of which are common to both `DOWNLOAD` and `UPLOAD`:

`LOG outVar`

Save logged output from the operation to the named variable. This can be useful to help diagnose problems when a download or upload fails.

`SHOW_PROGRESS`

When present, this option causes progress information to be logged as status messages. This can produce a fairly noisy CMake configure stage, so it is probably best to use this option only to temporarily help test a failing connection.

`TIMEOUT seconds`

Abort the operation if more than `seconds` have elapsed.

`INACTIVITY_TIMEOUT seconds`

This is a more specific kind of timeout. Some network connections may be of poor quality or may simply be very slow. It might be desirable to allow an operation to continue as long as it is making some sort of progress, but if it stalls for more than some acceptable limit, the operation should fail. The `INACTIVITY_TIMEOUT` option provides this capability, whereas `TIMEOUT` only allows the total time to be limited.

`TLS_VERIFY value`

This option accepts a boolean value indicating whether to perform server certificate verification when downloading from or uploading to a `https://` url. If this option is not provided, CMake looks for a variable named `CMAKE_TLS_VERIFY` instead. If neither the option nor the variable are defined, the default behavior is to not verify the server certificate. Note that `UPLOAD` support for this option was only added in CMake 3.18.

`TLS_CAINFO fileName`

A custom Certificate Authority file can be specified with this option. It only affects `https://` urls. If this option is not provided, CMake looks for a variable named `CMAKE_TLS_CAINFO` instead. Note that `UPLOAD` support for this option was only added in CMake 3.18.

`EXPECTED_HASH ALGO=value`

This option is only supported for `DOWNLOAD`. It specifies the checksum of the file being downloaded so that CMake can verify the contents. `ALGO` can be any one of the hashing algorithms CMake supports, the most commonly used being `MD5` and `SHA1`. Some older projects may use `EXPECTED_MD5` as an alternative to `EXPECTED_HASH MD5=⋯`, but new projects should prefer the `EXPECTED_HASH` form.

With CMake 3.7 or later, the following options are also available for both `DOWNLOAD` and `UPLOAD`:

`USERPWD username:password`

Provides authentication details for the operation. Be aware that hard-coding passwords is a security issue and in general should be avoided. If providing passwords with this option, the content should come from outside the project, such as from an appropriately protected file read from the user's local machine at configure time.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

`HTTPHEADER header`

Includes a HTTP header for the operation and can be repeated multiple times as needed to provide more than one header value. The following partial example demonstrates one of the motivating cases for this option:

```
file(DOWNLOAD "https://somebucket.s3.amazonaws.com/myfile.tar.gz" myfile.tar.gz
    EXPECTED_HASH SHA1=${myfileHash}
    HTTPHEADER "Host: somebucket.s3.amazonaws.com"
    HTTPHEADER "Date: ${timestamp}"
    HTTPHEADER "Content-Type: application/x-compressed-tar"
    HTTPHEADER "Authorization: AWS ${s3key}:${signature}"
)
```

The `file()`-based download and upload commands tend to find use more as part of install steps, packaging or test reporting, but they can also occasionally find use for other purposes. Examples include things like downloading bootstrap files at configure time or bringing a file into the build which cannot or should not be stored as part of the project sources (e.g. sensitive files that should only be accessible for certain developers, very large files, etc.). Later chapters provide specific scenarios where these commands are used with great effect.

19.7. Recommended Practices

A range of CMake functionality related to file handling has been presented in this chapter. The various methods can be used very effectively to carry out a range of tasks in a platform independent way, but they can also be misused. Establishing good patterns and applying them consistently throughout a project will help ensure new developers are exposed to better practices.

Consider using the `cmake_path()` command for path handling if the project's minimum CMake version is 3.20 or later. It generally uses a more consistent syntax than analogous capabilities of the much older `get_filename_component()` and `file()` commands. On the other hand, `cmake_path()` does not access the underlying file system, so it cannot be used if resolving symbolic links is required.

Note that the `if(IS_ABSOLUTE)` and `cmake_path(IS_ABSOLUTE)` commands interpret the path based on the *host* platform, but there are subtle differences between the two. They can yield different results for the same path in some cases. `cmake_path(IS_ABSOLUTE)` follows the implementation of the C++ `std::filesystem::path::is_absolute()` function, whereas `if(IS_ABSOLUTE)` uses its own logic with handling for some special cases. Take extra care if cross-compiling from a Windows host to a non-Windows target platform, or vice versa. If either command is used inappropriately, some paths may yield undefined behavior or give an opposite result to the value one may intuitively expect.

The `configure_file()` command is one that new developers often overlook, yet it is a key method of providing a file whose contents can be tailored according to variables determined at configure time, or even just to do a simple file copy. A common naming convention is for the file name part of the source and destination to be the same, except the source has an extra `.in` appended. Some IDE environments understand this convention and will provide appropriate syntax highlighting on the source file based on the file's extension without the `.in` suffix. The presence of the `.in` suffix not only serves as a clear reminder that the file needs to be transformed before use, it also prevents it from being accidentally picked up instead of the destination if CMake or the compiler look for files

Exclusively for sbmain17@gmail.com Transaction: 0098315591

in multiple directories. This is especially relevant when the destination is a C/C++ header and the current source and binary directories are both on the header search path.

Choosing the most appropriate command for copying files is not always clear. The following may serve as a useful guide when choosing between `configure_file()`, `file(COPY)` and `file(INSTALL)`:

- If file contents need to be modified to include CMake variable substitutions, `configure_file()` is the most concise way to achieve it.
- If a file just needs to be copied but its name will change, the syntax of `configure_file()` is slightly shorter than `file(COPY…)`, but either would be suitable.
- If copying more than one file or a whole directory structure, the `file(COPY)` or `file(INSTALL)` command must be used.
- If control over file or directory permissions is required as part of the copy, `file(COPY)` or `file(INSTALL)` must be used if the project's minimum CMake version is 3.19 or earlier.
- `file(INSTALL)` should only typically be used as part of install scripts. Prefer `file(COPY)` instead for other situations.

Prior to CMake 3.10, the `file(GENERATE…)` command had different handling of relative paths compared to most other commands provided by CMake. Rather than relying on developers being aware of this different behavior, projects should instead prefer to always specify the `INPUT` and `OUTPUT` files with an absolute path to avoid errors or files being generated in unexpected locations.

When downloading or uploading files with the `file(DOWNLOAD…)` or `file(UPLOAD…)` commands, security and efficiency aspects should be carefully considered. Strive to avoid embedding any sort of authentication details (usernames, passwords, private keys, etc.) in any file stored in a version control system for the project's sources. Such details should come from outside the project, such as through environment variables (still somewhat insecure), files found on the user's file system with appropriate permissions limiting access or a keychain of some kind. Make use of the `EXPECTED_HASH` option when downloading to re-use previously downloaded content from an earlier run and avoid a potentially time-consuming remote operation. If the downloaded file's hash cannot be known in advance, then the `TLS_VERIFY` option is highly recommended to ensure the integrity of the content. Also consider specifying a `TIMEOUT`, `INACTIVITY_TIMEOUT` or both to prevent a configure run from blocking indefinitely if network connectivity is poor or unreliable.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

Chapter 20. Specifying Version Details

Versioning is one of those things that frequently doesn't get the attention it deserves. The importance of what a version number communicates to users is often underestimated, resulting in users with unmet expectations or confusion about changes between releases. There are also the inevitable tensions between marketing and how a versioning strategy affects the technical implementation of builds, packaging and so on. Thinking about and establishing these things early places the project in a better position when it comes time to deliver the first public release. This chapter explores ways to implement an effective versioning strategy, taking advantage of CMake features to provide a robust, efficient process.

20.1. Project Version

A project version often needs to be defined near the beginning of the top level `CMakeLists.txt` file so that various parts of the build can refer to it. Source code may want to embed the project version so that it can be displayed to the user or recorded in a log file, packaging steps may need it to define release version details and so on. One could simply set a variable near the start of the `CMakeLists.txt` file to record a version number in whatever form is needed like so:

```
cmake_minimum_required(VERSION 3.0)
project(FooBar)
set(FooBar_VERSION 2.4.7)
```

If individual components need to be extracted, a slightly more involved set of variables may need to be defined. One example may look something like this:

```
cmake_minimum_required(VERSION 3.0)
project(FooBar)
set(FooBar_VERSION_MAJOR 2)
set(FooBar_VERSION_MINOR 4)
set(FooBar_VERSION_PATCH 7)
set(FooBar_VERSION
    ${FooBar_VERSION_MAJOR}.${FooBar_VERSION_MINOR}.${FooBar_VERSION_PATCH}
)
```

Different projects may use different conventions for the naming of variables. The structure of version numbers can also vary from project to project, with the resultant lack of consistency making it that much more difficult to bring together many projects as part of a larger collection or superbuild (discussed in Section 29.1, "Superbuild Structure"). CMake 3.0 introduced new functionality which makes specifying version details easier and brings some consistency to project version numbering. The `VERSION` keyword was added to the `project()` command, mandating a version number of the form *major.minor.patch.tweak* as the expected format. From that information, a set of variables are automatically populated to make the full version string as well as each version component individually available to the rest of the project. Where a version string is provided with some parts omitted (the *tweak* part is often left out, for example), the corresponding variables are left empty. The following table shows the automatically populated version variables when the `VERSION` keyword is used with the `project()` command:

Exclusively for sbmain17@gmail.com Transaction: 0098315591

PROJECT_VERSION	projectName_VERSION
PROJECT_VERSION_MAJOR	projectName_VERSION_MAJOR
PROJECT_VERSION_MINOR	projectName_VERSION_MINOR
PROJECT_VERSION_PATCH	projectName_VERSION_PATCH
PROJECT_VERSION_TWEAK	projectName_VERSION_TWEAK

The two sets of variables serve slightly different purposes. The project-specific projectName_⋯ variables can be used to obtain the version details anywhere from the current directory scope or below. A call like project(FooBar VERSION 2.7.3) results in variables named FooBar_VERSION, FooBar_VERSION_MAJOR and so on. Since no two calls to project() can use the same projectName, these project-specific variables won't be overwritten by other calls to the project() command. The PROJECT_⋯ variables, on the other hand, are updated every time project() is called, so they can be used to provide the version details of the most recent call to project() in the current scope or above. From CMake 3.12, an analogous set of variables also provides the version details set by the project() call in the top level CMakeLists.txt file. These variables are:

CMAKE_PROJECT_VERSION
CMAKE_PROJECT_VERSION_MAJOR
CMAKE_PROJECT_VERSION_MINOR
CMAKE_PROJECT_VERSION_PATCH
CMAKE_PROJECT_VERSION_TWEAK

This same pattern is also followed to provide variables for the project name, description and homepage url, the latter two being added in CMake versions 3.9 and 3.12 respectively. As a general guide, the PROJECT_⋯ variables can be useful for generic code (especially modules) as a way to define sensible defaults for things like packaging or documentation details. The CMAKE_PROJECT_⋯ variables are sometimes used for defaults too, but they can be a bit less reliable since their use typically assumes a particular top level project. The projectName_⋯ variables are the most robust, since they are always unambiguous in which project's details they will provide.

When working with projects that support CMake versions earlier than 3.0, it is sometimes the case that they will define their own version-related variables which clash with those automatically defined by CMake 3.0 and later. This can lead to CMP0048 policy warnings which highlight the conflict. The following shows an example of code which leads to such a warning:

```
cmake_minimum_required(VERSION 2.8.12)

set(FooBar_VERSION 2.4.7)
project(FooBar)
```

In the above, the FooBar_VERSION variable is explicitly set, but this variable name conflicts with the variable that the project() command would automatically define. The resultant policy warning is intended as an encouragement for the project to either use a different variable name or to update to a minimum CMake version of 3.0 and set the version details in the project() command instead.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

20.2. Source Code Access To Version Details

Once the version details are defined in the `CMakeLists.txt` file, a very common need is to make them available to source code compiled by the project. A number of different approaches can be used, each with their own strengths and weaknesses. One of the most common techniques used by those new to CMake is to add a compiler define at the top level of the project:

```
cmake_minimum_required(VERSION 3.0)
project(FooBar VERSION 2.4.7)
add_definitions(-DFOOBAR_VERSION=\"${FooBar_VERSION}\")
```

This makes the version available as a raw string able to be used like so:

```
void printVersion()
{
    std::cout << FOOBAR_VERSION << std::endl;
}
```

While this approach is fairly simple, adding the definition to the compilation of every single file in the project comes with some drawbacks. Apart from cluttering up the command line of every file to be compiled, it means that any time the version number changes, the whole project gets rebuilt. This may seem like a minor point, but developers who regularly switch between different branches in a source control system will almost certainly get very annoyed by all the unnecessary recompilations. A slightly better approach uses source properties to define the `FOOBAR_VERSION` symbol only for those files where it is needed. For example:

```
cmake_minimum_required(VERSION 3.0)
project(FooBar VERSION 2.4.7)

add_executable(FooBar main.cpp src1.cpp src2.cpp ...)

get_source_file_property(defs src1.cpp COMPILE_DEFINITIONS)
list(APPEND defs "FOOBAR_VERSION=\"${FooBar_VERSION}\"")
set_source_files_properties(src1.cpp PROPERTIES
    COMPILE_DEFINITIONS "${defs}"
)
```

This avoids adding the compiler definition to every file, instead only adding it to those files that need it. As mentioned in Section 9.5, “Source Properties”, however, there can be negative impacts on the build dependencies when setting individual source properties and these once again result in more files being rebuilt than should be necessary. Therefore, this approach may seem like an improvement, but often it won’t be.

Rather than passing the version details on the command line, another common approach is to use `configure_file()` to write a header file that supplies the version details. For example:

Exclusively for sbmain17@gmail.com Transaction: 0098315591

foobar_version.h.in

```
#include <string>

inline std::string getFooBarVersion()
{
    return "@FooBar_VERSION@";
}

inline unsigned getFooBarVersionMajor()
{
    return @FooBar_VERSION_MAJOR@;
}

inline unsigned getFooBarVersionMinor()
{
    return @FooBar_VERSION_MINOR@ +0;
}

inline unsigned getFooBarVersionPatch()
{
    return @FooBar_VERSION_PATCH@ +0;
}

inline unsigned getFooBarVersionTweak()
{
    return @FooBar_VERSION_TWEAK@ +0;
}
```

main.cpp

```
#include "foobar_version.h"
#include <iostream>

int main(int argc, char* argv[])
{
    std::cout << "VERSION = " << getFooBarVersion() << "\n"
              << "MAJOR   = " << getFooBarVersionMajor() << "\n"
              << "MINOR   = " << getFooBarVersionMinor() << "\n"
              << "PATCH   = " << getFooBarVersionPatch() << "\n"
              << "TWEAK   = " << getFooBarVersionTweak()
              << std::endl;
}
```

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.0)
project(FooBar VERSION 2.4.7)

configure_file(foobar_version.h.in foobar_version.h @ONLY)
add_executable(FooBar main.cpp)
target_include_directories(FooBar PRIVATE "${CMAKE_CURRENT_BINARY_DIR}")
```


Exclusively for sbmain17@gmail.com Transaction: 0098315591

The `+0` in `foobar_version.h.in` is necessary for the *minor*, *patch* and *tweak* parts to allow their corresponding variables to be empty in the case of those version components being omitted.

Providing version details through a header like this is an improvement over the previous techniques. The version details are not included on the command line of any source file's compilation and only those files that `#include` the `foobar_version.h` header will be recompiled when the version details change. Providing all of the different version components rather than just the version string also has no impact on command lines. Nevertheless, if the version number is needed in many different source files, this can still result in more recompilation than is really necessary. This approach can be further refined by moving the implementations out of the header into their own `.cpp` file and compiling that as its own library.

foobar_version.h

```
#include <string>

std::string getFooBarVersion();
unsigned    getFooBarVersionMajor();
unsigned    getFooBarVersionMinor();
unsigned    getFooBarVersionPatch();
unsigned    getFooBarVersionTweak();
```

foobar_version.cpp.in

```
#include "foobar_version.h"

std::string getFooBarVersion()
{
    return "@FooBar_VERSION@";
}

unsigned getFooBarVersionMajor()
{
    return @FooBar_VERSION_MAJOR@;
}

unsigned getFooBarVersionMinor()
{
    return @FooBar_VERSION_MINOR@ +0;
}

unsigned getFooBarVersionPatch()
{
    return @FooBar_VERSION_PATCH@ +0;
}

unsigned getFooBarVersionTweak()
{
    return @FooBar_VERSION_TWEAK@ +0;
}
```

Exclusively for sbmain17@gmail.com Transaction: 0098315591

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.0)
project(FooBar VERSION 2.4.7)

configure_file(foobar_version.cpp.in foobar_version.cpp @ONLY)
add_library(FooBar_version STATIC ${CMAKE_CURRENT_BINARY_DIR}/foobar_version.cpp)
target_include_directories(FooBar_version PUBLIC ${CMAKE_CURRENT_SOURCE_DIR})

add_executable(FooBar main.cpp)
target_link_libraries(FooBar PRIVATE FooBar_version)

add_library(FooToolkit mylib.cpp)
target_link_libraries(FooToolkit PRIVATE FooBar_version)
```

This arrangement has none of the drawbacks of the previous approaches. When the version details change, only one source file needs to be recompiled (the generated `foobar_version.cpp` file) and the `FooBar` and `FooToolkit` targets only need to be relinked. The `foobar_version.h` header never changes, so any file that depends on it does not become out of date when the version details change. No options are added to the compilation command line of any source file either, so no other recompilations are triggered as a result of changing version details.

In situations where the project provides a library and header as part of a release package, the above arrangement is also robust. The header does not contain the version details, the library does. Therefore, code using the library can call the version functions and be confident that the details they receive are those the library was built with. This can be helpful in complicated end user environments where multiple versions of a project might be installed and not necessarily structured how the project intended.

One variant of this approach is to make `FooBar_version` an object library rather than a static library. The end result is more or less the same, but there isn't much to be gained and it may feel less natural to some developers. Making it a shared library loses some of the robustness advantages and again introduces a little more complexity for little benefit. In general, a static library is the better choice.

If the version functions are to be exposed as part of the API for a broader shared library, then the additional concerns discussed in Section 21.5, "Symbol Visibility" and Section 21.6, "Mixing Static And Shared Libraries" may need to be considered. In such cases, it may be more appropriate to add the `foobar_version.cpp` file to that shared library directly rather than creating a separate static library for it.

20.3. Source Control Commits

It is not unusual for projects to want to record details related to their source control system. This might include the revision or commit hash of the sources at the time of the build, the name of the current branch or most recent tag and so on. The approach outlined above with version details provided through a dedicated `.cpp` file lends itself well to adding more functions to return such details. For example, the current git hash can be provided relatively easily:

Exclusively for sbmain17@gmail.com Transaction: 0098315591

foobar_version.cpp.in

```
std::string getFooBarGitHash()
{
    return "@FooBar_GIT_HASH@";
}
// Other functions as before...
```

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.0)
project(FooBar VERSION 2.4.7)

# The find_package() command is covered later in the Finding Things chapter.
# Here, it provides the GIT_EXECUTABLE variable after searching for the
# git binary in some standard/well-known locations for the current platform.
find_package(Git REQUIRED)
execute_process(
    COMMAND ${GIT_EXECUTABLE} rev-parse HEAD
    RESULT_VARIABLE result
    OUTPUT_VARIABLE FooBar_GIT_HASH
    OUTPUT_STRIP_TRAILING_WHITESPACE
)
if(result)
    message(FATAL_ERROR "Failed to get git hash: ${result}")
endif()

configure_file(foobar_version.cpp.in foobar_version.cpp @ONLY)
# Targets, etc....
```

A slightly more interesting example is measuring how many commits have occurred since a particular file changed. Consider embedding the project's version in a separate file rather than in the `CMakeLists.txt` file, where the only thing in this separate file is the project version number. A reasonable assumption can then be made that the file only changes when the version number changes. As a result, measuring the number of commits since that file changed on the current branch is generally a good measure of the number of commits since the last version update.

The following example moves the project version out to a separate file named `projectVersionDetails.cmake` and provides the number of commits through a new function in the generated `foobar_version.cpp` file. It demonstrates a pattern suitable for any project where the version is set by the top level `project()` call, but in a way that won't interfere with a parent project if it is incorporated into a larger project hierarchy (a topic discussed in Section 28.2, "FetchContent").

foobar_version.cpp.in

```
unsigned getFooBarCommitsSinceVersionChange()
{
    return @FooBar_COMMITS_SINCE_VERSION_CHANGE@;
}
// Other functions as before...
```

Exclusively for sbmain17@gmail.com Transaction: 0098315591

projectVersionDetails.cmake

```
# This file should contain nothing but the following line
# setting the project version. The variable name must not
# clash with the FooBar_VERSION* variables automatically
# defined by the project() command.
set(FooBar_VER 2.4.7)
```

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.0)
include(projectVersionDetails.cmake)
project(FooBar VERSION ${FooBar_VER})

find_package(Git REQUIRED)
execute_process(
    COMMAND ${GIT_EXECUTABLE} rev-list -1 HEAD projectVersionDetails.cmake
    RESULT_VARIABLE result
    OUTPUT_VARIABLE lastChangeHash
    OUTPUT_STRIP_TRAILING_WHITESPACE
)
if(result)
    message(FATAL_ERROR "Failed to get hash of last change: ${result}")
endif()

execute_process(
    COMMAND ${GIT_EXECUTABLE} rev-list ${lastChangeHash}..HEAD
    RESULT_VARIABLE result
    OUTPUT_VARIABLE hashList
    OUTPUT_STRIP_TRAILING_WHITESPACE
)
if(result)
    message(FATAL_ERROR "Failed to get list of git hashes: ${result}")
endif()
string(REGEX REPLACE "[\n\r]+" ";" hashList "${hashList}")
list(LENGTH hashList FooBar_COMMITS_SINCE_VERSION_CHANGE)

configure_file(foobar_version.cpp.in foobar_version.cpp @ONLY)
# Targets, etc....
```

The above approach works out the git hash of the last change to the version details file, then uses `git rev-list` to obtain the list of commit hashes for the whole repository since that commit. The commits are initially found as a string with one hash per line, which is then converted into a CMake list by replacing newline characters with the list separator (`;`). The `list()` command then simply counts how many items are in the list to give the number of commits. A simpler approach would use `git rev-list --count` to obtain the number directly, but older versions of git do not support the `--count` option, so the above method is preferable if older git versions need to be supported.

Other variations are also possible. Some projects use `git describe` to provide various details including branch names, most recent tag, etc., but note that tag and branch details can change without changing commits. If a branch or tag is moved or renamed, the build might not be repeatable. If version details only rely on file commit hashes, no such weakness is created. This also

Exclusively for sbmain17@gmail.com Transaction: 0098315591

gives the project freedom in creating, renaming or deleting tags as needed after builds have confirmed the commits have no errors (think of release tags being applied to commits after continuous integration builds, testing, etc. confirm there are no problems).

Source control systems like Subversion present other challenges. On the one hand, Subversion maintains a global revision number for the whole repository, so there is no need to first obtain commit hashes and then count them. But Subversion also has the complication that it allows mixing different revisions of different files. As a result, approaches like the one outlined above for git can be defeated by a developer checking out different revisions of files but leaving the project version file alone. This is not a scenario one would expect for an automated continuous integration system, but it may be more likely for a developer working locally on their own machine, depending on the way they like to work.

Another consideration of techniques like those above is what forces the generated version `.cpp` file to be updated. CMake ensures the configure step is re-run if the project version file changes, since it is brought into the main `CMakeLists.txt` file via an `include()` command. If, however, commits are made to other files, CMake will not be aware of them. It may be possible to implement hooks into the version control system (e.g. git's post-commit hook) to force CMake to re-run, but this is more likely to annoy developers than to help them. Ultimately, a compromise between convenience and robustness will typically be made. That said, the accuracy of the source control details will likely only be critical for releases and it should be easy enough to ensure that the release process explicitly invokes CMake.

20.4. Recommended Practices

Projects are not required to follow any particular versioning system, but by following the *major.minor.patch.tweak* format, certain functionality comes for free with CMake and new developers have an easier time understanding the versioning used by the project. As will be seen in later chapters (notably Chapter 27, *Packaging*), the version format is more important when making packaged releases, but since many projects report their own version number at run time, the version format affects the build as well.

The meaning of each of the numbers making up the version format is up to the project, but there are conventions that end users often expect. For example, a change in the *major* value usually means a significant release, often involving changes that are not backward compatible or that represent a change in direction for the project. If a *minor* value changes, users tend to see this as an incremental release, most likely adding new features without breaking existing behavior. When only the *patch* value changes, users may not see it as a particularly important change and expect it to be relatively minor, such as fixing some bugs but not introducing new functionality. The tweak value is often omitted and doesn't tend to have a common interpretation beyond being even less significant than *patch*. Note that these are just general observations, projects can and do give the version numbers completely different meanings. For ultimate simplicity, a project might use just a single number and nothing else, effectively specifying every release as a new *major* version. While this would be easy to implement, it would also provide less guidance to end users and require good quality release notes to manage user expectations between each version.

The `VERSION` keyword of the `project()` command is one example of how CMake provides extra convenience when the *major.minor.patch.tweak* format is used. The project provides a single

Exclusively for sbmain17@gmail.com Transaction: 0098315591

version string and the `project()` command automatically defines a set of variables making the various parts of the version number available. Some CMake modules may also use these variables as defaults for certain meta data, so it is generally advisable to set the project version with the `project()` command using the `VERSION` keyword. This keyword was added in CMake 3.0, but if supporting older CMake versions, this functionality still needs to be considered. Projects should not define variables whose names clash with the automatically defined ones or else later CMake versions will issue a warning. Avoid explicitly setting variables with names of the form `xxx_VERSION` or `xxx_VERSION_yyy` to prevent such warnings.

When defining the version number, consider doing so in its own dedicated file which CMake then pulls in via an `include()` command. This allows the project to take advantage of changes in version number aligning with changes in that file as seen by the project's source control system. To minimize unnecessary recompilation on version changes, generate a `.c` or `.cpp` file which contains functions that return version details rather than embedding those details in a generated header or as compiler definitions to be passed on the command line. Also ensure that names given to such functions incorporate something specific to the project or place them in a project-specific namespace. This allows the same pattern to be replicated across many projects which may later be combined into a single build without causing name clashes.

Establish versioning strategies and implementation patterns early in a project's life. This helps developers gain a clear understanding about how and when version details get updated and it encourages thinking about the release process well before the pressures of the first delivery. It also allows less efficient approaches to be weeded out early so that build efficiency is maximized in advance of releases where version numbers change and where build turnaround times may become more important.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

Chapter 21. Libraries

Compared to writing ordinary applications, creating and maintaining libraries is typically more involved, especially shared libraries. All the usual concerns about code correctness and maintainability still apply, but shared libraries in particular also bring with them additional considerations relating to API consistency, preserving binary compatibility between releases, symbol visibility and more. Furthermore, each platform typically has its own set of unique features and requirements, making cross-platform library development a challenging task.

For the most part, however, a core set of capabilities are supported by all major platforms, it's just that the way to define or use them varies. CMake provides a number of features which abstract away these differences so that developers can focus on the capabilities and leave the implementation details up to the build system.

21.1. Build Basics

The fundamental command for defining a library was covered in previous chapters and has the following form:

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

A shared library will be produced if either the `SHARED` or `MODULE` keyword is provided. Alternatively, if no `STATIC`, `SHARED`, `MODULE` or `OBJECT` keyword is given, a shared library will be produced if the `BUILD_SHARED_LIBS` variable has a value of true at the time `add_library()` is called.

The main difference between `SHARED` and `MODULE` is that `SHARED` libraries are intended for other targets to link against, whereas `MODULE` libraries are not. `MODULE` libraries are typically used for things like plugins or other optional libraries that can be loaded at runtime. The loading of such libraries is often dependent on an application configuration setting or detection of some system feature. Other executables and libraries do not normally link against a `MODULE` library.

On most Unix-based platforms, the file name of a `STATIC` or `SHARED` library will have `lib` prepended by default, whereas `MODULE` might not. Apple platforms also support frameworks and loadable bundles, which allow additional files to be bundled with the library in a well-defined directory structure. This is covered in detail in Section 23.3, "Frameworks".

On Windows platforms, library names do not have any `lib` prefix prepended, regardless of the type of library. Static library targets produce a single `.lib` archive, whereas shared library targets result in two separate files, one for the runtime (the `.dll` or dynamic link library) and the other for linking against at build time (i.e. the `.lib` import library). Developers sometimes confuse import and static libraries due to the same file suffix being used for both, but CMake generally handles them correctly without any special intervention.

When using GNU tools on Windows (e.g. with the MinGW or MSYS project generators), CMake has the ability to convert GNU import libraries (`.dll.a`) to the same format that Visual Studio produces (`.lib`). This can be useful if distributing a shared library built with GNU tools to enable it to be

Exclusively for sbmain17@gmail.com Transaction: 0098315591

linked to binaries built with Visual Studio. Note that Visual Studio must be installed for this conversion to be possible. The conversion is enabled by setting the `GNUtoMS` target property to true for a shared library. This target property is initialized by the value of the `CMAKE_GNUtoMS` variable at the time `add_library()` is called.

21.2. Linking Static Libraries

CMake handles some special cases specific to linking static libraries. If a library `A` is listed as a `PRIVATE` dependency for a static library target `B`, then `A` will effectively be treated as a `PUBLIC` dependency as far as linking is concerned (and *only* for linking). This is because the private `A` library will still need to be added to the linker command line of anything linking to `B` in order for symbols from `A` to be found at link time. If `B` was a shared library, the private library `A` that it depends on would not need to be listed on the linker command line. This is all handled transparently by CMake, so the developer typically doesn't need to concern themselves with the details beyond specifying the `PUBLIC`, `PRIVATE` and `INTERFACE` dependencies with `target_link_libraries()`.

In typical projects, static libraries will not contain cyclic dependencies where two or more libraries depend on each other. Nevertheless, some scenarios give rise to such situations and CMake will recognize and handle the cyclic dependency as long as the relevant linking relationships have been specified (i.e. by `target_link_libraries()`). A slightly modified version of the example from the CMake documentation highlights the behavior:

```
add_library(A STATIC a.cpp)
add_library(B STATIC b.cpp)
target_link_libraries(A PUBLIC B)
target_link_libraries(B PUBLIC A)
add_executable(Main main.cpp)
target_link_libraries(Main A)
```

In the above, the link command for `Main` will contain `A B A B`. This repetition is provided automatically by CMake without developer intervention, but in certain pathological cases, more than one repetition may be required. While CMake provides the `LINK_INTERFACE_MULTIPLICITY` target property for this purpose, such situations usually point to a need for the project to be restructured. `OBJECT` libraries may also be a useful tool for addressing such deep interdependencies, since they effectively act like a collection of sources rather than actual libraries. The ordering of object files on the linker command line is usually not important, whereas library ordering typically is.

21.3. Shared Library Versioning

A CMake project which does not expect its libraries to be used outside of the project itself doesn't typically need version information for any shared libraries it creates. The whole project tends to be updated together when deployed, so there are few issues about ensuring binary compatibility between releases, etc. But if the project provides libraries and other software could link against them, library versioning becomes very important. Library version details add greater robustness, allowing other software to specify the interface they expect to link against and have available to them at run time.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

Most platforms offer functionality for specifying the version number of a shared library, but the way it is done varies considerably. Platforms generally have the ability to encode version details into the shared library binary and this information is sometimes used to determine whether a binary can be used by another executable or shared library that links to it. Some platforms also have conventions for setting up files and symbolic links with different levels of the version number in their names. On Linux, for example, a common set of file and symbolic links for a shared library might look like this:

```
libMyStuff.so.2.4.3
libMyStuff.so.2 --> libMyStuff.so.2.4.3
libMyStuff.so   --> libMyStuff.so.2
```

CMake takes care of most of the platform differences with regard to version handling for shared libraries. When linking a target to a shared library, it will follow platform conventions when deciding which of the file or symlink names to link against. When building a shared library, CMake automates the creation of the full set of files and symlinks if version details are provided.

A shared library's version details are defined by the `VERSION` and `SOVERSION` target properties. The interpretation of these properties is different across the platforms CMake supports, but by following *semantic versioning* principles, these differences can be handled fairly seamlessly. Semantic versioning assumes a version number is specified in the form *major.minor.patch*, where each version component is an integer. The `VERSION` property would be set to the full *major.minor.patch*, whereas `SOVERSION` would be set to just the *major* part. As a project evolves and makes releases, semantic versioning implies that the version details should be modified as follows:

- When an incompatible API change is made, increment the *major* part of the version and reset the *minor* and *patch* parts to 0. This means the `SOVERSION` property will change every time there is an API breakage and *only* if there is an API breakage.
- When functionality is added in a backwards compatible manner, increment the *minor* part and reset the *patch* to 0. The *major* part remains unchanged.
- When a backwards compatible bug fix is made, increment the *patch* value and leave the *major* and *minor* parts unchanged.

If the version details of a shared library are modified according to these principles, API incompatibility issues at run time will be minimized on all platforms. Consider the following example, which produces the set of symbolic links shown earlier for Linux:

```
add_library(MyStuff SHARED source1.cpp ...)
set_target_properties(MyStuff PROPERTIES
    VERSION   2.4.3
    SOVERSION 2
)
```

On Apple platforms, the `otool -L` command can be used to print the version details encoded into the resultant shared library. The output for the shared library produced by the above example would report the version details as having a *compatibility version* of 2.0.0 and *current version* 2.4.3. Anything that linked against the `MyStuff` library would have the name `libMyStuff.2.dylib` encoded

Exclusively for sbmain17@gmail.com Transaction: 0098315591

into it as the name of the library to look for at run time. Linux platforms show a similar structure in their symbolic links for shared libraries and normal practice is to use just the *major* part for the library's soname.

CMake 3.17 added the `MACHO_COMPATIBILITY_VERSION` and `MACHO_CURRENT_VERSION` target properties to support advanced use cases for Apple platforms (typically related to matching `libtool` conventions). These additional properties allow file and symlink naming to be decoupled from the internal names embedded in the Mach-O binaries. Projects should rarely need this more complex functionality and are advised to avoid using these properties unless specific scenarios require them.

On Windows, CMake behavior is to extract a *major.minor* version from the `VERSION` property and encode that into the DLL as the DLL image version. Windows does not have the concept of a soname, so the `SOVERSION` property is not used. Nevertheless, following semantic versioning principles will at least ensure that the DLL version can be used to determine the compatibility of the library with binaries that link against it.

It should be noted that semantic versioning is not strictly required by any platform. Rather, it provides a well defined specification which brings some certainty around dependency management between shared libraries and the things that use them. It happens to closely reflect how library versions are usually interpreted on most Unix-based platforms and CMake aims to make the most of the `VERSION` and `SOVERSION` target properties to provide shared libraries which follow native platform conventions.

Projects should be aware that if only one of the `VERSION` and `SOVERSION` target properties are set, on most platforms the missing one is treated as though it had the same value as the one that was provided. This is unlikely to result in good version handling unless just a single number is used for the version number (i.e. no minor or patch parts). Such version numbering may be appropriate in certain cases, but projects should generally endeavor to follow the principles discussed above for more flexible and more robust runtime behavior.

21.4. Interface Compatibility

The `VERSION` and `SOVERSION` target properties allow API versioning to be specified at the binary level in a platform-independent manner. CMake also provides other properties which can be used to define requirements for compatibility between CMake targets when they are linked to one another. These can be used to describe and enforce details that version numbering alone cannot capture.

Consider a realistic example where a networking library only provides support for the `https://` protocol and other similar secure capabilities if an appropriate SSL toolkit is available. Other parts of the program may need to adjust their own functionality based on whether or not SSL is supported, while the program as a whole should be consistent about whether or not SSL features can be used. This can be enforced with an *interface compatibility* property.

A few different types of interface compatibility properties can be defined, but the simplest is a boolean property. The basic idea is that libraries specify the name of a property they will use to advertise a particular boolean state and then they define that property with the relevant value. When multiple libraries that are being linked together define the same property name for an interface compatibility, CMake will check that they specify the same value and issue an error if they are different. A basic example looks something like this:

Exclusively for sbmain17@gmail.com Transaction: 0098315591

```
add_library(Networking net.cpp)
set_target_properties(Networking PROPERTIES
    COMPATIBLE_INTERFACE_BOOL SSL_SUPPORT
    INTERFACE_SSL_SUPPORT YES
)

add_library(Util util.cpp)
set_target_properties(Util PROPERTIES
    COMPATIBLE_INTERFACE_BOOL SSL_SUPPORT
    INTERFACE_SSL_SUPPORT YES
)

add_executable(MyApp myapp.cpp)
target_link_libraries(MyApp PRIVATE Networking Util)
target_compile_definitions(MyApp PRIVATE
    $<$<BOOL:$<TARGET_PROPERTY:SSL_SUPPORT>>:HAVE_SSL>
)
```

Both library targets advertise that they define an interface compatibility for the property name `SSL_SUPPORT`. The `COMPATIBLE_INTERFACE_BOOL` property is expected to hold a list of names, each of which requires an associated property of the same name with `INTERFACE_` prepended to be defined on that target. When the libraries are used together as a link dependency for `MyApp`, CMake checks that both libraries define `INTERFACE_SSL_SUPPORT` with the same value. In addition, CMake will also automatically populate the `SSL_SUPPORT` property of the `MyApp` target with the same value too, which can then be used as part of a generator expression and made available to the source code of `MyApp` as a compile definition as shown. This allows the `MyApp` code to tailor itself to whether or not SSL support has been compiled into the libraries it uses. Continuing with the example, rather than `MyApp` simply detecting whether or not SSL support is available, it can specify a requirement by explicitly defining its `SSL_SUPPORT` property to hold the value that the libraries must be compatible with. In that case, rather than automatically populating the `SSL_SUPPORT` property of `MyApp`, CMake will compare the values and ensure the libraries are consistent with the specified requirement.

```
# Require libraries to have SSL support
set_target_properties(MyApp PROPERTIES SSL_SUPPORT YES)
```

The above examples are somewhat contrived, the same constraints could have been enforced in other ways. The real advantages of interface compatibility specifications start to emerge as a project becomes more complicated and its targets are spread across many directories or come from externally built projects. Interface compatibilities are assigned as properties of the targets, so they only need to be defined in one place and are then made available anywhere the target can be used without further effort. Consuming targets don't have to know the details of how the interface compatibility is determined, only the final decision stored in the target's `INTERFACE_⋯` properties.

CMake also supports interface compatibilities expressed as a string. These work essentially the same way as the boolean case except that the named properties are required to have exactly the same values and can hold any arbitrary contents. The earlier example can be modified to require that libraries use the same SSL implementation, not just agree on whether they support SSL or not:

Exclusively for sbmain17@gmail.com Transaction: 0098315591

```
add_library(Networking net.cpp)
set_target_properties(Networking PROPERTIES
    COMPATIBLE_INTERFACE_STRING SSL_IMPL
    INTERFACE_SSL_IMPL OpenSSL
)

add_library(Util util.cpp)
set_target_properties(Util PROPERTIES
    COMPATIBLE_INTERFACE_STRING SSL_IMPL
    INTERFACE_SSL_IMPL OpenSSL
)

add_executable(MyApp myapp.cpp)
target_link_libraries(MyApp PRIVATE Networking Util)
target_compile_definitions(MyApp PRIVATE
    SSL_IMPL=$<TARGET_PROPERTY:SSL_IMPL>
)
```

In the above, the `SSL_IMPL` property is used as a string interface compatibility with the libraries specifying that they use `OpenSSL` as their SSL implementation. Just as for the boolean case, the `MyApp` target could have defined its `SSL_IMPL` property to specify a requirement rather than letting CMake populate it with the value from the libraries.

The other kind of interface compatibility CMake supports is a numeric value. Numeric interface compatibilities are used to determine the *minimum* or *maximum* value defined for a property among a set of libraries rather than to require the properties to have the *same* value. This can be exploited to allow a target to detect things like a minimum protocol version it could support or to work out the largest temporary buffer size needed among the libraries it links to.

```
add_library(BigFast strategy1.cpp)
set_target_properties(BigFast PROPERTIES
    COMPATIBLE_INTERFACE_NUMBER_MIN PROTOCOL_VER
    COMPATIBLE_INTERFACE_NUMBER_MAX TMP_BUFFERS
    INTERFACE_PROTOCOL_VER 3
    INTERFACE_TMP_BUFFERS 200
)

add_library(SmallSlow strategy2.cpp)
set_target_properties(SmallSlow PROPERTIES
    COMPATIBLE_INTERFACE_NUMBER_MIN PROTOCOL_VER
    COMPATIBLE_INTERFACE_NUMBER_MAX TMP_BUFFERS
    INTERFACE_PROTOCOL_VER 2
    INTERFACE_TMP_BUFFERS 15
)

add_executable(MyApp myapp.cpp)
target_link_libraries(MyApp PRIVATE BigFast SmallSlow)
target_compile_definitions(MyApp PRIVATE
    MIN_API=$<TARGET_PROPERTY:PROTOCOL_VER>
    TMP_BUFFERS=$<TARGET_PROPERTY:TMP_BUFFERS>
)
```


Exclusively for sbmain17@gmail.com Transaction: 0098315591

In the above, PROTOCOL_VER is defined as a *minimum* numeric interface compatibility, so the PROTOCOL_VER property of MyApp will be set to the smallest value specified for the INTERFACE_PROTOCOL_VER property of the libraries it links to, which in this case is 2. Similarly, TMP_BUFFERS is defined as a *maximum* numeric interface compatibility and the MyApp TMP_BUFFERS property receives the largest value among the INTERFACE_TMP_BUFFERS property of its linked libraries, which is 200.

At this point, it would be natural to think about using the same property for both a minimum and maximum numeric interface compatibility to allow both the smallest and largest value to be detected in the parent. This is not possible because CMake does not (and cannot) allow the same property to be used with more than one kind of interface compatibility. If a property was used for multiple types of interface compatibilities, it would be impossible for CMake to know which type should be used to compute the value to be stored in the parent's result property. For example, if PROTOCOL_VER were both a minimum and maximum interface compatibility in the above example, CMake could not determine the value to store in the PROTOCOL_VER property of MyApp - should it store the minimum or maximum value? Instead, separate properties must be used to achieve this:

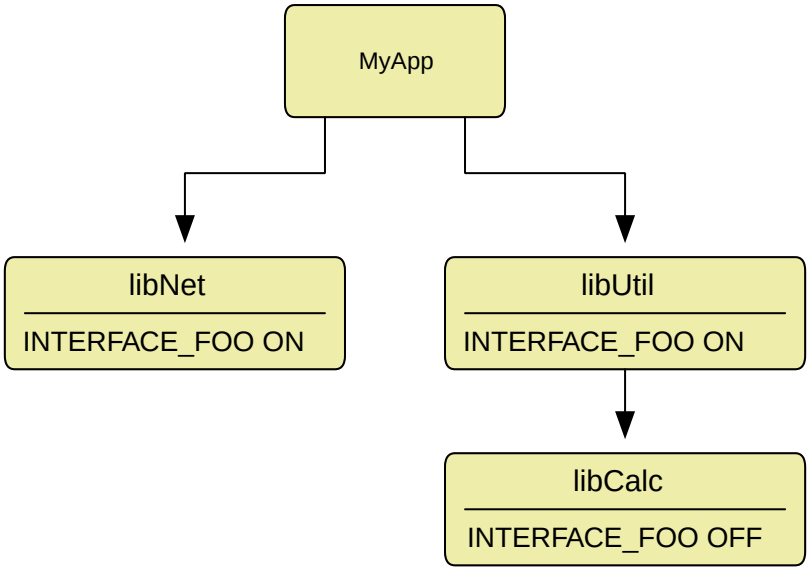
```
add_library(BigFast strategy1.cpp)
set_target_properties(BigFast PROPERTIES
    COMPATIBLE_INTERFACE_NUMBER_MIN PROTOCOL_VER_MIN
    COMPATIBLE_INTERFACE_NUMBER_MAX PROTOCOL_VER_MAX
    INTERFACE_PROTOCOL_VER_MIN 3
    INTERFACE_PROTOCOL_VER_MAX 3
)

add_library(SmallSlow strategy2.cpp)
set_target_properties(SmallSlow PROPERTIES
    COMPATIBLE_INTERFACE_NUMBER_MIN PROTOCOL_VER_MIN
    COMPATIBLE_INTERFACE_NUMBER_MAX PROTOCOL_VER_MAX
    INTERFACE_PROTOCOL_VER_MIN 2
    INTERFACE_PROTOCOL_VER_MAX 2
)

add_executable(MyApp myapp.cpp)
target_link_libraries(MyApp PRIVATE BigFast SmallSlow)
target_compile_definitions(MyApp PRIVATE
    PROTOCOL_VER_MIN=$<TARGET_PROPERTY:PROTOCOL_VER_MIN>
    PROTOCOL_VER_MAX=$<TARGET_PROPERTY:PROTOCOL_VER_MAX>
)
```

The result of the above example is that MyApp knows the range of protocol versions it needs to support based on the protocols used by the libraries it links to.

If one target defines an interface compatibility of any particular type, other targets are not required to define it too. Any target which does not define a matching interface compatibility is simply ignored for that particular property. This ensures libraries only need to define interface compatibilities that are relevant to them.

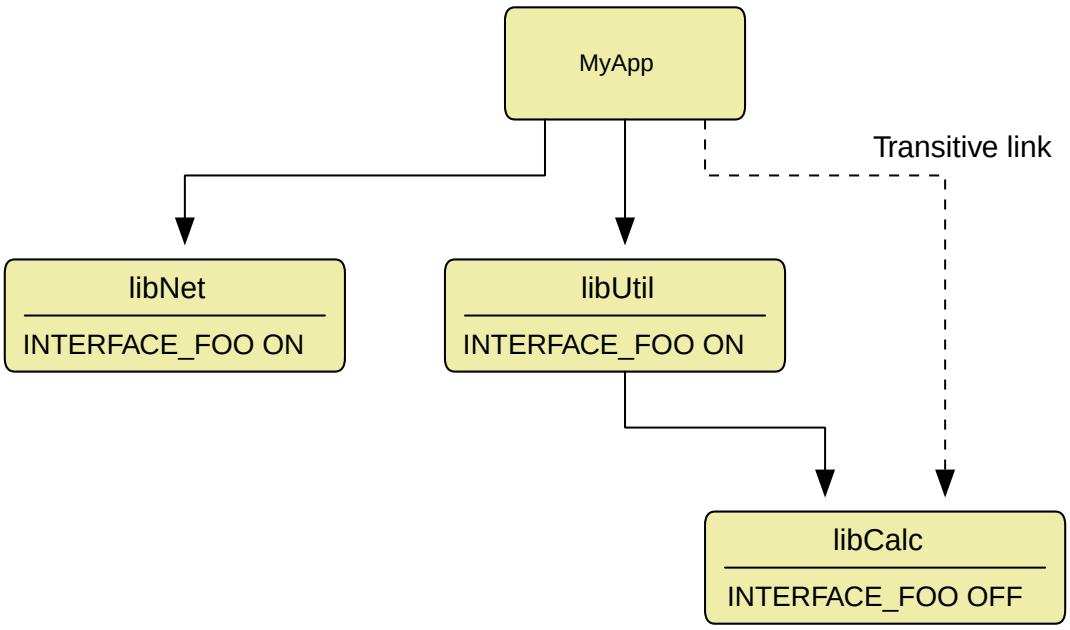
When there are multiple levels of library link dependencies, there are some subtle complexities to how interface compatibilities are handled. Consider the structure shown in the following diagram, which contains a number of library and executable targets and their direct link dependencies.

Exclusively for sbmain17@gmail.com Transaction: 0098315591



If all link dependencies are considered `PRIVATE`, then only `libNet` and `libUtil` are direct link dependencies of `MyApp`, so only those two libraries are required to have consistent values for their `INTERFACE_FOO` property. The value of that property in the `libCalc` library is not considered, since it is not a direct dependency of `MyApp`. Furthermore, the only direct link dependency of `libUtil` is `libCalc`, so the `INTERFACE_FOO` property of `libCalc` has no other library it is required to be consistent with. Even though both `libUtil` and `libCalc` define an interface compatibility for the same property name, because they are not both *direct* link dependencies of a common target, they are not required to have compatible values.

Now consider the situation where `libCalc` is a `PUBLIC` link dependency of `libUtil`. In that case, the final linking relationships will actually look like this:



Exclusively for sbmain17@gmail.com Transaction: 0098315591

When `libCalc` is a `PUBLIC` link dependency of `libUtil`, anything that links to `libUtil` will also link to `libCalc`. Thus, `libCalc` becomes a direct link dependency of `MyApp` and therefore it *does* participate in interface compatibility checking with `libNet` and `libUtil`. This means great care must taken when defining interface compatibilities to ensure that they accurately express the correct things, since their reach can extend out to targets beyond what may initially seem obvious when `PUBLIC` link relationships are involved.

21.5. Symbol Visibility

Simplistically, a library can be thought of as a container of compiled source code, providing various functions and global data which other code can call or use. For static libraries, the container is really just a collection of object files and the tool putting it together is sometimes referred to as an archiver or librarian. Shared libraries, on the other hand, are produced by the linker, which processes the object code, archives, etc. and decides what to include in the final shared library binary. Some functions and global data may be hidden, meaning they have been marked as okay for the linker to use to resolve internal code dependencies, but code outside of the shared library cannot call or use them. Other symbols are exported, so code both inside and outside of the shared library can access them. This is referred to as a symbol's *visibility*.

Compilers have different ways of specifying symbol visibility and they also have different default behaviors. Some make all symbols visible by default, whereas others hide symbols by default. Compilers also differ in the syntax used to mark individual functions, classes and data as visible or not, which adds to the complexity of writing portable shared libraries. In order to avoid some of that complexity, some developers opt to simply make all symbols visible and avoid having to explicitly mark any symbols for export. While this may initially seem like a win, it comes with a range of down sides:

- It is equivalent to saying every function, class, type, global variable, etc. is freely available for anything to use. This is rarely desirable, but may be acceptable if the project is content to rely on its documentation to define the symbols which should be considered public.
- By making all symbols visible, consuming code cannot be prevented from using things they shouldn't. Other code linking to the library may come to rely on some internal symbol, making it harder for the shared library to change its implementation or internal structure without breaking consuming projects.
- When all symbols are to be treated as visible, the linker cannot know whether each symbol will be used by anything, so it has to include them all in the final shared library. When only a subset of the symbols are exported, the linker has the opportunity to identify code which can never be used by the visible symbols and therefore discard it, often resulting in a much smaller binary, which then has the potential to load faster at run time.
- Languages like C++ which support templates have the potential to define a huge number of symbols. If all symbols are visible by default, this can result in the symbol table of a shared library growing quite large. In extreme cases, this can have a measurable impact on run time startup performance.
- Functions used in the internal implementation of the library may use names which expose details about what the library does or how it does it. This might be a security concern in some contexts, or it may reveal commercial IP that shouldn't be visible to those receiving the library.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

The above points highlight that symbol visibility is as much about enforcing the public-private nature of a library's API as it is about the low level mechanics of shared library performance and package size. Clearly, there are advantages to only exporting those symbols which should be considered public, but the compiler and platform specific nature of how to achieve that often presents a substantial hurdle for multi-platform projects. CMake considerably simplifies this process by abstracting away those differences behind a few properties, variables and a helper module.

21.5.1. Specifying Default Visibility

By default, Visual Studio compilers assume all symbols are hidden unless explicitly exported. Other compilers, such as GCC and Clang are the opposite, making all symbols visible by default and only hiding symbols if explicitly told to. If a project wishes to have the same default symbol visibility across all its compilers and platforms, one of these two approaches must be selected, but hopefully the disadvantages highlighted in the preceding section provide a compelling argument for choosing that symbols be hidden by default.

The first step to enforcing hidden default visibility is to define the `<LANG>_VISIBILITY_PRESET` set of properties on a shared library target. For the two most common languages where this functionality is used, the property names are `C_VISIBILITY_PRESET` and `CXX_VISIBILITY_PRESET` for C and C++ respectively. The value given to this property should be `hidden`, which changes the default visibility to hide all symbols. Other supported values include `default`, `protected` and `internal`, but these are less likely to be useful for cross-platform projects. They either specify what is already the default behavior or are variants of `hidden` with more specialized meanings in some contexts.

The second step is to specify that inlined functions should also be hidden by default. For C++ code making heavy use of templates, this can substantially reduce the size of the final shared library binary. This behavior is controlled by the target property `VISIBILITY_INLINES_HIDDEN` and applies to all languages. It should hold the boolean value `TRUE` to hide inline symbols by default.

Both `<LANG>_VISIBILITY_PRESET` and `VISIBILITY_INLINES_HIDDEN` can be specified on each shared library target, or a default can be set by the appropriate CMake variables. When a target is created, its `<LANG>_VISIBILITY_PRESET` property is initialized by the value of the CMake variable `CMAKE_<LANG>_VISIBILITY_PRESET` and its `VISIBILITY_INLINES_HIDDEN` property is initialized by the `CMAKE_VISIBILITY_INLINES_HIDDEN` variable. This is typically more convenient than setting the properties for each target individually.

For those projects wishing to make all symbols visible by default across all platforms, this only requires changing the default behavior of Visual Studio compilers. From version 3.4, CMake provides the `WINDOWS_EXPORT_ALL_SYMBOLS` target property which provides this behavior, but with caveats. Defining this property to a true value will cause CMake to write a `.def` file containing all symbols from all object files used to create the shared library and pass that `.def` file to the linker. This is a fairly brute force method which prevents the source code from selectively hiding any symbols, so it should only be used where *all* symbols should be made visible. This target property is initialized by the `CMAKE_WINDOWS_EXPORT_ALL_SYMBOLS` CMake variable when a shared library target is created.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

21.5.2. Specifying Individual Symbol Visibilities

Most common compilers support specifying the visibility of individual symbols, but the way they do so varies. In general Visual Studio uses one method and most other compilers follow the method used by GCC. The two share a similar structure, but they use different keywords. This means source code for languages like C, C++ and their derivatives can use a common preprocessor define for visibility control and projects can instruct CMake to provide the appropriate definition.

There are three primary cases where symbol visibility can be specified: classes, functions and variables. In the following example which contains declarations for each of these three cases, note the position of `MYTOOLS_EXPORT`:

```
class MYTOOLS_EXPORT SomeClass {...};   // Export non-private members of a class
MYTOOLS_EXPORT void someFunction();     // Make a free function visible
MYTOOLS_EXPORT extern int myGlobalVar;  // Make a global variable visible
```

When building the shared library containing the implementations of the above, `MYTOOLS_EXPORT` needs to be substituted with the relevant keywords specifying that the symbol should be *exported* for other libraries and executables to use. On the other hand, if the same declarations are read by code belonging to some other target outside of the shared library, then `MYTOOLS_EXPORT` must be substituted with the relevant keywords specifying that the symbol should be *imported*. On Windows, these keywords take the form `__declspec(...)`, whereas GCC and compatible compilers use `__attribute__(...)`.

Coming up with the right contents for `MYTOOLS_EXPORT` for all compilers and for both the exporting and importing cases can be somewhat messy. Add into the mix that developers might choose to build a library as either shared or static and the complexity grows. Thankfully, CMake provides the `GenerateExportHeader` module which handles all of these details in a very convenient fashion. This module provides the following function:

```
generate_export_header(target
    [BASE_NAME baseName]
    [EXPORT_FILE_NAME exportFileName]
    [EXPORT_MACRO_NAME exportMacroName]
    [DEPRECATED_MACRO_NAME deprecatedMacroName]
    [NO_EXPORT_MACRO_NAME noExportMacroName]
    [STATIC_DEFINE staticDefine]
    [NO_DEPRECATED_MACRO_NAME noDeprecatedMacroName]
    [DEFINE_NO_DEPRECATED]
    [PREFIX_NAME prefix]
    [CUSTOM_CONTENT_FROM_VARIABLE var]
)
```

Typically, none of the optional arguments are needed and only the shared library target name is provided. CMake writes out a header file in the current binary directory, using the target name in lowercase with `_export.h` appended as the header file name. The header provides a define for symbol export with a similarly structured name, this time using the uppercase target name with `_EXPORT` appended. The following demonstrates this typical usage:

Exclusively for sbmain17@gmail.com Transaction: 0098315591

CMakeLists.txt

```
# Hide things by default
set(CMAKE_CXX_VISIBILITY_PRESET     hidden)
set(CMAKE_VISIBILITY_INLINES_HIDDEN YES)

# NOTE: myTools.cpp must #include myTools.h
add_library(MyTools myTools.cpp)
target_include_directories(MyTools PUBLIC
    "${CMAKE_CURRENT_BINARY_DIR}"
)

# Write mytools_export.h to the current binary directory
include(GenerateExportHeader)
generate_export_header(MyTools)
```

myTools.h

```
#include "mytools_export.h"

class MYTOOLS_EXPORT SomeClass
{
    // ...
};
MYTOOLS_EXPORT void someFunction();
MYTOOLS_EXPORT extern int myGlobalVar;
```

The current binary directory is not part of the default header search path, so it needs to be added as a `PUBLIC` search path for the library to ensure the `mytools_export.h` header can be found by both the library's own source code and any other code from targets linking to the shared library.

If using the target name as part of the header file name or preprocessor define name is not desirable, the `BASE_NAME` option can be used to provide an alternative. It is transformed in the same way, being converted to lowercase and having `_export.h` appended for the file name and uppercase with `_EXPORT` appended for the preprocessor define.

CMakeLists.txt

```
include(GenerateExportHeader)
generate_export_header(MyTools BASE_NAME fooBar)
```

myTools.h

```
#include "foobar_export.h"

class FOOBAR_EXPORT SomeClass
{
    // ...
};
FOOBAR_EXPORT void someFunction();
FOOBAR_EXPORT extern int myGlobalVar;
```


Exclusively for sbmain17@gmail.com Transaction: 0098315591

If a different name should be used for the file and preprocessor define, then rather than using BASE_NAME, the EXPORT_FILE_NAME and EXPORT_MACRO_NAME options can be given. Unlike BASE_NAME, the names provided by these two options are used without any modification.

CMakeLists.txt

```
include(GenerateExportHeader)
generate_export_header(MyTools
    EXPORT_FILE_NAME  export_myTools.h
    EXPORT_MACRO_NAME API_MYTOOLS
)
```

myTools.h

```
#include "export_myTools.h"

class API_MYTOOLS SomeClass
{
    // ...
};
API_MYTOOLS void someFunction();
API_MYTOOLS extern int myGlobalVar;
```

The generate_export_header() function provides more than just this one preprocessor define, it also provides other preprocessor definitions which can be used to mark symbols as deprecated or to explicitly specify that a symbol should never be exported. The latter can be useful to prevent exporting parts of a class that is otherwise exported, such as a public member function intended for internal use within the shared library but not by code outside it. By default, the name of this preprocessor definition consists of the target name (or BASE_NAME if it is specified) with _NO_EXPORT appended, but an alternative name can be provided with the NO_EXPORT_MACRO_NAME option if desired.

CMakeLists.txt

```
include(GenerateExportHeader)
generate_export_header(MyTools
    NO_EXPORT_MACRO_NAME REALLY_PRIVATE
)
```

myTools.h

```
#include "mytools_export.h"

class MYTOOLS_EXPORT SomeClass
{
public:
    REALLY_PRIVATE void doInternalThings();
    // ...
};
```

The function's deprecation support works in a similar way, providing a preprocessor definition with the uppercased target (or BASE_NAME) name followed by _DEPRECATED, or allowing a custom name

Exclusively for sbmain17@gmail.com Transaction: 0098315591

to be specified via the DEPRECATED_MACRO_NAME option. The DEFINE_NO_DEPRECATED option can also be given, which will result in an additional preprocessor define being provided with a name consisting of the usual uppercased target or BASE_NAME followed by _NO_DEPRECATED. Like the other preprocessor defines, this name can also be overridden with the NO_DEPRECATED_MACRO_NAME option. With some compilers, symbols marked as deprecated can result in compile time warnings which draw attention to their use. This can be a helpful mechanism to encourage developers to update their code to no longer use the deprecated symbols. The following shows how the deprecation mechanisms can be used.

CMakeLists.txt

```
option(OMIT_DEPRECATED "Omit deprecated parts of MyTools")
if(OMIT_DEPRECATED)
    set(deprecatedOption "DEFINE_NO_DEPRECATED")
else()
    unset(deprecatedOption)
endif()
include(GenerateExportHeader)
generate_export_header(MyTools
    NO_DEPRECATED_MACRO_NAME OMIT_DEPRECATED
    ${deprecatedOption}
)
```

myTools.h

```
#include "mytools_export.h"

class MYTOOLS_EXPORT SomeClass
{
public:
#ifndef OMIT_DEPRECATED
    MYTOOLS_DEPRECATED void oldImpl();
#endif
    // ...
};
```

myTools.cpp

```
#include "myTools.h"

#ifndef OMIT_DEPRECATED
void SomeClass::oldImpl() { ... }
#endif
```

The above example provides a CMake cache variable to determine whether or not to compile the deprecated items. The developer has the ability to make this choice without editing any files, so verifying behavior with or without the deprecated part of an API is easy to do. This can be particularly useful if continuous integration builds have been set up to test both with and without deprecated parts of a library. It can also be useful in situations where the project is being used as a dependency of another project, allowing that other project's developers to test whether their code uses the deprecated symbols or not just by changing the CMake cache variable.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

A less common but nevertheless important case also deserves special mention. Some projects may wish to build both shared and static versions of the same library. In this case, the same set of source code needs to allow symbol exports to be enabled for the shared library build, but disabled for the static library build (also see the next section for why this won't always be the case). When both forms of library are required in the one build, they need to be different build targets, but the `generate_export_header()` function writes a header that is closely tied to a single target. In order to support this scenario, the generated header includes logic to check for the existence of one further preprocessor define before populating the export definition. The name of this special define follows the usual pattern once again, this time being the uppercased target or `BASE_NAME` followed by `_STATIC_DEFINE`, or having a custom name provided by the `STATIC_DEFINE` option. When this special preprocessor definition is defined, the export definition is forced to expand to nothing, which is typically what is needed when the target is being built as a static library. Without the special preprocessor definition, the export define has the usual contents and works as expected when building a shared library target.

When both shared and static libraries are being built for the same set of source files, the `generate_export_header()` function should be given the target that corresponds to the shared library. The special preprocessor define is then set only on the static library's target. The `BASE_NAME` option will also typically be used to make the various symbols intuitive to either form of the library rather than being specific to the shared library only. The following demonstrates the structure needed to achieve the desired result:

```
# Same source list, different library types
add_library(MyShared SHARED ${mySources})
add_library(MyStatic STATIC ${mySources})

# Shared target used for generating export header
# with the name mytools_export.h, which will be suitable
# for both the shared and static targets
include(GenerateExportHeader)
generate_export_header(MyShared BASE_NAME MyTools)

# Static target needs special preprocessor define
# to prevent symbol import/export keywords being added
target_compile_definitions(MyStatic PRIVATE
    MYTOOLS_STATIC_DEFINE
)
```

As is evident by the preceding discussion, the `generate_export_header()` function defines a number of different preprocessor definitions and there are opportunities for different targets to accidentally try to use the same names for at least some of them. To help reduce name collisions, the `PREFIX_NAME` option allows an additional string to be specified which will be prepended to the names of each preprocessor definition. When used, this option would typically be something related to the project as a whole, effectively putting all of a project's generated preprocessor names into something like a project-specific namespace.

The last option not yet discussed is `CUSTOM_CONTENT_FROM_VARIABLE`, which was only added in CMake 3.7. This option allows arbitrary content to be injected into the generated header near the end, after all of the various preprocessor logic has been added. When used, this option must be given the name of a variable whose contents should be injected, not the content itself.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

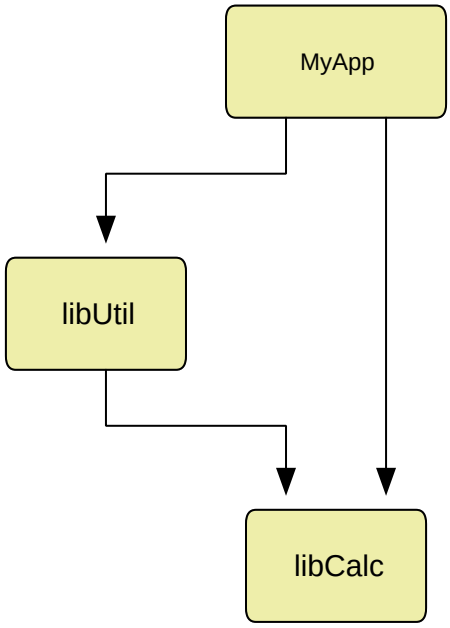
```
string(TIMESTAMP now)
set(customContents "/* Generated: ${now} */")
generate_export_header(MyTools
    CUSTOM_CONTENT_FROM_VARIABLE customContents
)
```

21.6. Mixing Static And Shared Libraries

When a project builds all its libraries as static, the build may appear to be a bit more forgiving about library link dependencies. The project may neglect to specify that one target requires another, but when various static libraries are linked into a final executable, the missing library dependencies are satisfied because they are explicitly listed for the executable in the required order. The build then succeeds, but probably only after a period of trial and error doing builds, having the linker complain about missing symbols, adding in more missing libraries or reordering the existing ones, etc.

This scenario results in success more by good fortune than by good design, but it is surprisingly common, especially with projects that define many small libraries. If link dependencies are specified for at least some of the static libraries, CMake automatically handles transitively linking those dependencies, so even if the `PRIVATE`/`PUBLIC` nature of the dependency is specified incorrectly, with a static library it is always treated as `PUBLIC` anyway and this sometimes makes builds work even though the link dependency isn't accurately described.

When library targets are defined as a mix of shared and static, the correctness of link dependencies becomes much more important. Consider the following set of targets:

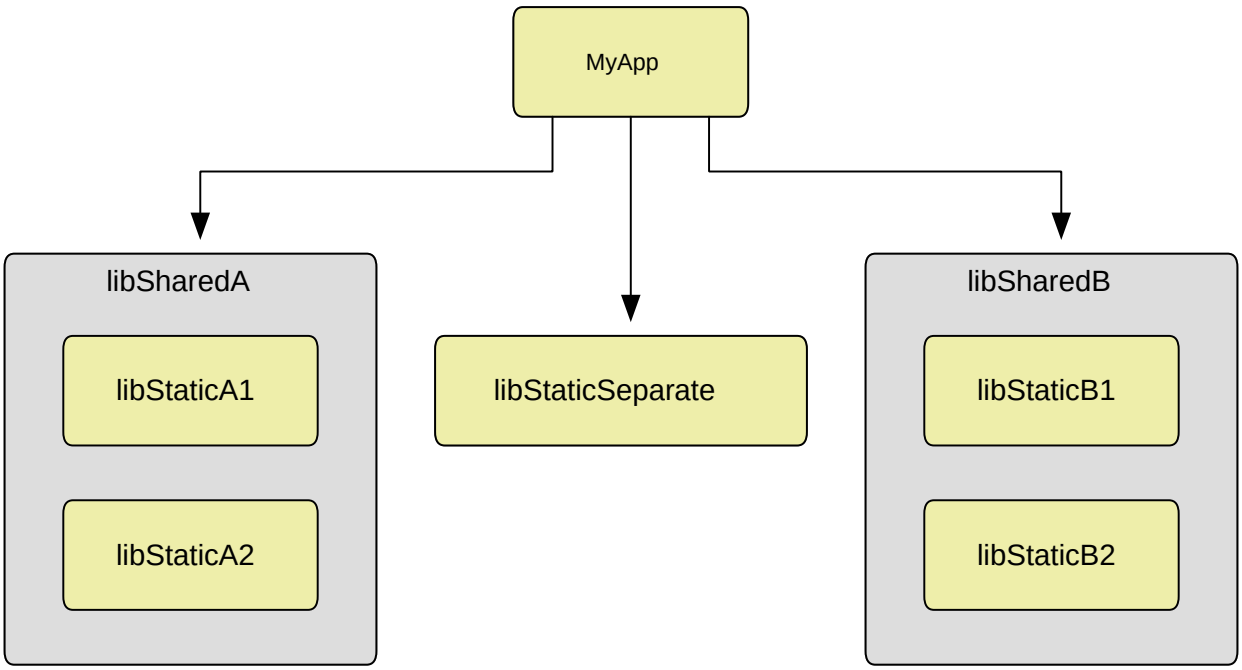


If `libUtil` and `libCalc` are static libraries, the above link dependency relationships are safe. If `libUtil` is a shared library, then the above link dependency arrangement opens up the possibility of duplicating data expected to have only one instance across a whole application. If `libCalc` defines

Exclusively for sbmain17@gmail.com Transaction: 0098315591

global data, such as might be common for a singleton or static data of a class, it may be possible for both `MyApp` and `libUtil` to have their own separate instances of that data. This becomes possible because both `MyApp` and `libUtil` require the linker to resolve symbols, so both invocations may decide the global data is required and set up an internal instance of it within that executable or shared library. If the global data is not an exported symbol, the linker won't see the instance already created in `libUtil` when it goes to link `MyApp`. The end result is that a second instance is created in `MyApp`, which is almost certain to cause hard-to-trace runtime issues. A typical manifestation of this is a variable magically appearing to change values across a function call from one executable or shared library into another shared library.

Situations similar to the above scenario can appear in a number of different forms, but the same underlying principle applies in each case. If a static library is linked into a shared library, that shared library should not be combined with any other library or executable that also links to that same static library. Ideally, if shared and static libraries are being mixed, then the static libraries should only ever exclusively be linked into one shared library and anything that needs something from one of those static libraries should link to the shared library instead. The shared library essentially has its own API and the static libraries may contribute to it.



Using static libraries to build up shared library content like this presents its own set of issues when it comes to symbol visibility. Ordinarily, the code from the static libraries would not be exported, so it would not appear as part of the shared library's exported symbols. One way to address this is to use the `generate_export_header()` function on the shared library as normal, then make the static library re-use the same export definitions. The key to making this work is to ensure the static library has a compile definition for the name of the shared library target with `_EXPORTS` appended, which is how the generated header detects whether the code is being built as part of the shared library or not.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

CMakeLists.txt

```
add_library(MyShared SHARED shared.cpp)
add_library(MyStatic STATIC static.cpp)

include(GenerateExportHeader)
generate_export_header(MyShared BASE_NAME mine)

target_link_libraries(MyShared PRIVATE MyStatic)
target_include_directories(MyShared PUBLIC ${CMAKE_CURRENT_BINARY_DIR})
target_include_directories(MyStatic PUBLIC ${CMAKE_CURRENT_BINARY_DIR})

# This makes the static library code appear to be part of the shared
# library as far as the generated export header is concerned
target_compile_definitions(MyStatic PRIVATE MyShared_EXPORTS)
```

shared.h

```
#include "mine_export.h"

MINE_EXPORT void sharedFunc();
```

static.h

```
#include "mine_export.h"

MINE_EXPORT void staticFunc();
```

The other factor to consider is whether the linker will discard code or data defined in the static library when it comes to linking the shared library. If it determines that nothing is using a particular symbol, the linker may discard it as an optimization. Special steps may need to be taken to prevent it from doing this. One choice is to make the shared library explicitly use every symbol to be retained from the shared libraries. This has the advantage that it would work for all compilers and linkers, but it may not be feasible for non-trivial projects. The alternative essentially requires linker-specific flags to be added, such as `--whole-archive` and `--no-whole-archive` for the `ld` linker on Unix systems, or `/WHOLEARCHIVE` with Visual Studio, but such functionality may not be available with all linkers. If ensuring the shared library uses each symbol exported by its static libraries isn't practical, it may be worth considering turning those static libraries into shared instead.

If a shared library only links to static libraries in a private fashion (meaning none of the static libraries' symbols need to be exported), then the situation is considerably easier. On some platforms, no further action is needed other than simply linking the shared library to the static libraries. On others, one or two minor wrinkles may arise which need to be addressed. On many 64-bit Unix systems, for example, code has to be compiled as *position independent* if it is to go into a shared library, whereas there is no such requirement for static libraries. If, however, a shared library links to a static library, then the static library *does* have to be built as position independent.

CMake provides the `POSITION_INDEPENDENT_CODE` target property as a way of transparently handling position independent behavior on those platforms that require it. When set to true, this causes that target's code to be built as position independent. By default, the property is `ON` for `SHARED` and `MODULE`

Exclusively for sbmain17@gmail.com Transaction: 0098315591

libraries and OFF for all other types of targets. The default can be overridden by setting the CMAKE_POSITION_INDEPENDENT_CODE variable, in which case it will be used to initialize the POSITION_INDEPENDENT_CODE target property when the target is created.

```
add_library(MyShared SHARED shared.cpp)
add_library(MyStatic STATIC static.cpp)
target_link_libraries(MyShared PRIVATE MyStatic)

set_target_properties(MyStatic PROPERTIES
    POSITION_INDEPENDENT_CODE ON
)

set(CMAKE_POSITION_INDEPENDENT_CODE ON)
add_library(MyOtherStatic STATIC other.cpp)
target_link_libraries(MyShared PRIVATE MyOtherStatic)
```

21.7. Recommended Practices

Use MODULE libraries for optional plugins to be loaded on demand and SHARED libraries for linking against. Use shared libraries where the symbols to be exposed to consumers of the library must be tightly controlled, either for API purposes or to hide sensitive implementation details. If aiming to deliver a library as part of a release package, shared libraries tend to be preferred over static libraries in most cases.

If a target uses something from a library, it should always link directly to that library. Even if the library is already a link dependency of something else the target links to, do not rely on an indirect link dependency for something a target uses directly. If that other target changes its implementation and it no longer links against the library, the main target will no longer build. Furthermore, express the right type of link dependency; PRIVATE, PUBLIC or INTERFACE. This ensures CMake correctly handles transitive link dependencies for both shared and static libraries. Specifying all the direct dependencies with the correct level of visibility is essential for ensuring CMake constructs a reliable linker command line with correct library ordering.

Using the correct link visibility has the added benefit that consuming targets don't have to know about all the different library dependencies used internally, they only need to link to a library and let that library define its own dependencies. CMake then takes care of ensuring all required libraries are specified in the correct order on the final linker command line. Resist the temptation to simply make all link dependencies PUBLIC, since this extends the visibility of otherwise private libraries into places where it may be undesirable. This becomes particularly important when packaging up a project for release or distribution.

Consider using a library versioning strategy as early as possible. Once a library has been released into the wild, the version number has some very specific meanings with regard to binary compatibility. Make use of the VERSION and SOVERSION target properties to specify the library version, even if initially these are set to some basic placeholders early in the life of the project. In the absence of any other strategy, one reasonable option is to start version numbering at 0.1.0, since people tend to interpret 0.0.0 as a default value or the version mistakenly not having been set, while 1.0.0 is sometimes taken to imply the first public release. Give strong consideration to adopting semantic versioning for handling version changes thereafter. Also keep in mind that

Exclusively for sbmain17@gmail.com Transaction: 0098315591

changes in library versions can have a surprisingly strong influence on things like release processes, packaging, etc. and developers need time to learn the implications of version numbers for shared libraries well in advance of those libraries being released publicly. Consider also whether the project version and library version should have any relationship to each other or not. It can be very difficult to change such a relationship once the first release is made, so be wary of linking them unless they have a strong association (a project delivering a coherent set of libraries as a SDK would be one such example of a strong association).

Some projects can optionally provide certain functionality if a particular supporting toolkit, library, etc. is available. To allow other parts of the build or indeed other consuming projects to detect or check consistency with that optional functionality or feature, interface compatibility details can be provided. Consider whether the feature in question needs to have visibility beyond the library, such as allowing consuming targets to detect whether or not the feature is supported or confirming whether the selected implementation provides all the capabilities required. Also consider whether the added complexity of specifying and using interface compatibilities brings with it sufficient benefits to make it worthwhile, as the deeper the library dependency hierarchy becomes, the harder it can be to use interface compatibilities effectively.

Give consideration to symbol visibility as early in the life of a project as possible, as it can be very difficult to go back and retrofit a project with symbol visibility details later. When creating libraries, develop the mindset of always thinking about whether a particular class, function or variable should be accessible to anything outside of the library. Think of anything that has external visibility as being very hard to change, whereas internal things can be more freely modified between releases as needed. Use hidden visibility as the default and explicitly mark each individual entity to be exported, ideally with macros provided by the `generate_export_header()` function so that CMake handles the various platform differences on the project's behalf. Also consider using the deprecation macros provided by that function to clearly identify those parts of a library's API that have been deprecated and which may be removed in a future version.

Take extra care when mixing shared and static libraries. Where possible, prefer to use one or the other rather than both, as this avoids some of the difficulties associated with build setting consistency and symbol visibility control. Where it makes sense to mix both library types, try to ensure that static libraries only get linked into one shared library and no other targets link to those static libraries. Treat the static libraries as being sub-groups within the shared library, with outside targets only ever linking to the shared library. Even better though, consider pulling the code up from the static libraries into the shared library directly instead, getting rid of the static libraries altogether. The techniques presented in Section 29.5.1, "Target Sources" demonstrate how to add sources to an existing target progressively, allowing the target sources to be conveniently accumulated across subdirectories.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

Chapter 22. Toolchains And Cross Compiling

When considering the process of building software and the tools involved, developers typically think about the compiler and linker. While these are the primary tools that developers are exposed to, there are a number of other tools, libraries and supporting files that also contribute to the process. Loosely speaking, this broader set of tools and other files is collectively referred to as the *toolchain*.

For desktop or traditional server applications, there usually isn't a great need to think too deeply about the toolchain. In most cases, deciding which release of the prevailing platform toolchain to use is about as complicated as it gets. CMake usually finds the toolchain without needing much help and the developer can get on with the task of writing software. For mobile or embedded development, however, the situation is quite different. The toolchain will normally need to be specified in some way by the developer. This can be as simple as specifying a different target system name, or it can be as complex as specifying the paths to individual tools and a target root file system. Special flags may also need to be set to make the tools produce binaries that will support the right chipset, have the required performance characteristics and so on.

Once a toolchain has been selected, CMake performs quite a bit of processing internally to test the toolchain to determine the features it supports, set various properties and variables, etc. This is the case even for a traditional build where the default toolchain is used, not just for builds that are cross-compiling. The results of these tests can be seen in CMake's output the first time it is run for a given build directory. An example for GCC may look something like this:

```
-- The C compiler identification is GNU 9.3.0
-- The CXX compiler identification is GNU 9.3.0
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working C compiler: /usr/bin/cc - skipped
-- Detecting C compile features
-- Detecting C compile features - done
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++ - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
```

The bulk of this processing usually occurs at the point where the first `project()` command is called and the results of the toolchain tests are then cached. The `enable_language()` command also triggers such processing when it enables a previously non-enabled language, as would another `project()` call that adds a previously non-enabled language. Once a language has been enabled, its cached details will always be used rather than re-testing the toolchain, even for subsequent CMake runs. This has at least two important consequences:

- Once a build directory has been configured with a particular toolchain, it cannot (safely) be changed. In certain situations, CMake may detect that the toolchain has been modified and discard its previous results, but this only discards cached details directly related to the toolchain. Any other cached quantities based on the cached toolchain details outside of the ones CMake knows about will not be reset. Therefore, the build directory should be completely

Exclusively for sbmain17@gmail.com Transaction: 0098315591

cleared before changing the toolchain (it may not be enough to just remove the `CMakeCache.txt` file, other details may be cached in different locations).

- Different toolchains cannot be mixed directly within the one project. CMake fundamentally sees a project as using a single toolchain throughout. In order to use multiple toolchains, one has to structure the project to perform parts of the build as external sub-builds (a technique discussed in Section 28.1, "ExternalProject" and Section 29.1, "Superbuild Structure").

22.1. Toolchain Files

If the default toolchain is not suitable, then the recommended way of specifying the desired toolchain details is with a *toolchain file*. This is just an ordinary CMake script which typically contains mostly `set(…)` commands. These would define the variables that CMake uses to describe the target platform, the location of the various toolchain components and so on. The name of the toolchain file is passed to CMake through the special cache variable `CMAKE_TOOLCHAIN_FILE` like so:

```
cmake -DCMAKE_TOOLCHAIN_FILE=myToolchain.cmake path/to/source
```

CMake 3.21 or later also supports a `--toolchain` command-line option or a fallback to a `CMAKE_TOOLCHAIN_FILE` environment variable, but the result is ultimately the same (i.e. the `CMAKE_TOOLCHAIN_FILE` cache variable is set).

```
cmake --toolchain myToolchain.cmake path/to/source
```

```
# Set once for the current shell
export CMAKE_TOOLCHAIN_FILE=myToolchain.cmake

# No toolchain specified, uses the environment variable
cmake path/to/source
```

A full absolute path can be used, or for a relative path like in the above examples, CMake first looks relative to the top of the build directory, then if not found there, relative to the top of the source directory. This toolchain file must be specified the first time CMake is run for the build directory, it cannot be added later or changed to point to a different toolchain. Since the variable itself is cached, there is no need to respecify it again for any subsequent CMake runs.

The toolchain file is read by the first call to the `project()` command, possibly more than once. The number of times it is read in subsequent runs may also be different to the first run. The toolchain file may also be read by temporary sub-projects that CMake sets up internally to test the toolchain when a language is enabled for the first time. Given these factors, a toolchain file should support being included multiple times and it should not contain logic that assumes it is being read only by the main project.

Developers should aim to make toolchain files minimal, setting only the things needed and making as few assumptions about what the project does as possible. Toolchain files should ideally be completely decoupled from the project and should even be reusable across different projects, since they should only be describing the toolchain, not how they interact with a particular project.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

The contents of a toolchain file can vary, but on the whole there are only a few main things they potentially need to do:

- Describe basic details of the target system.
- Provide paths to tools (often just to the compilers).
- Set the default flags for tools (usually just for compilers and perhaps linkers).
- Set the location of a target platform's root file system in the case of cross-compilation.

It is quite common to see other logic included in toolchain files as well, especially for influencing the behavior of the various `find_…()` commands (see Chapter 24, *Finding Things*). While there are situations where such logic may be appropriate, one can mount an argument that such logic can and should be part of the project instead in most cases. Only the project knows what it is trying to find, so the toolchain should not make assumptions about what the project wants to do.

22.2. Defining The Target System

The fundamental variables that describe the target system are:

- `CMAKE_SYSTEM_NAME`
- `CMAKE_SYSTEM_PROCESSOR`
- `CMAKE_SYSTEM_VERSION`

Of these, `CMAKE_SYSTEM_NAME` is the most important. It defines the type of platform being *targeted*, as opposed to `CMAKE_HOST_SYSTEM_NAME` which defines the platform on which the build is being *performed*. CMake itself always sets `CMAKE_HOST_SYSTEM_NAME`, but `CMAKE_SYSTEM_NAME` can be (and often is) set by toolchain files. One can think of `CMAKE_SYSTEM_NAME` as being what `CMAKE_HOST_SYSTEM_NAME` would be set to if CMake was run directly on the target platform. Thus, typical values include `Linux`, `Windows`, `QNX`, `Android` or `Darwin`, but for certain situations (e.g. bare metal embedded devices), a system name of `Generic` may be used instead. There are also variations on the typical platform names which can be appropriate in some situations, such as `WindowsStore` and `WindowsPhone`. If `CMAKE_SYSTEM_NAME` is set in a toolchain file, then CMake will also set the `CMAKE_CROSSCOMPILING` variable to true, even if it has the same value as `CMAKE_HOST_SYSTEM_NAME`. If `CMAKE_SYSTEM_NAME` is not set, it will be given the same value as the auto-detected `CMAKE_HOST_SYSTEM_NAME`.

`CMAKE_SYSTEM_PROCESSOR` is intended to describe the hardware architecture of the target platform. If not specified, it will be given the same value as `CMAKE_HOST_SYSTEM_PROCESSOR`, which is automatically populated by CMake. In cross-compiling scenarios or when building for a 32-bit platform on a 64-bit host of the same system type, this will result in `CMAKE_SYSTEM_PROCESSOR` being incorrect. Therefore, it is advisable to set `CMAKE_SYSTEM_PROCESSOR` if the architecture doesn't match the build host, even if the project seems to build okay without it. Wrong decisions based on an incorrect `CMAKE_SYSTEM_PROCESSOR` value can lead to subtle problems that may not be easy to detect or diagnose.

The `CMAKE_SYSTEM_VERSION` variable has different meanings depending on what `CMAKE_SYSTEM_NAME` is set to. For example, with a system name of `WindowsStore`, `WindowsPhone` or `WindowsCE`, the system version will be used to define which Windows SDK to use. Values might be more general like 8.1 or 10.0, or they might define a very specific release, such as 10.0.10240.0. As another example, if `CMAKE_SYSTEM_NAME` is set to `Android`, then `CMAKE_SYSTEM_VERSION` will typically be interpreted as the

Exclusively for sbmain17@gmail.com Transaction: 0098315591

default Android API version and must be a positive integer. For other system names, it is not unusual to see CMAKE_SYSTEM_VERSION set to something arbitrary like 1, or to not be set at all. The toolchains section of the CMake documentation provides examples of different uses of CMAKE_SYSTEM_VERSION, but the meaning and the set of allowable values for the variable are not always clearly defined. For this reason, projects are advised to exercise caution if implementing logic that depends on the value of CMAKE_SYSTEM_VERSION.

Normally, these three CMAKE_SYSTEM_⋯ variables fully describe the target system, but there are exceptions:

- With CMake 3.13 and earlier, all Apple platforms use Darwin for the CMAKE_SYSTEM_NAME, even for iOS, tvOS or watchOS. The actual target system is then selected by the CMAKE_OSX_SYSROOT variable, which selects the base SDK to be used for the build. The target device is determined based on the SDK chosen, but the developer can still choose between device or simulator at build time. This is a complex topic and is covered in detail in Section 23.5, "Build Settings". Support for the dedicated CMAKE_SYSTEM_NAME values iOS, tvOS and watchOS was added in CMake 3.14 to better distinguish the different platforms and make them more consistent with how other platforms are handled.
- CMAKE_SYSTEM_PROCESSOR and CMAKE_SYSTEM_VERSION are not particularly meaningful for Apple platforms and usually remain unset.
- The CMAKE_SYSTEM_PROCESSOR variable is typically not set when targeting Android platforms. This is discussed further in Section 22.7, "Android" below.

Furthermore, some project generators support their own native platform names. For such generators, instead of setting the CMAKE_SYSTEM_NAME variable, a native platform name can be specified via a few different methods. The simplest and most direct is to specify the native platform along with the generator details on the cmake command line using the -A option. For example, the Visual Studio generator can be instructed to target the x64 platform like so:

```
cmake -G "Visual Studio 2019" -A x64
```

The chosen platform will be available to projects through the CMAKE_GENERATOR_PLATFORM CMake variable. Alternatively, developers may choose to use a toolchain file and to set the CMAKE_GENERATOR_PLATFORM CMake variable directly (projects should never set this CMake variable themselves). If using CMake 3.15 or later, another choice is to provide the platform via the CMAKE_GENERATOR_PLATFORM environment variable instead.

22.3. Tool Selection

Of all the tools used in the build, the compiler is probably the most important from the developer's perspective. The path to the compiler is controlled by the CMAKE_<LANG>_COMPILER variable, which can be set in a toolchain file or on the command line to manually control the compiler used, or it can be omitted to allow CMake to choose one automatically. If the name of an executable is provided manually without a path, CMake will search for it using find_program() (covered in Section 24.3, "Finding Programs"). If a full path to a compiler is provided, it will be used directly. If no compiler is manually specified, CMake will select a compiler based on an internal set of defaults for the target platform and generator.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

From CMake 3.19, CMAKE_<LANG>_COMPILER can be a list. The first item in the list is the compiler to use, just as described above. The remaining list items are compiler options that have to be present for the compiler to work. Do not add non-mandatory options via this variable. Note also that CMAKE_<LANG>_COMPILER should not be changed after the first CMake run.

Most languages also have support for setting the compiler by specifying an environment variable instead of having to set CMAKE_<LANG>_COMPILER. These usually follow common conventions, such as CC for a C compiler, CXX for a C++ compiler, FC for a Fortran compiler and so on. These environment variables will only have an effect the first time CMake is run in a build directory and only if the corresponding CMAKE_<LANG>_COMPILER variable is not set by a toolchain file or on the CMake command line.

Some generators support their own separate toolset specifications which work differently to the above methods. These toolsets can be selected using the -T option on the cmake command line, or if using CMake 3.15 or later, by setting the CMAKE_GENERATOR_TOOLSET environment variable. They can also be selected in a toolchain file by setting the CMAKE_GENERATOR_TOOLSET CMake variable (projects should never set this variable themselves). The available toolsets and the supported syntax are specific to each generator, but the following examples demonstrate some of the possibilities.

Build 32-bit executables but use 64-bit compiler and linker tools:

```
cmake -G "Visual Studio 2019" -A Win32 -T host=x64 ...
```

Use the clang-cl compilers from the LLVM Visual Studio extension:

```
cmake -G "Visual Studio 2019" -T llvm ...
```

For some generators, having multiple instances of the build tool installed may mean the developer needs to specify which instance to use. A typical example is a developer trying out a preview version of Visual Studio and then later installing the release version without deleting the preview version. When using CMake 3.11 or later, the CMAKE_GENERATOR_INSTANCE variable may be set in a toolchain file to control the specific instance that will be used. With CMake 3.15 or later, the CMAKE_GENERATOR_INSTANCE environment variable can be set instead. Few generators support this feature, currently only those for Visual Studio 2017 or later.

With the toolchain specified, CMake will identify the compiler and try to determine its version. This compiler information will be made available through the CMAKE_<LANG>_COMPILER_ID and CMAKE_<LANG>_COMPILER_VERSION variables respectively. The compiler ID is a short string used to differentiate one compiler from another, with common values being GNU, Clang, AppleClang, MSVC and Intel. The CMake documentation for CMAKE_<LANG>_COMPILER_ID gives the full list of supported IDs. If the compiler version can be determined, it will have the usual *major.minor.patch.tweak* form, where not all version components need to be present (e.g. 4.9 would be a valid version).

In addition to the CMAKE_<LANG>_COMPILER_ID and CMAKE_<LANG>_COMPILER_VERSION variables, analogous generator expressions without the leading CMAKE_ part are also supported. Either the variables or the generator expressions can be used to conditionally add content only for certain compilers or compiler versions. For example, GCC 7 introduced a new -fcode-hoisting option and the following shows both ways of adding it for C++ compilation only if it is available:

Exclusively for sbmain17@gmail.com Transaction: 0098315591

```
# Conditionally add -fcode-hoisting option using variables
if(CMAKE_CXX_COMPILER_ID STREQUAL "GNU" AND
   CMAKE_CXX_COMPILER_VERSION VERSION_GREATER_EQUAL 7)
    target_compile_options(SomeTarget PRIVATE -fcode-hoisting)
endif()

# Same thing using generator expressions instead
set(isGNU     $<CXX_COMPILER_ID:GNU>)
set(newEnough $<VERSION_GREATER_EQUAL:$<CXX_COMPILER_VERSION>,7>)
target_compile_options(SomeTarget PRIVATE
    $<$<AND:${isGNU},${newEnough}>:-fcode-hoisting>
)
```

The compiler ID is the most robust way to identify the compiler used. The one case projects may need to be aware of is that prior to CMake 3.0, the Apple Clang compiler was treated the same as the upstream Clang and both had the compiler ID `Clang`. From CMake 3.0 onward, Apple's compiler has the compiler ID `AppleClang` instead so that it can be differentiated from upstream Clang. Policy `CMP0025` was added to allow the old behavior to be used for those projects that require it.

Once the path to the compiler has been determined, CMake is able to work out the appropriate set of default flags for the compiler and linker. These are made available to the project through the variables `CMAKE_<LANG>_FLAGS`, `CMAKE_<LANG>_FLAGS_<CONFIG>`, `CMAKE_<TARGETTYPE>_LINKER_FLAGS` and `CMAKE_<TARGETTYPE>_LINKER_FLAGS_<CONFIG>`, which were covered back in Section 15.4, "Compiler And Linker Variables". Developers can add their own flags into the set of default values for these using variables of the same name but with `_INIT` appended. These `⋯_INIT` variables are only ever used to set the initial defaults, they have no effect once CMake has been run once and the actual values have been saved in the cache.

A common mistake is to set the non-`⋯INIT` variables in a toolchain file (i.e. setting `CMAKE_<LANG>_FLAGS` rather than `CMAKE_<LANG>_FLAGS_INIT`). This has the undesirable effect of discarding or hiding any changes the developer might make to these variables in the cache. The toolchain file may also be re-read as part of later `project()` or `enable_language()` calls, thereby discarding any changes to these variables made by the project itself. Setting the `⋯INIT` variables instead ensures that only the initial default values are affected and any subsequent changes to the non-`⋯_INIT` variables via any method are retained.

As an example, consider a toolchain file a developer might use to set up their build with special compiler flags for debugging (this can be a useful way of re-using some complex developer-only logic across multiple projects without having to add it to each project). The following chooses GNU compilers and adds flags that enable most warnings:

```
set(CMAKE_C_COMPILER   gcc)
set(CMAKE_CXX_COMPILER g++)

set(extraOpts "-Wall -Wextra")
set(CMAKE_C_FLAGS_DEBUG_INIT   ${extraOpts})
set(CMAKE_CXX_FLAGS_DEBUG_INIT ${extraOpts})
```


Exclusively for sbmain17@gmail.com Transaction: 0098315591

Unfortunately, there are some inconsistencies in how CMake combines developer-specified ⋯_INIT options with the defaults it normally provides. In most cases, CMake will append further options to those specified by ⋯INIT variables, but with some platform/compiler combinations (particularly older or less frequently used ones), developer-specified ⋯_INIT values can be discarded. This stems from the history of these variables, which used to be for internal use only and always unilaterally set the ⋯_INIT values. From CMake 3.7, the ⋯_INIT variables were documented for general use and the behavior was switched to appending rather than replacing for the commonly used compilers. The behavior for very old or no longer actively maintained compilers was left unmodified.

Some compilers act more as compiler drivers and expect a command line argument specifying the platform/architecture to compile for (Clang and QNX qcc are two such examples). For compilers that CMake recognizes as requiring such arguments, the CMAKE_<LANG>_COMPILER_TARGET variable can be set in a toolchain file to specify the target. Where supported, this should be used instead of trying to manually add the flags with CMAKE_<LANG>_FLAGS_INIT.

Another less common situation is where the compiler toolchain does not include other supporting utilities like archivers or linkers. These compiler drivers typically support a command line argument that can be used to specify where these tools can be found. CMake provides the CMAKE_<LANG>_COMPILER_EXTERNAL_TOOLCHAIN variable which can be used to specify the directory in which these utilities are located.

22.4. System Roots

In many cases, the toolchain is all that is needed, but sometimes projects may require access to a broader set of libraries, header files, etc. as they would be found on the target platform. A common way of handling this is to provide the build with a cut down version (or even a full version) of the root filesystem for the target platform. This is referred to as a *system root* or just *sysroot* for short. A sysroot is basically just the target platform's root file system mounted or copied to a path that can be accessed through the host's file system. Toolchain packages often provide a minimal sysroot containing various libraries, etc. needed for compiling and linking.

CMake has fairly extensive and easy to use support for sysroots. Toolchain files can set the CMAKE_SYSROOT variable to the sysroot location and with that information alone, CMake can find libraries, headers, etc. preferentially in the sysroot area over same-named files on the host (this is covered in detail in Section 24.1.2, "Cross-compilation Controls"). In many cases, CMake will also automatically add the necessary compiler/linker flags to the underlying tools to make them aware of the sysroot area. For more complex scenarios where different sysroots need to be provided for compiling and linking (e.g. as used by the Android NDK with unified headers), toolchain files can set CMAKE_SYSROOT_COMPILE and CMAKE_SYSROOT_LINK instead when using CMake 3.9 or later.

In some arrangements, developers may choose to mount the full target file system under a host mount point and use that as their sysroot. This could be mounted as read-only, or if not it may still be desirable to leave it unmodified by the build. Therefore, when the project has been built, it may need to be installed to somewhere else rather than writing to the sysroot area. CMake provides the CMAKE_STAGING_PREFIX variable which can be used to set a staging point below which any install commands will install to (see Section 26.1.2, "Base Install Location" for a discussion of this area). This staging area could be a mount point for a running target system and the installed binaries could then be tested immediately after installation. Such an arrangement is particularly useful

Exclusively for sbmain17@gmail.com Transaction: 0098315591

when cross compiling on a fast host for a target system that would otherwise be slow to build on (e.g. building on a desktop machine for a Raspberry Pi target). Section 24.1.2, "Cross-compilation Controls" also discusses how CMAKE_STAGING_PREFIX affects the way CMake searches for libraries, headers and so on.

22.5. Compiler Checks

When a project() or enable_language() call triggers testing of compiler and language features, the try_compile() command is called internally to perform various checks. If a toolchain file has been provided, it is read by each try_compile() invocation, so the test project will be configured in a similar way to the main build. CMake will pass through some relevant variables automatically, such as CMAKE_<LANG>_FLAGS, but toolchain files may want other variables to be passed through to the test build as well. Since the main build will read the toolchain file first, the toolchain file itself can define which variables should be passed through to test builds. This is done by adding the names of the variables to the CMAKE_TRY_COMPILE_PLATFORM_VARIABLES variable (do not set this in the project, only in a toolchain file). Use list(APPEND) rather than set() so that any variables added by CMake are not lost. It won't matter if CMAKE_TRY_COMPILE_PLATFORM_VARIABLES ends up containing duplicates, it only matters that the desired variable names are present.

The try_compile() command normally compiles and links test code to produce an executable. In some cross compiling scenarios, this can present a problem if running the linker requires custom flags or linker scripts, or is otherwise not desirable to invoke (cross compiling for a bare metal target platform may have such a restriction). If using CMake 3.6 or later, the command can be told to create a static library instead by setting CMAKE_TRY_COMPILE_TARGET_TYPE to STATIC_LIBRARY. This avoids the need for the linker, but it still requires an archiving tool. CMAKE_TRY_COMPILE_TARGET_TYPE can also have the value EXECUTABLE, which is the default behavior anyway if no value is set. Prior to CMake 3.6, the now deprecated CMakeForceCompiler module had to be used to prevent try_compile() from being invoked at all, but CMake now relies heavily on these tests to work out what features the compilers support, so the use of CMakeForceCompiler is now actively discouraged.

While it is not invoked during compiler checks, the try_run() command is closely related to try_compile() and its behavior is affected by cross-compilation. try_run() is effectively a try_compile() followed by an attempt to run the executable just built. When CMAKE_CROSSCOMPILING is set to true, CMake modifies its logic for running the test executable. If the CMAKE_CROSSCOMPILING_EMULATOR variable is set, CMake will prepend it to the command that would otherwise have been used to run the executable on the target platform and uses that to run the executable on the host platform. If CMAKE_CROSSCOMPILING_EMULATOR is not set when CMAKE_CROSSCOMPILING is true, CMake requires the toolchain or project to manually set some cache variables. These variables provide the exit code and the output from stdout and stderr that would be obtained had the executable been able to be run on the target platform. Having to provide these manually is clearly inconvenient and error-prone, so projects should generally try hard to avoid calling try_run() in cross-compiling situations where CMAKE_CROSSCOMPILING_EMULATOR cannot be set. For cases where these manually defined variables cannot be avoided, the CMake documentation for the try_run() command provides the necessary details regarding the variables to be set. Further uses of CMAKE_CROSSCOMPILING_EMULATOR are also discussed in Section 25.7, "Cross-compiling And Emulators".

Exclusively for sbmain17@gmail.com Transaction: 0098315591

22.6. Examples

The examples that follow have been selected to highlight the concepts discussed in this chapter. The toolchains section of the CMake reference documentation contains further examples for a variety of different target platforms.

22.6.1. Raspberry Pi

Cross compiling for the Raspberry Pi is a good introduction to the way CMake handles cross compilation in general. The first step is to obtain the compiler toolchain, a common way being to use a utility like crosstool-NG. The rest of this example will use `/path/to/toolchain` to refer to the top of the toolchain directory structure.

A typical toolchain file for the Raspberry Pi might look something like this:

```
set(CMAKE_SYSTEM_NAME      Linux)
set(CMAKE_SYSTEM_PROCESSOR ARM)

set(CMAKE_C_COMPILER   /path/to/toolchain/bin/armv8-rpi3-linux-gnueabihf-gcc)
set(CMAKE_CXX_COMPILER /path/to/toolchain/bin/armv8-rpi3-linux-gnueabihf-g++)

set(CMAKE_SYSROOT      /path/to/toolchain/armv8-rpi3-linux-gnueabihf/sysroot)
```

If the host has a mount point for a running target device, it could be used to make testing the binaries built by the project relatively straightforward. For example, assume `/mnt/rpiStage` is a mount point that attaches to a running Raspberry Pi (this would preferably point to some local directory rather than the system root so that it could be wiped or otherwise modified in arbitrary ways without destabilizing the running system). A toolchain file would specify this mount point as a staging area like so:

```
set(CMAKE_STAGING_PREFIX /mnt/rpiStage)
```

The project's binaries could then be installed to this staging area and run directly on the device (see Section 26.1.2, "Base Install Location").

22.6.2. GCC With 32-bit Target On 64-bit Host

GCC allows 32-bit binaries to be built on 64-bit hosts by adding the `-m32` flag to both the compiler and linker commands. The following toolchain example still allows the GCC compilers to be found on the `PATH`, adding just the extra flag to the initial set used by the compilers and linker. Depending on one's point of view, this arrangement could be seen as cross-compiling or not. Therefore, setting `CMAKE_SYSTEM_NAME` could also be seen as optional, since setting it forces `CMAKE_CROSSCOMPILING` to have the value true. Either way, the `CMAKE_SYSTEM_PROCESSOR` should still be set since the goal of this toolchain file is specifically to target a processor different to that of the host.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

```
set(CMAKE_SYSTEM_NAME      Linux)
set(CMAKE_SYSTEM_PROCESSOR i686)

set(CMAKE_C_COMPILER   gcc)
set(CMAKE_CXX_COMPILER g++)

set(CMAKE_C_FLAGS_INIT   -m32)
set(CMAKE_CXX_FLAGS_INIT -m32)

set(CMAKE_EXE_LINKER_FLAGS_INIT    -m32)
set(CMAKE_SHARED_LINKER_FLAGS_INIT -m32)
set(CMAKE_MODULE_LINKER_FLAGS_INIT -m32)
```

One way to confirm that the build is indeed 32-bit is with the `CMAKE_SIZEOF_VOID_P` variable, which is computed by CMake automatically as part of its toolchain setup. For 64-bit builds, this will have a value of 8, whereas for 32-bit builds, it will be 4.

```
math(EXPR bitness "${CMAKE_SIZEOF_VOID_P} * 8")
message("${bitness}-bit build")
```

22.7. Android

Cross-compiling for Android can be a bit more involved than the cases discussed above. There are a number of Android-specific settings that affect the build, not just the usual target platform, toolchain locations and flags. Some combinations of CMake and Android NDK versions also have compatibility issues, so developers need to select their tools and toolkits carefully.

22.7.1. Historical Context

CMake has its own built-in support for Android, but the Android NDK also has its own expectations around how the build should be set up. Both CMake and the NDK developed their respective support for each other more or less independently and in parallel. This resulted in problems at various points where incompatibilities emerged.

The NDK r19 release introduced changes that broke CMake's built-in Android support. CMake 3.16.0 contained changes which restored the ability to use the built-in Android support again, but some issues remained in certain scenarios. With the NDK r23 release and CMake 3.21 or later, these incompatibilities have finally been resolved. The interested reader is directed to the following for details on the above:

- https://gitlab.kitware.com/cmake/cmake/issues/18787
- https://github.com/android-ndk/ndk/issues/463

Users should be able to reliably use the toolchain file provided by the NDK r23 or later when using CMake 3.21 or later. Developers who must use NDK releases in the range r18—r22 are advised to prefer using CMake 3.20 or later where possible. The 3.20 release contained fixes relevant to these NDK versions.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

The rest of the Android-related material in the sections that follow assume NDK r23 or later and CMake 3.21 or later are being used.

22.7.2. Using The NDK

The recommended way to build for Android is to use the toolchain file provided by the Android NDK. The name of the file is typically something like `android.toolchain.cmake`. Depending on the host platform and how the NDK has been installed, the toolchain file can be found in a variety of locations. A typical arrangement would see the file placed in the `build/cmake` subdirectory below the base install directory of the NDK.

The toolchain file takes care of setting a number of things. `CMAKE_SYSTEM_NAME` will always be set to `Android`. The Clang compilers provided by the NDK will be used (NDK r18 removed the ability to select the `gcc` toolchain). `CMAKE_SYSROOT` will be set to the appropriate directory within the NDK.

The architecture and ABI can be left at defaults chosen by the NDK, but it is recommended that the developer explicitly set them. This ensures it is very clear what is being built for. This can be done by setting the `CMAKE_ANDROID_ARCH_ABI` variable to one of the following values (others may be supported, check the NDK documentation or toolchain file):

- `armeabi-v7a`
- `arm64-v8a`
- `x86`
- `x86_64`

If `armeabi-v7a` is selected, the following two variables are also relevant:

- `CMAKE_ANDROID_ARM_NEON` can be set to true to enable building with NEON support, or false to build without it. If this variable is not set, NEON support will be enabled by default.
- `CMAKE_ANDROID_ARM_MODE` controls the type of processor to build for. Set it to true to build for 32-bit ARM processors, otherwise the build will target 16-bit Thumb processors.

CMake will set `CMAKE_SYSTEM_PROCESSOR` to an appropriate value based on the above and information provided by the NDK.

The Android API level is controlled by a variable named `ANDROID_PLATFORM` (note the lack of any `CMAKE_` prefix). If `ANDROID_PLATFORM` is not set, the API level is set to the minimum level supported by the NDK. Note that this minimum version may be unsuitable for the chosen architecture and ABI, so it is advisable not to rely on it. Instead, the API level should be specified as a number to ensure the API used is well-defined. It is recommended that an API level of 23 or higher be used to avoid potential reliability issues with Android PackageManager's native library loading (see https://github.com/KeepSafe/ReLinker). The special string `latest` can also be used, which selects the latest API level supported by the NDK, but this is less clear and not as traceable.

`CMAKE_ANDROID_STL_TYPE` specifies the C++ STL implementation to be used. Earlier NDK releases supported a range of different options, but projects should now use either `c++_shared` or `c++_static`. Only use the latter if your application consists of a single shared library. If the application does not use the C++ STL at all, the value `none` can be used instead, but this would be unusual.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

`CMAKE_ANDROID_RTTI` and `CMAKE_ANDROID_EXCEPTIONS` control whether to enable rtti and exceptions respectively. They are boolean variables whose defaults are determined by the choice of STL implementation. Note that the NDK toolchain file in r23 contains a bug which means `CMAKE_ANDROID_EXCEPTIONS` might not be set correctly in some circumstances, if not set directly by the developer. It is therefore recommended that this variable always be set to ensure the expected behavior is obtained.

From the above, it can be seen that there are a number of variables that will typically need to be set before the NDK toolchain file is read. One could specify them on the `cmake` command line, like so:

```
cmake -DCMAKE_TOOLCHAIN_FILE=/path/to/ndk/build/cmake/android.toolchain.cmake \
      -DCMAKE_ANDROID_ARCH_ABI=arm64-v8a \
      -DANDROID_PLATFORM=24 \
      -DCMAKE_ANDROID_STL_TYPE=c++_shared \
      -DCMAKE_ANDROID_RTTI=YES \
      -DCMAKE_ANDROID_EXCEPTIONS=YES \
      ...
```

Alternatively, a wrapper toolchain file could be used:

my_android_toolchain.cmake:

```
set(CMAKE_ANDROID_ARCH_ABI arm64-v8a)
set(ANDROID_PLATFORM 24)
set(CMAKE_ANDROID_STL_TYPE c++_shared)
set(CMAKE_ANDROID_RTTI YES)
set(CMAKE_ANDROID_EXCEPTIONS YES)

include(/path/to/ndk/build/cmake/android.toolchain.cmake)
```

```
cmake -DCMAKE_TOOLCHAIN_FILE=my_android_toolchain.cmake ...
```

CMake caches information computed during the first run in a build directory. If any of the variables mentioned in this section need to be changed, the build directory contents should be deleted first. If this is not done, the compiler flags may not be updated to reflect the new settings in the changed variables.

22.7.3. Android Studio

Certain tools may enforce the use of their own internal toolchain file, making it potentially harder for developers to specify any of the above settings. Android Studio is one such example, forcing a particular toolchain file which overrides much of CMake's own logic. The gradle builds are set up to create an external CMake build that uses the Ninja generator and the NDK provided through the Android SDK manager. While direct access to the toolchain file is not enabled, the gradle build does provide a range of gradle variables which are translated into their CMake equivalents. Developers should consult the tool's documentation to understand how different CMake versions can be used and how to influence the behavior of the CMake build.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

22.7.4. ndk-build

For developers using `ndk-build` (which is essentially just a wrapper around GNU `make`) rather than gradle, CMake 3.7 added the ability to export an `Android.mk` file as part of the CMake build using `export()` or as part of the install step with `install()`. Exporting during the build is straightforward:

```
export(TARGETS target1 [target2...] ANDROID_MK fileName)
```

The `fileName` will typically be `Android.mk` with some path prepended to put it at the location required by `ndk-build`. Each of the named targets will be included in the generated file along with the relevant usage requirements such as include flags, compiler defines, etc. This is typically what a project will want to do if it needs to support being part of a parent `ndk-build`. For the case where the CMake project will be packaged up and wants to make itself easy to incorporate into any `ndk-build`, the `install()` command offers the required functionality (see Section 26.3, “Installing Exports”).

22.7.5. Visual Studio Generators

The Ninja and Makefiles generators integrate well with the NDK toolchain file. When using a Visual Studio generator, CMake 3.19 and later supports the NDK as well. Setup should work reasonably smoothly for NDK versions installed directly via the Visual Studio installer, but these tend to be quite old and are therefore of limited usefulness. The Ninja and Makefiles generators are likely to give better results if using more recent NDK versions installed outside of Visual Studio.

For earlier CMake versions, Android support in Visual Studio required the use of the Nvidia Nsight Tegra Visual Studio Edition. This method has been essentially unmaintained for quite a few CMake releases, so it’s reliability may be questionable. Developers are encouraged to use the NDK instead.

22.8. Recommended Practices

Toolchain files can seem a little intimidating at first, but much of this comes from many examples and projects putting too much logic in them. Toolchain files should be as minimal as possible to support the required tools and they should generally be reusable across different projects. Logic specific to a project should be in the project’s own `CMakeLists.txt` files.

When writing toolchain files, developers should ensure that the contents do not assume they will only be executed once. CMake may process the toolchain file multiple times depending on what the project does (e.g. multiple calls to `project()` or `enable_language()`). The toolchain file may also be used for temporary builds "off to the side" as part of `try_compile()` calls, so they should make no assumptions about the context in which they are being used.

Avoid using the deprecated `CMakeForceCompiler` module to set the compiler to be used in the build. This module was popular when using older CMake versions, but newer versions rely heavily on testing the toolchain and working out the features it supports. The `CMakeForceCompiler` module was mainly intended for cases where the compiler was not known to CMake, but use of such compilers with recent CMake versions will likely result in non-trivial limitations. It is recommended to work with the CMake developers to add the required support for such compilers.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

Be careful not to discard or mishandle the contents of variables that may already be set by the time the toolchain file is processed. A common error is to modify variables like `CMAKE_<LANG>_FLAGS` rather than `CMAKE_<LANG>_FLAGS_INIT`, which can discard values manually set by developers or interact poorly with values already populated when the toolchain file is processed multiple times.

When targeting Android platforms, prefer to use the NDK and the toolchain file it provides, or a simple wrapper around that toolchain file. To ensure a fully working, reliable build, use NDK r23 or later and CMake 3.21 or later. Earlier versions had incompatibilities between the two and setup was more involved. Avoid the once popular *taka-no-me* toolchain file frequently referred to by online examples. It is overly complicated, has known issues and has not been maintained for a number of years. Also use Android API 23 or later to avoid known issues with the Android PackageManager's native library loader.

Projects should generally avoid using the `CMAKE_CROSSCOMPILING` variable for any of its logic. This variable can be misleading, since it can be set to true even when the target and host platform are the same, or false when they are different. The inconsistency of its value makes it unreliable. Project authors should also be aware that some multi configuration generators (e.g. Xcode) allow the target platform to be selected at build time, so CMake logic based around whether cross-compiling or not needs to be written very carefully to handle the different situations in which the project may be generated.

Toolchain files often contain commands to modify where CMake searches for programs, libraries and other files. See Chapter 24, *Finding Things* for recommended practices related to this area.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

Chapter 23. Apple Features

Apple platforms have a number of unique characteristics which directly affect the way software is built. While simple command-line applications for macOS can be built in similar ways to other Unix-based platforms, those applications with a graphical user interface are usually provided in an Apple-specific format known as an *application bundle* (or just *app bundle*). These bundles are more than a single executable file, they are a standardized directory structure containing a variety of files associated with the application. These app bundles are intended to be self-contained, able to be moved around as a unit and placed anywhere on a user's file system.

There is an analogous situation for libraries too. Standalone static and shared libraries can be created much like those on other Unix-based platforms, but they can also be built as part of a *framework*, which is essentially the library equivalent of an app bundle. Frameworks have their own standardized directory structure and may contain files other than just the library binaries. They may even support multiple versions within that directory structure. Libraries intended to be loaded at runtime can instead be built as a loadable bundle, which corresponds to Apple's `CFBundle` functionality.

Bundles and frameworks are essential parts of the machinery used to produce content for Apple's app store. Another key aspect is code signing, a process which verifies the integrity and origin of software and is a mandatory part of app store distribution. Code entitlements are also an integral part of the build process and govern which Apple features the code may use. These entitlements are part of the information sealed by the code signing process and must be defined at build time if the default entitlement set (which is empty) is not appropriate.

Together, these features present unique challenges for CMake projects. The sections that follow provide the tools for understanding and handling these areas, or in some cases, highlight the current limitations of CMake's support. It should also be noted that formal support for tvOS and watchOS was only added in CMake 3.14 (macOS and iOS have been formally supported before that).

23.1. CMake Generator Selection

The technologies and tools used to produce frameworks and bundles is constantly evolving, with major Apple OS releases often introducing new features and changing the requirements around signing, distribution, etc. The processes and technologies are tightly integrated into Xcode as the primary tool Apple expects developers to use, with developers also typically expected to upgrade to the current Xcode release rather than staying with past major releases. Areas like resource compilation, code signing, etc. are automatically handled as part of building applications and frameworks, many aspects of which are unique to the Apple ecosystem.

For CMake projects, this means that the Xcode generator is the most reliable and most convenient for building with the Xcode toolchain. Other generators such as Makefiles or Ninja tend to lack some of the automation of the Xcode generator, or they may lag behind implementing support for some of the more recent Xcode features. With the exception of basic unsigned desktop applications not intended for distribution through the app store, developers will be more or less required to use the Xcode generator to get a build that supports the necessary features. Also note that the fast-moving nature of Apple platforms means that developers will also generally want to be using fairly recent CMake and Xcode releases to keep up with the changes. The arrival of Apple Silicon is a good

Exclusively for sbmain17@gmail.com Transaction: 0098315591

example, where CMake 3.19.2 and Xcode 12.2 should be considered the minimum versions when building on or for that architecture. These are also the minimum requirements for CMake to be able to use Xcode's modern build system.

One of the unique benefits of the Xcode generator is that it supports setting arbitrary Xcode project attributes. Most project settings can be modified in a key-value fashion on a per-target basis using target properties of the form `XCODE_ATTRIBUTE_XXX`, where `XXX` is the name of an Xcode property. These names are defined in the Apple documentation, but a potentially more convenient way to find them is to open an existing Xcode project, go to the build settings of a target and select the menu item **Editor > Show Setting Names**. Alternatively, one can click on a build setting of interest and the *Quick Help* assistant editor shows the setting name along with a description. A third method is to obtain a listing of a project's build settings from the command line using `xcodebuild`:

```
xcodebuild -showBuildSettings
```

CMake also supports variables of the form `CMAKE_XCODE_ATTRIBUTE_XXX`, but their relationship to the corresponding target properties is different to the usual CMake arrangement. These variables are used to set or override global defaults in the Xcode project rather than being used to initialize target properties. They only have an effect in the top level `CMakeLists.txt` file, their value is ignored in all other directory scopes. When a `XCODE_ATTRIBUTE_XXX` target property is set, it overrides the global default.

The following example shows some of the more commonly used attributes:

```
# Set the default signing identity and team ID to use for
# all targets, must be in the top level of the project
set(CMAKE_XCODE_ATTRIBUTE_CODE_SIGN_IDENTITY "Apple Development")
set(CMAKE_XCODE_ATTRIBUTE_DEVELOPMENT_TEAM   XYZ123ABCD)

# Target-specific settings, can be in any directory scope
set_target_properties(MyApp PROPERTIES
    XCODE_ATTRIBUTE_TARGETED_DEVICE_FAMILY     1,2
    XCODE_ATTRIBUTE_IPHONEOS_DEPLOYMENT_TARGET 10.0
)
```

This feature can also be used to set an Xcode attribute for only one particular build type by appending `[variant=ConfigName]` to the property name. Other suffix types can be appended to the property name too for even more specific attribute settings, but their use would be unusual. Even `[variant=…]` suffixes would not often be needed. The following example gives an idea of the sort of use cases where this feature might be useful:

```
set_target_properties(MyApp PROPERTIES
    XCODE_ATTRIBUTE_GCC_UNROLL_LOOPS[variant=Release] YES
    XCODE_ATTRIBUTE_ENABLE_TESTABILITY[variant=Debug] YES
)
```

Some projects may require setting quite a few attributes in order to get the desired Xcode behavior and features, whereas other projects may be quite simple and require only a minimal number of

Exclusively for sbmain17@gmail.com Transaction: 0098315591

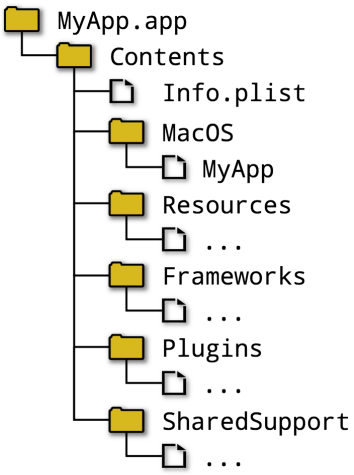
additional settings. Some attributes are only needed in very specific circumstances, whereas others are so common they are (or should be) present in almost every Apple-focused project. A number of these are discussed in the rest of this chapter, including some of those used in the above examples.

23.2. Application Bundles

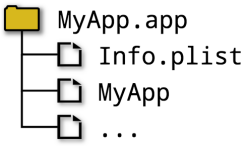
Application bundles are a cornerstone of the Apple ecosystem. Understanding the structure of an application bundle and how this affects the way the build should be configured is essential for any developer targeting Apple platforms.

23.2.1. Bundle Structure

The structure of an application bundle for macOS is different to that for iOS, tvOS and watchOS. The macOS structure separates out various categories of files into different subdirectories and looks something like the following (applications might have only some of the subdirectories shown):



In contrast, the bundle structure for iOS, tvOS and watchOS is flattened, having very little in the way of a defined structure:



When building an app bundle, CMake somewhat abstracts away these structural differences, allowing at least some things to be handled the same way whether the bundle is being built for macOS or for iOS, tvOS or watchOS. Developers should be aware, however, that some areas of that abstraction have not been implemented correctly until more recent CMake releases (the handling of resources being a specific example), so using the latest CMake release is strongly advised.

An application is identified as being a bundle by adding the `MACOSX_BUNDLE` keyword to `add_executable()`:

```
add_executable(MyApp MACOSX_BUNDLE ...)
```

Exclusively for sbmain17@gmail.com Transaction: 0098315591

This sets the MACOSX_BUNDLE target property to true, which non-Apple platforms simply ignore. A project can alternatively set the CMAKE_MACOSX_BUNDLE variable to true and all subsequently defined executable targets have their MACOSX_BUNDLE target property set as well, but it would be more common and clearer to use the MACOSX_BUNDLE keyword with each add_executable() command instead (projects typically define only a small number of application bundles, often only one).

Somewhat confusingly, MACOSX_BUNDLE applies not just to macOS, but also to iOS, tvOS and watchOS. The keyword predates the non-desktop Apple platforms, hence the desktop-specific name. Rather than creating new keywords for each of the other platforms, the use of the existing keyword was expanded to cover all of the Apple platforms. This same pattern of expanding OSX-specific keywords and variables to cover all the Apple platforms can be seen in a number of other cases as well, but note that this is not universal across all OSX-related variables and properties. Those for which this holds true are highlighted in this chapter.

23.2.2. Bundle Info.plist Files

Every application bundle must have at least an Info.plist file and a main executable (MyApp in the directory structure examples above). By default, CMake will provide a basic Info.plist file from a template file shipped with CMake itself. In most cases, however, projects will want to provide their own Info.plist so that they have full control over the app configuration. When the app uses storyboard or interface builder files, providing a custom Info.plist is pretty much required so that the relevant key entries like NSMainStoryboardFile are present. The MACOSX_BUNDLE_INFO_PLIST target property can be set to the name of a file to use as the Info.plist template (for all Apple platforms, not just macOS). The default template file is called MacOSXBundleInfo.plist.in and can be found in CMake's modules directory. It may serve as a useful starting point for custom templates.

Regardless of whether a target uses the default Info.plist template or one provided by the project, CMake will copy the template file into the app bundle, performing some specific substitutions along the way. In the template file, any content of the form ${XXX} will be substituted by the value of the XXX target property if XXX is one of the properties in the table below. Each of these properties is mapped to a particular key in the default Info.plist file, so if the project provides its own template file and uses these variables, it should generally follow the same mapping.

Property	Info.plist Key	Example
MACOSX_BUNDLE_BUNDLE_NAME	CFBundleName	MyApp
MACOSX_BUNDLE_BUNDLE_VERSION	CFBundleVersion	2.4.7
MACOSX_BUNDLE_COPYRIGHT	NSHumanReadableCopyright	© 2020 MyCompany
MACOSX_BUNDLE_GUI_IDENTIFIER	CFBundleIdentifier	com.example.myapp
MACOSX_BUNDLE_SHORT_VERSION_STRING	CFBundleShortVersionString	2.4.7
MACOSX_BUNDLE_LONG_VERSION_STRING	CFBundleLongVersionString	*see below*
MACOSX_BUNDLE_INFO_STRING	CFBundleGetInfoString	*see below*
MACOSX_BUNDLE_ICON_FILE	CFBundleIconFile	*see below*

Exclusively for sbmain17@gmail.com Transaction: 0098315591

Apple no longer documents `CFBundleLongVersionString` as one of the `Info.plist` keys, so projects may choose to not provide it. Their documentation also states that `NSHumanReadableCopyright` has replaced `CFBundleGetInfoString` and that `CFBundleIconFile` is deprecated and recommends using `CFBundleIconFiles` or `CFBundleIcons` instead. `CFBundleIconFile` is still honored if neither of the other alternatives is set.

If multiple app targets are being defined, a project may set variables with exactly the same names as the properties in the above table and the variables will be used to initialize the target properties. Note that this differs from the usual CMake convention of variables having a `CMAKE_⋯` prefix before the target property they act as defaults for.

When a project provides its own `Info.plist` template file, it is not required to make any use of the above target properties. It is perfectly valid to hard-code values instead. Note, however, that `CFBundleVersion` and `CFBundleShortVersionString` may need to be derived from version details specified within the `CMakeLists.txt` files, so setting these via `MACOSX_BUNDLE_BUNDLE_VERSION` and `MACOSX_BUNDLE_SHORT_VERSION_STRING` substitutions may still be the most convenient approach.

The Apple requirements around the version numbers have evolved over time, with *major.minor.patch* now the documented format for both `CFBundleShortVersionString` and `CFBundleVersion`. In the case of `CFBundleVersion`, only the *major* part is required — the *minor* and *patch* version components are optional. One strategy is to set `CFBundleVersion` to the job number from a continuous integration system, which will ensure that subsequent builds for the same `CFBundleShortVersionString` during the pre-release phase will have unique and increasing build numbers. When updating an already installed app with the same `CFBundleShortVersionString`, this will ensure the operating system sees the latest build as a newer version.

The following CMake code and sample `Info.plist` file show one way to provide version numbers that meet Apple's requirements:

```
# CI systems typically provide some form of job ID as an
# environment variable. This example works for gitlab, but
# other CI systems are likely to be similar. When not run
# under a CI system, this will leave BUILD_VERSION unset.
set(BUILD_VERSION $ENV{CI_JOB_ID})

if(BUILD_VERSION STREQUAL "")
    # This is a local build, not through CI system
    set(BUILD_VERSION 0)
endif()

add_executable(MyApp MACOSX_BUNDLE ...)

set_target_properties(MyApp PROPERTIES
    MACOSX_BUNDLE_BUNDLE_VERSION       "${BUILD_VERSION}"
    MACOSX_BUNDLE_SHORT_VERSION_STRING "${PROJECT_VERSION}"
    MACOSX_BUNDLE_INFO_PLIST
        "${CMAKE_CURRENT_SOURCE_DIR}/Info.plist"
)
```

Exclusively for sbmain17@gmail.com Transaction: 0098315591

Info.plist

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
  "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0"><dict>
  <key>CFBundleDevelopmentRegion</key><string>en</string>
  <key>CFBundleExecutable</key>
    <string>$(EXECUTABLE_NAME)</string>
  <key>CFBundleIconFile</key>
    <string>${MACOSX_BUNDLE_ICON_FILE}</string>
  <key>CFBundleIdentifier</key>
    <string>${MACOSX_BUNDLE_GUI_IDENTIFIER}</string>
  <key>CFBundleInfoDictionaryVersion</key>
    <string>6.0</string>
  <key>CFBundleName</key>
    <string>${MACOSX_BUNDLE_BUNDLE_NAME}</string>
  <key>CFBundlePackageType</key><string>APPL</string>
  <key>CFBundleShortVersionString</key>
    <string>${MACOSX_BUNDLE_SHORT_VERSION_STRING}</string>
  <key>CFBundleVersion</key>
    <string>${MACOSX_BUNDLE_BUNDLE_VERSION}</string>
  <key>LSMinimumSystemVersion</key>
    <string>$(MACOSX_DEPLOYMENT_TARGET)</string>
  <key>NSHumanReadableCopyright</key>
    <string>${MACOSX_BUNDLE_COPYRIGHT}</string>
  <key>NSMainStoryboardFile</key><string>Main</string>
  <key>NSPrincipalClass</key><string>NSApplication</string>
</dict></plist>
```

In the example `Info.plist` file above, note how some field values are provided as CMake variables using the `${}` syntax, but the `CFBundleExecutable` and `LSMinimumSystemVersion` are provided using Xcode variable substitution syntax `$()` instead. These two fields will be populated by Xcode itself based on other information provided in the project file and the scheme being built. The value for `LSMinimumSystemVersion` will be derived from the `CMAKE_OSX_DEPLOYMENT_TARGET` variable in the case of a macOS app, or from the `XCODE_ATTRIBUTE_IPHONEOS_DEPLOYMENT_TARGET` target property for iOS (but note that CMake 3.11 and later can use `CMAKE_OSX_DEPLOYMENT_TARGET` for all platforms, see discussion in Section 23.5, "Build Settings" further below). Projects can instead hard-code a value directly in the `Info.plist` file, if that is more convenient.

When using Storyboard or interface builder files, the `Info.plist` file should contain one of the keys `NSMainStoryboardFile`, `NSMainNibFile` or `UIMainStoryboardFile` (see the Apple documentation for details on the meaning and appropriate use of these keys). Such an entry tells the operating system which UI element to use when launching the app. In the above example, the `NSMainStoryboardFile` field has the value `Main`, which specifies that a `Main.storyboard` UI will be used when the app starts. This is discussed further in the next section.

23.2.3. Sources, Resources And Other Files

With an appropriate `Info.plist` file defined, attention can be turned to the source files to be compiled and linked into the bundle. In addition to the usual C/C++ sources, Apple platforms also support Objective C/C++ source files. These typically have a `.m` or `.mm` file suffix and can be listed as

Exclusively for sbmain17@gmail.com Transaction: 0098315591

sources in `add_executable()` and `target_sources()` commands just like ordinary C/C++ files (see Section 29.5, "Defining Targets" for more on defining target sources). Most of CMake's generators will recognize these file suffixes and compile the files appropriately, not just the Xcode generator. Starting from CMake 3.16, `OBJC` and `OBJCXX` are recognized as distinct languages and can be enabled separately from `C` and `CXX`. They also support their own language-specific sets of CMake variables like `CMAKE_OBJC_FLAGS`, `CMAKE_OBJCXX_STANDARD` and so on.

Another group of source files unique to Apple platforms are those used to define the user interface. Storyboard or interface builder files are like sources, but they require some additional handling to compile them as resources and put the compiled result in the appropriate place in the app bundle. Only the Xcode generator implements this automatic compilation and copying to the appropriate location, so the use of Makefile or Ninja generators is generally not recommended when an app bundle has these files.

Storyboard and interface builder sources need to be listed as sources in `add_executable()` or `target_sources()`. To get them to be automatically compiled and copied to the appropriate location in the bundle, they also need to be listed in the `RESOURCE` target property or have their `MACOSX_PACKAGE_LOCATION` source property set to `Resources`. The main difference between these two approaches is what happens if the target is installed (Chapter 26, *Installing* is dedicated to this broad topic). Any file listed in the `RESOURCE` target property will be copied to the `RESOURCE` destination given in the `install(TARGET)` command, regardless of what type of source file it is. This is usually undesirable for source files that need to be compiled, but is suitable for other arbitrary files to be added to the app bundle. On the other hand, source files that have their `MACOSX_PACKAGE_LOCATION` set to `Resources` will not be installed if Xcode recognizes them as source files that should be compiled. All other types of source files will be installed. Section 23.8.3, "Combined Device And Simulator Binaries" discusses a specific scenario where this difference in behavior is particularly relevant.

```
set(compileRes
    Base.lproj/Main.storyboard
    Base.lproj/LaunchScreen.storyboard
    Assets.xcassets     ①
)
set(directCopyRes
    defaultConfig.xml
    inventoryDb.dat
)
add_executable(MyApp MACOSX_BUNDLE
    AppDelegate.m
    AppDelegate.h
    ViewController.m
    ViewController.h
    main.m
    ${compileRes}
    ${directCopyRes}
)
set_target_properties(MyApp PROPERTIES
    RESOURCE "${directCopyRes}"    ②
)
set_source_files_properties(${compileRes} PROPERTIES
    MACOSX_PACKAGE_LOCATION Resources
)
```

Exclusively for sbmain17@gmail.com Transaction: 0098315591

① While CMake doesn't officially support listing directories as sources, it is an important technique for allowing things like asset catalogs to be compiled.

② Note the quotes to ensure the list of files is passed as a single value for the `RESOURCE` property.

A special case applies when setting the `MACOSX_PACKAGE_LOCATION` to a path starting with `Resources` and the target is being built for iOS, tvOS or watchOS. Because app bundles for these platforms use a flattened structure, CMake will strip off the `Resources` part of the path. Prior to CMake 3.9, this behavior was implemented incorrectly and it was not always possible to get a file into the desired location.

For cases where source files need to be copied into the app bundle with some sort of directory structure, or where they need to be located somewhere other than the `Resources` directory, the `MACOSX_PACKAGE_LOCATION` method must be used. For example:

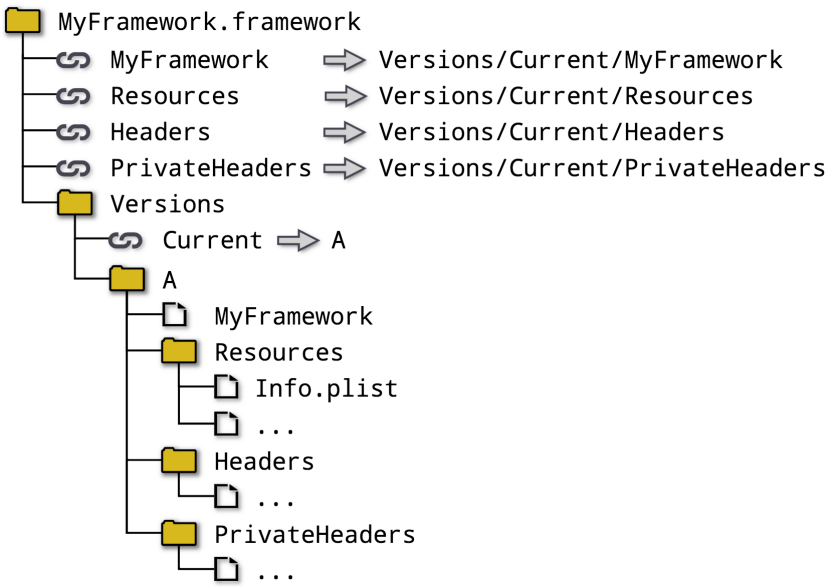
```
set(sharedRes defaultConfig.xml defaultInventoryDb.dat)

add_executable(MyApp MACOSX_BUNDLE ... ${sharedRes})

set_source_files_properties(${sharedRes} PROPERTIES
    MACOSX_PACKAGE_LOCATION SharedSupport/defaults
)
```

23.3. Frameworks

Frameworks share some similarities with application bundles, but they also have a number of unique features. A framework contains a main library, but unlike an application bundle, on macOS there may be multiple versions of the library. In addition to the usual resources, frameworks also support headers and in the case of macOS, both the resources and headers are version-specific.

23.3.1. Framework Structure

A typical example of the macOS framework structure looks something like this.

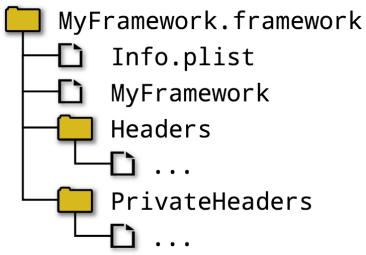


Exclusively for sbmain17@gmail.com Transaction: 0098315591

The top level of the framework always has a name that ends with `.framework` and typically the only non-symlinked contents in that top level directory is the `Versions` subdirectory (umbrella frameworks being the exception, but those are outside the scope of framework support being considered here). Everything else at that level is usually a symlink to something in the current version's subdirectory.

Within the `Versions` directory, each version of the library gets its own subdirectory whose name is the version. In most cases, these directory names are just `A`, `B`, etc. Use of numeric versions is another common convention, which aligns more closely with how shared libraries are versioned on other platforms. Regardless of the style of versioning, a symlink called `Current` points to the most recent version and it acts like a default version for the framework. Each version is expected to have a `Resources` directory that contains at least an `Info.plist` file, which provides configuration details about that particular version (discussed further below). There will also be a library (which is usually a shared library, but it can be static) and often `Headers` and potentially `PrivateHeaders` subdirectories as well.

In comparison, the structure on iOS, tvOS and watchOS is flattened and does not typically support versions:



CMake supports the creation of frameworks (single-version only in the case of macOS) and it provides features for handling the version details. There is also support for `Info.plist` files which follows a similar approach to that used for application bundles. The first step is to define a library in the usual way and then mark it as a framework by setting the `FRAMEWORK` target property to true. With CMake 3.15 or later, the `FRAMEWORK` target property is initialized with the value of the `CMAKE_FRAMEWORK` variable, while for 3.14 and earlier, the target property is initially unset. Most frameworks are defined as shared libraries, but as of CMake 3.8, static libraries can also be built as frameworks. The `FRAMEWORK` target property is ignored on non-Apple platforms.

For macOS only, the framework version can be specified using the `FRAMEWORK_VERSION` target property, or if omitted a default version of `A` will be set. Non-macOS Apple platforms will ignore the `FRAMEWORK_VERSION` property if it is set, producing the same flattened, unversioned framework structure produced by Xcode when it creates frameworks for these platforms.

```
add_library(MyFramework SHARED foo.cpp)

set_target_properties(MyFramework PROPERTIES
    FRAMEWORK          TRUE
    FRAMEWORK_VERSION 5
)
```

Exclusively for sbmain17@gmail.com Transaction: 0098315591

23.3.2. Framework Info.plist Files

The `Info.plist` file template is specified in the same way as for application bundles, except the target property is called `MACOSX_FRAMEWORK_INFO_PLIST` (this is supported for all Apple platforms, not just macOS):

```
set_target_properties(MyFramework PROPERTIES
    MACOSX_FRAMEWORK_INFO_PLIST
        "${CMAKE_CURRENT_SOURCE_DIR}/Info.plist"
)
```

As for application bundles, if a framework `Info.plist` file is not explicitly provided, a default one is automatically generated. Whether the project provides its own `Info.plist` or it relies on the default, CMake will perform a similar substitution as for application bundles when copying it into the framework. The following target properties will be substituted where the `Info.plist` file refers to them (the expected associated key name in the `Info.plist` file is also listed):

Property	Info.plist Key
`MACOSX_FRAMEWORK_BUNDLE_VERSION`	`CFBundleVersion`
`MACOSX_FRAMEWORK_ICON_FILE`	`CFBundleIconFile`
`MACOSX_FRAMEWORK_IDENTIFIER`	`CFBundleIdentifier`
`MACOSX_FRAMEWORK_SHORT_VERSION_STRING`	`CFBundleShortVersionString`

Unlike for application bundles, the default framework `Info.plist` file is likely to be sufficient in many cases, so the project can usually just set the above four target properties and let CMake provide an appropriate `Info.plist` file.

23.3.3. Headers

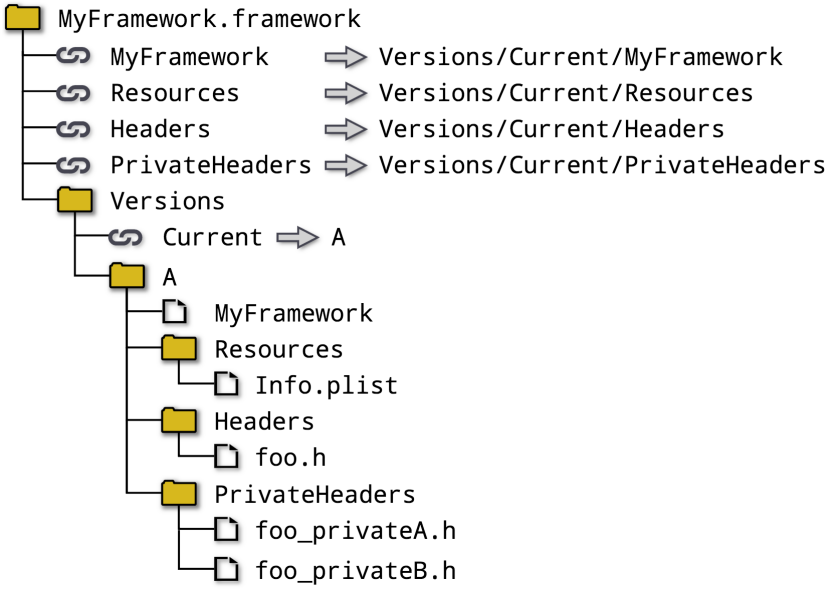
Frameworks often contain the headers associated with the framework's library. This allows the framework to be treated as a self-contained bundle which other software can build against. Framework headers are separated into public and private groups, with only public headers intended to be directly included or imported by consuming projects. The private headers are usually needed as internal implementation details by the public headers and frameworks often do not include any private headers at all.

CMake supports specifying the set of public and private headers with the `PUBLIC_HEADER` and `PRIVATE_HEADER` target properties respectively. Both properties contain a list of header files and all files mentioned must also be explicitly listed as sources for the target or they won't be copied into the framework. Files listed in `PUBLIC_HEADER` will be copied into the framework's `Headers` directory with paths stripped, while files listed in `PRIVATE_HEADER` will be copied into the `PrivateHeaders` directory, again with any paths stripped. If paths need to be preserved, these target properties cannot be used and the headers have to be added using techniques such as via `MACOSX_PACKAGE_LOCATION` as described in Section 23.2.3, "Sources, Resources And Other Files".

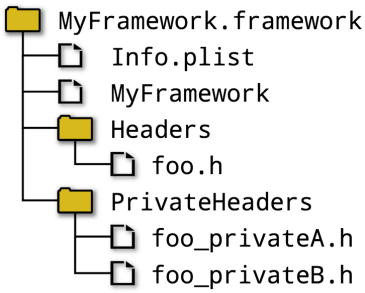
Exclusively for sbmain17@gmail.com Transaction: 0098315591

```
add_library(MyFramework SHARED
    foo.cpp
    foo.h
    foo_privateA.h
    nested/foo_privateB.h
)

set_target_properties(MyFramework PROPERTIES
    FRAMEWORK      TRUE
    PUBLIC_HEADER  foo.h
    PRIVATE_HEADER "foo_privateA.h;nested/foo_privateB.h"
)
```

The above example would result in the following structure on macOS:



The same example on iOS would result in a more flattened structure:



Note that the `PUBLIC_HEADER` and `PRIVATE_HEADER` target properties are also used when installing targets on non-Apple platforms (see Section 26.2.3, “Apple-specific Targets”).

23.4. Loadable Bundles

In addition to application bundles and frameworks, Apple also supports loadable bundles for macOS. These are often used as plugins or to provide optional features which might or might not be supported at run time. The structure of a loadable bundle is the same as that of an application bundle, but the top level directory usually has the extension `.bundle` or `.plugin` (any extension is

Exclusively for sbmain17@gmail.com Transaction: 0098315591

technically permitted). CMake supports the creation of loadable bundles through the `MODULE` library type and the `BUNDLE` target property. By default, loadable bundles will be given the extension `bundle`, but this can be overridden with the `BUNDLE_EXTENSION` target property.

```
add_library(MyBundle MODULE ...)
set_target_properties(MyBundle PROPERTIES
    BUNDLE           TRUE
    BUNDLE_EXTENSION plugin
)
```

All of the target properties relating to application bundles can also be used for loadable bundles.

23.5. Build Settings

When building a project for Apple platforms, a number of properties work together to define what platform to build for and to specify minimum platform version requirements. Unlike other CMake generator types, the Xcode generator allows a number of these to be specified at build time by the developer rather than being known at configure time, a characteristic which can be one of the more difficult aspects to handle correctly for both new and experienced CMake users alike.

For single configuration generators, the target device is known exactly at configure time, but for Xcode, some platforms support both the device and device simulators. Furthermore, some of these devices have multiple architectures. In the case of iOS, this can mean up to five different target platform combinations. In order to allow developers to switch between different device targets and SDKs at build time, CMake projects must be careful to not over-specify or incorrectly specify these details.

23.5.1. SDK Selection

The selection of the SDK for iOS, tvOS and watchOS is one area where many online examples exhibit considerable complexity and often result in locking the developer out of the ability to switch between device and simulator builds without re-running CMake. With recent versions of CMake and Xcode, however, specifying the SDK should be relatively straightforward. This is especially so with improvements added in CMake 3.14, with potentially the only required setup being to set the `CMAKE_SYSTEM_NAME` variable to `iOS`, `tvOS` or `watchOS`. A toolchain file may not even be needed, `CMAKE_SYSTEM_NAME` can be set directly on the `cmake` command line, although some workarounds may still be needed depending on the Xcode version (see Section 23.5.4, “Compiler Test Workarounds”). With CMake 3.13 or earlier, a similar result can be achieved by setting the `CMAKE_OSX_SYSROOT` variable to one of `iphoneos`, `appletvos` or `watchos`, although the support for tvOS and watchOS is less complete and should be considered unreliable.

With either of these methods, Xcode will choose the latest SDK for that platform and it will allow switching between device and simulator builds without having to re-run CMake. Furthermore, Xcode can automatically populate the set of supported architectures based on the chosen SDK, so the project shouldn’t need to specify them directly (see Section 23.8.2, “Intel And Apple Silicon Binaries” for more on this topic). This gives the developer the most control over what they want to build without having to re-run CMake.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

The `-sdk` option can be given to choose the device or simulator at build time. For example, if `CMAKE_SYSTEM_NAME` was set to `iOS`, then building for the simulator could be done like so:

```
xcodebuild -sdk iphonesimulator
```

For the curious, the available SDKs can be obtained by running the following command:

```
xcodebuild -showsdks
```

23.5.2. Deployment Target

For either method of setting `CMAKE_OSX_SYSROOT` or `CMAKE_SYSTEM_NAME` to the desired platform, the project will have its deployment target set to the most recent one the SDK or host system supports by default. This will often be undesirable, since projects typically want to remain compatible with a specific minimum OS version.

For macOS, the `OSX_DEPLOYMENT_TARGET` target property controls the minimum macOS version the target will support. A default value can be specified for this target property using the `CMAKE_OSX_DEPLOYMENT_TARGET` variable, but this must be set before the first `project()` command is called. Furthermore, `CMAKE_OSX_DEPLOYMENT_TARGET` needs to be a cache variable if it is being set directly in the top level `CMakeLists.txt` file, otherwise it will be overwritten by the compiler checks performed by the `project()` command.

An alternative strategy is to use a toolchain file and set `CMAKE_OSX_DEPLOYMENT_TARGET` within it, but the use of toolchain files for macOS builds would be rather uncommon and this variable is something that the project should define, not the developer. One more approach would be to set the `CMAKE_OSX_DEPLOYMENT_TARGET` cache variable on the `cmake` command line, but this also puts the responsibility on the developer to remember to set it and to provide the correct value, making it less attractive.

Prior to CMake 3.11, when targeting platforms other than macOS, the `CMAKE_OSX_DEPLOYMENT_TARGET` variable had no effect. To control the minimum deployment target version for iOS before CMake 3.11, use the `XCODE_ATTRIBUTE_IPHONEOS_DEPLOYMENT_TARGET` target property instead. A default value for this target property can be set using the `CMAKE_XCODE_ATTRIBUTE_IPHONEOS_DEPLOYMENT_TARGET` variable and unlike for macOS, this variable can be set after the first `project()` call. From CMake 3.11 onward, `CMAKE_OSX_DEPLOYMENT_TARGET` can be used to define the minimum deployment target version for any of the Apple platforms, not just macOS. If a target ends up with both `OSX_DEPLOYMENT_TARGET` and `XCODE_ATTRIBUTE_IPHONEOS_DEPLOYMENT_TARGET` target properties defined, the latter takes precedence when using the Xcode generator.

```
cmake_minimum_required(VERSION 3.9)

# Set the deployment target for macOS with any CMake version or all Apple
# platforms when using CMake 3.11 or later. Must be set before first call
# to project() and it must be a cache variable.
set(CMAKE_OSX_DEPLOYMENT_TARGET 10.11 CACHE STRING "")
project(AppleProject)
```

Exclusively for sbmain17@gmail.com Transaction: 0098315591

```
# Set the deployment target for iOS with any CMake version.

# Set defaults for all targets in the project
set(CMAKE_XCODE_ATTRIBUTE_IPHONEOS_DEPLOYMENT_TARGET 9.0)

# Build an app with the deployment target explicitly set
add_executable(MyApp MACOSX_BUNDLE ...)
set_target_properties(MyApp PROPERTIES
    XCODE_ATTRIBUTE_IPHONEOS_DEPLOYMENT_TARGET 10.0
)
```

23.5.3. Device Families

In the case of iOS, projects will also likely want to specify the device families being targeted. Apple denotes devices with integer values specified in the `TARGETED_DEVICE_FAMILY` attribute. For iOS, valid values are 1 for iPhone (and technically iPod touch too) or 2 for iPad. If the app should support both iPhone and iPad, then specify both values separated by a comma. If this attribute is not set, it will default to 1. Xcode will use this value to add a `UIDeviceFamily` entry in the app's `Info.plist` file automatically, so avoid setting this entry in any custom `Info.plist` supplied by the project.

```
# An iOS app that supports only iPad
add_executable(MyiPadApp MACOSX_BUNDLE ...)
set_target_properties(MyiPadApp PROPERTIES
    XCODE_ATTRIBUTE_TARGETED_DEVICE_FAMILY 2
)

# An iOS app that supports both iPhone and iPad
add_executable(RunEverywhereApp MACOSX_BUNDLE ...)
set_target_properties(RunEverywhereApp PROPERTIES
    XCODE_ATTRIBUTE_TARGETED_DEVICE_FAMILY 1,2
)
```

23.5.4. Compiler Test Workarounds

When building for Apple platforms other than macOS, the compiler checks performed by CMake and any `try_compile()` invocations initiated by the project have the potential to trigger a variety of problems. Many of these problems come from code signing being enabled during compiler checks or `try_compile()` invocations, but signing is usually not necessary nor desirable in those contexts. The steps to avoid these problems depend on the combination of the Xcode and CMake versions used, but in almost all cases, the workarounds are most easily implemented with a toolchain file.

Xcode 11 changed the handling of certain signing-related project options, among other things. To be able to build successfully with Xcode 11 for iOS, watchOS or tvOS, the preferred CMake version should be 3.18.2 or later. This requires no workarounds to get a successful build. CMake 3.14 or later can still be used (earlier versions will fail), but the `CMAKE_XCODE_ATTRIBUTE_CODE_SIGNING_ALLOWED` variable needs to be set to false inside `try_compile()` calls to prevent signing. This variable should *not* be set for the main part of the build, since the main build's targets will likely need to be signed. The following toolchain file demonstrates how this can be achieved:

Exclusively for sbmain17@gmail.com Transaction: 0098315591

```
# Any of these platforms can be used with this approach
#set(CMAKE_SYSTEM_NAME watchOS)
#set(CMAKE_SYSTEM_NAME tvOS)
set(CMAKE_SYSTEM_NAME iOS)

# Only disable code signing for try_compile() calls. Leave
# signing details alone for the main part of the build.
get_property(__in_try_compile GLOBAL
    PROPERTY IN_TRY_COMPILE
)
if(__in_try_compile)
    set(CMAKE_XCODE_ATTRIBUTE_CODE_SIGNING_ALLOWED NO)
endif()
unset(__in_try_compile)
```

CMake 3.18.2 and later automatically sets `CMAKE_XCODE_ATTRIBUTE_CODE_SIGNING_ALLOWED` to false inside `try_compile()` calls, so the above is not necessary in that case (but also not harmful). Only the `CMAKE_SYSTEM_NAME` needs to be set for that Xcode/CMake combination, which can be done as a command line option or in a toolchain file.

When using Xcode 10 or earlier with CMake 3.14 or later, the above toolchain file is also not needed, but again it would not harm the build.

When using CMake 3.13 or earlier, the situation is more complex. The following toolchain file works for iOS with Xcode 10, 9 and potentially earlier. It works at least as far back as CMake 3.7, although that should really be considered too old for building iOS apps today:

```
set(CMAKE_OSX_SYSROOT iphoneos)

if(NOT DEFINED CMAKE_XCODE_ATTRIBUTE_CODE_SIGN_IDENTITY)
    set(CMAKE_XCODE_ATTRIBUTE_CODE_SIGN_IDENTITY "")
endif()

set(CMAKE_TRY_COMPILE_TARGET_TYPE STATIC_LIBRARY)
```

The above prevents code signing in code checks and `try_compile()` invocations, but still leaves the code signing identity untouched in the main build (when those details are provided). Setting `CMAKE_TRY_COMPILE_TARGET_TYPE` prevents the checks from trying to build bundles, which would require additional information such as a bundle ID. Other combinations of options can also be used, but the above is the simplest. For tvOS and watchOS, variations on the above may work with CMake versions prior to 3.14, but official support for those platforms is only provided with CMake 3.14 or later.

23.6. Code Signing

The above covers the main build-related settings that need to be defined for most Apple projects. For simple unsigned macOS apps, they may be enough on their own, but most projects will need further configuration to sign the build products before they can be useful.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

Xcode functionality related to code signing has evolved with each successive major Xcode release. The move toward automatic management of code signing and provisioning has made it easier to get signed applications built with CMake, but it still requires an understanding of the signing process to set the appropriate properties and variables. It should be noted that in Xcode 8, the way that signing and provisioning works changed significantly, leaving many examples which demonstrate methods for Xcode 7 and earlier no longer reflecting best practice.

23.6.1. Signing Identity And Development Team

For signing and provisioning to work, the app must have a valid bundle ID and at least two other key pieces of information: the development team ID and the code signing identity. These need to be specified as Xcode attributes, which follow the usual pattern of being set on individual targets through target properties or through CMake variables to specify global defaults. Since both quantities would typically need to be the same throughout the build, it is generally advisable to set them as variables at the top of the project rather than per target.

The `XCODE_ATTRIBUTE_DEVELOPMENT_TEAM` target property or alternatively the corresponding `CMAKE_XCODE_ATTRIBUTE_DEVELOPMENT_TEAM` variable should be set to the development team ID, which is a short string typically of around 10 characters. The most convenient approach is usually to set the `CMAKE_XCODE_ATTRIBUTE_DEVELOPMENT_TEAM` variable very early in the very top `CMakeLists.txt` file, usually just after the first `project()` command. Depending on the project, the developer might or might not need the ability to change this value. For example, if the project is company software that will always be built by an employee, then the team ID will likely never change, whereas an open source project available to the general public will almost certainly be built by developers with their own development team ID. For cases where the team ID should never change, defining `CMAKE_XCODE_ATTRIBUTE_DEVELOPMENT_TEAM` as an ordinary variable is sufficient, but where it is expected that the developer may need to change it, it should be defined as a cache variable so that a default value can be given but developers can override it without editing the `CMakeLists.txt` file.

Similarly, the `XCODE_ATTRIBUTE_CODE_SIGN_IDENTITY` target property or the corresponding `CMAKE_XCODE_ATTRIBUTE_CODE_SIGN_IDENTITY` variable specifies the signing identity. In the past, this should almost always have been the string `Mac Developer` for macOS applications or `iPhone Developer` for iOS, tvOS or watchOS applications. Apple have since changed their certificate handling and starting with Xcode 11, the string `Apple Development` should work for all platforms. These values will direct Xcode to select the most appropriate signing identity for the specified development team. In unusual circumstances, the signing identity may need to be set to a string which identifies a particular code signing identity in the developer's keychain, but the onus is then on the developer to ensure that this identity belongs to the specified development team.

The following example shows how a `CMakeLists.txt` might be structured for a macOS application which allows the developer to change the team ID and the signing identity:

```
cmake_minimum_required(VERSION 3.14)
project(macOSexample)

set(CMAKE_XCODE_ATTRIBUTE_DEVELOPMENT_TEAM   "ABC12345DE"        CACHE STRING "")
set(CMAKE_XCODE_ATTRIBUTE_CODE_SIGN_IDENTITY "Apple Development" CACHE STRING "")
```


Exclusively for sbmain17@gmail.com Transaction: 0098315591

For an iOS application where the team ID is not expected to be changed, but where the developer might still want control over the signing identity (e.g. to test a different identity in their keychain), only the identity would need to be a cache variable:

```
cmake_minimum_required(VERSION 3.14)
project(iOSexample)

set(CMAKE_XCODE_ATTRIBUTE_DEVELOPMENT_TEAM   "ABC12345DE")
set(CMAKE_XCODE_ATTRIBUTE_CODE_SIGN_IDENTITY "Apple Development" CACHE STRING "")
```

One might be tempted to move the code signing details into a toolchain file to avoid having to set `CMAKE_XCODE_ATTRIBUTE_CODE_SIGN_IDENTITY` altogether, but as highlighted in Section 23.5.4, "Compiler Test Workarounds", this can have negative consequences. The try-compile test that CMake performs as part of the first `project()` command would then require a valid provisioning profile, which in turn would require a valid bundle ID. It is not generally going to be desirable to have such bundle IDs and provisioning profiles being created in the team account. The try-compile tests do not need to perform code signing, so a toolchain file should not be used to enable signing globally.

If the app links to shared frameworks that are also built by the project, do not enable code signing for those frameworks. The recommended way to add such frameworks to an app bundle is to embed them with the *Code sign on copy* option enabled. This is discussed in Section 23.10, "Embedding Frameworks".

23.6.2. Provisioning Profiles

When configured as described in the preceding section, Xcode will automatically select an appropriate provisioning profile. If an appropriate profile doesn't exist, the Xcode IDE can automatically create one. The `xcodebuild` command line tool also provides the `-allowProvisioningUpdates` option for Xcode 9 or later. Automatic provisioning can be more convenient compared to the manual process of earlier Xcode versions where provisioning profiles had to be created manually through the online developer portal. For automatic provisioning to work, however, the developer still needs an account on Apple's developer portal. This can be a problem for continuous integration builds or for team environments where individual developers might not have such an account. In those situations, manual provisioning may ultimately still be preferred.

The `XCODE_ATTRIBUTE_PROVISIONING_PROFILE_SPECIFIER` target property allows the provisioning profile to be specified directly. It should be given the provisioning profile's name rather than its UUID because the profile will need to be updated from time to time, such as when a device or developer certificate is added, updated or removed. The profile's name will remain constant, but its UUID will change with each update. The name to use is the one shown for the profile in the developer portal.

23.6.3. Entitlements

Apple applications also have an associated set of entitlements. These control which features the operating system will allow the app to use, such as Siri, push notifications and so on. In the project settings within the Xcode IDE, users are able to go to the *Capabilities* tab of an app target and turn on the capabilities required. The associated entitlements are then enabled in the plist file that

Exclusively for sbmain17@gmail.com Transaction: 0098315591

Xcode automatically generates, the target is linked to any required frameworks and the capability is added to the app ID in the team account. With a CMake-generated project, this *Capabilities* tab is effectively bypassed. Instead, the CMake project is expected to provide its own entitlements plist file if the default entitlements are not sufficient. The project must also handle linking of any required frameworks itself and no changes are made to the app ID. In practice, for many applications these are fairly mild restrictions, with only framework linking presenting some wrinkles.

Specifying entitlements is done by setting the `XCODE_ATTRIBUTE_CODE_SIGN_ENTITLEMENTS` target property to the name of an appropriate entitlements file like so:

```
set_target_properties(MyApp PROPERTIES
    XCODE_ATTRIBUTE_CODE_SIGN_ENTITLEMENTS
        ${CMAKE_CURRENT_LIST_DIR}/MyApp.entitlements
)
```

As an example, an entitlements file which adds Siri to the default entitlements can be quite simple:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
 "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
    <key>com.apple.developer.siri</key>
    <true/>
</dict>
</plist>
```

23.7. Creating And Exporting Archives

In order to distribute an app via the App Store, an Enterprise distribution portal or ad hoc distribution, an archive first needs to be created. While CMake doesn't create a build target for creating such an archive, the `xcodebuild` tool can be used with a project generated by CMake to accomplish the task. The `archive` build action requires only a few options to be able to build the necessary targets for release and create an archive. There are a few ways to specify what to archive, but a fairly simple approach is to name the project, scheme and the name of the output, which can be done like so:

```
xcodebuild archive                      \
           -project MyProject.xcodeproj \
           -scheme MyApp                \
           -archivePath MyApp.xcarchive
```

CMake creates the `.xcodeproj` file when using the Xcode generator. Prior to CMake 3.9, the user then had to load the project in the Xcode IDE to create the build schemes. This presented a problem for headless continuous integration builds where the IDE cannot be accessed, so to address this situation, CMake 3.9 introduced the `CMAKE_XCODE_GENERATE_SCHEME` variable. When this variable is set to true, CMake will also generate schema files for the build, which then allows the name of the app target to be specified for the `-scheme` option and the archive task has all the information it needs.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

The above command will build the `MyApp` target for the Release configuration for all supported architectures, sign it (still with the developer signing identity), and then create an archive named `MyApp.archive` in the current directory.

Archiving may fail if certain install attributes are not set appropriately. The Apple developer documentation contains a few troubleshooting guidelines which may help overcome the more common situations, some of the more relevant ones being to ensure the target's `INSTALL_PATH` and `SKIP_INSTALL` attributes are set correctly for the target type. In a CMake project aimed at producing a signed application for distribution, a target's `XCODE_ATTRIBUTE_SKIP_INSTALL` property must be set to `YES` for libraries and embedded frameworks and to `NO` for applications. Where it is set to `NO`, the `XCODE_ATTRIBUTE_INSTALL_PATH` must also be provided and it should generally be given the value `$(LOCAL_APPS_DIR)`. Failure to follow this advice will typically result in the archiving step producing a generic archive rather than an application archive.

```
# Apps must have install step enabled
set_target_properties(macOSApp PROPERTIES
    XCODE_ATTRIBUTE_SKIP_INSTALL  NO
    XCODE_ATTRIBUTE_INSTALL_PATH  "$(LOCAL_APPS_DIR)"
)
```

After the application archive has been created, it needs to be exported for distribution. This is achieved by running the `xcodebuild` tool again, this time providing the archive just created, an options plist file and the location to write the output to. The basic form of the command is:

```
xcodebuild -exportArchive                          \
           -archivePath MyApp.xcarchive            \
           -exportOptionsPlist exportOptions.plist \
           -exportPath Products
```

The `-archivePath` option points to the archive file created by the earlier invocation of `xcodebuild` and the `-exportPath` option specifies the directory in which to create the final output file. Everything else about the export step is defined by the plist file given to the `-exportOptionsPlist` option. The full set of supported keys can be found in the tool's help documentation (`xcodebuild -help`), but a minimal plist file might look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
 "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
    <key>method</key>
    <string>app-store</string>
</dict>
</plist>
```

The *method* specifies the intended distribution channel and is expected to be one of the following:

Exclusively for sbmain17@gmail.com Transaction: 0098315591

- `app-store`
- `ad-hoc`
- `enterprise`
- `development`
- `developer-id`
- `package`

The default is `development`, but the methods more likely to be of interest are `app-store`, `enterprise` or `ad-hoc`. When exporting an archive, the tool will select an appropriate distribution signing identity for the chosen method and re-sign the app. The developer is expected to have already created or downloaded the appropriate distribution signing identity and provisioning profile (most easily done within the Xcode IDE, but can be done manually for continuous integration servers, etc.).

23.8. Universal Binaries

When building for Apple platforms, it will often be desirable to build for multiple architectures. For a given device or desktop platform, there may be multiple generations of CPU type and it would usually be desirable to provide binaries for each architecture. For example, the set of architectures for iOS typically includes some combination of `armv7`, `armv7s`, `arm64` and `arm64e`, depending on the SDK used and the deployment version being targeted. Another example is for macOS, where it is likely that both Intel (`x86_64`) and Apple Silicon (`arm64`) architectures will need to be supported. The device platforms also typically each come with a set of simulators for development and testing purposes. These simulators run on the host machine's architecture.

There are two main scenarios where it is desirable to build for multiple architectures:

- Producing an archive for upload to the App Store or a release package for direct distribution. An app bundle should preferably contain slices for each of the supported architectures of that platform. For example, an iOS app with a deployment target of iOS 11 or later would ideally have both `arm64` and `arm64e` slices in its binaries.
- Producing a bundle or framework that should contain both device and simulator binaries. Continuing with the iOS theme, for a bundle or framework targeting iOS 11 or later, this would mean including `arm64`, `arm64e` and `x86_64` architectures. In the case of frameworks, this would be more appropriate for passing along to other developers or customers for use in their own project's build.

These scenarios present different problems for CMake builds and require different solutions.

23.8.1. Device-only Bundles

For an app bundle without any embedded frameworks, producing a universal binary doesn't require any additional changes to the project's `CMakeLists.txt` files or specifying any new cache variables. It can be achieved just by indicating which architectures to build for on the build command line and overriding the `ONLY_ACTIVE_ARCH` setting. The final app bundle with the combined binaries will be signed using the same settings as if doing a single architecture build. In the following two examples, both commands are equivalent:

Exclusively for sbmain17@gmail.com Transaction: 0098315591

```
# Building via CMake
cmake --build . --config Release -- \
    ONLY_ACTIVE_ARCH=NO             \
    -arch arm64                     \
    -arch arm64e
```

```
# Invoking xcodebuild directly
xcodebuild -configuration Release \
    ONLY_ACTIVE_ARCH=NO           \
    -arch arm64                   \
    -arch arm64e
```

When building via `cmake`, all options after the `--` are passed directly to the underlying build tool (`xcodebuild` when using the Xcode generator). If building from within the Xcode IDE, the `ONLY_ACTIVE_ARCH` build setting for the app bundle's target can be set to `NO` and the build is then performed in the usual way.

When producing an archive, `cmake --build` cannot be used and the `xcodebuild` tool must be invoked directly. The same architecture information needs to be given when producing an archive as well. The following extends the example from Section 23.7, "Creating And Exporting Archives" to build an archive as a universal binary:

```
xcodebuild archive                      \
           -project MyProject.xcodeproj \
           -scheme MyApp                \
           -archivePath MyApp.xcarchive \
           ONLY_ACTIVE_ARCH=NO          \
           -arch arm64                  \
           -arch arm64e
```

The archive can then be exported and uploaded in the same way as for a single architecture app.

It is very common to see online tutorials and examples recommend setting the architectures in the project's `CMakeLists.txt` files or as cache variables on the `cmake` command line. They set the `CMAKE_OSX_ARCHITECTURES` variable and sometimes also set `CMAKE_XCODE_ATTRIBUTE_ONLY_ACTIVE_ARCH` to false. These have the effect of changing the default behavior of the build to always produce universal binaries with the specified architectures. For example:

```
cmake -G  Xcode                                        \
      -D  CMAKE_TOOLCHAIN_FILE=../toolchain.cmake      \
      -D "CMAKE_OSX_ARCHITECTURES:STRING=arm64;arm64e"  \
      -D  CMAKE_XCODE_ATTRIBUTE_ONLY_ACTIVE_ARCH:BOOL=NO \
      path/to/source
```

Whether this is desirable will depend on the situation. For day-to-day development, it is likely that the developer will only need to build one architecture. In that case, it would be a nuisance if the project hard-coded the architectures and forced building universal binaries, since it would build more than the developer needs each time. Instead, the project should omit those details and let the

Exclusively for sbmain17@gmail.com Transaction: 0098315591

developer decide if they want that behavior by default or not. The above example shows how the developer or an automated script can set the defaults using CMake cache variables alone. One could make the argument that the project could specify the set of architectures, but still leave the developer to choose whether to set `CMAKE_XCODE_ATTRIBUTE_ONLY_ACTIVE_ARCH` or not. If the project does this, it should not force the value of `CMAKE_OSX_ARCHITECTURES`, but rather set it as a cache variable which can still be overridden if the developer or a script driving the build needs to.

Developers should be aware of a side effect of setting `CMAKE_OSX_ARCHITECTURES`. With one exception (see next section), the compiler checks will test the architectures listed in that variable instead of just the default architecture. This not only increases the configure time, it can break the configure step if the deployment version used by those compiler checks is incompatible with any of the architectures requested. If not specified globally through a cache variable set before the first `project()` command, the latest version supported by the SDK will be used for the deployment version. In the case of iOS, the deployment version is required to be 10 or less for the `armv7` architecture, which means using any reasonably current Xcode version will break by default without explicitly setting the deployment version. Adding something like `-DCMAKE_OSX_DEPLOYMENT_TARGET=10` to the `cmake` command line may be needed to overcome this situation.

23.8.2. Intel And Apple Silicon Binaries

With a recent CMake and Xcode, producing macOS universal binaries with slices for Intel and Apple Silicon is very similar to the device-only case. In fact, it can be handled the same way, by setting `CMAKE_OSX_ARCHITECTURES` to the desired architectures. Instead of listing the individual architectures though, it may be more convenient to use the special Xcode variable `$(ARCHS_STANDARD)`. This will expand to the full set of architectures supported by that particular version of Xcode for the selected SDK (macOS by default). When using at least CMake 3.19.2 and Xcode 12.2, the following is sufficient to produce a universal binary containing Intel and Apple Silicon slices:

```
cmake -G Xcode -D "CMAKE_OSX_ARCHITECTURES:STRING=$(ARCHS_STANDARD)" path/to/source
```

CMake 3.19 also ensures that the initial compiler checks only test the host architecture when targeting the macOS platform with the Xcode generator. This is in contrast to targeting one of the device platforms, where all architectures specified by `CMAKE_OSX_ARCHITECTURES` would be tested.

If `CMAKE_OSX_ARCHITECTURES` is not set, CMake will only build for one architecture by default. The choice may be influenced by the host machine's architecture, by whether CMake is running under Rosetta and by whether CMake itself has been built as a universal binary. With CMake 3.19.2 or later, the official CMake packages contain universal binaries. The default architecture is then chosen by which slice is being executed. On an Intel host, this will be the Intel slice. On an Apple Silicon host, this will normally be the Apple Silicon slice, but if CMake is being forced to run under Rosetta, it will select the Intel architecture instead. If required, the user can override the default choice by setting a CMake cache variable named `CMAKE_APPLE_SILICON_PROCESSOR` to either `x86_64` or `arm64`. Alternatively, a `CMAKE_APPLE_SILICON_PROCESSOR` environment variable can be used in the same way instead. Avoid CMake versions before 3.19.2 if building on an Apple Silicon host.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

23.8.3. Combined Device And Simulator Binaries

The methods of the previous sections are suitable when only a single SDK is needed for the build. When building for both a device and its simulators, there are two SDKs involved and producing universal binaries requires more steps. One can either do two separate builds and then use the `lipo` command to manually stitch the resultant binaries together, or the project can be configured to do the building and stitching automatically as part of an install. The project needs to provide the required `install()` commands for this to work, which is covered in Chapter 26, *Installing*. Particularly relevant material can be found in Section 26.2.3, "Apple-specific Targets".

To direct CMake to build and install both device and simulator builds, the `CMAKE_IOS_INSTALL_COMBINED` variable needs to be set to true. This can also be set on a per-target basis using the `IOS_INSTALL_COMBINED` target property, which is initialized by the value of the `CMAKE_IOS_INSTALL_COMBINED` variable when the target is defined, but setting the variable would typically be more convenient. Using this feature, universal binaries combining device and simulator slices could be created with commands like the following:

```
cmake -G  Xcode                                        \
      -D  CMAKE_TOOLCHAIN_FILE=someToolchain.cmake     \
      -D "CMAKE_OSX_ARCHITECTURES=arm64;arm64e;x86_64" \
      -D  CMAKE_XCODE_ATTRIBUTE_ONLY_ACTIVE_ARCH=NO    \
      -D  CMAKE_IOS_INSTALL_COMBINED=YES               \
      -D  CMAKE_INSTALL_PREFIX=path/to/staging/area    \
      path/to/source

cmake --build . --config Release --target install
```

If building on an Apple Silicon host with Xcode 12 or later, it may also be necessary to prevent the `arm64` architecture from being added for simulator builds. This can avoid the situation where `lipo` tries to add `arm64` slices for both the device and simulator. One way to achieve this is by adding an exclusion, which is a new feature for Xcode 12. The following example shows one method for targeting iOS using the variant form of Xcode project settings:

```
set(CMAKE_XCODE_ATTRIBUTE_EXCLUDED_ARCHS[sdk=iphonesimulator*] arm64)
```

23.9. Linking Frameworks

App bundles often link against frameworks to re-use functionality. These frameworks could be provided by the SDK, by external providers or they could be built by the project. Frameworks provided by external parties are the most straightforward to incorporate into the project. They are linked using `target_link_libraries()` just like any other library. They will first need to be located, which is usually done by a call to `find_package()` or `find_library()` (see Chapter 24, *Finding Things*).

Frameworks provided by the Xcode SDK are also fairly straightforward to handle. Xcode automatically adds the relevant search path for the SDK's frameworks, so they can be linked using the form `-framework FrameworkName` and don't need to be found explicitly. To prevent CMake from erroneously merging these options, quoting must be used to keep the option together with the name of the framework to link.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

```
target_link_libraries(MyApp PRIVATE
    "-framework ARKit"
    "-framework HomeKit"
)
```

It gets more complicated when linking a framework built by the project. When creating an archive, Xcode uses completely different paths to build the project compared to an ordinary build. The frameworks built during the archive process will not be at the locations CMake expects them to be. When using CMake 3.19 or later, the linker command lines are constructed in such a way that the full path to the framework isn't used directly. This allows the usual `target_link_libraries()` command to be used and linking will succeed despite the different paths. For CMake 3.18 and earlier, the wrong path breaks archive builds, so linking needs to be done without referring to the CMake target directly. The `-framework` linker option can once again be used for this, following it with the output name of the framework. This output name is usually the same as the target name, but can be changed with the `OUTPUT_NAME` target property. Some kind of dependency also needs to be added to ensure the framework is built before the executable is linked. The following example assumes the default output name for simplicity:

```
# Workaround for CMake 3.18 and earlier
target_link_libraries(MyApp PRIVATE "-framework MyFwk")
add_dependencies(MyApp MyFwk)
```

A drawback to losing the direct linking to the CMake target is that usage requirements will not be propagated from the framework target to the executable target. None of the framework's `PUBLIC` or `INTERFACE` properties will be applied to the executable target. If these are relevant, the project will need to find another way to achieve this. One method would be to create a separate interface library that both the framework and the executable targets link to. That is fairly fragile and inconvenient, so prefer to use CMake 3.19 or later where possible.

The technique outlined above can be extended to cases where a framework links against another framework:

```
add_executable(MyApp ...)
add_library(MyFwk ...)
add_library(OtherFwk ...)

target_link_libraries(MyFwk PRIVATE "-framework OtherFwk")
target_link_libraries(MyApp PRIVATE "-framework MyFwk")
add_dependencies(MyFwk OtherFwk)
add_dependencies(MyApp MyFwk)
```

The same caveat applies regarding loss of usage requirements propagating from `OtherFwk` to `MyFwk` using this approach.

CMake 3.19 introduced support for using Xcode's *Link Binary With Libraries* build phase. A new target property `XCODE_LINK_BUILD_PHASE_MODE` can be used to control the way CMake links libraries and frameworks into some target types. The default value of this property results in the same behavior as earlier CMake versions, except it no longer embeds paths in linker flags in a way that

Exclusively for sbmain17@gmail.com Transaction: 0098315591

would break an `xcodebuild` archive operation. This is why the above fragile workaround can be avoided with CMake 3.19 or later.

A more familiar Xcode experience can be obtained by setting a target's `XCODE_LINK_BUILD_PHASE_MODE` property to `BUILT_ONLY`. Any other CMake targets that are built by the project and linked into the target will then be linked via the *Link Binary With Libraries* build phase instead of using raw linker flags. The property can alternatively be set to `KNOWN_LOCATION`, in which case any item CMake knows the path to will be linked via this build phase, not just CMake targets built by the project. Examples of items that would only be linked via the build phase using `KNOWN_LOCATION` but not `BUILT_ONLY` include imported targets and externally provided libraries specified as full paths.

When building frameworks for any of the Apple device platforms, be aware that a long-standing bug in CMake versions before 3.20.1 caused mishandling of a framework's install name. The full absolute path to the framework would be embedded in the framework's binary, but that path wouldn't exist on the device. Any app using that framework would therefore be unable to run. Use CMake 3.20.1 or later to avoid this problem. For further details, see the discussions in Section 26.2.3, "Apple-specific Targets".

23.10. Embedding Frameworks

For shared frameworks, there are further steps to consider beyond linking. The project will need to embed the shared framework into the app bundle so that it is available at run time. CMake 3.20 added direct support for this with the Xcode generator. Projects can specify a list of frameworks to embed with the `XCODE_EMBED_FRAMEWORKS` target property. These can be CMake targets or paths to frameworks (they can also be paths to bare libraries, but that would be unusual and not consistent with normal practice of using frameworks). Xcode will copy the frameworks to a standard location within the bundle by default. If required, the project can override this location with the `XCODE_EMBED_FRAMEWORKS_PATH` property, but that should not normally be necessary.

When embedding a framework, the project needs to decide whether or not it wants the framework headers to be included. If the target being embedded into is an application bundle, then the headers would likely be unnecessary and should be excluded. On the other hand, if the target being embedded into is itself a framework, then it may be reasonable to include the headers. The project can specify the behavior it wants with the boolean `XCODE_EMBED_FRAMEWORKS_REMOVE_HEADERS_ON_COPY` property on the target being embedded into. It will apply to all frameworks being embedded into that target.

Similarly, the project will likely want to indicate whether to code-sign the frameworks as part of embedding them. This can be done with the boolean `XCODE_EMBED_FRAMEWORKS_CODE_SIGN_ON_COPY` property on the target being embedded into. If enabling this feature, ensure that the frameworks are not signed when they are built or else Xcode will complain about trying to sign an already-signed framework.

The following example demonstrates how to apply the above for an application with a single embedded framework:

Exclusively for sbmain17@gmail.com Transaction: 0098315591

```
set(CMAKE_BUILD_WITH_INSTALL_RPATH)

add_library(MyFwk SHARED ...)
add_executable(MyApp MACOSX_BUNDLE ...)
target_link_libraries(MyApp PRIVATE MyFwk)

set_target_properties(MyFwk PROPERTIES
    FRAMEWORK                           TRUE
    XCODE_ATTRIBUTE_CODE_SIGNING_ALLOWED FALSE
    ...
)

set_target_properties(MyApp PROPERTIES
    # CMake 3.20 or later required
    XCODE_EMBED_FRAMEWORKS                       MyFwk
    XCODE_EMBED_FRAMEWORKS_CODE_SIGN_ON_COPY     TRUE
    XCODE_EMBED_FRAMEWORKS_REMOVE_HEADERS_ON_COPY TRUE
    ...
)
```

Further tweaks to the above example are needed to complete the full picture. Target properties like `XCODE_ATTRIBUTE_SKIP_INSTALL` and `XCODE_ATTRIBUTE_INSTALL_PATH` need to be set appropriately, as discussed in Section 23.7, "Creating And Exporting Archives". The binaries also need to be able to find each other at run time, which requires embedding further details in the binaries themselves. That topic is addressed in Section 26.2.2, "RPATH" and Section 26.2.3, "Apple-specific Targets".

On a side note, CMake 3.21 extended the set of `XCODE_EMBED_`⋯ properties to support embedding app extensions as well. Properties starting with `XCODE_EMBED_APP_EXTENSIONS` work in a similar way to the `XCODE_EMBED_FRAMEWORKS` properties discussed above, apart from using a different default for the ⋯`_REMOVE_HEADERS_ON_COPY` properties. See the official CMake documentation for specific details.

Embedding frameworks with CMake 3.19 and earlier is much less straightforward. More manual (and more fragile) steps are needed to achieve a similar result. When using the Xcode generator, a workaround is available by also listing the shared frameworks as sources for the app bundle's executable target. They need to be specified in an unusual way to support all use cases reasonably robustly.

Ordinarily, embedding a source is a simple matter of listing its path in a call to `add_executable()` or `target_sources()`. For externally provided frameworks, this is the case because the location of that framework is typically fixed. The `MACOSX_PACKAGE_LOCATION` source file property can then be used to copy the framework into the app bundle at build time, just like any other resource or file. The following shows how that might be done for a macOS platform where frameworks are usually collected under a `Frameworks` directory within the app bundle:

```
add_executable(MyApp MACOSX_BUNDLE ... ${pathToFramework})
target_link_libraries(MyApp PRIVATE ${pathToFramework})

set_source_files_properties(${pathToFramework} PROPERTIES
    MACOSX_PACKAGE_LOCATION Frameworks
)
```


Exclusively for sbmain17@gmail.com Transaction: 0098315591

When the shared framework is built by the project, the situation is more complicated because framework locations for an archive operation are different to those of a regular build. Variables populated by Xcode have to be used instead of mentioning the target directly. The first step is being able to reliably locate where the shared framework will be built, whether for a regular build or an archive action. The `CONFIGURATION_BUILD_DIR` Xcode project variable points to the directory in which the framework will end up, so its final location can be captured like so:

```
add_library(MyFwk SHARED ...)

set(fwkLocation "/\$(CONFIGURATION_BUILD_DIR)/MyFwk.framework")
```

The reason for the leading forward slash (/) before `$(CONFIGURATION_BUILD_DIR)` is discussed further below.

The next step is to add the framework as a source to the executable target and set up an appropriate dependency on the original framework target. The following defines a custom command that CMake will treat as creating the framework, but it is the `DEPENDS` part that actually enforces the relationship:

```
add_custom_command(OUTPUT ${fwkLocation}
    DEPENDS MyFwk
    VERBATIM
)
add_executable(MyApp MACOSX_BUNDLE ... ${fwkLocation})
```

It should be noted that the above will only work when the framework and the executable targets are defined in the same directory scope. This is a limitation of how `add_custom_command(OUTPUT)` interacts with targets that use the outputs of that command.

Now that the framework is seen as a source for the executable, the same method can be used to place it in the bundle as for any other file. The following will embed the framework in the standard location for a macOS app bundle:

```
set_source_files_properties(${fwkLocation} PROPERTIES
    MACOSX_PACKAGE_LOCATION Frameworks
)
```

In some places above, CMake attempts to work out whether the source file has been given using an absolute or relative path. Its logic assumes a relative path unless the path starts with a forward slash, or in some cases a generator expression. The `$(CONFIGURATION_BUILD_DIR)` variable provided by Xcode already supplies the full path where needed, so in order to prevent CMake from prepending anything, a leading forward slash is included before `$(CONFIGURATION_BUILD_DIR)`. This will produce two forward slashes at the start of the path used during processing, but all tools involved handle that without error.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

For externally provided frameworks or frameworks built by the project, code signing on copy can be achieved by adding an Xcode file attribute for the framework as a source file property:

```
set_source_files_properties(${fwkLocation} PROPERTIES
    XCODE_FILE_ATTRIBUTES "CodeSignOnCopy"
)
```

23.11. Limitations

CMake's handling of entitlements is fairly rudimentary. It falls short of the automation that the Xcode IDE provides in the *Capabilities* target properties tab, where turning on a particular capability also takes care of adding any required frameworks and automatically updates app ID details as needed. CMake's support still allows all entitlements to be specified, but the process is entirely manual. The project is responsible for defining the entitlements in raw plist format and it must also manually link in any required frameworks itself. Nonetheless, the handling of entitlements is at least possible without the workarounds or steps becoming too burdensome. Any frameworks required by the entitlements are system-provided, so they do not need to be embedded with the application.

On a more practical, day-to-day level, a word of caution is in order regarding a CMake behavior that isn't always obvious. With the Xcode generator, when CMake writes the Xcode project, it creates a utility target called `ZERO_CHECK`. Most other targets in the project depend on `ZERO_CHECK` and its sole purpose is to work out if CMake needs to be re-run before doing the rest of the build. Unfortunately, if CMake *is* re-run by `ZERO_CHECK`, the rest of that build still uses the old project details, which can result in subtle errors due to targets being built with out-of-date settings. Rebuilding a second time should always ensure any such incorrectly built targets are rebuilt properly, but it can be easy to miss. Developers may want to explicitly build the `ZERO_CHECK` target or re-run CMake first after modifying `CMakeLists.txt` files or anything else that would cause CMake to be re-run automatically, or simply build twice.

A more subtle problem related to `ZERO_CHECK` exists if the project contains multiple calls to the `project()` command. Targets defined below the second or later `project()` calls may not have their dependency on `ZERO_CHECK` set up correctly. The `CMAKE_XCODE_GENERATE_TOP_LEVEL_PROJECT_ONLY` variable can be set to true to prevent this problem, which will also have the useful side effect of speeding up the CMake stage, but support for this variable was only added in CMake 3.11.

23.12. Recommended Practices

CMake is able to handle projects targeting Apple platforms, but the limitations need to be considered carefully. If an app bundle needs to embed shared frameworks, the framework embedding features available for the Xcode generator since CMake 3.20 are by far the simplest solution (use CMake 3.20.1 or later to avoid an RPATH-related bug if targeting iOS, tvOS or watchOS). Using older CMake versions will require more work to set up and the steps involved are not intuitive. If shared frameworks are not needed, then CMake's support from 3.14 onward is generally enough to automate the process without too much effort, again as long as the Xcode generator is used. For Xcode 12 or later though, CMake 3.19.2 should be considered the bare minimum. Other generators such as Makefiles or Ninja are fine for building an unsigned macOS

Exclusively for sbmain17@gmail.com Transaction: 0098315591

application, but for other platforms or for signed applications, these generators typically lack some of the features needed to easily produce the final package for distribution. Except for unsigned macOS application development, use of the Xcode generator for Apple development is strongly advised.

Much of the information available in online tutorials and examples is relatively out of date when it comes to using CMake for Apple platforms. In particular, it is very common to see fairly complex toolchain files for iOS, but much of the logic contained in such toolchain files is either questionable or should be moved to the project itself. Developers are strongly encouraged to set their minimum CMake version no lower than 3.14 and take advantage of the improved support for Apple platforms. This makes toolchain files vastly simpler or even unnecessary — just setting `CMAKE_SYSTEM_NAME` to `iOS`, `tvOS` or `watchOS` would often be sufficient. Logic related to Xcode project settings, device- or platform-specific configuration, etc. should go in the project itself.

One of the things that tutorials and examples often do is specify the target architecture by setting the `CMAKE_OSX_ARCHITECTURES` variable. When using the Xcode generator with projects targeting iOS, watchOS or tvOS, this can prevent the developer from being able to switch freely between device and simulator builds. The target architecture is selectable at build time when working in the Xcode IDE or when building at the command line. Projects should generally avoid setting `CMAKE_OSX_ARCHITECTURES` to a specific list of architectures. Instead, let Xcode supply the standard set of architectures based on the selected SDK. When using the Xcode generator, this can be achieved by setting `CMAKE_OSX_ARCHITECTURES` to `$(ARCHS_STANDARD)`. If a project wants to specify the architectures for the convenience of its developers, it should do so as a cache variable and not force overwriting an existing value. This will ensure that developers still have control.

SDK selection is ultimately determined by the value of `CMAKE_OSX_SYSROOT`. When using CMake 3.14 or later, prefer to select the platform via `CMAKE_SYSTEM_NAME`, as this will take care of setting `CMAKE_OSX_SYSROOT` to a suitable default value for the corresponding device SDK. If using CMake 3.13 or earlier cannot be avoided, the developer has to specify the SDK in `CMAKE_OSX_SYSROOT`, which should typically be the generic device string `iphoneos` for iOS builds. Xcode recognizes a matching simulator when a device SDK is chosen, so the device and simulator builds will both be available to the developer.

While it is possible to specify the SDK version as part of the value given to `CMAKE_OSX_SYSROOT`, there is usually little reason to do so. It is much more likely that the deployment target should be set via `MACOSX_DEPLOYMENT_TARGET` or `XCODE_ATTRIBUTE_IPHONEOS_DEPLOYMENT_TARGET` than the SDK version be set. It is the deployment target that ultimately determines whether the application will be able to run on the target and this is independent of the SDK used to build it (assuming the SDK supports that deployment target, of course). Since the latest available version of the SDK will be used by default, there is little to be gained by requiring the build to use a specific SDK version and it can even be harmful. When a particular SDK version is specified, not all developer machines may have it available, since it will depend on which Xcode version is being used. Some developers carry over older SDKs to newer Xcode versions to try to work around this, but that should not be necessary and is actively discouraged.

Some examples also set `CMAKE_XCODE_ATTRIBUTE_ONLY_ACTIVE_ARCH` to true so as to only build the currently selected architecture in the Xcode IDE. Again, this is a decision that should typically be left up to the developer at build time rather than being enforced by the project. The default value of `CMAKE_XCODE_ATTRIBUTE_ONLY_ACTIVE_ARCH` depends on whether or not `CMAKE_OSX_ARCHITECTURES` has been

Exclusively for sbmain17@gmail.com Transaction: 0098315591

explicitly set, but in general the developer can/should set this according to their own needs. They can do so within the project in the Xcode IDE, or they can override the project setting when building on the command line with the `-arch` option to `xcodebuild`. The `-arch` option can be specified multiple times to build more than one architecture and have `xcodebuild` generate universal binaries for those architectures automatically.

One situation where it may make sense to restrict the build to just one architecture is where the project contains targets that link to libraries or frameworks that do not provide universal binaries (i.e. they were only built for a single target platform). In this case, since those libraries or frameworks only support a single platform, the project can only be built for that platform. Similarly, when using `find_library()` or `find_package()` (covered in Chapter 24, *Finding Things*), these commands inherently assume they are building for a single platform, so they do not attempt to define the things they find in a way that supports switching between multiple target platforms.

Some projects may choose to use CMake's install functionality rather than assume Xcode does everything needed for a distributable bundle at build time. For such cases, the `IOS_INSTALL_COMBINED` target property can be set to true to build both device and simulator versions of the target and to combine them into a single universal binary during the install step. The `CMAKE_IOS_INSTALL_COMBINED` cache variable is a more convenient way to enable this behavior project-wide.

The build output from Xcode can be quite verbose, so developers may choose to use a tool like `xcpretty` to hide much of the detail (this is more common for scripted builds to reduce log sizes). Unfortunately, this particular tool will typically hide the output of any of CMake's custom post-build steps, even if those custom steps cause a build error. When such custom steps fail, it can therefore be very difficult to work out the cause of the failure, so it is advisable to either avoid the use of this tool or at least make it easy to switch it off in scripts to help diagnose build problems. The `-quiet` option to the `xcodebuild` command may be an alternative to reduce log output without hiding warnings or errors, but it may also hide too much detail.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

Part III: The Bigger Picture

For the lucky few, a project may be independent of anything else and only needs to satisfy mild quality constraints or perhaps none at all for throw-away experiments. The more likely scenario is that, at some point, the project needs to move beyond its own isolated existence and interact with external entities. This occurs in two directions:

Dependencies

The project may depend on other externally provided files, libraries, executables, packages and so on.

Consumers

Other projects may wish to consume the project in a variety of ways. Some may want to incorporate it at the source level, others may expect a pre-built binary package to be available. Another possibility is the assumption that the project is installed somewhere on the system.

Making a project available either as a standalone package or for consumption by other projects also brings an expectation of a certain level of quality. Automated testing is usually a critical part of any robust software delivery strategy, which means it must be easy to define and execute tests and also to report on the results.

The CMake suite of tools provides assistance with all of the above. It provides commands that operate at a lower level for finding individual files, libraries, etc. and it also provides modules that build on these commands to give a higher level entry point for dependency management. The CTest framework provides a rich set of automated testing capabilities, while CPack considerably eases the process of creating packages in various formats. This part of the book covers these externally focused topics, showing how to get the most out of what CMake offers while also highlighting common mistakes and pitfalls.

Towards the end of this part of the book, two chapters bring the reader full circle back to thinking about how to organize a project. Doing this well requires an appreciation for both the build level features and how a project will interact with other projects. With the benefit of the knowledge gained from the earlier chapters of the book, these two chapters show how to structure and define a project to be flexible, robust and easier for developers to work with. A variety of techniques are presented which can significantly improve the overall build performance.

The final chapter of the book covers the special topic of using CMake with Qt projects. It draws on many of the techniques and features introduced throughout the book and covers Qt-specific features provided by both CMake and Qt.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

Chapter 24. Finding Things

A project of at least modest size will likely rely on things provided by something outside the project itself. For example, it may expect a particular library or tool to be available, or it may need to know the location of a specific configuration or header file for a library it uses. At a higher level, the project may want to find a complete package that potentially defines a range of things including targets, functions, variables and anything else a regular CMake project might define.

To assist with this, CMake provides a variety of features which allow projects to find various things and to make themselves easy to find and be incorporated into other projects. Various `find_…()` commands provide the ability to search for specific files, libraries or programs, or indeed for an entire package. CMake modules also add the ability to use `pkg-config` to provide information about external packages, while other modules facilitate writing package files for other projects to consume. This chapter covers CMake's support for searching for something already available on the file system. The ability to download missing dependencies is covered in Chapter 28, *External Content* and preparing a project for being found by other projects is addressed in Section 26.8, "Writing A Config Package File".

The basic idea of searching for something is relatively straightforward, but as will become apparent, the details of how the search is conducted can be quite involved. In many cases, the default behaviors are appropriate, but an understanding of the search locations and their ordering can allow projects to tailor the search to account for non-standard behaviors and unusual circumstances.

24.1. Finding Files And Paths

Conceptually, the most basic search task is to find a specific file and the most direct way to achieve this is with the `find_file()` command. It also serves as a good introduction to the whole family of `find_…()` commands, since they all share many of the same options and have similar behavior. The full syntax of this command is as follows:

```
find_file(outVar
    name | NAMES name1 [name2...]
    [HINTS path1 [path2...] [ENV var]...]
    [PATHS path1 [path2...] [ENV var]...]
    [PATH_SUFFIXES suffix1 [suffix2 ...]]
    [NO_DEFAULT_PATH]
    [NO_PACKAGE_ROOT_PATH]
    [NO_CMAKE_PATH]
    [NO_CMAKE_ENVIRONMENT_PATH]
    [NO_SYSTEM_ENVIRONMENT_PATH]
    [NO_CMAKE_SYSTEM_PATH]
    [CMAKE_FIND_ROOT_PATH_BOTH |
     ONLY_CMAKE_FIND_ROOT_PATH |
     NO_CMAKE_FIND_ROOT_PATH]
    [DOC "description"]
    [REQUIRED]  # CMake 3.18 or later
    [NO_CACHE]  # CMake 3.21 or later
)
```


Exclusively for sbmain17@gmail.com Transaction: 0098315591

The command can search for a single file name or it can be given a list of names with the NAMES option. A list can be useful when the file being searched for may have a few variations on its name, such as different operating system distributions choosing different naming conventions, incorporating version numbers or not, accounting for a file changing names from one release to another and so on. The names should be listed in preferred order, since the search will stop at the first one found (the complete set of search locations is checked for a particular name before moving on to the next name). When specifying names that contain some form of version numbering, the CMake documentation recommends listing the name(s) without version details ahead of those that do so that locally built files are more likely to be found ahead of files provided by the operating system.

The search will be conducted over a set of locations checked according to a well-defined order. Most locations have an associated option which will cause that location to be skipped if the option is present, thereby allowing the search to be tailored as needed. The following table summarizes the search order:

Location	Skip Option
Package root variables	NO_PACKAGE_ROOT_PATH
Cache variables (CMake-specific)	NO_CMAKE_PATH
Environment variables (CMake-specific)	NO_CMAKE_ENVIRONMENT_PATH
Paths specified via the HINTS option	
Environment variables (system-specific)	NO_SYSTEM_ENVIRONMENT_PATH
Cache variables (platform-specific)	NO_CMAKE_SYSTEM_PATH
Paths specified via the PATHS option	

Package root variables

The first location searched only applies when find_file() is invoked from a script invoked as part of a find_package() call (discussed later in this chapter). It was initially added as a search location in CMake 3.9.0, but was removed in 3.9.1 due to backward compatibility issues. It was then re-added again in CMake 3.12 with the problems addressed. Further discussion of this search location is deferred to Section 24.5, “Finding Packages” where its use is more relevant.

Cache variables (CMake-specific)

The CMake-specific cache variable locations are derived from the cache variables CMAKE_PREFIX_PATH, CMAKE_INCLUDE_PATH and CMAKE_FRAMEWORK_PATH. Of these, CMAKE_PREFIX_PATH is perhaps the most convenient, as setting it works not just for find_file(), but also for all the other find_…() commands. It represents a base point below which a typical directory structure of bin, lib, include and so on is expected and each find_…() command appends its own subdirectory to construct search paths. In the case of find_file(), for each entry in CMAKE_PREFIX_PATH, the directory <prefix>/include will be searched. If the CMAKE_LIBRARY_ARCHITECTURE variable is set, then the architecture-specific directory <prefix>/include/${CMAKE_LIBRARY_ARCHITECTURE} will be searched first to ensure architecture-specific locations take precedence over generic locations. The CMAKE_LIBRARY_ARCHITECTURE variable is normally set automatically by CMake and projects should not generally try to set it themselves.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

For the cases where a more specific include or framework path needs to be searched and it is not part of a standard directory layout or package, the `CMAKE_INCLUDE_PATH` and `CMAKE_FRAMEWORK_PATH` variables can be used. They each provide a list of directories to be searched, but unlike `CMAKE_PREFIX_PATH`, no `include` subdirectory is appended. `CMAKE_INCLUDE_PATH` is supported by `find_file()` and `find_path()`, whereas `CMAKE_FRAMEWORK_PATH` is supported by those two commands and by `find_library()`. Other than that, these two sets of paths are handled in the same way. See Section 24.1.1, "Apple-specific Behavior" further below for additional details.

When using CMake 3.16 or later, the `CMAKE_FIND_USE_CMAKE_PATH` variable can be used to control the default behavior of whether to consider CMake-specific cache variables in the search. If the variable is not defined or is set to true, the search will consider the cache variables. If set to false, the search will ignore them.

Environment variables (CMake-specific)

The CMake-specific environment variable locations are very similar to the cache variable locations. The three environment variables `CMAKE_PREFIX_PATH`, `CMAKE_INCLUDE_PATH` and `CMAKE_FRAMEWORK_PATH` are treated in the same way as the same-named cache variables, except that on Unix platforms, each list item will be separated by a colon (:) instead of a semi-colon (;). This is to allow the environment variables to use platform-specific path lists defined in the same style as other path lists for each platform.

When using CMake 3.16 or later, the `CMAKE_FIND_USE_CMAKE_ENVIRONMENT_PATH` variable can be used to control the default behavior in a similar way as for `CMAKE_FIND_USE_CMAKE_PATH`.

Environment variables (system-specific)

The system-specific environment variables are `INCLUDE` and `PATH`. Both may contain a list separated by the platform-specific path separator (colon on Unix, semi-colon on Windows), with each item being added to the set of search locations (`INCLUDE` is added before `PATH`).

On Windows only (including Cygwin), the `PATH` entries will be further processed in a more complex manner. For each item in the `PATH` environment variable, a base path will be computed by dropping any trailing `bin` or `sbin` subdirectory from the end. If `CMAKE_LIBRARY_ARCHITECTURE` is defined, `<base>/include/${CMAKE_LIBRARY_ARCHITECTURE}` is added. After that, the `<base>/include` path is added to the set of search paths regardless of whether `CMAKE_LIBRARY_ARCHITECTURE` is defined. In the search path ordering, these paths are placed immediately before the unmodified `PATH` item itself. For example, if `CMAKE_LIBRARY_ARCHITECTURE` was set to `somearch` and the `PATH` environment variable contained `C:\foo\bin;D:\bar`, the following ordered set of search paths would be added:

- `C:\foo\include\somearch`
- `C:\foo\include`
- `C:\foo\bin`
- `D:\bar\include\somearch`
- `D:\bar\include`
- `D:\bar`

When using CMake 3.16 or later, the `CMAKE_FIND_USE_SYSTEM_ENVIRONMENT_PATH` variable can be used to control the default behavior in a similar way as for `CMAKE_FIND_USE_CMAKE_PATH`.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

Cache variables (platform-specific)

The platform-specific cache variable locations are very similar to those used for the CMake-specific ones. The names change slightly but the pattern is the same. The variable names are `CMAKE_SYSTEM_PREFIX_PATH`, `CMAKE_SYSTEM_INCLUDE_PATH` and `CMAKE_SYSTEM_FRAMEWORK_PATH`. These platform-specific variables are not intended to be set by the project or the developer. Rather, they are set automatically by CMake as part of setting up the platform toolchain so that they reflect locations specific to the platform and compilers being used. The exception to this is where a developer provides their own toolchain file, in which case it may be appropriate to set these variables within the toolchain file.

When using CMake 3.16 or later, the `CMAKE_FIND_USE_CMAKE_SYSTEM_PATH` variable can be used to control the default behavior in a similar way as for `CMAKE_FIND_USE_CMAKE_PATH`.

`HINTS` and `PATHS`

Each of the various groups of variables discussed above is intended to be set by something outside of the project, but the `HINTS` and `PATHS` options are where the project itself should inject additional search paths. The main difference between `HINTS` and `PATHS` is that `PATHS` are generally fixed locations that never change and don't depend on anything else, whereas `HINTS` are usually computed from other values, such as the location of something already found previously or a path dependent on a variable or property value. `PATHS` are the last directories searched, but `HINTS` are searched before any platform- or system-specific locations.

Both `HINTS` and `PATHS` support specifying environment variables which may contain a list of paths in the host's native format (i.e. colon-separated for Unix systems, semi-colon separated on Windows). This is done by preceding the name of the environment variable with `ENV`, such as `PATHS ENV FooDirs`.

All but the `HINTS` and `PATHS` search locations have an associated skip option of the form `NO_⋯_PATH` which can be used to skip just that set of locations. In addition, the `NO_DEFAULT_PATH` option can be used to bypass all but the `HINTS` and `PATHS` locations, forcing the command to search just specific places controlled by the project. These `NO_⋯` options override any default provided by `CMAKE_FIND_USE_⋯` variables.

The `PATH_SUFFIXES` option can be used to provide a list of additional subdirectories to check below each search location. Each search location is used with each suffix in turn, then without any suffix at all before moving on to the next search location. Use this option with care, as it greatly expands the total number of locations to be searched.

In many cases, projects only need to specify a single file name to search for and the complexities of the search order are not of particular interest. Perhaps just a few additional paths to search might need to be provided (equivalent to the `PATHS` option). In such cases, a shorter form of the command can be used:

```
find_file(outVar name [path1 [path2...]])
```

Whether the short or long form is used, the ordering of the search locations is designed to search in more specific locations ahead of more generic ones. While this is typically the desired behavior, there may be situations where this is not the case.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

For example, a project may wish to always look in specific paths first ahead of any search locations provided through cache or environment variables. Projects can enforce a different priority by calling `find_file()` multiple times with different options controlling the search locations. Once the file is found, the location is cached and all subsequent calls will skip their search. This is where the various `NO_⋯_PATH` options are most useful. For example, the following enforces searching in the location `/opt/foo/include` first and only if not found there will the full set of default locations be searched:

```
find_file(FOO_HEADER foo.h
    PATHS /opt/foo/include
    NO_DEFAULT_PATH
)
find_file(FOO_HEADER foo.h)
```

An important requirement for this to work is that the same result variable must be used for each call. It is that variable that gets set and that controls skipping subsequent calls once the file has been found. The `DOC` option can be used to specify documentation for the cache variable storing the result, but projects frequently omit it. Choosing a variable name that is self-documenting should make the need for explicit documentation unnecessary. By convention, these variables are usually all uppercase and use underscores to separate words.

When using CMake 3.20 or earlier, certain corner cases related to the result variable may yield surprising behavior. Care must be exercised when a non-cache variable with the same name as the result variable already exists before `find_file()` is called. The non-cache variable might be ignored, depending on whether a cache variable also exists and whether such a cache variable has a type (see Section 5.3, “Cache Variables”). CMake 3.21 added policy `CMP0125`, which ensures that a non-cache variable will be handled more like how one would intuitively expect. The `NO_CACHE` option was also added in CMake 3.21 as a way to assign the result only to a non-cache variable, but projects should generally avoid using it. `NO_CACHE` adversely affects the performance of the configure stage by forcing `find_file()` to always repeat the search on every run.

If the file could not be found, the value stored in the variable will evaluate to false if used in an `if()` expression. The test only needs to be made after the final call to `find_file()`.

```
find_file(FOO_HEADER foo.h
    PATHS /opt/foo/include
    NO_DEFAULT_PATH
)
find_file(FOO_HEADER foo.h)

if(NOT FOO_HEADER)
    message(FATAL_ERROR "Could not find foo.h")
endif()
```

The above form works for any CMake version, but from CMake 3.18, the `REQUIRED` option can be used to express the logic more concisely:

Exclusively for sbmain17@gmail.com Transaction: 0098315591

```
find_file(FOO_HEADER foo.h
    PATHS /opt/foo/include
    NO_DEFAULT_PATH
)
find_file(FOO_HEADER foo.h REQUIRED)
```

24.1.1. Apple-specific Behavior

Although the `find_file()` command can be used to find any file, it has its origins in searching for header files. This is why some of the default search paths have an `include` subdirectory appended. On Apple platforms, frameworks sometimes contain their own header files (see Section 23.3, "Frameworks") and the `find_file()` command has additional behaviors related to searching in the appropriate subdirectories within them. For each search location, the command may treat the location as a framework, as an ordinary directory or both. The behavior is controlled by the `CMAKE_FIND_FRAMEWORK` variable, which is expected to hold one of the following values:

- `FIRST`
- `LAST`
- `ONLY`
- `NEVER`

`FIRST` means to treat the search location as though it was the top directory of a framework and to append the appropriate subdirectories to descend into the `Headers` location within it. If the named file cannot be found there, then the search location is treated as an ordinary directory rather than a framework and searched again. `LAST` reverses that order, `ONLY` will not treat the location as an ordinary directory and `NEVER` will skip the step that treats the location as a framework. The default for Apple systems is `FIRST`, which is usually the desired behavior.

24.1.2. Cross-compilation Controls

For cross-compiling scenarios, the set of search locations becomes considerably more complex. Cross compiling toolchains are often collected under their own directory structure to keep them separate from the default host toolchain. When conducting searches for a particular file, it is generally desirable to first look in the toolchain's directory structure ahead of those of the host so that a target platform-specific version of the file will be found. This is especially important when finding programs and libraries. Even for finding files, it may be the case that the content of files could change between platforms (e.g. a platform-specific configuration header).

To support cross-compilation scenarios, the entire set of search locations can be re-rooted to a different part of the file system. The `CMAKE_FIND_ROOT_PATH` variable can be set to a list of additional directories at which to re-root the set of search locations (i.e. prepend each item in the list to every search location). The `CMAKE_SYSROOT` variable can also affect the search root in a similar way. This variable is intended to specify a single directory acting as the system root for a cross-compiling scenario and it should only be set in a toolchain file, never by a project itself. It affects flags used during compilation as well. From CMake 3.9, the more specialized variables `CMAKE_SYSROOT_COMPILE` and `CMAKE_SYSROOT_LINK` also have a similar effect. If any of the non-rooted locations are already under one of the locations specified by `CMAKE_FIND_ROOT_PATH`, `CMAKE_SYSROOT`, `CMAKE_SYSROOT_COMPILE` or

Exclusively for sbmain17@gmail.com Transaction: 0098315591

CMAKE_SYSROOT_LINK, it will not be re-rooted. A non-rooted path that sits under a path specified by the variable CMAKE_STAGING_PREFIX will also not be re-rooted. Furthermore, an undocumented behavior of all find_…() commands is to not re-root any non-rooted path that starts with a ~ character (this is intended to avoid re-rooting directories that sit under the user's home directory).

The default order of searching among the re-rooted and non-rooted locations is controlled by the CMAKE_FIND_ROOT_PATH_MODE_INCLUDE variable. That can also be overridden on a per-call basis by providing one of the CMAKE_FIND_ROOT_PATH_BOTH, ONLY_CMAKE_FIND_ROOT_PATH or NO_CMAKE_FIND_ROOT_PATH options to the find_file() command. The following table summarizes the effects of this mode variable, the associated options and the final search order:

Find Mode	find_file() Option	Search order
BOTH	CMAKE_FIND_ROOT_PATH_BOTH	• CMAKE_FIND_ROOT_PATH • CMAKE_SYSROOT_COMPILE • CMAKE_SYSROOT_LINK • CMAKE_SYSROOT • All non-rooted locations
NEVER	NO_CMAKE_FIND_ROOT_PATH	• All non-rooted locations
ONLY	ONLY_CMAKE_FIND_ROOT_PATH	• CMAKE_FIND_ROOT_PATH • CMAKE_SYSROOT_COMPILE • CMAKE_SYSROOT_LINK • CMAKE_SYSROOT • Any non-rooted locations already under one of the re-rooted locations or under CMAKE_STAGING_PREFIX

It may be desirable to force find_file() to ignore particular paths that may contain a matching file known to be unsuitable. In a cross-compiling scenario, ignoring some specific host paths may be needed to ensure target-specific rather than host-specific files are found. Projects may set the CMAKE_IGNORE_PATH variable to a list of directories to exclude from the search. These paths are not recursive, so they cannot be used to exclude a whole section of a directory structure, they need to specify each directory explicitly. The CMAKE_SYSTEM_IGNORE_PATH variable does the same thing, but it is intended to be populated by the toolchain setup. Both of these …IGNORE_PATH variables apply regardless of whether cross-compiling or not, but it would be unusual for them to be set when not cross-compiling.

Developers should also be aware that find_file() can only provide one location, but some cross compiling situations support build arrangements that can switch between device and simulator builds without re-running CMake. This means that if the results of find_file() depend on which of the two is being used, they are unreliable. This aspect is even more important for finding libraries and is discussed in more detail in Section 24.4, "Finding Libraries" further below.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

24.2. Finding Paths

A project may wish to find the directory containing a particular file rather than the actual file itself. The `find_path()` command provides this functionality and is identical to `find_file()` in every way except that the directory of the file to be found is stored in the result variable.

24.3. Finding Programs

Finding programs is only slightly different to finding files. The `find_program()` command takes exactly the same set of arguments as `find_file()`, as well as one more optional argument, `NAMES_PER_DIR`. The `find_program()` command also supports a similar short form. The following describes the differences for `find_program()` compared to `find_file()`, and while it may seem complicated, for the most part it just describes the differences one might logically expect but with a few exceptions highlighted:

Cache variables (CMake-specific)

- When searching under `CMAKE_PREFIX_PATH`, `find_file()` appends `include` to each item. `find_program()` instead appends `bin` and `sbin` as search locations to be checked. The `CMAKE_LIBRARY_ARCHITECTURE` variable has no effect for `find_program()`.
- `CMAKE_PROGRAM_PATH` replaces `CMAKE_INCLUDE_PATH` but is otherwise used in exactly the same way. `CMAKE_PROGRAM_PATH` is used only by `find_program()`.
- `CMAKE_APPBUNDLE_PATH` replaces `CMAKE_FRAMEWORK_PATH` but is otherwise used in exactly the same way. It is used only by `find_program()` and `find_package()`.

Environment variables (system-specific)

- The search locations for standard system environment variables are handled in a considerably simpler manner. `INCLUDE` has no meaning for `find_program()` and each item in the `PATH` is checked without any modification. The behavior is the same on all platforms.

General

- Normally, all search locations are checked for a given name before moving on to search for the next name in the list when the `NAMES` option is used to provide multiple names. The `find_program()` command supports a `NAMES_PER_DIR` option which reverses this order, checking each name for a particular search location before moving on to the next location. The `NAMES_PER_DIR` option was added in CMake 3.4.
- On Windows (including Cygwin and MinGW), file extensions `.com` and `.exe` are automatically checked as well, so there is no need to provide such extensions as part of the program name to find. These extensions are checked first before names without the extensions. Note that `.bat` and `.cmd` files will not be searched for automatically.
- Whereas `find_file()` uses `CMAKE_FIND_FRAMEWORK` to determine the search order between framework and non-framework paths, `find_program()` uses `CMAKE_FIND_APPBUNDLE`. It provides similar control between app bundle and non-bundle paths for Apple platforms. The supported values are the same for both variables and they have the expected equivalent meaning for bundles. Whereas finding files will look in a `Headers` subdirectory, finding programs will look in the `Contents/MacOS` subdirectory and set the result to the executable within the app bundle.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

- `CMAKE_FIND_ROOT_PATH_MODE_INCLUDE` has no effect on `find_program()`, it is replaced by the `CMAKE_FIND_ROOT_PATH_MODE_PROGRAM` variable which has the equivalent effect but applies exclusively to `find_program()` only. When cross-compiling, it is usually the case that it is a host platform tool being sought rather than a program on the target platform, so `CMAKE_FIND_ROOT_PATH_MODE_PROGRAM` is frequently set to `NEVER`.

24.4. Finding Libraries

Finding libraries is also similar to finding files. The `find_library()` command supports the same set of options as `find_file()` plus an additional `NAMES_PER_DIR` option. The following differences apply:

Cache variables (CMake-specific)

- When searching under `CMAKE_PREFIX_PATH`, `find_file()` appends `include` to each item, whereas `find_library()` instead appends `lib`. The `CMAKE_LIBRARY_ARCHITECTURE` variable is also honored in the same way as for `find_file()`.
- `CMAKE_LIBRARY_PATH` replaces `CMAKE_INCLUDE_PATH` but is otherwise used in exactly the same way. `CMAKE_LIBRARY_PATH` is used only by `find_library()`. The `CMAKE_FRAMEWORK_PATH` variable is used in exactly the same way as for `find_file()`.

Environment variables (system-specific)

- The search locations for standard system environment variables are handled in a very similar way to `find_file()`. Instead of `INCLUDE`, the `LIB` environment variable is consulted. Furthermore, the search locations based on `PATH` follow the same complex logic as for `find_file()`, except that `lib` is appended to each prefix rather than `include`. Just as for `find_file()`, the complex `PATH` logic only applies on Windows.

General

- The `NAMES_PER_DIR` option has exactly the same meaning as it does for `find_program()`. It is only available with CMake 3.4 or later.
- Both `find_file()` and `find_library()` use `CMAKE_FIND_FRAMEWORK` to determine the search order between framework and non-framework paths. In the case of `find_library()`, if a framework is found then the name of the top level `.framework` directory is stored in the result variable.
- `CMAKE_FIND_ROOT_PATH_MODE_INCLUDE` has no effect on `find_library()`, it is replaced by the `CMAKE_FIND_ROOT_PATH_MODE_LIBRARY` variable which has the equivalent effect but applies exclusively to `find_library()`. On Apple platforms, consider carefully before setting `CMAKE_FIND_ROOT_PATH_MODE_LIBRARY` to `ONLY`, as libraries may be built as fat binaries which support multiple target platforms. These fat binaries may not reside under target platform-specific paths, so it may still be necessary to search host platform paths to find them.

Further behavioral differences apply when using `find_library()`. Platforms have different conventions for library names, such as prepending `lib` on most Unix platforms. The file extensions also vary considerably across platforms. DLLs on Windows can have an associated import library with a different file extension.

The `find_library()` command does its best to abstract away most of these differences, allowing projects to specify just the base name of the library as the name to search for. Where a directory contains both static and shared libraries, the shared library will be the one found. Most of the time,

Exclusively for sbmain17@gmail.com Transaction: 0098315591

this abstraction works well, but in some circumstances it can be useful to override this behavior. One common case is to give priority to static libraries ahead of shared libraries, potentially only on some platforms and not others. The following naive example would prefer a static `foobar` library ahead of shared on Linux, but not on macOS or Windows:

```
# WARNING: Not robust!
find_library(FOOBAR_LIBRARY NAMES libfoobar.a foobar)
```

Keep in mind that the priority override only applies to libraries found within a particular directory. If the set of search locations is such that a directory containing just a shared library is searched before a directory that contains a static library, then the above technique will not result in the static library being found. The more robust way to ensure that a static library is given priority over shared libraries across all search locations is to use multiple calls to `find_library()` like so:

```
# Better, static library now has priority across all search locations
find_library(FOOBAR_LIBRARY libfoobar.a)
find_library(FOOBAR_LIBRARY foobar)
```

Note that such techniques cannot be used on Windows because static libraries and the import library for shared libraries (i.e. DLLs) have the same file name, including suffix (e.g. `foobar.lib`). Therefore, the file name cannot be used to differentiate between the two types of libraries.

Another complication unique to library handling is that many platforms support both 32- and 64-bit architectures. There may be both 32- and 64-bit versions of libraries installed to different locations, but with the same file names. The directory structure used to separate the different architectures on such multilib systems can vary, even between distributions for the same platform. For example, some distributions place 64-bit libraries under `lib` directories and 32-bit libraries under `lib32`. Others place 64-bit libraries under `lib64` and the 32-bit libraries under `lib`. Other platforms use yet another variation, a `libx32` subdirectory. CMake is generally aware of the variations and when setting up the platform defaults, it populates the global properties `FIND_LIBRARY_USE_LIB32_PATHS`, `FIND_LIBRARY_USE_LIB64_PATHS` and `FIND_LIBRARY_USE_LIBX32_PATHS` with appropriate values to control which architecture-specific directories should be searched first, if any. Projects can override these with their own custom prefix using the `CMAKE_FIND_LIBRARY_CUSTOM_LIB_SUFFIX` variable, but the need for this should be very rare.

When an architecture-specific suffix is active (whether from one of the above global properties or from the `CMAKE_FIND_LIBRARY_CUSTOM_LIB_SUFFIX` variable), the logic used to augment the search locations with architecture-specific locations is non-trivial. Any directory anywhere in the search location path that ends with `lib` is augmented with an architecture-specific equivalent. This occurs recursively throughout the path, so a search location like `/opt/mylib/foo/lib` may result in the set of search locations being expanded out to `/opt/mylib64/foo/lib64`, `/opt/mylib64/foo/lib`, `/opt/mylib/foo/lib64` and `/opt/mylib/foo/lib` on some 64-bit systems. Even if a search location does not end with `lib`, it will still be augmented with an architecture-suffixed location, so a search location `/opt/foo` may result in `/opt/foo64` and `/opt/foo` being searched on some 64-bit systems.

The details of the architecture-specific search path augmentation are not typically something developers need to concern themselves with. In those situations where undesirable libraries are being found or desired libraries are being missed, it may be more straightforward to coerce the

Exclusively for sbmain17@gmail.com Transaction: 0098315591

result using variables like `CMAKE_LIBRARY_PATH` rather than trying to manipulate the architecture-specific logic. A detailed knowledge of the intricacies involved is not typically needed, a simple awareness of the above points should generally be sufficient, if for no other reason than to reduce some of the mystery around how CMake finds libraries in architecture-specific locations.

Special care needs to be exercised when working with a CMake generator that supports switching between device and simulator configurations at build time. Any `find_library()` results would generally be unusable for such cases, since they could only ever find a library for either the device or the simulator, but not both. Even if CMake is re-run, it would retain its cached results and so would not update the library location unless the relevant cache entry was manually deleted first. This is a particularly common problem with Xcode builds where projects might want to use `find_library()` to locate various frameworks or common libraries such as zlib. For these situations, projects have little choice but to specify the linker flags directly without paths instead, leaving the linker to find the library on its search path. For Apple frameworks, this means specifying two values since frameworks are added using `-framework <FrameworkName>`. For ordinary libraries like zlib, the more traditional `-lz` would be sufficient.

24.5. Finding Packages

The various `find_…()` commands discussed in the preceding sections all focus on finding one specific item. Quite often, however, these items are just one part of a larger package and the package as a whole may have its own characteristics that projects could be interested in, such as a version number or support for certain features. Projects will generally want to find the package as a single unit rather than piece together its different parts manually.

There are two main ways packages are defined in CMake, either as a module or through config details. Config details are usually provided as part of the package itself and they are more closely aligned with the functionality of the various `find_…()` commands discussed in the preceding sections. Modules, on the other hand, are typically defined by something unrelated to the package (usually by CMake or by projects themselves) and as a result, they are harder to keep up to date as the package evolves over time.

Module and config files typically define variables and imported targets for the package. These may provide the location of programs, libraries, flags to be used by consuming targets and so on. Functions and macros can also be defined. There is no set of requirements for what will be provided, but there are some conventions which are stated in the CMake developer manual. Project authors must consult the documentation of each module or package to understand what is provided. As a general guide, older modules tend to provide variables that follow a fairly consistent pattern, whereas newer modules and config implementations usually define imported targets. Where both variables and imported targets are provided, projects should prefer the latter due to their superior robustness and better integration with CMake's transitive dependency features.

Projects normally look for a package using the `find_package()` command, which has a short form and a long form. The short form should generally be preferred because of its greater simplicity and because it supports both module and config packages, whereas the long form does not support modules. The long form does, however, provide more control over the search, making it preferable in certain situations.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

The short form has only a few options and can be summarized as follows:

```
find_package(packageName
    [version [EXACT] ]
    [QUIET] [REQUIRED]
    [ [COMPONENTS] component1 [component2...] ]
    [OPTIONAL_COMPONENTS component3 [component4...] ]
    [MODULE]
    [NO_POLICY_SCOPE]
)
```

The optional `version` argument indicates that the package must be of the specified version or higher, or match exactly if `EXACT` is also given. When using CMake 3.19 or later, the `version` can alternatively be specified as a version *range* instead. Version ranges are expressed in the form `versionMin...versionMax` or `versionMin...<versionMax`. The `versionMin` part of the range is treated just like a single version number. It is the minimum required version of the package. For the upper end of the version constraint, the first form requires the package version to be no greater than `versionMax`, whereas the second form requires it to be strictly less than `versionMax`. The `EXACT` keyword cannot be given if a version range is used. Be aware that packages may not know about version ranges. Their module or config file may have a fairly old implementation that pre-dates CMake's version range support. In these cases, the packages will typically ignore the `versionMax` part of the requirement. Section 26.8.1, "Config Files For CMake Projects" discusses further implications for how packages may handle version ranges.

A package may be optional, meaning the project can use it if available or work without it if an appropriate package cannot be found. If a package is mandatory, the `REQUIRED` option should be provided so that CMake will halt with an error if an appropriate package could not be found. Unlike the other `find_⋯()` commands, all CMake versions support the `REQUIRED` option for `find_package()`.

Normally, `find_package()` will log messages for failures, but the `QUIET` option can be used to suppress them for optional packages (failure messages for a mandatory package cannot be suppressed). `QUIET` will also suppress messages that would normally be printed the first time a package is found. A typical use for `QUIET` is to prevent messages for a missing optional package so that developers are less likely to think it is an error.

The component-related options allow a project to indicate what parts of the package they are interested in. Not all packages support components, it is up to the module or config implementation whether or not components are defined and what the components represent. An example where components may be useful is a large package like Qt where not all components might be installed. It may not be enough for a project to just say it wants Qt, it may also need to say which parts of Qt. The `find_package()` command allows the project to specify components as mandatory with the `COMPONENTS` arguments or as optional with the `OPTIONAL_COMPONENTS` arguments. For example, the following call requires Qt 5.9 or later, the `Gui` component must be available and `DBus` is optional:

```
find_package(Qt5 5.9 REQUIRED
    COMPONENTS Gui
    OPTIONAL_COMPONENTS DBus
)
```

Exclusively for sbmain17@gmail.com Transaction: 0098315591

When the REQUIRED option is present, the COMPONENTS keyword can be omitted and the mandatory components placed after REQUIRED. This is common when there are no optional components:

```
find_package(Qt5 5.9 REQUIRED Gui Widgets Network)
```

If a package defines components but no components are given to find_package(), it is up to the module or config definition how this is handled. For some packages, it may be treated as though all components were listed, for others it may be interpreted as no components are required (basic details of the package may still be defined though, such as base libraries, package version, etc.). Another possibility is that the lack of components could be treated as an error. Given the variation in behavior, developers should consult the documentation for the package they wish to find.

The remaining options of the short form are less frequently used. The NO_POLICY_SCOPE keyword is a historical hangover from the CMake 2.6 era and projects should avoid using it. The MODULE keyword restricts the call to searching only for modules and not config packages. Projects should generally avoid using this option since they should not have to concern themselves with the implementation details of how a package is defined, only with stating the requirements on the package. When MODULE is not present, the short form of the find_package() command will first search for a matching module, then if no such module is found it will search instead for a config package. CMake 3.15 added support for the CMAKE_FIND_PACKAGE_PREFER_CONFIG variable, which can be set to true to reverse the search preference (it is unset by default to preserve the pre-3.15 behavior).

Modules were first discussed back in Chapter 11, *Modules*. While non-package modules are incorporated into a project using the include() command, package modules have a file name of the form Find<packageName>.cmake and are intended to be processed by a call to find_package() instead. For this reason, they are commonly referred to as *Find modules*. Both include() and find_package() respect the CMAKE_MODULE_PATH variable as a list of directories that CMake should search in before the set of modules that come as part of every CMake release.

Find modules are responsible for implementing all aspects of the find_package() call, including locating the package, performing version checks, fulfilling component requirements and logging or not logging messages as appropriate. Not all find modules honor these responsibilities and they may choose to ignore some or all of the information provided beyond the package name, so as always, consult the module documentation to confirm the expected behavior.

Find modules are usually implemented in terms of calls to the various find_⋯() commands. As a result, they can sometimes be affected by the cache and environment variables relevant to those commands. The CMAKE_PREFIX_PATH variable is especially convenient for influencing find modules because each path specified acts as a base point below which each find_⋯() command appends its own command-specific subdirectories. For packages that follow a reasonably standard layout, adding just the base install location of the package to CMAKE_PREFIX_PATH is often enough for the find module to find all the package components it needs.

Compared to find modules, packages with config details offer a much richer, more robust way for projects to retrieve information about that package. A much more extensive set of find_package() options are available in config mode, with the full long form of the command having many similarities to the other find_⋯() commands:

Exclusively for sbmain17@gmail.com Transaction: 0098315591

```
find_package(packageName
    [version [EXACT] ]
    [QUIET | REQUIRED]
    [ [COMPONENTS] component1 [component2...] ]
    [OPTIONAL_COMPONENTS component3 [component4...] ]
    [NO_MODULE | CONFIG]
    [NO_POLICY_SCOPE]
    [NAMES name1 [name2 ...] ]
    [CONFIGS fileName1 [fileName2...] ]
    [HINTS path1 [path2 ... ] ]
    [PATHS path1 [path2 ... ] ]
    [PATH_SUFFIXES suffix1 [suffix2 ...] ]
    [CMAKE_FIND_ROOT_PATH_BOTH |
     ONLY_CMAKE_FIND_ROOT_PATH |
     NO_CMAKE_FIND_ROOT_PATH]
    [<skip-options>]    # See further below
)
```

When `find_package()` is called with an option only supported by the long form, the search for a Find module is skipped. The `NO_MODULE` or `CONFIG` keywords allow a call that would otherwise match the short form to be treated as the long form and hence only search for config details (both keywords are equivalent).

When searching for config details, `find_package()` looks for a file named `<packageName>Config.cmake` or `<lowercasePackageName>-config.cmake` by default. The `CONFIGS` option can be used to specify a different set of file names to search for instead, but this is discouraged. Non-default file names require every project wanting to find that package to be aware of the non-default file name.

When a config file is found, `find_package()` also looks for an associated version file in the same directory. The version file has `Version` or `-version` appended to the base name, so `FooConfig.cmake` would result in looking for a version file named `FooConfigVersion.cmake` or `FooConfig-version.cmake`, while `foo-config.cmake` would result in looking for `foo-configVersion.cmake` or `foo-config-version.cmake`. Packages are not required to provide a version file, but they usually do. If version details are included in a call to `find_package()` but there is no version file for that package, the version requirements are deemed to have failed.

The locations searched follow a similar pattern to the other `find_…()` commands, except package registries are also supported. Each search location is then treated as a possible package install base point below which a variety of subdirectories may be searched:

```
<prefix>/
<prefix>/(cmake|CMake)/
<prefix>/<packageName>*/
<prefix>/<packageName>*/(cmake|CMake)/
<prefix>/(lib/<arch>|lib*|share)/cmake/<packageName>*/
<prefix>/(lib/<arch>|lib*|share)/<packageName>*/
<prefix>/(lib/<arch>|lib*|share)/<packageName>*/(cmake|CMake)/
<prefix>/<packageName>*/(lib/<arch>|lib*|share)/cmake/<packageName>*/
<prefix>/<packageName>*/(lib/<arch>|lib*|share)/<packageName>*/
<prefix>/<packageName>*/(lib/<arch>|lib*|share)/<packageName>*/(cmake|CMake)/
```

Exclusively for sbmain17@gmail.com Transaction: 0098315591

The following are also checked on Apple platforms:

```
<prefix>/<packageName>.framework/Resources/
<prefix>/<packageName>.framework/Resources/CMake/
<prefix>/<packageName>.framework/Versions/*/Resources/
<prefix>/<packageName>.framework/Versions/*/Resources/CMake/
<prefix>/<packageName>.app/Contents/Resources/
<prefix>/<packageName>.app/Contents/Resources/CMake/
```

In the above, `<packageName>` is treated case-insensitively and the `lib/<arch>` subdirectories are only searched if `CMAKE_LIBRARY_ARCHITECTURE` is set. The `lib*` subdirectories represent a set of directories that may include `lib64`, `lib32`, `libx32` and `lib`, the last of which is always checked. If the `NAMES` option is given to `find_package()`, all of the above directories are checked for each name provided.

The set of search location base points checked follow the order defined in the following table, which shares many similarities to the other `find_⋯()` commands. Most search locations can be disabled by adding the associated `NO_⋯` keyword:

Location	Skip Option
Package root variables	`NO_PACKAGE_ROOT_PATH`
Cache variables (CMake-specific)	`NO_CMAKE_PATH`
Environment variables (CMake-specific)	`NO_CMAKE_ENVIRONMENT_PATH`
Paths specified via the `HINTS` option	
Environment variables (system-specific)	`NO_SYSTEM_ENVIRONMENT_PATH`
User package registry	`NO_CMAKE_PACKAGE_REGISTRY`
Cache variables (platform-specific)	`NO_CMAKE_SYSTEM_PATH`
System package registry	`NO_CMAKE_SYSTEM_PACKAGE_REGISTRY`
Paths specified via the `PATHS` option	

Package root variables

As for the other `find_⋯()` commands, support for package root variables was added as a search location in CMake 3.9.0, removed in 3.9.1 due to backward compatibility issues and re-added again in CMake 3.12. Each time `find_package()` is called, it pushes `<packageName>_ROOT` CMake and environment variables onto an internally maintained stack of paths. These paths are used in exactly the same way as `CMAKE_PREFIX_PATH`, not just for the current call to `find_package()`, but all `find_..()` commands that might be called as part of the `find_package()` processing. In practice, this means if a `find_package()` call loads a Find module, then any `find_⋯()` commands the Find module calls internally will use each path in the stack as though it was a `CMAKE_PREFIX_PATH` first before checking any other paths.

For example, say a `find_package(Foo)` call resulted in `FindFoo.cmake` being loaded. Any `find_⋯()` command within `FindFoo.cmake` would first search `${Foo_ROOT}` and `$ENV{Foo_ROOT}` (if they were set) before moving on to check other search locations. If `FindFoo.cmake` contained a call like `find_package(Bar)` that resulted in `FindBar.cmake` being loaded, then the stack would contain `${Bar_ROOT}`, `$ENV{Bar_ROOT}`, `${Foo_ROOT}` and `$ENV{Foo_ROOT}`. This feature means nested Find

Exclusively for sbmain17@gmail.com Transaction: 0098315591

modules will search the prefix locations of each of their parent Find modules first, so that information doesn't have to be manually propagated down via `CMAKE_PREFIX_PATH` or another similar method. For the most part, projects can ignore this functionality, since it should work transparently without any specific action by the project. It should mostly just be thought of as an automatic convenience.

Cache variables (CMake-specific)

The CMake-specific cache variable locations are derived from the cache variables `CMAKE_PREFIX_PATH`, `CMAKE_FRAMEWORK_PATH` and `CMAKE_APPBUNDLE_PATH`. These work the same way as for the other `find_…()` commands except that `CMAKE_PREFIX_PATH` entries already correspond to package install base points, so no directories like `bin`, `lib`, `include`, etc. are appended.

Environment variables (CMake-specific)

These have the same relationship to the cache variables above as other `find_…()` commands. The environment variables `CMAKE_PREFIX_PATH`, `CMAKE_INCLUDE_PATH` and `CMAKE_FRAMEWORK_PATH` all use the platform-specific path separator (colons on Unix platforms, semi-colons on Windows). An additional environment variable `<packageName>_DIR` is also checked before the other three.

Environment variables (system-specific)

The only supported system-specific environment variable is `PATH`. Each entry is used as a package install base point, except any trailing `bin` or `sbin` is removed. This is the point at which default system locations like `/usr` are likely to be searched on most systems.

Cache variables (platform-specific)

The platform-specific cache variable locations follow the same pattern as the other `find_…()` commands, providing `…_SYSTEM_…` equivalents of the CMake-specific cache variables. The names of these system variables are `CMAKE_SYSTEM_PREFIX_PATH`, `CMAKE_SYSTEM_FRAMEWORK_PATH` and `CMAKE_SYSTEM_APPBUNDLE_PATH` and they are not intended to be set by the project.

`HINTS` and `PATHS`

These work exactly the same way as the other `find_…()` commands except they do not support items of the form `ENV someVar`.

Package registries

Unique to `find_package()`, the user and system package registries are intended to provide a way to make packages easily findable without having them installed in standard system locations. See Section 24.5.1, "Package Registries" further below for a more detailed discussion.

The various `NO_…` options work the same way as for the other `find_…()` commands, allowing each group of search locations to be bypassed individually. The `NO_DEFAULT_PATH` keyword causes all but the `HINTS` and `PATHS` to be bypassed. With CMake 3.16 or later, the various `CMAKE_FIND_USE_…` variables also have the same effects as for the other `find_…()` commands. These variables allow the default behavior of each search location to be controlled individually. The `PATH_SUFFIXES` option has the expected effect too, accepting further subdirectories to check below each search location.

The `find_package()` command also supports the same search re-rooting logic as the other `find_…()` commands. `CMAKE_SYSROOT`, `CMAKE_STAGING_PREFIX` and `CMAKE_FIND_ROOT_PATH` are all considered in the same way as the other commands and the meanings of the `CMAKE_FIND_ROOT_PATH_BOTH`,

Exclusively for sbmain17@gmail.com Transaction: 0098315591

ONLY_CMAKE_FIND_ROOT_PATH and NO_CMAKE_FIND_ROOT_PATH options are also equivalent. The default re-root mode when none of these three options is provided is controlled by the CMAKE_FIND_ROOT_PATH_MODE_PACKAGE variable which has the predictable set of valid values (ONLY, NEVER or BOTH).

Unlike the other find_…() commands, when looking for a config file, find_package() does not necessarily stop searching at the first package it finds that matches the criteria. Parts of the search consider a family of search locations and the search results may return multiple matches for that particular sub-branch of the search. Typically this might occur if there are multiple versions of the package installed under some common directory, each of which has a versioned subdirectory below that common point. In such cases, the CMAKE_FIND_PACKAGE_SORT_ORDER and CMAKE_FIND_PACKAGE_SORT_DIRECTION variables are consulted to sort the candidates based on their version details. CMAKE_FIND_PACKAGE_SORT_DIRECTION must have the value DEC or ASC to indicate a descending (choose the newest) or ascending (choose the oldest) sort direction respectively, while CMAKE_FIND_PACKAGE_SORT_ORDER controls the type of sorting and has documented values of NAME, NATURAL or NONE. If set to NONE or not set at all, no sorting is performed and the first valid package found will be used. The NAME setting sorts lexicographically, while NATURAL sorts by comparing sequences of digits as whole numbers. The following table demonstrates the difference between the last two methods when sorting in descending order, which is the default behavior if CMAKE_FIND_PACKAGE_SORT_DIRECTION is not set:

NAME	NATURAL
1.9	1.10
1.10	1.9
1.0	1.0

In practice, the intricacies of the search logic are usually well beyond the level of detail needed to use the find_package() command effectively. As long as a package follows one of the more common directory layouts and sits under one of the higher level base install locations, the find_package() command will usually find its config file without further help.

Once a suitable config file for a package has been found, the <packageName>_DIR cache variable will be set to the directory containing that file. Subsequent calls to find_package() will then look in that directory first and if the config file still exists, it is used without further searching. <packageName>_DIR is ignored if there is no longer a config file for the package at that location. This arrangement ensures that subsequent calls to find_package() for the same package are much faster, even from one invocation of CMake to the next, but the search is still performed if the package is removed. Be aware, however, that the caching of the package location can also mean that CMake might not get an opportunity to become aware of a newly added package in a more preferable location. For example, the operating system might come with a fairly old version of a package pre-installed. The first time CMake is run on a project, it finds that old version and stores its location in the cache. The user sees that an old version is being used and decides to install a newer version of the package under some other directory, adds that location to CMAKE_PREFIX_PATH and re-runs CMake. In this scenario, the old version will still be used because the cache still points to the older package's location. The <packageName>_DIR cache entry would need to be removed or the old version uninstalled before the newer version's location would be considered.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

Further controls are available to influence the handling of specific packages. It is possible to disable every non-REQUIRED call to `find_package()` for a given `packageName` by setting the `CMAKE_DISABLE_FIND_PACKAGE_<packageName>` variable to true early in the project, ideally at the top level or as a cache variable. This can be thought of as a way of turning off an optional package, preventing it from being found via `find_package()` calls. Note that it will not prevent such calls if they include the `REQUIRED` keyword.

With CMake 3.22 or later, a `CMAKE_REQUIRE_FIND_PACKAGE_<packageName>` variable is also supported. Setting it to true for a particular `<packageName>` forces all `find_package()` calls for that package to behave as though they used the `REQUIRED` keyword. This can be used to catch situations where a package is expected to be available, forcing CMake to halt with an error if it is missing. Testing of logic that relies on optional packages is an example scenario where this variable may be useful.

24.5.1. Package Registries

Packages tend to be found in standard system locations or in directories CMake has been told about through `CMAKE_PREFIX_PATH` or similar. For non-system packages, it can be tedious or undesirable to have to specify the location for each package if they don't all share a common install prefix. CMake supports a form of package registry which allows references to arbitrary locations to be collected together in one place. This allows the user to maintain an account- or system-wide registry which CMake will consult automatically without further direction. The locations referenced by the registry don't have to be a full package install, they can also be a directory within a build tree for the package (or any other directory for that matter) as long as the required files are there.

On Windows, two registries are provided. A user registry is stored in the Windows registry under the `HKEY_CURRENT_USER` key, while a system package registry is stored under `HKEY_LOCAL_MACHINE`:

```
HKEY_CURRENT_USER\Software\Kitware\CMake\Packages\<packageName>\
HKEY_LOCAL_MACHINE\Software\Kitware\CMake\Packages\<packageName>\
```

For a given `packageName`, each entry under that point is an arbitrary name holding a `REG_SZ` value. The value is expected to be a directory in which a config file for that package can be found. On Unix platforms, there is no system package registry, only a user package registry stored under the user's home directory and entries under that point have the same meaning as for Windows:

```
~/.cmake/packages/<packageName>/
```

CMake provides very little assistance with how to actually create these entries on any platform. No automated mechanism is provided for installed packages, but the `export()` command can be used within a project's `CMakeLists.txt` files to add parts of a project's build tree to the user registry:

```
export(PACKAGE packageName)
```

This command can add the specified package to the user package registry and point that registry entry to the current binary directory associated with wherever `export()` was called (see below for conditions that can prevent this). It is then up to the project to ensure that an appropriate config file

Exclusively for sbmain17@gmail.com Transaction: 0098315591

for the package exists in that directory. If no such config file exists and a `find_package()` call is made for that package for any project, the registry entry will be automatically removed if permissions allow it. It is common practice for the name of each entry in the package registry to be the MD5 hash of the directory path it points to. This avoids name collisions and is the naming strategy employed by the `export(PACKAGE)` command.

Adding locations from a build tree to the package registry has its dangers. While `export(PACKAGE)` is available to add a location to the registry, there is no corresponding mechanism to remove it again other than to manually delete the registry entry or to remove the package config file from the build directory. It can be easy to forget to do this, so an old build tree left behind from past experiments can easily be picked up unexpectedly. The use of `export(PACKAGE)` also has the potential to play havoc with continuous integration systems by making projects pick up build trees of other projects built on the same machine.

Because of the dangers associated with `export(PACKAGE)`, developers will frequently want to disable it. CMake provides two ways to achieve this, one using an opt-out method available since CMake 3.1 and another introduced in CMake 3.15 which uses a superior opt-in mechanism. With CMake 3.14 or earlier, the `export(PACKAGE)` command will modify the package registry unless the `CMAKE_EXPORT_NO_PACKAGE_REGISTRY` variable is set to true. Because that variable is undefined by default, the `export(PACKAGE)` command will modify the package registry by default. In CMake 3.15, the default behavior was changed via policy `CMP0090` such that when that policy is set to `NEW`, the `export(PACKAGE)` command will be disabled unless the `CMAKE_EXPORT_PACKAGE_REGISTRY` variable is set to true (note the different variable name). If policy `CMP0090` is set to `OLD` or is not set, then the CMake 3.14 and earlier behavior is used. For most practical scenarios, developers can set `CMAKE_EXPORT_NO_PACKAGE_REGISTRY` to true and regardless of policy settings or CMake version, the `export(PACKAGE)` command will be disabled.

Whereas `CMAKE_EXPORT_NO_PACKAGE_REGISTRY` and `CMAKE_EXPORT_PACKAGE_REGISTRY` control writing to the registry, CMake provides a separate set of variables to control reading from it. The `CMAKE_FIND_PACKAGE_NO_PACKAGE_REGISTRY` and `CMAKE_FIND_PACKAGE_NO_SYSTEM_PACKAGE_REGISTRY` variables can be used to specify the default behavior of `find_package()` calls, controlling whether to read from the user and system package registries respectively. CMake 3.16 deprecated both variables, replacing them with `CMAKE_FIND_USE_PACKAGE_REGISTRY` and `CMAKE_FIND_USE_SYSTEM_PACKAGE_REGISTRY` to maintain more consistent naming and behavior with the other `CMAKE_FIND_USE_…` variables mentioned earlier in this chapter. These newer variables take precedence over the old ones where both the old and new variables are defined.

While the two sets of `CMAKE_EXPORT_…` and `CMAKE_FIND_…` variables described above are complementary, the `CMAKE_FIND_…` variables are more effective at isolating a build from the package registry and are usually more relevant for developers.

In practice, package registries are not often used. The limited help provided for adding and removing entries means maintaining the registry is somewhat of a manual process. When a package is installed via the host's standard package management system, it could conceivably add itself to either the system or user registry as appropriate, then the package's uninstaller could remove that same entry. While the package locations are well defined and their definition is conceptually easy, few packages bother to do the work to register and unregister themselves. The various different ways a package may find its way onto an end user's machine makes it somewhat difficult to implement such register/unregister features robustly and simply.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

24.5.2. FindPkgConfig

The `find_package()` command will generally be the preferred method for finding and incorporating a package into a CMake project, but in certain cases the results can be less than ideal. Some Find modules are yet to be updated to more modern practices and do not provide imported targets, relying instead on defining a collection of variables that consuming projects must handle manually. Other modules may fall behind the latest package releases, leading to incompatibilities or incorrect information being provided.

In some instances, a package may have support for `pkg-config`, a tool that provides similar information to `find_package()` but in a different form. If such `pkg-config` details are available, then the `PkgConfig` Find module may be used to read that information and provide it in a more CMake-friendly way. Imported targets can be automatically created, freeing projects from having to handle various variables manually. The `pkg-config` details are also likely to match the installed version of the package, since they are typically provided by the package itself.

The `FindPkgConfig` module locates the `pkg-config` executable and defines a few functions that invoke it to find and extract details about packages that have `pkg-config` support. If the module finds the executable, it sets the `PKG_CONFIG_FOUND` variable to true and the `PKG_CONFIG_VERSION_STRING` variable to the tool's version (except for CMake versions before 2.8.8). The `PKG_CONFIG_EXECUTABLE` variable is set to the location of the tool. CMake 3.22 and later also sets `PKG_CONFIG_ARGN` to any additional arguments to pass to the executable with each call. The user can explicitly set `PKG_CONFIG_EXECUTABLE` and `PKG_CONFIG_ARGN` if the module's defaults need to be overridden.

In practice, projects should rarely need to use the `PKG_CONFIG_EXECUTABLE` or `PKG_CONFIG_ARGN` variables. The module defines two functions which wrap the tool to provide a more convenient way to query package details. These two functions, `pkg_check_modules()` and `pkg_search_module()`, accept exactly the same set of options and have similar behavior. The main difference between the two is that `pkg_check_modules()` checks all the modules given in its argument list, whereas `pkg_search_module()` stops at the first one it finds that satisfies the criteria. The use of the term *module* rather than *package* is established in the history of these commands and may cause some confusion, but they have no direct relationship to regular CMake modules and can essentially be thought of as packages.

```
pkg_check_modules(prefix
    [REQUIRED] [QUIET]
    [IMPORTED_TARGET [GLOBAL] ]
    [NO_CMAKE_PATH]
    [NO_CMAKE_ENVIRONMENT_PATH]
    moduleSpec1 [moduleSpec2...]
)
pkg_search_module(prefix
    [REQUIRED] [QUIET]
    [IMPORTED_TARGET [GLOBAL] ]
    [NO_CMAKE_PATH]
    [NO_CMAKE_ENVIRONMENT_PATH]
    moduleSpec1 [moduleSpec2...]
)
```

Exclusively for sbmain17@gmail.com Transaction: 0098315591

The behavior of these functions has some similarities to find_package(). The REQUIRED and QUIET arguments have the same effect here as they do for the find_package() command. With CMake 3.1 or later, CMAKE_PREFIX_PATH, CMAKE_FRAMEWORK_PATH and CMAKE_APPBUNDLE_PATH are all considered as search locations in the same way too and the NO_CMAKE_PATH and NO_CMAKE_ENVIRONMENT_PATH keywords also have the same meaning here. The PKG_CONFIG_USE_CMAKE_PREFIX_PATH variable can be set to change the default behavior for whether or not these search locations are considered (it will be treated as a boolean switch to turn the search locations on or off), but projects should generally avoid it unless they need to support CMake versions older than 3.1.

The IMPORTED_TARGET option is only supported with CMake 3.6 or later. When given, if the requested module is found then an imported target with the name PkgConfig::<prefix> is created. This imported target will have interface details populated from the module's .pc file, providing such things as header search paths, compiler flags, etc. For this reason, it is highly recommended that this option be used if the minimum CMake version required by the project is 3.6 or later. If using CMake 3.13 or later, the GLOBAL keyword can also be added to make imported targets have global visibility instead of only to the current directory scope and below.

The functions expect one or more moduleSpec arguments to define what to search for. These can be a bare module/package name or they can combine the name with a version requirement. Such version requirements have the form name=version, name<=version or name>=version. With CMake 3.13 or later, < and > are also supported. When no version requirement is included, any version is accepted.

Upon return, the functions set a number of variables in the calling scope by calling pkg-config with the appropriate option(s) to extract the relevant part of the package details. Where multiple items are returned by a set of options (e.g. multiple libraries or multiple search paths), the corresponding variable will hold a CMake list.

Variable	pkg-config **options used**
prefix_LIBRARIES	--libs-only-l
prefix_LIBRARY_DIRS	--libs-only-L
prefix_LDFLAGS	--libs
prefix_LDFLAGS_OTHER	--libs-only-other
prefix_INCLUDE_DIRS	--cflags-only-I
prefix_CFLAGS	--cflags
prefix_CFLAGS_OTHER	--cflags-only-other
prefix_STATIC_LIBRARIES	--static --libs-only-l
prefix_STATIC_LIBRARY_DIRS	--static --libs-only-L
prefix_STATIC_LDFLAGS	--static --libs
prefix_STATIC_LDFLAGS_OTHER	--static --libs-only-other
prefix_STATIC_INCLUDE_DIRS	--static --cflags-only-I
prefix_STATIC_CFLAGS	--static --cflags
prefix_STATIC_CFLAGS_OTHER	--static --cflags-only-other

The above variables are only set if the module requirements are satisfied. The canonical way to check this is using the prefix_FOUND and prefix_STATIC_FOUND variables. For pkg_check_modules(), all

Exclusively for sbmain17@gmail.com Transaction: 0098315591

moduleSpec requirements must be satisfied for these variables to have a value of true, whereas pkg_search_module() only has to find one matching moduleSpec. With CMake 3.16 or later, pkg_search_module() populates the <prefix>_MODULE_NAME with the module that was found.

For pkg_check_modules(), some additional per-module variables are also set when modules are found successfully. In the following, if only one moduleSpec is given then YYY = prefix, otherwise YYY = prefix_moduleName.

YYY_VERSION

The version of the module found, extracted from output of the --modversion option.

YYY_PREFIX

The module's prefix directory. This is obtained by querying for a variable named prefix, which most .pc files typically define and which pkg-config provides by default anyway.

YYY_INCLUDEDIR

The result of querying for a variable named includedir. This is a common but not required variable.

YYY_LIBDIR

The result of querying for a variable named libdir. Again, this is a common but not required variable.

In CMake 3.4 and later, the FindPkgConfig module provides an additional function which can be used to extract arbitrary variables from .pc files:

```
pkg_get_variable(resultVar moduleName variableName)
```

This is used internally by pkg_check_modules() to query the prefix, includedir and libdir variables, but projects can use it to query the value of any arbitrary variable. Note that before CMake 3.15, pkg_get_variable() contained a bug which resulted in it effectively ignoring CMAKE_PREFIX_PATH, so consider making CMake 3.15 the minimum version if relying on this functionality.

For most common systems, the functions provided by the FindPkgConfig module work fairy reliably. The implementations of those functions do, however, rely on features introduced in pkg-config version 0.20.0. Some older systems (e.g. Solaris 10) come with older versions of pkg-config which result in all calls to the FindPkgConfig functions failing to find any modules successfully and no error message is logged to highlight that the pkg-config version is too old.

24.6. Debugging find_...() Calls

As the preceding sections demonstrate, the logic for the locations and names CMake searches for the various find_⋯() commands is complex. When a search returns something unexpected or it fails to find something that was thought to exist, it can be non-trivial to work out what went wrong. To help with this, CMake 3.17 added a new --debug-find command-line option which enables logging for calls to the built-in find_⋯() commands. This output may include a brief summary of the search settings and a list of each location and name that was checked. If a find_⋯() command call uses a cached value rather than actually performing a search, that call may produce no debug output.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

The `--debug-find` option applies to the whole build, so for large projects with many `find_⋯()` calls, the verbose output can be overwhelming. If developers only want to target a specific call or section of a project, a more effective strategy is to only enable `find_⋯()` command debugging around the specific calls of interest. This can be achieved by setting a variable called `CMAKE_FIND_DEBUG_MODE` to true before the calls of interest and to false after them (support for this variable was also added in CMake 3.17). For example:

```
set(CMAKE_FIND_DEBUG_MODE TRUE)
find_program(...)
set(CMAKE_FIND_DEBUG_MODE FALSE)
```

The debugging output is meant as a development aid for human consumption. It should not be used as input to any script or other form of automated processing, since the format and contents could change from one CMake version to another.

24.7. Recommended Practices

From CMake 3.0, there has been a conscious shift toward the use of imported targets to represent external libraries and programs rather than populating variables. This allows such libraries and programs to be treated as a coherent unit, collecting together not just the location of the relevant binary, but in the case of libraries, the associated header search paths, compiler defines and further library dependencies that consuming targets will need are also part of the imported target. This makes external libraries and programs as easy to use within a project as any other regular target the project defines. This shift in focus means that finding packages has become much more important than finding individual files, paths, etc. and there is an increasing push for projects to make themselves consumable by other CMake projects as packages. Finding individual files, etc. still has its uses and it is helpful to understand how that can be done, but developers should see it as a stepping stone to packages and/or imported targets rather than an end in itself. Wherever possible, prefer to find packages rather than individual things within packages.

When finding packages, most complications that arise are related to situations where multiple versions are installed in different locations. The user may not be aware of all the installed versions or there may be expectations about which one should be found ahead of the others. Rather than the project trying to predict such situations, it is generally more advisable to not deviate too far from the default search behavior and let the user provide their own overrides via cache or environment variables. `CMAKE_PREFIX_PATH` is usually the most convenient way to do this due to the way CMake automatically searches a range of common directory layouts below each prefix path listed.

Do not rely heavily on the version range support for `find_package()`. This feature is only available with CMake 3.19 or later and older packages will typically ignore the upper end of the version range constraint. Consider the upper limit to be advisory rather than strictly enforced.

All the `find_⋯()` commands except `find_package()` work in a similar way. By default, they cache a successful result to avoid having to repeat the whole find operation the next time the `find_⋯()` command is asked to find the same thing. This is cached even across multiple CMake invocations. Given the potentially large number of locations and directory entries each call may search through, the caching mechanism can save a non-trivial amount of time when there are many such `find_⋯()` invocations throughout the project. There are, however, at least two consequences of this that

Exclusively for sbmain17@gmail.com Transaction: 0098315591

developers need to be aware of. Firstly, once a `find_file()`, `find_path()`, `find_program()` or `find_library()` command succeeds, it will stop searching for all subsequent invocations, even if running the command would return a different result or if the entity found previously no longer exists. If the entity is removed, this can result in build errors that can only be rectified by removing the out-of-date entries from the cache. Developers often just delete their entire cache and rebuild again from scratch rather than trying to figure out which cache variables need to be removed. The other aspect of this find behavior that developers should be aware of is that where a call to one of these `find_…()` commands fails to find the desired entity, the search will be repeated for *every* call, even within the same project. An unsuccessful call is *not* cached. If a project has many such calls, this can slow down the configure step. Developers should therefore carefully consider how the project uses `find_…()` commands and try to minimize the likelihood and number of unsuccessful searches. If the minimum CMake version can be set to 3.21 or later, policy `CMP0125` also allows some subtle surprising behaviors to be avoided.

The situation with `find_package()` is a little more complicated. If the package is found via a Find module, then it is likely that all the above concerns will also apply to the package, since the logic is likely to be built upon the other `find_…()` commands. If the package is instead found via config mode, then `find_package()` will cache a successful result and check that location first on subsequent invocations. If the package no longer has an appropriate config file at that location, the command proceeds with its normal search logic. This unique behavior for config mode is much more robust and is closer to what developers would naturally want.

A particularly tricky situation where the caching of `find_…()` results can lead to subtle problems is with continuous integration systems. If incremental builds are being used where the CMake cache of a previous run is kept, then changes made in a project to the way it searches for things might not be reflected in the build. Only when the CMake cache is cleared might such changes take effect. The caching often also means that no details are logged about the entity being found, so the build output gives little clue about the use of the old search details. One might therefore be tempted to require all CI builds to build from scratch, but this may not be feasible for longer builds. A strategy which may help reduce the problem is to schedule a daily build job at a time of low CI load where the build tree is cleared and then the project is built as per normal. This will still keep the incremental behavior during regular hours and it will usually make any cache-related problems self-resolving within a day. The effectiveness of this strategy is reduced during periods where changes are being made on a branch and CI builds are alternating between that branch and other branches, but one would hope that such periods are not common and can be tolerated as long as developers are made aware of potential consequences during that time.

The package registry features of the `find_package()` command should be approached with caution. They have the potential to give unexpected results for continuous integration systems where projects may want to find packages that are also built on the same machine. Unfortunately, there is no environment variable that can be set to disable the use of the registries, but it can be enforced by the projects themselves by setting the `CMAKE_FIND_PACKAGE_NO_PACKAGE_REGISTRY` CMake variable to `OFF` (CI jobs would not normally have the required permissions to modify the system package registry, so setting `CMAKE_FIND_PACKAGE_NO_SYSTEM_PACKAGE_REGISTRY` as well should be unnecessary). In practice, few projects write into the package registry, so unless it is known that such a project might be using the CI system, the need to add this CMake variable to every potentially affected project is low. Projects should also avoid making calls to `export(PACKAGE)` within CI jobs (arguably they should avoid such calls in general).

Exclusively for sbmain17@gmail.com Transaction: 0098315591

Use of the `FindPkgConfig` module should be reserved only for those situations where `find_package()` is not suitable. Typically this is for a package where CMake provides a find module, but that find module is fairly old and does not provide imported targets, or where it falls behind the more recent package releases. The `FindPkgConfig` module is also useful for searching for packages that CMake knows nothing about and where the package does not provide its own CMake config file, but it does provide a `pkg-config` (i.e. `.pc`) file.

When using a toolchain file for cross-compilation, prefer to set `CMAKE_SYSROOT` rather than `CMAKE_FIND_ROOT_PATH`. While both affect the search paths of the various `find_⋯()` commands in the same way, only `CMAKE_SYSROOT` also ensures that the compiler and linker flags are properly augmented so that header inclusions and library linking work correctly.

In cross-compiling scenarios, it is also typical that searches for programs expect to find binaries that will run on the host, whereas searches for files and libraries typically expect to find things for the target. Therefore, it is very common to see the following in toolchain files to enforce such behavior by default:

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

One could argue that this should be set in the project rather than relying on it being set in a toolchain file. Technically the developer is free to use any toolchain file if they wish and it is the project that implicitly relies on default behavior that it then chooses to override or not. An added complexity here is that toolchain files may be re-read for each `project()` or `enable_language()` call, so if a project wants to enforce a particular combination of defaults, it would have to do so after every such call. A reasonable compromise, therefore, is for projects to include the above block before its first `project()` call and for toolchain writers to also include it. Then, if toolchain authors do not include such a block, at least the project still gets sensible defaults. If a toolchain file changes the defaults to something else, they will then be applied consistently throughout the whole project. Developers should be very wary of using settings other than those shown in the example just above, since it is such a common pattern than projects frequently assume it.

For situations where the developer is able to switch between device and simulator builds without re-running CMake (e.g. when using Xcode for an iOS project), avoid calls to `find_library()`. Any results obtained by such calls can only ever point to one of either the device or simulator library, not both. Add the underlying linker flags that link only by name and not by path in such cases, such as `-framework ARKit` or `-lz`. If the frameworks or libraries cannot be found on the default linker search path, the project will also need to provide linker options to extend the search paths to allow them to be found.

It is quite common for online examples and blog posts to show conflicting recommendations over whether to use `CMAKE_MODULE_PATH` or `CMAKE_PREFIX_PATH` to control where CMake searches for things. An easy way to remember the difference is that `CMAKE_MODULE_PATH` is only used by CMake when searching for `FindXXX.cmake` files or when a module is brought in via an `include()` command. For everything else, including searching for config package files, `CMAKE_PREFIX_PATH` is used.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

Chapter 25. Testing

A natural follow-on to building a project is to test the artifacts it created. The CMake software suite includes the CTest tool which can be used to automate the testing phase, or even the entire process of configuring, building, testing and even submitting results to a dashboard. This chapter first covers the simpler case of how to use CMake to define tests and execute them using the `ctest` command line tool. Automating the entire configure-build-test process uses much of that same knowledge and is discussed later in the chapter.

25.1. Defining And Executing A Simple Test

The first step to defining tests in a CMake project is to call `enable_testing()` somewhere in the top level `CMakeLists.txt` file. This would typically be done early, soon after the first `project()` call. The effect of this function is to direct CMake to write out a CTest input file in the `CMAKE_CURRENT_BINARY_DIR` with details of all the tests defined in the project (more accurately, those tests defined in the current directory scope and below). `enable_testing()` can be called in a subdirectory without error, but without a call to `enable_testing()` at the top level, the CTest input file will not be created at the top of the build tree, which is where it is normally expected to be.

Defining individual tests is done with the `add_test()` command:

```
add_test(NAME testName
         COMMAND command [arg...]
         [CONFIGURATIONS config1 [config2...]]
         [WORKING_DIRECTORY dir]
         [COMMAND_EXPAND_LISTS]   # CMake 3.16 or later
)
```

By default, the test will be deemed to pass if the command returns an exit code of 0, but more flexible pass/fail handling is supported and is discussed in the next section. Prior to CMake 3.19, the `testName` should not contain any spaces, quotes or other special characters. With CMake 3.19 or later, these constraints are removed when policy `CMP0110` is set to `NEW`. CMake 3.18.0 originally introduced the same capability, but it was reverted in 3.18.1 after it was found to break some projects. The capability was reintroduced again in 3.19.0, this time with a policy to ensure backward compatibility for projects that relied on the old behavior.

The `command` can be a full path to an executable or it can be the name of an executable target defined in the project. When a target name is used, CMake automatically substitutes the real path to the executable. This is particularly useful when using multi configuration generators like Xcode, Visual Studio or Ninja Multi-Config where the location of the executable will be configuration-dependent.

```
cmake_minimum_required(VERSION 3.0)
project(CTestExample)
enable_testing()

add_executable(TestApp testapp.cpp)
add_test(NAME noArgs COMMAND TestApp)
```

Exclusively for sbmain17@gmail.com Transaction: 0098315591

The automatic substitution of a target with its real location does not extend to the command arguments, only the command itself supports such substitution. If the location of a target needs to be given as a command line argument, generator expressions can be used. For example:

```
add_executable(App1 ...)
add_executable(App2 ...)

add_test(NAME WithArgs COMMAND App1 $<TARGET_FILE:App2>)
```

When running the tests, the user can specify which configuration should be tested. When the project is using a single configuration generator, the configuration does not have to match the build type. In particular, if no configuration is provided, an empty configuration is assumed. Without the optional `CONFIGURATIONS` keyword, the test will be run for all configurations regardless of the build type or what configuration has been requested by the user. If the `CONFIGURATIONS` keyword is given, only for those configurations listed will the test be run. Note that an empty configuration is still considered valid, so for the test to run in that scenario, an empty string would have to be one of the `CONFIGURATIONS` listed.

For example, to add a test that should only be executed for configurations that have debug information, the `Debug` and `RelWithDebInfo` configurations can be listed. Adding the empty string also makes the test run when no configuration is specified when running the tests:

```
add_test(NAME DebugOnly
    COMMAND TestApp
    CONFIGURATIONS Debug RelWithDebInfo ""
)
```

In most cases, the `CONFIGURATIONS` keyword is not needed and the test would be executed for all configurations, including the empty one.

By default, the test will run in the `CMAKE_CURRENT_BINARY_DIR` directory, but the `WORKING_DIRECTORY` option can be used to make the test run in some other location. An example of where this can be useful is to run the same executable in different directories to pick up different sets of input files without having to specify them as command line arguments.

```
add_test(NAME Foo
    COMMAND TestApp
    WORKING_DIRECTORY ${CMAKE_CURRENT_LIST_DIR}/Foo
)
add_test(NAME Bar
    COMMAND TestApp
    WORKING_DIRECTORY ${CMAKE_CURRENT_LIST_DIR}/Bar
)
```

If specifying a working directory, always use an absolute path. If a relative path is given, it will be interpreted as being relative to the directory in which `ctest` itself was launched, but that might not be the top of the build tree. In order to ensure the working directory is predictable, projects should avoid using a relative `WORKING_DIRECTORY`.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

If the specified working directory does not exist when the test is run, CMake versions 3.11 and earlier will not issue an error message and will still run the test, even though it fails to change the working directory. CMake 3.12 and later will catch the error and treat the test as failed. Regardless of what version of CMake is being used, it is the project's responsibility to ensure the working directory exists and has appropriate permissions.

CMake 3.16 added support for the `COMMAND_EXPAND_LISTS` keyword, which has the same effect as the same-named option for the `add_custom_command()` and `add_custom_target()` commands. When this keyword is present, any list given as a command name or argument is expanded. This list expansion occurs after any generator expressions are evaluated. One of the primary motivations for this feature is to avoid passing an unwanted empty string as a command argument after a generator expression expands to nothing. For example:

```
add_test(NAME Expander
    COMMAND someCommand $<$<CONFIG:Debug>:-g>
    COMMAND_EXPAND_LISTS
)
```

If the above test is run for non-Debug configurations, `COMMAND_EXPAND_LISTS` ensures that after the generator expression expands to nothing, no empty argument is added to the command line.

A reduced form of the `add_test()` command is also supported for backward compatibility reasons:

```
add_test(testName command [args...])
```

This form should not be used in new projects, since it lacks some of the features of the full `NAME` and `COMMAND` form. The main differences are that generator expressions are not supported and if `command` is the name of a target, CMake will not automatically substitute the location of its binary.

To run the tests, the `ctest` command line tool is used. It would normally be run from the top of the build directory, although with CMake 3.20 or later, a `--test-dir` command line option can be given to specify the directory for which tests should be run. By default, `ctest` will execute all defined tests one at a time, logging a status message as each test is started and completed, but hiding all test output. An overall summary of the tests will be printed at the end. Typical output would look something like this:

```
Test project /path/to/build/dir
    Start 1: FooWithBar
1/2 Test #1: FooWithBar..............   Passed    0.00 sec
    Start 2: FooWithoutBar
2/2 Test #2: FooWithoutBar...........   Passed    0.00 sec

100% tests passed, 0 tests failed out of 2

Total Test time (real) =   0.02 sec
```

If using a multi configuration generator like Xcode, Visual Studio or Ninja Multi-Config, `ctest` needs to be told which configuration to test. This is done by including the `-C configType` option where

Exclusively for sbmain17@gmail.com Transaction: 0098315591

`configType` will be one of the supported build types (`Debug`, `Release`, etc.). For single configuration generators, the `-C` option is not mandatory, since the build can only produce one configuration, so there is no ambiguity for where to find the binaries to execute. Nevertheless, it can still be useful to specify a configuration to avoid the less intuitive behavior of excluding tests that are defined to only run under certain configurations and where the empty string is not among those listed.

`ctest` can be instructed to show all test output and various other details about the run with the `-V` option. `-VV` shows an increased level of verbosity, but this is typically only needed by developers working on `ctest` itself. Even the `-V` level of verbosity is usually more detail than users want to see. It is more likely that only the output of failed tests needs to be shown, which can be achieved by passing the `--output-on-failure` option. Alternatively, developers can set the `CTEST_OUTPUT_ON_FAILURE` environment variable (the value isn't used, `ctest` merely checks if `CTEST_OUTPUT_ON_FAILURE` has been set). With CMake 3.18 or later, the `--stop-on-failure` option can also be given to end the test run immediately upon the first error encountered.

By default, each test will be run with the same environment as the `ctest` command. If a test requires changes to its environment, this can be done through the `ENVIRONMENT` test property. This property is expected to be a list of `NAME=VALUE` items that define environment variables to be set before running the test. Changes are local to that test only and do not affect other tests.

```
set_tests_properties(FooWithoutBar PROPERTIES
    ENVIRONMENT "FOO=bar;HAVE_BAZ=1"
)
```

With CMake 3.22 or later, the `ENVIRONMENT_MODIFICATION` test property can also be used. Instead of replacing an environment variable with a value determined at configure time, `ENVIRONMENT_MODIFICATION` can be used to apply a list of changes to one or more environment variables based on the variable's value when `ctest` runs. A change is specified in the form `varName=operation:value`. A variety of operations are supported, but some of the more useful ones include `string_append`, `string_prepend`, `path_list_append`, `path_list_prepend`, `cmake_list_append` and `cmake_list_prepend`. The `string_⋯` operations have the expected behavior of appending and prepending to the variable's existing value. The `cmake_list_⋯` operations do likewise, except they add a semicolon separator between the new value and the existing value as well. The `path_list_⋯` operations do the same, except the separator will be the one the host platform uses for `PATH`-like environment variables (i.e. a semicolon on Windows, a colon everywhere else).

```
# In this example, Algo is assumed to be a shared library defined elsewhere in
# the project and whose binary will be in a different directory to test_Algo
add_executable(test_Algo ...)
target_link_libraries(test_Algo PRIVATE Algo)

add_test(NAME CheckAlgo COMMAND test_Algo)

set_property(TEST CheckAlgo PROPERTY
    ENVIRONMENT_MODIFICATION
        SOME_VAR=string_append:ExtraPart
        QT_LOGGING_RULES=cmake_list_append:*.debug=true
)
```

continued...

Exclusively for sbmain17@gmail.com Transaction: 0098315591

```
if(WIN32)
    # Ensure the Algo DLL can be found at runtime
    set_property(TEST CheckAlgo APPEND PROPERTY
        ENVIRONMENT_MODIFICATION
            PATH=path_list_prepend:$<TARGET_FILE_DIR:Algo>
    )
endif()
```

With CMake 3.21 or earlier, situations where an environment variable needs to modify rather than replace an existing value are less straightforward. If the environment should be based on the one in which CMake is run rather than the `ctest` command, then the form `$ENV{SOMEVAR}` can be used to obtain existing values. For example:

```
if(WIN32)
    set(algoDir  "$<SHELL_PATH:$<TARGET_FILE_DIR:Algo>>")
    set(execPath "PATH=${algoDir}$<SEMICOLON>$ENV{PATH}")
    set_tests_properties(CheckAlgo PROPERTIES
        ENVIRONMENT "${execPath}"
    )
endif()
```

Modifying the environment based on the actual environment being used to invoke `ctest` rather than CMake is more involved. It can be achieved with a combination of `cmake -E env` invoking a script, with CMake-provided locations being passed as variables to the `cmake -E env` part, then the script does the actual task of augmenting the run-time environment using those values and invoking the test executable. Such an arrangement is complex, can be fragile and should be avoided unless there is a definite need to support such a use case for CMake 3.21 or earlier.

As a convenience primarily for IDE applications, when testing has been enabled, CMake defines a custom build target that invokes `ctest` with a default set of arguments. For the Xcode and Visual Studio generators, this target will be called `RUN_TESTS` and it will pass the currently selected build type as the configuration to `ctest`. For other generators, the target is simply called `test` and if it is a single configuration generator, that target does not specify any configuration when invoking `ctest`.

With CMake 3.16 or earlier, there is no facility to specify which tests will be executed or any other custom options to pass to `ctest` when using the `RUN_TESTS` or `test` build target. CMake 3.17 introduced the `CMAKE_CTEST_ARGUMENTS` variable, which can be used to prepend arbitrary options to the `ctest` command line for that build target.

25.2. Pass / Fail Criteria And Other Result Types

Basing the result of a test purely on the exit code of the test command can be quite restrictive. Another supported alternative is to specify regular expressions to match against the test output. The `PASS_REGULAR_EXPRESSION` test property can be used to specify a list of regular expressions, at least one of which the test output must match for the test to pass. These regular expressions frequently span across multiple lines. Similarly, the `FAIL_REGULAR_EXPRESSION` test property can be set to a list of regular expressions. If any of these match the test output, the test fails, even if the output also matches a `PASS_REGULAR_EXPRESSION` or the exit code is 0. A test can have both `PASS_REGULAR_EXPRESSION`

Exclusively for sbmain17@gmail.com Transaction: 0098315591

and FAIL_REGULAR_EXPRESSION set, just one of the two or neither. If PASS_REGULAR_EXPRESSION is set and is not empty, the exit code is not considered when determining whether the test passes or fails.

```
# Ignore exit code, check output to determine the pass/fail status
set_tests_properties(test_Foo PROPERTIES
    FAIL_REGULAR_EXPRESSION "warning|Warning|WARNING"
    PASS_REGULAR_EXPRESSION [[
Checking some condition for test_Foo: passed
+.*
All checks passed]]
)
```

Sometimes a test may need to be skipped, perhaps for reasons that only the test itself can determine. The SKIP_RETURN_CODE test property can be set to a value the test can return to indicate that it was skipped rather than failed. A test that exits with the SKIP_RETURN_CODE will override any other pass/fail criteria.

CMakeLists.txt

```
add_executable(test_Foo test_Foo.cpp ...)
add_test(NAME Foo COMMAND test_Foo)

set_tests_properties(Foo PROPERTIES
    SKIP_RETURN_CODE 2
)
```

test_Foo.cpp

```
int main(int argc, char* argv[])
{
    if (shouldSkip())
        return 2; // Skipped

    if (runTest())
        return 0; // Passed

    return 1; // Failed
}
```

Output from the above test may look similar to the following:

```
Test project /path/to/build/dir
    Start 1: Foo
1/1 Test #1: Foo ....................***Skipped   0.00 sec

100% tests passed, 0 tests failed out of 1

Total Test time (real) =   0.01 sec

The following tests did not run:
      1 - Foo (Skipped)
```


Exclusively for sbmain17@gmail.com Transaction: 0098315591

From CMake 3.16, a SKIP_REGULAR_EXPRESSION can also be specified. This works in exactly the same way as PASS_REGULAR_EXPRESSION and FAIL_REGULAR_EXPRESSION, causing the test to be skipped if the output matches any of the skip expressions. A skip regular expression also takes precedence over any pass or fail criteria.

When at least one test fails or is not run for some reason, a summary of all such tests and their status is printed at the end. A test that indicates it should be skipped via its return code is still counted in the total number of tests. These skipped tests are not considered failures with CMake 3.9 or later, but they *are* considered failures with CMake 3.8 and earlier. Regardless of CMake version, a test may also be skipped for other reasons which could be deemed a failure, such as a test dependency failing to be met (discussed in Section 25.6, "Test Dependencies" below).

With CMake 3.9 or later, a DISABLED test property is also supported. This can be used to mark a test as temporarily disabled, which will allow it to be defined, but not executed or even counted in the total number of tests. It will not be considered a test failure, but it will still be shown in the test results with an appropriate status message. Note that such tests should not normally remain disabled for extended periods, the feature is intended as a temporary way to disable a problematic or incomplete test until it can be fixed. The following simple example demonstrates the DISABLED test behavior:

```
add_test(NAME FooWithBar ...)
add_test(NAME FooWithoutBar ...)

set_tests_properties(FooWithoutBar PROPERTIES DISABLED YES)
```

The ctest output for the above may look something like this:

```
Test project /path/to/build/dir
    Start 1: FooWithBar
1/2 Test #1: FooWithBar ..............   Passed    0.00 sec
    Start 2: FooWithoutBar
2/2 Test #2: FooWithoutBar ...........***Not Run (Disabled)   0.00 sec

100% tests passed, 0 tests failed out of 1

Total Test time (real) =   0.01 sec

The following tests did not run:
      2 - FooWithoutBar (Disabled)
```

In some cases, a test may be expected to fail. Rather than disabling the test, it may be more appropriate to mark the test as expecting failure so that it continues to be executed. The WILL_FAIL test property can be set to true to indicate this, which will then invert the pass/fail result. This has the added advantage that if the test starts to pass unexpectedly, ctest will consider that a failure and the developer is immediately aware of the change in behavior.

Another aspect of a test's pass/fail status is how long it takes to complete. The TIMEOUT test property, if set, specifies the number of seconds the test is allowed to run before it will be terminated and marked as failed. The ctest command line also accepts a --timeout option which has the same effect

Exclusively for sbmain17@gmail.com Transaction: 0098315591

for any test without a TIMEOUT property set (i.e. it acts as a default timeout). Furthermore, a time limit can also be applied to the entire set of tests as a whole by specifying the --stop-time option to ctest. The argument after --stop-time must be a real time of day rather than a number of seconds, with local time assumed if no timezone is given.

```
add_test(NAME t1 COMMAND ...)
add_test(NAME t2 COMMAND ...)

set_tests_properties(t2 PROPERTIES TIMEOUT 10)
```

```
ctest --timeout 30 --stop-time 13:00
```

In the above example, the default per-test timeout is set to 30 seconds on the ctest command line. Since t1 has no TIMEOUT property set, it will have a 30 second timeout, whereas t2 has its TIMEOUT property set to 10, which will override the default set on the ctest command line. The tests will be given until 1pm local time to complete.

In some circumstances, a test may need to wait for a particular condition before it starts the test proper. It may be desirable to apply a timeout to just the part of the run after that condition has been met and the real test begins. With CMake 3.6 or later, the TIMEOUT_AFTER_MATCH test property is available to support this behavior. It expects a list containing two items, the first being the number of seconds to be used as a timeout after the condition is met and the second is a regular expression to be matched against the test output. When the regular expression is found, the test's timeout countdown and start time is reset and the timeout value is set to the first list item.

For example, the following will apply an overall timeout of 30 seconds to the test, but once the string Condition met appears in the test output, the test will have 10 seconds to complete from that point and the original 30 second timeout condition will no longer apply:

```
set_tests_properties(t2 PROPERTIES
    TIMEOUT 30
    TIMEOUT_AFTER_MATCH "10;Condition met"
)
```

If the test took 25 seconds for the condition to be satisfied, the overall time of the test could be as long as 35 seconds, but because the test's start time is also reset, ctest would report a time between 0 and 10 seconds (i.e. the time for the condition to be met is not counted). If, on the other hand, the condition fails to be met within 30 seconds, the test will show an overall test time of about 30 seconds.

The reported times for the above can be somewhat confusing. Therefore, where possible, the use of TIMEOUT_AFTER_MATCH should generally be avoided in favor of other ways to handle preconditions. Section 25.6, "Test Dependencies" and Section 25.4, "Parallel Execution" further below discuss better alternative methods.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

25.3. Test Grouping And Selection

In larger projects, it is quite common to want to run just a subset of all defined tests. The developer may be focusing on a particular failing test and may not be interested in all the other tests while working on that problem. CMake offers a few different ways to narrow down the set of tests to run.

25.3.1. Regular Expressions

One way to execute just a specific subset of tests is by giving the `-R` and `-E` options to `ctest`. These options each specify a regular expression to be matched against test names. The `-R` option selects tests to be included in the test set, whereas `-E` excludes tests. Both options can be specified to combine their effects.

```
add_test(NAME FooOnly    COMMAND ...)
add_test(NAME BarOnly    COMMAND ...)
add_test(NAME FooWithBar COMMAND ...)
add_test(NAME FooSpecial COMMAND ...)
add_test(NAME Other_Foo  COMMAND ...)
```

```
ctest -R Only                     # Run just FooOnly and BarOnly
ctest -E Bar                      # Run all but FooWithBar
ctest -R '^Foo' -E FooSpecial     # Run all tests starting with Foo except FooSpecial
ctest -R 'FooSpecial|Other_Foo'   # Run only FooSpecial and Other_Foo
```

Sometimes it isn't always easy to work out a regular expression to capture just the desired tests, or a developer may just want to see all the tests that have been defined without running them. The `-N` option instructs `ctest` to only print the tests rather than run them, which can be a useful way to check that the regular expressions yield the desired set of tests.

```
ctest -N

Test project /path/to/build/dir
  Test #1: FooOnly
  Test #2: BarOnly
  Test #3: FooWithBar
  Test #4: FooSpecial
  Test #5: Other_Foo

Total Tests: 5
```

```
ctest -N -R 'FooSpecial|Other_Foo'

Test project /path/to/build/dir
Test #4: FooSpecial
Test #5: Other_Foo

Total Tests: 2
```

Exclusively for sbmain17@gmail.com Transaction: 0098315591

25.3.2. Test Numbers

As each test is added, it is given a test number. This number will remain the same between runs unless another test is added or removed before it in the project. The `ctest` output shows this number beside the test. When using the `-N` option, tests are listed in the order they have been defined by the project, but the tests might not necessarily be executed in that order. Tests to be run can be selected by test number rather than name using the `-I` option. This method is rather fragile, since the addition or removal of a single test can change the number assigned to any number of other tests. Even passing a different configuration via the `-C` option to `ctest` can result in the test numbers changing. In most cases, matching by name will be preferable.

One situation where test numbers can be useful is where two tests have been given exactly the same name. Except when defined in the same directory, both tests are accepted without any warnings being issued. While duplicate test names should generally be avoided, in hierarchical projects involving externally provided tests, this may not always be possible.

The `-I` option expects an argument which has a somewhat complicated form. The most direct form involves specifying test numbers on the command line, separated by commas with no spaces:

```
ctest -I [start[,end[,stride[,testNum[,testNum...]]]]]
```

To specify just individual test numbers, the `start`, `end` and `stride` can be left blank like so:

```
ctest -I ,,,3,2     # Selects tests 2 and 3 only
```

The same details can be read from a file instead of being specified on the command line by giving the name of the file to the `-I` option. This can be useful if regularly running the same complicated set of tests and no tests are being added or removed:

testNumbers.txt

```
,,,3,2
```

```
ctest -I testNumbers.txt
```

25.3.3. Labels

Selecting tests individually by name or number can become cumbersome if a large set of related tests needs to be executed. Tests can be assigned an arbitrary list of labels using the `LABELS` test property and then tests can be selected by these labels. The `-L` and `-LE` options are analogous to the `-R` and `-E` options respectively, except they operate on test labels rather than test names. Continuing with the same tests defined in the earlier example:

Exclusively for sbmain17@gmail.com Transaction: 0098315591

```
set_tests_properties(FooOnly    PROPERTIES LABELS "Foo")
set_tests_properties(BarOnly    PROPERTIES LABELS "Bar")
set_tests_properties(FooWithBar PROPERTIES LABELS "Foo;Bar;Multi")
set_tests_properties(FooSpecial PROPERTIES LABELS "Foo")
set_tests_properties(Other_Foo  PROPERTIES LABELS "Foo")
```

```
ctest -L Bar

Test project /path/to/build/dir
    Start 2: BarOnly
1/2 Test #2: BarOnly .................   Passed    1.52 sec
    Start 3: FooWithBar
2/2 Test #3: FooWithBar ..............   Passed    1.02 sec

100% tests passed, 0 tests failed out of 2

Label Time Summary:
Bar      =   2.53 sec*proc (2 tests)
Foo      =   1.02 sec*proc (1 test)
Multi    =   1.02 sec*proc (1 test)

Total Test time (real) =   2.54 sec
```

Labels not only enable convenient grouping for test execution, they also provide grouping for basic execution time statistics. As seen in the above example output, the `ctest` command prints a label summary when any tests in the set of executed tests has its `LABELS` property set. This allows the developer to get an idea how each label group is contributing to the overall test time. The `proc` part of the `sec*proc` units refers to the number of processors allocated to tests (described in Section 25.4, "Parallel Execution" below). A test that ran for 3 seconds and required 4 processors would report a value of 12. The label time summary can be suppressed with the `--no-label-summary` option.

With CMake 3.22 or later, labels can also be added dynamically to a test when the test runs. See the discussion in Section 25.9.4, "Test Measurements And Results" for how to do this.

25.3.4. Repeating Tests

Another common need is to re-run just those tests that failed the last time `ctest` was run. This can be a convenient way to re-check just the relevant tests after making a small fix or to re-run tests that failed due to some temporary environmental condition. The `ctest` command supports a `--rerun-failed` option which provides this behavior without needing any test names, numbers or labels to be given.

Sometimes a particular test or set of tests only fails intermittently, so the test(s) may need to be run many times to try to reproduce a failure. Rather than running `ctest` itself over and over, the `--repeat-until-fail` option can be given with the upper limit on the number of times each test can be repeated. If a test fails, it will not be re-run again for that `ctest` invocation.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

```
ctest -L Bar --repeat-until-fail 3

Test project /path/to/build/dir
    Start 2: BarOnly
    Test #2: BarOnly .................   Passed    1.52 sec
    Start 2: BarOnly
    Test #2: BarOnly .................***Failed    0.00 sec
    Start 3: FooWithBar
    Test #3: FooWithBar ..............   Passed    1.02 sec
    Start 3: FooWithBar
    Test #3: FooWithBar ..............   Passed    1.02 sec
    Start 3: FooWithBar
2/2 Test #3: FooWithBar ..............   Passed    1.02 sec

50% tests passed, 1 tests failed out of 2

Label Time Summary:
Bar      =   1.02 sec*proc (2 tests)
Foo      =   1.02 sec*proc (1 test)
Multi    =   1.02 sec*proc (1 test)

Total Test time (real) =   4.59 sec

The following tests FAILED:
          2 - BarOnly (Failed)
Errors while running CTest
```

The label summary doesn't accumulate the total time for the repeated tests, it only uses the time of a test's last execution. The total test time does, however, count all repeats.

CMake 3.17 expanded the repeat capabilities to re-run tests covering more situations. A new `ctest` option `--repeat mode:n` was added, where `n` is the maximum number of times a test will be run and `mode` is one of the following:

`until-fail`

This corresponds to the `--repeat-until-fail` option and is provided for consistency.

`until-pass`

This mode re-runs a test until it passes. Ordinarily, tests should always pass, but occasionally this option may be useful during development or to investigate a problem. It should not be relied upon for a more permanent arrangement.

`after-timeout`

Certain types of tests may experience an occasional failure due to external environmental factors. For example, a test case may need to perform a network operation and it may sometimes take longer than expected or even block indefinitely. Tests that interact with websites across the internet are especially susceptible to such timeouts. This option can be used to allow such tests to be retried if they timeout. A low value for `n` should typically be used to avoid having repeated timeouts significantly extend the overall test time.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

25.4. Parallel Execution

Maximizing the test throughput can be an important consideration for large projects or where tests take a non-trivial amount of time to complete. The ability to run tests in parallel is a key feature of `ctest` and is enabled using command line options that are very similar to the standard `make` tool. The `-j` option can be used to specify an upper limit on how many tests can be run simultaneously. Unlike most `make` implementations, a value must be supplied or the option will have no effect. As an alternative, the `CTEST_PARALLEL_LEVEL` environment variable can be used to specify the number of jobs, but the command line option takes precedence if both are used. This arrangement is particularly useful for continuous integration builds, since `CTEST_PARALLEL_LEVEL` can be set to the number of CPU cores on each machine, freeing every project from having to compute the optimal number of jobs themselves. For those projects that need to restrict the number of parallel jobs, they can still override `CTEST_PARALLEL_LEVEL` with the `-j` command line option.

A related option is `-l` which is used to specify a desirable upper limit on the CPU load. `ctest` will try to avoid starting a new test if it may cause the load to go above this limit. Unfortunately, the shortcomings of this option are immediately apparent at the start of testing. Typically, `ctest` will initially launch as many tests as the job limit from `-j` or `CTEST_PARALLEL_LEVEL` settings allow, exceeding any limit specified by `-l`. The measured CPU load usually has a lag, which allows `ctest` to start too many tests initially before the measured load increases. To prevent this occurring, the number of parallel jobs specified by `-j` or `CTEST_PARALLEL_LEVEL` should be set to no more than the limit imposed by `-l`. If neither `-j` nor `CTEST_PARALLEL_LEVEL` is set, the `-l` option will have no effect. Despite these limitations, the `-l` option can still be useful in helping to reduce CPU overload on shared systems where other processes may also be competing for CPU resources.

By default, `ctest` will assume each test consumes one CPU. For test cases that use more than one CPU, their `PROCESSORS` test property can be set to indicate how many CPUs they are expected to use. `ctest` will then use that value when determining whether enough CPU resources are free before starting the test. If `PROCESSORS` is set to a value higher than the job limit, `ctest` will behave as though it was set to the job limit when determining whether the test can be started.

The effect of these options can be seen in the following examples, which use the same set of tests as defined earlier (test summaries have been omitted for brevity).

```
ctest -j 5

Test project /path/to/build/dir
    Start 5: Other_Foo
    Start 2: BarOnly
    Start 3: FooWithBar
    Start 1: FooOnly
    Start 4: FooSpecial
1/5 Test #4: FooSpecial ..............   Passed    0.12 sec
2/5 Test #1: FooOnly .................   Passed    0.52 sec
3/5 Test #3: FooWithBar ..............   Passed    1.01 sec
4/5 Test #2: BarOnly .................   Passed    1.52 sec
5/5 Test #5: Other_Foo ...............   Passed    2.02 sec
```

Exclusively for sbmain17@gmail.com Transaction: 0098315591

Five tests were defined and the job limit was given on the command line as 5, so `ctest` was able to start all tests immediately. The result of each test was recorded as it completed, not in the order they were started. If the job limit is reduced to 2, the output may be more like the following:

```
ctest -j 2

Test project /path/to/build/dir
    Start 5: Other_Foo
    Start 2: BarOnly
1/5 Test #2: BarOnly .................   Passed    1.52 sec
    Start 3: FooWithBar
2/5 Test #5: Other_Foo ...............   Passed    2.01 sec
    Start 1: FooOnly
3/5 Test #1: FooOnly .................   Passed    0.52 sec
    Start 4: FooSpecial
4/5 Test #3: FooWithBar ..............   Passed    1.02 sec
5/5 Test #4: FooSpecial ..............   Passed    0.12 sec
```

With a large number of tests and a high job limit, the logging of each individual test start and completion can be difficult to follow. The overall test summary at the end of the run then becomes much more important, with each test that didn't pass listed along with its result. The `--progress` option to `ctest`, which was added in CMake 3.13, can also help reduce the output and focus on the important details. It collapses the start and completion progress messages down to a single line, similar to the output of the Ninja build tool.

25.5. Managing Test Resources

Tests sometimes need to ensure that no other test is running in parallel with them. They may be performing an action that is sensitive to other activities on the machine or they may create conditions that would interfere with other tests. To enforce this constraint, the test's `RUN_SERIAL` property can be set to true.

`RUN_SERIAL` is a brutal constraint which can have a strong impact on test throughput. The `RESOURCE_LOCK` test property is often a better alternative. It provides a list of resources the test needs exclusive access to. These resources are arbitrary strings which `ctest` does not interpret in any way, except to ensure no other test with any of those resources listed in its own `RESOURCE_LOCK` property will run at the same time. This is a great way to serialize tests that need exclusive access to something (e.g. a database, shared memory) without blocking tests that do not use that resource.

```
set_tests_properties(FooOnly FooSpecial Other_Foo PROPERTIES
    RESOURCE_LOCK Foo
)
set_tests_properties(BarOnly PROPERTIES
    RESOURCE_LOCK Bar
)
set_tests_properties(FooWithBar PROPERTIES
    RESOURCE_LOCK "Foo;Bar"
)
```


Exclusively for sbmain17@gmail.com Transaction: 0098315591

The following sample output (again with the test summary omitted) shows that even though the job limit of 5 would allow all tests to be executed simultaneously, `ctest` delays starting some tests until the resources they need are available.

```
ctest -j 5

Test project /path/to/build/dir
    Start 5: Other_Foo
    Start 2: BarOnly
1/5 Test #2: BarOnly .................   Passed    1.52 sec
2/5 Test #5: Other_Foo ...............   Passed    2.02 sec
    Start 3: FooWithBar
3/5 Test #3: FooWithBar ..............   Passed    1.01 sec
    Start 1: FooOnly
4/5 Test #1: FooOnly .................   Passed    0.52 sec
    Start 4: FooSpecial
5/5 Test #4: FooSpecial ..............   Passed    0.12 sec
```

The `RESOURCE_LOCK` test property is a good fit when a test needs exclusive access to something, but when more fine-grained control over test resources is needed, the `RESOURCE_GROUPS` test property available with CMake 3.16 or later may be more appropriate. Resource groups enable projects to define not just *what* resources a test needs, but also *how much* of each resource is required. This allows controlled sharing of resources and can be useful for a variety of interesting scenarios:

- Tests that require non-trivial amounts of memory can specify how much memory they need. The `ctest` scheduler will keep track of the memory resources it has allocated to all currently running tests and ensure that the available memory resources are not exceeded, delaying test execution until that test's memory requirements can be met.
- Multiple tests may need access to a GPU, but they could potentially share a GPU with other jobs as long as there are only a few of them sharing a given GPU. The end user's machine might also have one GPU or it might have multiple GPUs installed. The project only needs to define how many slots of a GPU each test needs and the `ctest` scheduler will take care of distributing the tests across however many GPUs are available to it at run time.
- There could be many tests all needing to communicate with a particular service. In order to prevent flooding that service, the number of tests communicating with it can be capped. For each such test, the project specifies that the test requires a resource representing that service. The user running the test suite controls the upper limit on how many tests are allowed to communicate with the service at the same time.

Compared to `RESOURCE_LOCK`, setting up resource groups is considerably more involved, requiring a number of separate steps:

- Define the resources a test must have in order to run. This must be done by the project.
- Define the resources available on the system. This can be done by the user running the tests or by the script used to invoke a CDash run (discussed in Section 25.9, "CDash Integration").
- Write tests to make use of resource details passed to it by `ctest` via environment variables.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

25.5.1. Defining Test Resource Requirements

A test defines its required resources as a list of resource *groups*. Each group consists of one or more `name:value` pairs, with multiple pairs separated by commas. The `name` part of the pair is referred to as the *resource type*. In the following example, the group specifies a requirement for 16 units of `mem_gb` and 4 `cpus`:

```
mem_gb:16,cpus:4
```

A group can also be preceded by a count of how many of that whole group are needed. It is given as an integer followed by a comma and then the group definition as described above. When this count value is omitted, as in the above example, it is assumed to be 1. The following demonstrates how to specify that the test requires 3 sets of resources where each set needs 2 `gpus` and 4 `workers`:

```
3,gpus:2,workers:4
```

A group can also list a particular resource type more than once. Consider the following example:

```
gpus:2,gpus:4
```

The above group needs a total of 6 `gpus`, but they can be split across two separate instances of the `gpus` resource type (instances are discussed in the next section). It is permitted for 2 `gpus` to come from one instance and 4 from the other. They could also be satisfied with 6 `gpus` all on the one instance if it has the slots available.

A test may need multiple sets of resource groups where the groups are not all the same. The `RESOURCE_GROUPS` property accepts a list for exactly this purpose. For example:

```
set_property(TEST ParallelCoordinator PROPERTY
    RESOURCE_GROUPS
        producers:1,consumers:1
        producers:1,consumers:4
        producers:4,consumers:1
        4,producers:1,consumers:1
)
```

The above specification results in the `ParallelCoordinator` test requiring a total of seven resource groups. The test won't be executed unless enough resources are available to satisfy all seven groups at once.

Being able to split the total resource requirements across multiple groups is essential for supporting certain system resource configurations. The next section discusses scenarios that take advantage of this capability.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

25.5.2. Specifying Available System Resources

In order for `ctest` to allocate resources to tests, it needs to be told what resources are available on the system. The system resources are specified in a JSON file, which is passed to `ctest` in one of the following ways (listed in order of precedence):

- Using the `RESOURCE_SPEC_FILE` keyword in a call to `ctest_test()` within a CDash script (see Section 25.9, "CDash Integration").
- Setting the `CTEST_RESOURCE_SPEC_FILE` variable either in a CDash script or as a `ctest -D` command line option when running a CDash script. This variable acts as a default value for the `RESOURCE_SPEC_FILE` keyword in calls to `ctest_test()`, but it is only supported with CMake 3.18 or later.
- Providing the `--resource-spec-file` command line option to `ctest`.
- Setting `CTEST_RESOURCE_SPEC_FILE` as a CMake variable (only supported with CMake 3.18 or later). To ensure the user always has control, this variable should only be set using a `cmake -D` command line option rather than hard-coding it directly in the project.

The format of this JSON file can be thought of as follows (a more formal description can be found in the CMake documentation of the `ctest` command):

```
{
  "version": {
    "major": 1,
    "minor": 0
  },
  "local": [
    {
      "resource1": [ ... ],
      "resource2": [ ... ],
      ...
    }
  ]
}
```

The `version` object must exist at the top level and it must contain both a `major` and a `minor` element. The first release of the resource allocation feature in CMake 3.16 requires `major` to be 1 and `minor` to be 0. Future releases may support other version combinations.

The other top level JSON element must be named `local` and it must be an array with exactly one element (future CMake releases may allow more). That array element is a JSON object that defines each of the resource types provided by the system on which the tests will run. The name of each resource type must be all lowercase and may contain numbers and underscores, but it must not start with a number. Each resource type is specified as an array of items of the following format:

Exclusively for sbmain17@gmail.com Transaction: 0098315591

```
{
  "id": "name",
  "slots": numericValue
}
```

The `id` is the name used to identify this particular instance of the resource. This name must be unique across all instances for this resource type and it must only contain lowercase letters, numbers or underscores. The name is not required to start with a letter or underscore and it can contain just a number if desired. The `slots` specifies the amount of the resource that this instance provides and it must be given as an integer.

Example resource spec file

```
{
  "version": {
    "major": 1,
    "minor": 0
  },
  "local": [
    {
      "mem_gb": [
        {
          "id": "pool_0",
          "slots": 64
        }
      ],
      "gpus": [
        {
          "id": "0",
          "slots": 2
        },
        {
          "id": "1",
          "slots": 2
        }
      ],
      "workers": [
        {
          "id": "0",
          "slots": 8
        },
        {
          "id": "1",
          "slots": 4
        }
      ]
    }
  ]
}
```

The above example demonstrates that values for the `id` only need to be unique within the same resource type. Both `gpus` and `workers` have ids of 0 and 1, but that is okay because they are different resource types.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

Because units cannot be specified with the value given to `slots`, it may be advisable to include units in the label for the resource type where the value's meaning requires some sort of unit. In the example above, the `mem_gb` resource type's name makes it clear that the slots are to be interpreted as gigabytes. In the case of `gpus` and `workers`, no units are needed, since it is already clear that the `slots` value is a count of those resources. For resource type names that don't require units, the convention is that names should generally be in plural form.

When `ctest` tries to satisfy the resource requirements of a test from the available pool of resources, it does not merge all the test's resource groups together. Rather, it iterates over the resource groups one by one and tries to satisfy each group individually. For each group, each `name:value` pair is assessed. `ctest` will look up the system allocations for the resource type and try to find an item in that resource type's array that has enough slots unallocated to satisfy that resource requirement. Importantly, `ctest` will not combine slots from multiple array elements to try to meet one `name:value` pair's resource requirements.

An example helps demonstrate the allocation logic. Consider the resource spec file given above and a resource group defined as `gpus:4`. The system has a combined total of 4 slots of the `gpus` resource type, but they are split across two separate items with ids 0 and 1. Because `ctest` is only allowed to satisfy a `name:value` resource requirement from a single element of a resource type's array, this requirement cannot be satisfied and the test will fail to run. Conversely, if a test had a resource group definition of `4,gpus:1`, it requires 4 separate groups where each group needs one `gpus` slot. This *can* be satisfied and two groups can even share one array item (e.g. two groups can share the resource with id 0 and the other two groups can share resource id 1). A group defined as `gpus:1,gpus:1` could be satisfied three different ways. It could receive both slots from id 0, both slots from id 1, or one slot from each.

25.5.3. Using Resources Allocated To A Test

The test receives information about the resources allocated to it through a number of environment variables. The most basic of these is `CTEST_RESOURCE_GROUP_COUNT`, which will holds the total number of resource groups the test specified. If this environment variable is not defined, it means that no resource spec file was provided when `ctest` was invoked. It is up to the test to then decide what to do in such cases. If the test cannot run without resource allocations being provided, the test should either fail or it should indicate that it has been skipped (e.g. by a return code that matches the `SKIP_RETURN_CODE` test property or by output that matches a `SKIP_REGULAR_EXPRESSION`.

For each resource group, there will be a set of environment variables with the pattern `CTEST_RESOURCE_GROUP_<num>` which contains a list of resource types allocated for that group. In order to see exactly which resources of a given resource type were allocated, another set of environment variables of the form `CTEST_RESOURCE_GROUP_<num>_<resourceType>` must be consulted. The `<resourceType>` will be the uppercased name of the resource type. The contents of these variables will be a list containing one or more items of the form `id:X,slots:Y`, which can be read as "Y slots from id X".

To illustrate, one example in the previous section specified a set of resource groups as `4,gpus:1`. This might lead to the test receiving a set of environment variables like the following:

Exclusively for sbmain17@gmail.com Transaction: 0098315591

```
CTEST_RESOURCE_GROUP_COUNT=4
CTEST_RESOURCE_GROUP_0=gpus
CTEST_RESOURCE_GROUP_1=gpus
CTEST_RESOURCE_GROUP_2=gpus
CTEST_RESOURCE_GROUP_3=gpus
CTEST_RESOURCE_GROUP_0_GPUS=id:0,slots:1
CTEST_RESOURCE_GROUP_1_GPUS=id:0,slots:1
CTEST_RESOURCE_GROUP_2_GPUS=id:1,slots:1
CTEST_RESOURCE_GROUP_3_GPUS=id:1,slots:1
```

For another example, consider a test that has one resource group, where that group is defined as `gpus:2,gpus:2,workers:4`. A possible set of environment variables it could receive would be:

```
CTEST_RESOURCE_GROUP_COUNT=1
CTEST_RESOURCE_GROUP_0=gpus,workers
CTEST_RESOURCE_GROUP_0_GPUS=id:0,slots:2;id:1,slots:2
CTEST_RESOURCE_GROUP_0_WORKERS=id:0,slots:4
```

Note how two list items are returned for `CTEST_RESOURCE_GROUP_0_GPUS` because the resource group listed `gpus:2` twice.

It is up to the test how it uses the information provided through the environment variables, but at the very least it should always confirm whether `CTEST_RESOURCE_GROUP_COUNT` is defined.

25.6. Test Dependencies

Tests can be used to do more than simply verify a particular condition, they can also be used to enforce them. For example, one test may need a server to connect to so that it can verify a client implementation. Rather than relying on the developer to ensure such a server is available, another test case can be created which ensures a server is running. The client test then needs to have some kind of dependency on the server test to make sure they are run in the correct order.

The `DEPENDS` test property allows a form of this constraint to be expressed by holding a list of other tests that must complete before that test can run. The above client/server example could loosely be expressed as follows:

```
set_tests_properties(ClientTest1 ClientTest2
    PROPERTIES DEPENDS StartServer
)
set_tests_properties(StopServer
    PROPERTIES DEPENDS "ClientTest1;ClientTest2"
)
```

A weakness with the `DEPENDS` test property is that while it defines a test order, it does not consider whether the pre-requisite tests pass or fail. In the above example, if the `StartServer` test case fails, the `ClientTest1`, `ClientTest2` and `StopServer` tests will still run. These tests will then likely fail and the test output will show all four tests as failed, where in reality only the `StartServer` test failed and the others should have been skipped.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

CMake 3.7 added support for test fixtures, a concept which allows dependencies between tests to be expressed much more rigorously. A test can indicate it requires a particular fixture by listing that fixture name in its `FIXTURES_REQUIRED` test property. Any other test with that same fixture name in its `FIXTURES_SETUP` test property must complete successfully before the dependent test will be started. If any of the setup tests for a fixture fail, all of the tests that require that fixture will be marked as skipped. Similarly, a test can list a fixture in its `FIXTURES_CLEANUP` test property to indicate that it must be run after any other test with that same fixture listed in its `FIXTURES_SETUP` or `FIXTURES_REQUIRED` property. These cleanup tests do not require the setup or fixture-requiring tests to pass, since cleanup may be needed even if the earlier tests fail.

All three fixture-related test properties accept a list of fixture names. These names are arbitrary and do not have to relate to the test names, resources they use or any other property. The fixture names should make clear to developers what they represent and so, while not required to, they often do have the same value as those used for `RESOURCE_LOCK` properties.

Consider the earlier client/server example. This can be expressed rigorously using fixtures with the following properties:

```
set_tests_properties(StartServer
    PROPERTIES FIXTURES_SETUP Server
)
set_tests_properties(ClientTest1 ClientTest2
    PROPERTIES FIXTURES_REQUIRED Server
)
set_tests_properties(StopServer
    PROPERTIES FIXTURES_CLEANUP Server
)
```

In the above, `Server` is the name of the fixture, `ClientTest1` and `ClientTest2` will only run if `StartServer` passes and `StopServer` will run last regardless of the result of any of the other three tests. If parallel execution is enabled, `StartServer` will run first, the two client tests will run simultaneously and `StopServer` will only run after both client tests have been completed or skipped.

Another benefit of fixtures can be seen when the developer is running only a subset of tests. Consider the scenario where the developer is working on `ClientTest2` and is not interested in running `ClientTest1`. When dependencies between tests are expressed using `DEPENDS`, the developer is responsible for ensuring they also include required tests in the test set, which means they need to understand all the relevant dependencies. This would lead to the `ctest` command line:

```
ctest -R "StartServer|ClientTest2|StopServer"
```

When fixtures are used, `ctest` automatically adds any setup or cleanup tests to the set of tests to be executed in order to satisfy fixture requirements. This means the developer need only specify the test they want to focus on and leave the dependencies to `ctest`:

```
ctest -R ClientTest2
```

Exclusively for sbmain17@gmail.com Transaction: 0098315591

When using the `--rerun-failed` option, this same mechanism ensures that setup and cleanup tests are automatically added to the test set to satisfy fixture dependencies of the previously failed tests.

A fixture may have zero or more setup tests and zero or more cleanup tests. Fixtures may define setup tests with no cleanup tests and vice versa. While not particularly useful, a fixture can have no setup or cleanup tests at all, in which case the fixture has no effect on the tests to be executed or when the tests will run. Similarly, a fixture can have setup and/or cleanup tests associated with it but no tests that require it. These situations can arise during development when tests are being defined or temporarily disabled. For the case of a fixture having no tests that require it, a bug in CMake 3.7 allowed that fixture's cleanup tests to run before the setup tests, but that bug was fixed in the 3.8.0 release.

A more involved example demonstrates how fixtures can be used to express more complex test dependencies. Expanding the previous example, suppose one client test requires just a server, whereas another requires both a server and a database to be available. This is succinctly expressed by defining two fixtures: `Server` and `Database`. For the latter, it is acceptable to simply check whether there is a database available and fail if not, so the `Database` fixture requires no cleanup test. The `Server` and `Database` fixtures are not related, so they need no dependencies between them. These constraints can be expressed like so:

```
# Setup/cleanup
set_tests_properties(StartServer
    PROPERTIES FIXTURES_SETUP Server
)
set_tests_properties(StopServer
    PROPERTIES FIXTURES_CLEANUP Server
)
set_tests_properties(EnsureDbAvailable
    PROPERTIES FIXTURES_SETUP Database
)

# Client tests
set_tests_properties(ClientNoDb
    PROPERTIES FIXTURES_REQUIRED Server
)
set_tests_properties(ClientWithDb
    PROPERTIES FIXTURES_REQUIRED "Server;Database"
)
```

While having `ctest` automatically add fixture dependencies into the test execution set is generally a useful feature, there are also times where this can be undesirable. Continuing with the above example, the developer may want to leave the server running and keep executing just one client test multiple times. They may be making changes, recompiling the code and checking whether the client test passes with each change. To support this level of control, CMake 3.9 introduced the `-FS`, `-FC` and `-FA` options to `ctest`, each of which requires a regular expression that will be matched against fixture names. The `-FS` option is used to disable adding fixture setup dependencies for those fixtures that match the regular expression provided. `-FC` does the same for cleanup tests and `-FA` combines both, disabling both setup and cleanup tests that match. A common situation is to disable adding any setup/cleanup dependencies at all, which can be done by giving a regular expression of a single period (.).

Exclusively for sbmain17@gmail.com Transaction: 0098315591

The following demonstrates various examples of the fixture control options and their effects:

Command line	Tests in execution set
`ctest -FS Server -R ClientNoDb`	`ClientNoDb`, `StopServer`
`ctest -FC Server -R ClientNoDb`	`ClientNoDb`, `StartServer`
`ctest -FA Server -R ClientNoDb`	`ClientNoDb`
`ctest -FS . -R Client`	`ClientNoDb`, `ClientWithDb`, `StopServer`
`ctest -FA . -R Client`	`ClientNoDb`, `ClientWithDb`

25.7. Cross-compiling And Emulators

When an executable target defined by the project is used as the command for `add_test()`, CMake automatically substitutes the location of the built executable. For a cross-compiling scenario, this won't typically work, since the host cannot usually run binaries built for a different platform directly.

To help with this, CMake provides a `CROSSCOMPILING_EMULATOR` target property which can be set to a script or executable to be used to launch the target. CMake will prepend this to the target binary when forming the command to run, so the real target binary becomes the first argument to the emulator script or executable. This enables tests to be run even when cross-compiling. With CMake 3.15 or later, `CROSSCOMPILING_EMULATOR` can be a list to allow arguments to be included in the items inserted before the target binary.

The `CROSSCOMPILING_EMULATOR` doesn't have to be an actual emulator, it just has to be a command that can be run on the host to launch the target executable. While a dedicated emulator for the target platform is the obvious use case, one could also set it to a script that copies the executable to a target machine and runs it remotely (e.g. over a SSH connection). Whichever method is used, developers should be aware that the startup time for an emulator or for preparing to run the binary could be non-trivial and may have an impact on the test timing measurements. This can, in turn, mean that test timeout settings may need to be revised.

The default value for the `CROSSCOMPILING_EMULATOR` target property is taken from the `CMAKE_CROSSCOMPILING_EMULATOR` variable, which is the usual way the emulator details would be specified rather than setting each target's property individually. The variable would typically be set in the toolchain file, since it affects things like `try_run()` commands in a similar way to how it affects tests and custom commands as described above. See the discussion in Section 22.5, "Compiler Checks" for more on this aspect of the variable's effects.

Even when not cross-compiling, CMake will still honor a non-empty `CROSSCOMPILING_EMULATOR` target property and prepend it to the command line for tests and custom commands executing that target. This can be quite useful, allowing the property to be temporarily set to a launch script to assist with things like debugging or for data-gathering. It is not recommended to use this technique as a permanent feature of a project's build, but it may be useful in certain development situations.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

25.8. Build And Test Mode

`ctest` can not only execute a set of tests, it can drive an entire configure, build and test pipeline. There are two main methods for doing this; a more basic, standalone way and a more powerful approach closely associated with a dashboard reporting tool. The more basic approach is to invoke the `ctest` tool with the `--build-and-test` command line option, which uses the following form:

```
ctest --build-and-test sourceDir buildDir
      --build-generator generator
      [options...]
      [--test-command testCommand [args...]]
```

Without any `options`, the above will run CMake with the specified `sourceDir` and `binaryDir` and use the specified `generator`. All three of these must be specified. If the CMake run was successful, `ctest` will then build the `clean` target and lastly it will build the default `all` target. To run tests as well after the build step, the last option on the command line must be `--test-command` with its associated `testCommand` and optionally some arguments. This can be another invocation of `ctest` to run all tests, as demonstrated in the following example.

```
ctest --build-and-test sourceDir buildDir  \
      --build-generator Ninja              \
      --test-command ctest -j 4
```

The above carries out a full configure-clean-build-test pipeline. Various options are provided which can be used to modify which parts of the pipeline are run and how they are run. For example, `--build-nocmake` and `--build-noclean` disable the configure and clean steps respectively. The `--build-two-config` option will invoke CMake twice, which handles certain special cases where a second CMake pass is needed to fully configure a project. When using a generator like Visual Studio, it may be necessary to specify extra generator details with `--build-generator-platform` and `--build-generator-toolset`, which will be passed through as the `-A` and `-T` options respectively to `cmake` for the configure step. Some generators like Xcode may require the project name to be given so it can find the project file generated by the configure stage, which can be done with the `--build-project` option. The target to build in the build step can be set using the `--build-target` option and the build tool can be overridden by passing `--build-makeprogram` with the alternative tool.

As can be seen in the above, all of the options related to the `--build-and-test` mode begin with `--build`. While most options have intuitive names, the common `--build` prefix can lead to some unfortunate confusing anomalies. An option with the name `--build-options` exists which may initially seem to be related to the build step, but is actually used to pass command line options to the `cmake` command. It also has the additional constraint that it must be last on the command line, unless `--test-command` is also given, in which case `--build-options` must precede `--test-command`.

The following example should clarify these constraints. It adds two cache variable definitions to the `cmake` invocation and also runs the full test suite after the build step.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

```
ctest --build-and-test sourceDir buildDir                              \
      --build-generator Ninja                                          \
      --build-options -DCMAKE_BUILD_TYPE=Debug -DBUILD_SHARED_LIBS=ON \
      --test-command ctest -j 4
```

There are a few other `--build-⋯` options, but the above covers the most useful ones. The other remaining option that should be mentioned is `--test-timeout`, which places a time limit (in seconds) on how long the test command is allowed to run before it is forced to terminate.

It is situation-dependent whether controlling the whole pipeline using a single `ctest` command is better or worse than invoking each of the tools needed for each stage explicitly. The last example above could just as easily be done with the following equivalent sequence of commands on Unix:

```
mkdir -p buildDir
cd buildDir
cmake -G Ninja -DCMAKE_BUILD_TYPE=Debug -DBUILD_SHARED_LIBS=ON sourceDir
cmake --build . --target clean
cmake --build .
ctest -j 4
```

Invoking each tool individually allows them to be run with the full set of options, whereas the `ctest --build-and-test` approach has only a very limited ability to control the build stage.

One situation where build and test mode is particularly convenient is where a project needs to perform a complete configure-build-test cycle off to the side, separate from the main build. Since the whole cycle can be controlled by a single `ctest` invocation, it can be used as the `COMMAND` part of a call to `add_test()`, making the process of adding a basic CMake project to the main project's test suite relatively straightforward. CMake itself uses the `ctest` build and test mode extensively in its own test suite in exactly this manner.

The following example shows how a separate build can be used to test the API provided by a library built by the main project:

```
add_library(Decoder foo.c bar.c)

add_test(NAME Decoder.api
    COMMAND ${CMAKE_CTEST_COMMAND}
            --build-and-test  ${CMAKE_CURRENT_LIST_DIR}/test_api
                              ${CMAKE_CURRENT_BINARY_DIR}/test_api
            --build-generator ${CMAKE_GENERATOR}
            --build-options   -DDECODER_LIB=$<TARGET_FILE:Decoder>
            --test-command    ${CMAKE_CTEST_COMMAND}
)
```

The `test_api` source directory would contain its own `CMakeLists.txt` file whose sole purpose is to configure a build that links against the Decoder library, the absolute path to which is set in the `DECODER_LIB` variable (this is just one of a few ways to pass the library location to the test project).

Exclusively for sbmain17@gmail.com Transaction: 0098315591

An interesting thing about this sort of test is that it can also be used to verify that a particular test project does *not* build or to verify that configuring fails with a particular fatal error (e.g. a missing symbol). Such expected fatal build errors cannot be tested in the main project, since it would cause the main project's build to fail. Separating the test build off to the side means it can fail without affecting the main build and the test can verify the failure with an appropriate `FAIL_REGULAR_EXPRESSION` or a non-zero return code.

Another scenario where such tests can be helpful is to test the output of a code generator created by the main project. Test fixtures can be used to set up a pair of tests, one to generate the code and the other to perform a test build with it. This is particularly helpful if the code generator creates files that `cmake` would normally read, such as `CMakeLists.txt` files. For example:

```
add_executable(CodeGen generator.cpp)

add_test(NAME GenerateCode COMMAND CodeGen)
add_test(NAME BuildGeneratedCode
    COMMAND ${CMAKE_CTEST_COMMAND}
            --build-and-test  ${CMAKE_CURRENT_LIST_DIR}/test_gen
                              ${CMAKE_CURRENT_BINARY_DIR}/test_gen
            --build-generator ${CMAKE_GENERATOR}
            --test-command    ${CMAKE_CTEST_COMMAND}
)

set_tests_properties(GenerateCode
    PROPERTIES FIXTURES_SETUP Generator
)
set_tests_properties(BuildGeneratedCode
    PROPERTIES FIXTURES_REQUIRED Generator
)
```

Build and test mode could also be used to verify CMake utility scripts by including them in a small test project and invoking its functionality as appropriate. In effect, this provides a fairly convenient way to implement unit testing of CMake scripts that avoids having to put such tests into the configure stage of the main project.

While build and test mode is certainly useful for cases like those mentioned above, it lacks the flexibility of a fully scripted run where the full set of options are available for each individual command. The next section introduces an alternative way of invoking `ctest` which offers more powerful handling of the entire pipeline, including some useful additional reporting capabilities.

25.9. CDash Integration

CTest has a long history and close relationship with another product called CDash, which is also developed by the same company behind CMake and CTest. CDash is a web-based dashboard which collects results from a software build and test pipeline driven by `ctest`. It collects warnings and errors from each stage of the pipeline and shows per-stage summaries with the ability to click through to each individual warning or error. A history of past pipelines allows trends to be observed over time and to compare runs.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

CMake itself has its own fairly extensive dashboard which tracks nightly builds, builds associated with merge requests and so on. A few minutes exploring a sample dashboard will be helpful in understanding the material covered in this section:

https://open.cdash.org/index.php?project=CMake

25.9.1. Key CDash Concepts

Three important concepts tie together how CTest and CDash execute pipelines and report results: *steps* (sometimes also referred to as *actions*), *models* (also sometimes called *modes*) and *groups* (previously called *tracks* with CMake 3.15 and earlier). Steps are the sequence of actions that a pipeline performs. The main set of defined actions in the order they would normally be invoked is:

- Start
- Update
- Configure
- Build
- Test
- Coverage
- MemCheck
- Submit

Not all actions have to be executed, some may not be supported or do not need to be run. Loosely speaking, each row in the CDash dashboard corresponds to a single pipeline and will typically show a summary of each action taken (a commit hash, a total of warnings, errors, failures, etc.).

Each pipeline must be associated with a model, which is used to define certain behaviors, such as whether or not to continue with later steps after a particular step fails. The model also provides a default set of actions when no specific action is requested. The supported models are:

Nightly

Intended to be invoked once per day, usually by an automated job during a time when the executing machine is less busy. The default set of actions includes all the steps listed above except *MemCheck*. If the *Update* step fails, the rest of the steps will still be executed.

Continuous

Very similar to *Nightly* except that it is intended to be run multiple times a day as needed, usually in response to a change being committed. It defines the same set of default actions as *Nightly*, but if the *Update* step fails, the later steps will not be executed.

Experimental

As the name suggests, this model is intended for ad hoc experiments executed by developers as needed. Its default set of actions includes all steps except *Update* and *MemCheck*. If a model other than one of the three defined models is specified or if no model is specified at all, it will be treated as *Experimental*.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

The *group* controls which group the pipeline results will be shown under in the dashboard results. Group names can be anything the project or developer wishes to use, but if no group is specified, it will be set to the same as the model. This has led to a common misunderstanding that the model controls the grouping in the dashboard, but it is the group that does this. The *Coverage* and *MemCheck* actions are a special case, they effectively ignore the group and their dashboard results are shown in their own dedicated groups (*Coverage* and *Dynamic Analysis* respectively). With CMake 3.22 or later, in addition to populating the dedicated *Dynamic Analysis* group, the *MemCheck* action will also submit test results in the same way as the *Test* action, for which the nominated group *will* be used.

25.9.2. Executing Pipelines And Actions

For a project with the necessary configuration files in place (covered in the next section), entire pipelines or individual steps can be invoked using the following form of the `ctest` command:

```
ctest [-M Model] [-T Action] [--group Group] [otherOptions...]
```

If using CMake 3.16 or earlier, `--track Track` must be used rather than `--group Group`.

At least one or both of the *Model* and *Action* must be specified. As a convenience, the `-M` and `-T` options can be combined into a single `-D` option like so:

```
ctest -D Model[Action] [--group Group] [otherOptions...]
```

Arguments to `-D` can omit the action or append it to the *Model*. Examples of valid arguments include *Continuous*, *NightlyConfigure*, *ExperimentalBuild* and so on. The `-T` and `-D` options can be specified multiple times to list multiple steps in the one `ctest` invocation if desired.

> Note that `-D` is also used to define `ctest` variables and the `ctest` command will treat any *Model* or *ModelAction* it doesn't recognize as setting a variable instead. It is advisable to use the `-M` and `-T` options rather than `-D` to minimize opportunities for a mistyped command line option to be misinterpreted.

A nightly run using the default set of steps and reporting its results under the default group *Nightly* is trivially invoked as:

```
ctest -M Nightly
```

For the same thing but with results reported under a group called *Nightly Master*:

```
ctest -M Nightly --group "Nightly Master"
```

Consider a custom *Experimental* pipeline consisting of just *Configure*, *Build* and *Test* steps with results grouped under *Simple Tests*. This requires the set of steps to be explicitly specified, since it differs from the default set of actions defined for an *Experimental* model (no *Coverage* step is being

Exclusively for sbmain17@gmail.com Transaction: 0098315591

executed). This can be done as either a sequence of `ctest` invocations with one step per invocation, or they could all be listed together using multiple `-T` options on the one command line. Both forms are shown for comparison:

Separate commands

```
ctest -T Start -M Experimental --group "Simple Tests"
ctest -T Configure
ctest -T Build
ctest -T Test
ctest -T Submit
```

One command

```
ctest -M Experimental --group "Simple Tests" \
      -T Start -T Configure -T Build -T Test -T Submit
```

The first step should be a *Start* action, which is used to initialize the pipeline details and to record the model and group names that later steps will use. These details do not need to be repeated for any of the later steps if splitting each action out to its own separate `ctest` invocation. The last step would be a *Submit* action, assuming the goal is to submit the final set of results to a dashboard.

All output from the above is collected under a `Testing` subdirectory below the directory in which `ctest` is invoked. The *Start* action writes out a file named `TAG` which contains at least two lines, the first being a date-time for the start of the run in the form `YYYYMMDD-hhmm` and the second being the group name. CMake 3.12 adds a third line containing the model name.

As each step after the *Start* action is executed, it will create its own output file at `Testing/YYYYMMDD-hhmm/<Action>.xml` and a log file at `Testing/Temporary/Last<Action>_YYYYMMDD-hhmm.log`. In the case of the *MemCheck* step, the `<Action>` part will be `DynamicAnalysis` rather than `MemCheck` in these file names (with CMake 3.22 or later, a second `Testing/YYYYMMDD-hhmm/DynamicAnalysis-Test.xml` file will also be created). The *Submit* action collects the XML output files and some of the log files and submits them to the nominated dashboard.

To attach a build note to the whole pipeline, use the `-A` or `--add-notes` option with the *Submit* step to specify the file names to upload, separated by semi-colons if multiple files are being added. This can be a useful way to record extra details about that particular pipeline, such as information from a continuous integration system that initiated the run.

```
ctest -T Submit --add-note JobNote.txt
```

An `--extra-submit` option is also supported, but it is intended more for internal use by `ctest`. It is not a general file upload mechanism, but is often mistakenly assumed to serve that purpose. It should not be used by developers or projects directly.

While the above functionality is intended primarily for integration with CDash, it can also be used for other scenarios too. For example, the Jenkins CI system has a plugin that allows it to read the Test action's `Test.xml` output file and record test results in a similar way to CDash. Instead of running `ctest` in the ordinary way, it can be invoked as a dashboard run with just the *Test* action.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

The Jenkins plugin then only needs to be told where to find the `Test.xml` file and it can read the test results. When used this way, even the *Start* action can be omitted. `ctest` will silently perform the equivalent of a *Start* action with an *Experimental* model if one of the other steps is executed without any prior *Start* action. Projects may want to clear any previous contents of the `Testing` directory before doing so to ensure only the results of the current run are picked up by Jenkins.

When passing the XML output file of an action to a tool other than CDash, it may be necessary to instruct `ctest` to not compress the output it captures. By default, the action's output is compressed and written to the XML file in an ASCII-encoded form, but this can be be prevented by passing the `--no-compress-output` option to `ctest`. Only use this option if it is necessary, since it will result in larger output files.

Another situation where dashboard steps can be useful without CDash is to take advantage of the support for code coverage or memory checking (Valgrind, Purify, various sanitizers, etc.). These dashboard actions can make invoking the relevant tool and collecting results easier. See the next section for details on how to setup and use these tools.

25.9.3. CTest Configuration

Preparing a project for CDash integration is mostly handled by a `CTest` module provided by CMake. It should be included soon after the `project()` command in the top level `CMakeLists.txt` file. This is important because the module writes various files into the current build directory at the point it is included, and developers typically expect to be able to run `ctest` from the top level build directory.

```
cmake_minimum_required(VERSION 3.0)
project(CDashExample)

# ... set any variables to customize CTest behavior

include(CTest)

# ... Define targets and tests as usual
```

The `CTest` module defines a `BUILD_TESTING` cache variable which defaults to true. It is used to decide whether the module calls `enable_testing()` or not, so the project does not have to make its own call to `enable_testing()` as well. This cache variable can also be used by the project to perform certain processing only if testing is enabled. If the project has many tests that take a long time to build, this can be a useful way to avoid adding them to the build when they are not needed.

```
cmake_minimum_required(VERSION 3.0)
project(CDashExample)

include(CTest)

# ... define regular targets

if(BUILD_TESTING)
    # ... define test targets and add tests
endif()
```


Exclusively for sbmain17@gmail.com Transaction: 0098315591

The `CTest` module defines build targets for each *Model* and for each *ModelAction* combination. These targets execute `ctest` with the `-D` option set to the target name and are intended as a convenient way to execute the whole pipeline or just one dashboard action from within an IDE application. The targets don't offer any real advantage over invoking `ctest` directly if working from the command line.

The more important task performed by the `CTest` module is to write out a configuration file called `DartConfiguration.tcl` in the build directory. The name of this file is historical, with Dart being the original name of the CDash project. This file records basic details like the source and build directory locations, information about the machine on which the build is being performed, the toolchain used, the location of various tools and other defaults. It will also contain the details of the CDash server, but in order for it to do so, the project needs to provide a `CTestConfig.cmake` file at the top of the source tree with the relevant contents. A suitable `CTestConfig.cmake` file can be obtained from CDash itself (requires administrator privileges), but it is usually not difficult to create one manually. A minimal example which works for all CMake versions would look something like this:

```
# Name used by CDash to refer to the project
set(CTEST_PROJECT_NAME "MyProject")

# Time to use for the start of each day. Used by CDash to group results by day,
# usually set to midnight in the local timezone of the CDash server.
set(CTEST_NIGHTLY_START_TIME "01:00:00 UTC")

# Details of the CDash server to submit to
set(CTEST_DROP_METHOD     "http")
set(CTEST_DROP_SITE       "my.cdash.org")
set(CTEST_DROP_LOCATION   "/submit.php?project=${CTEST_PROJECT_NAME}")
set(CTEST_DROP_SITE_CDASH YES)

# Optional, but recommended so that command lines can be seen in the CDash logs
set(CTEST_USE_LAUNCHERS YES)
```

From CMake 3.14, the various `CTEST_DROP_⋯` options can be replaced by a single `CTEST_SUBMIT_URL` option. This is much simpler and more readable, so if the minimum CMake version of the project is at least 3.14, this should be preferred. The equivalent for the above example would be:

```
set(CTEST_SUBMIT_URL "http://my.cdash.org/submit.php?project=${CTEST_PROJECT_NAME}")
```

The `DartConfiguration.tcl` file written out by the `CTest` module contains options for dashboard actions. Most are set to appropriate values by default, but the *Coverage* and *MemCheck* steps have options that may be of interest. These are controlled by CMake variables which the developer can manipulate in the CMake cache or in the `CMakeLists.txt` file before the `CTest` module is included.

The *Coverage* step is assumed to be invoking `gcov` and the `CTest` module will search for a command by that name. The `COVERAGE_COMMAND` cache variable holds the result of that search, but it can be modified by the developer if needed. A second cache variable `COVERAGE_EXTRA_FLAGS` is used to hold the options that should immediately follow the `COVERAGE_COMMAND`, so the developer has the ability to control both the command used and the options passed to it.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

The *MemCheck* step is more interesting. A number of different memory checkers are supported, including Valgrind, Purify, BoundsChecker, Dr Memory (with CMake 3.17 or later), Cuda Sanitizer (with CMake 3.19 or later) and various other sanitizers. For the first five, they can be selected by setting `MEMORYCHECK_COMMAND` to the location of the relevant executable. `ctest` will then identify the checker from the executable name. For Valgrind, the `VALGRIND_COMMAND_OPTIONS` variable can also be set to override the options given to `valgrind` itself. Dr Memory has a similar capability with the `DRMEMORY_COMMAND_OPTIONS` variable. To use one of the sanitizers, set `MEMORYCHECK_TYPE` to one of the following strings (`MEMORYCHECK_COMMAND` will then be ignored):

- `AddressSanitizer`
- `LeakSanitizer`
- `MemorySanitizer`
- `ThreadSanitizer`
- `UndefinedBehaviorSanitizer`

`ctest` will then launch test executables as normal but with the relevant environment variables set to enable the requested sanitizer. Note that sanitizers require building with the relevant compiler and linker flags (typically `-fsanitize=XXX` and perhaps `-fno-omit-frame-pointer`). For further details on the relevant flags and what the various sanitizers do, consult the Clang or GCC documentation.

The above details are enough to be able to perform various dashboard actions and submit results to a CDash server, but there is a chicken-and-egg problem. The *Update* and *Configure* steps need to have already been performed to obtain the `DartConfiguration.tcl` file. Therefore, details of those two steps cannot be captured, or in the case of the *Configure* step, the output from the first `cmake` run are lost and one can only get the output from re-running CMake in an already-configured build directory. Nevertheless, all the other steps will have their output captured and that may be enough in some situations.

For example, when using a continuous integration system like Gitlab CI or Jenkins, the initial clone or update of the source tree can be handled by the CI system itself. An initial `cmake` run can be performed and then the rest of the steps can be run as dashboard actions. The final results can be submitted to a CDash server, read directly by the CI system or possibly both.

To capture a complete pipeline, including the initial clone or update of a source tree and first configure step, one has to write a custom `ctest` script to define the required setup details and call the relevant `ctest` functions. This can be an involved process and isn't typically necessary if already using another CI system. If the clone/update step doesn't need to be captured, then the complexity of the custom script is reduced. When used this way, `ctest` is invoked with the `-S` option and the name of the script to execute.

The following demonstrates a fairly straightforward example:

```
ctest -S MyCustomCTestJob.cmake
```


Exclusively for sbmain17@gmail.com Transaction: 0098315591

MyCustomCTestJob.cmake

```
# Re-use CDash server details we already have
include(${CTEST_SCRIPT_DIRECTORY}/CTestConfig.cmake)

# Basic information every run should set, values here are just examples
site_name(CTEST_SITE)
set(CTEST_BUILD_NAME         ${CMAKE_HOST_SYSTEM_NAME})
set(CTEST_SOURCE_DIRECTORY   "${CTEST_SCRIPT_DIRECTORY}")
set(CTEST_BINARY_DIRECTORY   "${CTEST_SCRIPT_DIRECTORY}/build")
set(CTEST_CMAKE_GENERATOR    Ninja)
set(CTEST_CONFIGURATION_TYPE RelWithDebInfo)

# Dashboard actions to execute, always clearing the build directory first
ctest_empty_binary_directory(${CTEST_BINARY_DIRECTORY})
ctest_start(Experimental)
ctest_configure()
ctest_build()
ctest_test()
ctest_submit()
```

The following more interesting example shows how custom scripts allow more flexible pipeline behavior to be defined:

```
include(${CTEST_SCRIPT_DIRECTORY}/CTestConfig.cmake)

site_name(CTEST_SITE)
set(CTEST_BUILD_NAME         "${CMAKE_HOST_SYSTEM_NAME}-ASan")
set(CTEST_SOURCE_DIRECTORY   "${CTEST_SCRIPT_DIRECTORY}")
set(CTEST_BINARY_DIRECTORY   "${CTEST_SCRIPT_DIRECTORY}/build")
set(CTEST_CMAKE_GENERATOR    Ninja)
set(CTEST_CONFIGURATION_TYPE RelWithDebInfo)
set(CTEST_MEMORYCHECK_TYPE   AddressSanitizer)
set(configureOpts
    "-DCMAKE_CXX_FLAGS_INIT=-fsanitize=address -fno-omit-frame-pointer"
    "-DCMAKE_EXE_LINKER_FLAGS_INIT=-fsanitize=address -fno-omit-frame-pointer"
)
ctest_empty_binary_directory(${CTEST_BINARY_DIRECTORY})
ctest_start(Experimental GROUP Sanitizers)
ctest_configure(OPTIONS "${configureOpts}")
ctest_submit(PARTS Start Configure)
ctest_build()
ctest_submit(PARTS Build)
ctest_memcheck()
ctest_submit(PARTS MemCheck)
ctest_upload(FILES
    ${CTEST_BINARY_DIRECTORY}/mytest.log
    ${CTEST_BINARY_DIRECTORY}/anotherFile.txt
)
ctest_submit(PARTS Upload Submit)
if(NOT CMAKE_VERSION VERSION_LESS "3.14")
    ctest_submit(PARTS Done)
endif()
```

Exclusively for sbmain17@gmail.com Transaction: 0098315591

In this more production-quality example, rather than waiting to the very end of the run before submitting results to the dashboard, results are submitted progressively after each step (useful if some steps take a long time). The executables are built with address sanitizer support and the address sanitizer check is run instead of regular testing. Some extra files are also uploaded at the end.

The *Done* part was only added in CMake 3.14 and is used to tell CDash that the job is complete. It is used by recent versions of CDash to enable them to report a more reliable total duration for the job. Earlier CDash versions will simply ignore it. When not submitting by parts, the *Done* part is handled automatically as part of the *Submit* action or a call to `ctest_submit()` with no parts specified.

Each of the various `ctest_⋯` commands is detailed in the CMake documentation, along with `CTest` and CMake variables that can be used to customize each step or affect the processing in various ways. The above should be a good base script that can be used to experiment with the different parameters and variables.

Creating a script that also handles cloning/updating the project adds more complexity. Projects often have their own special ways of doing this and they typically need to decide how things like *Nightly* and *Continuous* builds should be scheduled. Supporting things like automated builds for merge requests will depend heavily on the capabilities of the repository hosting the project. For those interested in exploring this path, a recommended way to get started is to find a project using a similar repository hosting arrangement and use it as a guide. Some projects include the custom script in their repository for ease of access (many projects from Kitware do this and the scripts have been documented reasonably well).

25.9.4. Test Measurements And Results

The above example briefly showed how file uploads can be incorporated into a custom CTest script. The `ctest_upload()` command provides a basic mechanism for recording files to upload with the build results. The upload is executed as part of a subsequent call to `ctest_submit()`. Sometimes, however, file uploads should be associated with a particular test rather than the whole scripted run. For this, CMake provides the `ATTACHED_FILES` and `ATTACHED_FILES_ON_FAIL` test properties. Both hold a list of files to be uploaded and associated with that particular test, the only difference is that the latter contains files that are only uploaded if the test fails. This is a very useful way to record additional information about the failure to allow further investigation.

```
add_executable(CodeGen ...)
add_test(NAME GenerateFile COMMAND CodeGen)

set_tests_properties(GenerateFile PROPERTIES
    ATTACHED_FILES_ON_FAIL
        ${CMAKE_CURRENT_BINARY_DIR}/generated.c
        ${CMAKE_CURRENT_BINARY_DIR}/generated.h
)
```

Tests can also record a single measurement value which will be recorded and tracked in CDash. A measurement generally has the form `key=value`, although the `=value` part can be omitted to use an assumed default value of 1. The measurement is recorded as a test property like so:

Exclusively for sbmain17@gmail.com Transaction: 0098315591

```
set_tests_properties(PerfRun PROPERTIES
    MEASUREMENT mySpeed=${someValue}
)
```

Because the measurement value has to be defined before the test is even run, this has limited usefulness. Much more useful is a feature which has long been supported, but only been documented since CMake 3.21, where measurements can be embedded in the test output in a form similar to HTML tags. `ctest` scans the test output for these measurements, extracts the relevant data and uploads it to CDash as part of the test results. These measurements are then displayed in a result table near the top of the test details page. The simplest type of measurement is defined by the following form:

```
<DartMeasurement name="key" type="someType">value</DartMeasurement>
```

The `name` attribute will be used as the label for the measurement in the results table and the `type` attribute will typically be something like `text/string`, `text/link` (for URLs) or `numeric/double`. The `value` is whatever text or numerical content makes sense for the measurement. For numerical values, CDash provides a facility to plot the history of each measurement across recent test runs, which is very useful for spotting changes in behavior over time.

Another form can be used to embed a file rather than a specific value:

```
<DartMeasurementFile name="key" type="someType">filePath</DartMeasurementFile>
```

This second form is most useful for uploading images, where the `type` attribute would be something like `image/png` or `image/jpeg`. The `filePath` should be the absolute path to the file to be uploaded.

CDash recognizes a few special measurement names when it comes to images. These can be used to help compare expected and actual images, with CDash even providing a useful interactive UI element for overlapped comparisons. The recognized `name` attributes and their meanings include:

`TestImage`

This is interpreted as the image generated by the test. It can be thought of as the test output and will be shown both on its own and also as part of the interactive comparison image.

`ValidImage`

This is equivalent to the expected image for the test. It should generally be of the same dimensions as the `TestImage`, but is not necessarily required to be of the same image format. It will be included in the interactive image only. `BaselineImage` can also be used as the `name` and it means the same thing as `ValidImage`.

`DifferenceImage2`

Various tools can be employed to generate an image that represents the difference between two other images. Where the test provides such an image file, it can use this name to include it in the test output measurements uploaded to CDash. It will be incorporated into the interactive comparison image.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

Names other than the above and types other than images can be uploaded, but bugs in CMake 3.20 and earlier may make that unreliable. For best results, use 3.21 or later if possible. CMake 3.21 also provides better handling for non-image files when the `type` is set to `file`. The file will then be uploaded as an attachment, just like the `ATTACHED_FILES` test property would. With any other `type`, it will be treated as a named measurement instead and may not display as appropriately in CDash.

CMake 3.21 also added the ability for the test output to override the test's details field in CDash. The default contents normally state whether the test completed and what the test result was, but the test can use this facility to provide more specific information. It should still be relatively short though, ideally no more than one line.

```
<CTestDetails>Replacement test details go here</CTestDetails>
```

Starting with CMake 3.22, a test can also dynamically add labels at runtime by including the relevant tags in its output like so:

```
<CTestLabel>Some dynamic label</CTestLabel>
<CTestLabel>Another label</CTestLabel>
```

These labels are uploaded to CDash along with the test results, just like regular labels assigned to the test at configure time with the `LABELS` test property (see Section 25.3.3, "Labels"). These dynamic labels are also included in the results summary at the end of the `ctest` run.

Because dynamic labels are only assigned when the test is executed, they cannot be used to include or exclude tests to be run. If the user specifies `ctest` options like `-L` or `-LE` to control which tests to execute, only the statically assigned labels set through the `LABELS` test property are considered. If a test adds a label dynamically to its output and that same label is set statically on another test, it may be a source of confusion for users if they then try to filter the test set based on that label. Tests they thought they had filtered out might still show up in the output, or those they thought were included might be missing. Consider using non-overlapping sets of labels for static and dynamic cases to avoid setting up this situation.

25.10. Output Control

While CDash can be very effective for collecting and tracking test results, many projects use other tools for such purposes. To support these workflows, CMake 3.21 added the ability to provide test results in the widely supported JUnit XML format. JUnit XML files are often able to be imported into CI tools and various test reporting software.

For non-dashboard runs, a JUnit result file can be generated by adding an option to the `ctest` command line, specifying the file name to write the results to:

```
ctest --output-junit /path/to/resultFile.xml ...
```


Exclusively for sbmain17@gmail.com Transaction: 0098315591

For dashboard runs, an additional argument to the `ctest_test()` command achieves a similar thing:

```
ctest_test(OUTPUT_JUNIT /path/to/resultFile.xml)
```

Features like named measurements and file attachments are not supported by the JUnit output. It therefore isn't a complete replacement for what can be accomplished with CDash, but the reduced feature set may still be good enough for many projects.

Tests can sometimes produce a lot of output. A dashboard run will truncate the reported output beyond a configurable limit, both in the default CDash results and in JUnit XML file output (where enabled). By default, the output from any test that passes will be truncated at 1024 bytes. For failing tests, the output will be truncated at 307,200 bytes (i.e. 300kB). These limits can be overridden by setting variables in the dashboard script. `CTEST_CUSTOM_MAXIMUM_PASSED_TEST_OUTPUT_SIZE` and `CTEST_CUSTOM_MAXIMUM_FAILED_TEST_OUTPUT_SIZE` can be used to specify the number of bytes to truncate at instead of the default values. These variables have long been supported by CMake, but have only been officially documented since CMake 3.4.

With CMake 3.21 or later, individual tests can override the output limits by logging the special string `CTEST_FULL_OUTPUT` somewhere in their output. The entire output from that test will then be included in the test results, regardless of any output limit. Use this feature sparingly, as it has the potential to significantly increase the amount of data reported and uploaded.

25.11. GoogleTest

CMake/`ctest` provide support for building, executing and determining pass/fail status of tests. The project is responsible for providing the test code itself and this is where testing frameworks like GoogleTest can be useful. Such frameworks complement the features provided by CMake and `ctest` to facilitate the writing of clear, well-structured test cases that integrate well into the way CMake and `ctest` work.

CMake has supported GoogleTest via a `FindGTest` module for quite a long time. The module searches for a pre-built GoogleTest location and creates variables that projects can use to incorporate GoogleTest into their build. From CMake 3.5, import targets are also provided, which are strongly preferred over the use of variables. Using these import targets results in much more robust handling of usage requirements and properties. A simple example of how to use the module with CMake 3.5 or later would be similar to the following:

```
add_executable(MyGTestCases ...)

find_package(GTest REQUIRED)
target_link_libraries(MyGTestCases PRIVATE GTest::GTest)

add_test(NAME MyGTestCases COMMAND MyGTestCases)
```

The import target takes care of ensuring the relevant header search path is used when building `MyGTestCases` and that things like the appropriate threading library is linked in if needed. The above works on all platforms, hiding a fair amount of complexity associated with different names,

Exclusively for sbmain17@gmail.com Transaction: 0098315591

runtimes, flags, etc. that are used on the different platforms and compilers. If using the variables defined by the module instead of the import targets, these things mostly have to be handled manually, which is a fairly fragile task.

An even more robust approach is to incorporate GoogleTest's sources directly into the build rather than relying on having pre-built binaries available. This ensures that GoogleTest is built with exactly the same compiler and linker settings as the rest of the project, which avoids many of the subtle issues that can arise when using pre-built GoogleTest binaries. Projects can do this in a number of ways, each with their advantages and drawbacks. Embedding a copy of the sources and headers in the project is the simplest, but it disconnects the project from improvements that may be made to GoogleTest in the future. The GoogleTest git repository can be added to the project as a git submodule, but that too comes with its own robustness issues. A third option of downloading the GoogleTest sources as part of the configure step is discussed in detail in Section 28.2, "FetchContent" and doesn't have the drawbacks of the other methods (it is also made very easy with features added in CMake 3.11).

A test executable that uses GoogleTest typically defines more than one test case. The usual pattern of running the executable once and assuming it is a single test case isn't really appropriate. Ideally, each GoogleTest test case should be visible to `ctest` so that each one can be run and assessed individually. The `FindGTest` module provides a `gtest_add_test()` function which scans the source code looking for uses of the relevant GoogleTest macros and extracts out each individual test case as its own `ctest` test. The form of this command has traditionally been the following:

```
gtest_add_tests(executable "extraArgs" sourceFiles..)
```

From CMake 3.1, the list of `sourceFiles` to scan can be replaced by the keyword `AUTO`. The sources are then obtained by assuming `executable` is a CMake target and using its `SOURCES` target property.

In CMake 3.9, it was recognized that projects may want to use the `gtest_add_tests()` function with GoogleTest built by the project itself. This meant the project didn't need a Find module, so the function was moved out to a new `GoogleTest` module and `FindGTest` then included it to maintain backward compatibility. An improved form of the function with keyword arguments was also added as part of that work:

```
gtest_add_tests(
    TARGET target
    [SOURCES src1...]
    [EXTRA_ARGS arg1...]
    [WORKING_DIRECTORY dir]
    [TEST_PREFIX prefix]
    [TEST_SUFFIX suffix]
    [SKIP_DEPENDENCY]
    [TEST_LIST outVar]
)
```

The old form is still supported, but projects should prefer to use the new form instead where possible, since it is more flexible and more robust. For example, the same target can be given to multiple calls to `gtest_add_tests()` with different arguments, with each call having a different

Exclusively for sbmain17@gmail.com Transaction: 0098315591

TEST_PREFIX and/or TEST_SUFFIX to differentiate the sets of tests that get generated. The new form also provides the set of tests found when the TEST_LIST option is given. With the test names available, the project is able to modify the tests' properties as needed. The following example demonstrates these various capabilities:

```
# Assume GoogleTest is already part of the build, so we don't need
# FindGTest and can reference the gtest target directly
include(GoogleTest)
add_executable(TestDriver ...)
target_link_libraries(TestDriver PRIVATE gtest)

# Run the TestDriver twice with two different arguments
gtest_add_tests(
    TARGET      TestDriver
    EXTRA_ARGS  --algo=fast
    TEST_SUFFIX .Fast
    TEST_LIST   fastTests
)
gtest_add_tests(
    TARGET      TestDriver
    EXTRA_ARGS  --algo=accurate
    TEST_SUFFIX .Accurate
    TEST_LIST   accurateTests
)
```

```
set_tests_properties(${fastTests}     PROPERTIES TIMEOUT 3)
set_tests_properties(${accurateTests} PROPERTIES TIMEOUT 20)

set(betaTests ${fastTests} ${accurateTests})
list(FILTER betaTests INCLUDE REGEX Beta)
set_tests_properties(${betaTests} PROPERTIES LABELS Beta)
```

The above example creates two sets of tests and applies different timeout limits to them. The test names will have different suffixes in each group. Without the TEST_SUFFIX options, the second call to gtest_add_tests() would fail because it would try to create tests with the same name as the first call. The example also sets a Beta label to some tests regardless of which test set they belong to.

While gtest_add_tests() works well for simple cases and source files that don't have unusual formatting, it doesn't cope with parameterized tests or tests defined through custom macros. It also requires re-running CMake to re-scan the source files whenever the test sources change. If the CMake step isn't fast, it can be frustrating when working on the test code as CMake will be forced to re-run for the next build after each change. The SKIP_DEPENDENCY option prevents that behavior and relies on the developer manually re-running CMake to update the set of tests, but this is more a temporary workaround when actively working on a test than something that should be left permanently in place.

In CMake 3.10, a new function was added to address the shortcomings of gtest_add_tests() by asking the executable to list its tests. This query is performed during the build or when running ctest rather than scanning the source code during the configure stage. Because of this, CMake does not need to be re-run whenever the test source is changed, parameterized tests are handled

Exclusively for sbmain17@gmail.com Transaction: 0098315591

robustly and there is no restriction on the formatting or the way that the tests are defined. The trade-off is that the list of tests is not available during the CMake run.

```
gtest_discover_tests(target
    [EXTRA_ARGS arg1...]
    [WORKING_DIRECTORY dir]
    [TEST_PREFIX prefix]
    [TEST_SUFFIX suffix]
    [NO_PRETTY_TYPES]
    [NO_PRETTY_VALUES]
    [PROPERTIES name1 value1...]
    [TEST_LIST var]
    [TEST_FILTER filter]         # CMake 3.22 or later
    [DISCOVERY_TIMEOUT seconds]  # See notes below
    [DISCOVERY_MODE]             # CMake 3.18 or later
)
```

With CMake 3.22 or later, the set of tests reported by the executable can be constrained using the `TEST_FILTER` option. When the executable is asked to list its tests with `--gtest_list_tests`, the specified `filter` is also passed as `--gtest_filter=filter`. This allows a subset of the available tests to be selected.

By default, when generating the names of parameterized tests, the function will attempt to use type or value names rather than a numerical index. This will generally result in much more readable and useful names, but for those cases where this is undesirable, the `NO_PRETTY_TYPES` and `NO_PRETTY_VALUES` options can be used to suppress the substitution and just use the index values. Note that if `TEST_FILTER` is used, the filter is matched against the original test names with indexes as reported by `--gtest_list_tests`, not the pretty test names used by `ctest`.

The `DISCOVERY_TIMEOUT` option refers to the time taken to run the executable to obtain the list of tests. The default of 5 seconds should be sufficient for all but those executables with a huge number of tests or some other behavior that causes it to take a long time to return the test list. This option was originally added in CMake 3.10.1 with the keyword name `TIMEOUT`, but it was found to cause name clashes with the `TIMEOUT` test property in a way that led to unexpected but legal behavior. The keyword was changed to `DISCOVERY_TIMEOUT` in CMake 3.10.3 to prevent those scenarios.

Since the list of tests is not returned to the caller, it is not possible to call `set_tests_properties()` or `set_property()` to modify properties of the discovered tests. Instead, `gtest_discover_tests()` allows properties and their values to be specified as part of the call, which are then written into the `ctest` input file to be applied when `ctest` is run. While not providing all the flexibility of being able to iterate through the set of discovered tests in CMake and processing them individually, the ability to set properties of the discovered tests as a whole is usually all that is needed and is not typically a significant restriction. The main exception to this is that it is not possible to set test properties that have names which correspond to keywords in the `gtest_discover_tests()` command, or where properties require values that are lists. A custom `ctest` script must be used to handle such cases, an example of which is given below.

The `TEST_LIST` option works differently for `gtest_discover_tests()` than for `gtest_add_tests()`. In this case, the variable name given with this option is used in the `ctest` input file written out by CMake rather than being available to CMake directly. The `TEST_LIST` option would only be needed if the

Exclusively for sbmain17@gmail.com Transaction: 0098315591

project adds some of its own custom logic to the generated `ctest` input file and wants to refer to the list of generated tests. Even then, only if the same target is being used in multiple calls to `gtest_discover_tests()` would this be necessary. A default variable name of `<target>_TESTS` is used if not set by a `TEST_LIST` option.

Custom code can be added by appending file names to the list of files held in the `TEST_INCLUDE_FILES` directory property. Projects must not overwrite this directory property, they should only append to it since `gtest_discover_tests()` uses the property to build up the set of files to be read by `ctest`.

The following example shows how to use a custom file to manipulate properties on discovered tests and implement the same equivalent logic as the earlier example for `gtest_add_tests()`, including a workaround for the `TIMEOUT` name clash corner case:

```
gtest_discover_tests(
    TestDriver
    EXTRA_ARGS  --algo=fast
    TEST_SUFFIX .Fast
    TEST_LIST   fastTests
)
gtest_discover_tests(
    TestDriver
    EXTRA_ARGS  --algo=accurate
    TEST_SUFFIX .Accurate
    TEST_LIST   accurateTests
)

set_property(DIRECTORY APPEND PROPERTY
    TEST_INCLUDE_FILES ${CMAKE_CURRENT_LIST_DIR}/customTestManip.cmake
)
```

customTestManip.cmake

```
# Set here to work around the TIMEOUT keyword clash for the
# gtest_discover_tests() call, works for all CMake versions
set_tests_properties(${fastTests}     PROPERTIES TIMEOUT 3)
set_tests_properties(${accurateTests} PROPERTIES TIMEOUT 20)

set(betaTests ${fastTests} ${accurateTests})
list(FILTER betaTests INCLUDE REGEX Beta)
set_tests_properties(${betaTests} PROPERTIES LABELS Beta)
```

Using a custom `ctest` script adds a little more complexity to the project, but it allows full control over test properties. There is no concern about name clashes with `gtest_discover_tests()` and properties with list values can be handled safely.

With CMake 3.17 and earlier, the executable is queried for the list of tests as a `POST_BUILD` step during the build. With CMake 3.18 or later, it is possible to defer querying the list of tests until `ctest` is run. This has at least the following advantages:

- It might not always be possible to run the test executable during the build stage. On the other hand, the test executable *must* be runnable at test time, either natively or via an emulator.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

- Because the query happens during the test stage rather than during the build, the cost of performing the query is only paid during testing. This improves the developer turnaround time when they are working on just getting the test code to compile. In the cross-compilation case where emulator start-up time may be non-trivial, the potential benefits of this deferral could be significant.
- On Windows, executing the test list query during the build would require the `PATH` environment variable to be set such that all of the test executable's DLLs can be found. This is not only inconvenient, it also has the potential to change the way things are built. Deferring the query to the test stage means executables don't have to be runnable at build time, thereby avoiding these problems.

The `DISCOVERY_MODE` option controls when the test list query is performed. It gets its default value from the `CMAKE_GTEST_DISCOVER_TESTS_DISCOVERY_MODE` variable. If that variable is not set, the mode defaults to `POST_BUILD` to preserve backward compatibility. Instead of using `POST_BUILD`, the mode can be set to `PRE_TEST`, which delays the query until the test stage. In practice, the `PRE_TEST` mode should always be preferable. It will generally be more convenient to set the discovery mode project-wide via the variable instead of explicitly setting it in every call to `gtest_discover_tests()`.

25.12. Recommended Practices

Aim to make the name of each test short, yet sufficiently specific to the nature of the test. This makes it easy to narrow down a test set using regular expressions with the `-R` and `-E` options given to `ctest`. Avoid including `test` in the name, since it adds extra content to the test output with no benefit.

Assume that the project may one day be incorporated into a much larger hierarchy of projects which may have many other tests. Aim to use test names that are specific enough to reduce the chances of name clashes. More importantly, prefer to give parent projects control over whether to add the tests at all and define the default behavior to only add tests if there is no parent project. Use a non-cache variable to implement the control so that a parent project can choose whether to expose it in the cache or not. A suitable variable name would be `TEST_XXX` where `XXX` is the uppercased project name. The following example demonstrates such an arrangement for a top level project called `FooBar`:

```
if(TEST_FOOBAR OR CMAKE_SOURCE_DIR STREQUAL FooBar_SOURCE_DIR)
    add_test(...)
endif()
```

To further improve integration with parent projects, consider using the `LABELS` test property to include a project-specific label for each test. These per-project labels should allow tests to be easily included or excluded by regular expressions given to `ctest` via `-L` and `-LE` options. Tests can have multiple labels, so this places no restriction on how else labels can be used, but it may be difficult to ensure that the project-specific label is rigorously set on all of a project's tests.

Another good use of labels is to identify tests that are expected to take a long time to run. Developers and continuous integration systems may want to run these less frequently, so being able to exclude them based on test labels can be very convenient. Consider adding a label to tests that

Exclusively for sbmain17@gmail.com Transaction: 0098315591

run for a non-trivial amount of time and that don't need to run as often. In the absence of any other existing convention, a label of `LongRunning` is a good choice.

As well as using regular expression matching against test names and labels, it is also possible to narrow the set of tests down to a particular directory and below. Instead of running `ctest` from the top of the build tree, it can be run from subdirectories below it. Only those tests defined from that directory's associated source directory and below will be known to `ctest`. To be able to take full advantage of this, tests should not all be collected together in one place and defined with no directory structure. It may be useful to keep tests close to the source code they are testing so that the natural directory structure of the source code can be re-used to also give structure to the tests. If the source code is ever moved around, this approach also makes it easier to move the associated tests with it.

It can be tempting to write tests that simply turn on a lot of logging and then use pass/fail regular expressions to determine success. This can be a fairly fragile approach, as developers frequently change logged output under the assumption that it is just for informational purposes. Adding timestamps into the logged output further complicates that approach. Rather than relying on matching logged output, where possible prefer to make the test code itself determine the success or failure status by explicitly testing the expected pre- and post-conditions, intermediate values, etc. Testing frameworks such as GoogleTest make writing and maintaining such tests considerably easier and are strongly recommended (which framework is less important than at least using *some* suitable framework).

If using the GoogleTest framework, consider using the `gtest_add_tests()` and `gtest_discover_tests()` functions provided by the `GoogleTest` module. If the test code is simple enough for `gtest_add_tests()` to find all tests, it offers the simplest and most flexible way of manipulating individual test properties, but it can be less convenient while working on the test code itself since it can require re-running CMake frequently. If the project can require CMake 3.10.3 or later as a minimum version, then `gtest_discover_tests()` may be more suitable. The main drawback to this function is that setting test properties to values that are lists requires more work, which is particularly relevant if following the advice above regarding the use of test labels. If supporting CMake versions before 3.9 is required, only `gtest_add_tests()` can be used and only the simpler form of the command. The project will also need to use the `FindGTest` module rather than the `GoogleTest` module, which adds further complexity if GoogleTest is being built as part of the project itself. Projects are therefore strongly advised to move to CMake 3.9 or later if using GoogleTest and ideally 3.10.3 or later. The `CMAKE_GTEST_DISCOVER_TESTS_DISCOVERY_MODE` variable should also be set to `PRE_TEST` to give more robust and more convenient behavior for `gtest_discover_tests()` (advantages apply with CMake 3.18 or later).

For projects where cross-compiling for a different target platform is a possibility, consider whether tests can be written to run under an emulator or be copied and executed on a remote system via a script or an equivalent mechanism. CMake's `CMAKE_CROSSCOMPILING_EMULATOR` variable and the associated `CROSSCOMPILING_EMULATOR` target property can be used to implement either of these strategies. Ideally, `CMAKE_CROSSCOMPILING_EMULATOR` would be set in the toolchain file used for the cross-compilation.

Make the most of the support for parallel test execution in `ctest`. Where tests are known to use more than one CPU, set those tests' `PROCESSORS` property to provide better guidance to `ctest` for how to schedule them. If tests need exclusive access to a shared resource, use the `RESOURCE_LOCK` property

Exclusively for sbmain17@gmail.com Transaction: 0098315591

to control access to that resource and avoid using the `RUN_SERIAL` test property unless there is no other alternative. `RUN_SERIAL` can have a big negative impact on parallel test performance and is rarely justified apart from quick, temporary developer experiments. If the machine on which `ctest` is being run may have other processes contributing to the CPU load, consider using the `-l` option to help limit the CPU over-commit. This can be especially useful on developer machines where developers may be building and running tests for multiple projects simultaneously.

If the minimum CMake version can be set to 3.7 or later, prefer to use test fixtures to define dependencies between tests. Define test cases to setup and clean up resources required by other tests, to start and stop services and so on. When running with a reduced test set as a result of regular expression matching or options like `--rerun-failed`, `ctest` automatically adds the required fixture tests to the test set. Fixtures also ensure that tests whose dependencies fail are skipped, unlike the `DEPENDS` test property which merely controls test order without enforcing a success requirement. To gain fine-grained control over which tests will be automatically added to the test set to satisfy fixture dependencies, use CMake 3.9 or later for the `ctest` options `-FS`, `-FC` and `-FA` added in that release. Projects can still require only CMake 3.7 as a minimum version. Also prefer to use fixtures over the `TIMEOUT_AFTER_MATCH` test property due to the clearer dependency relationship and timing control.

The `ctest` build and test mode can be a useful way of incorporating small test builds off to the side as test cases in the main project's test suite. These can be especially effective when some of those test builds need to verify that certain situations lead to configure or build errors. Since test cases can be defined as expected to fail, they can verify such conditions without making the main project's build fail. Consider using the `ctest` build and test mode as the `COMMAND` part of a call to `add_test()` to define such test cases.

For running the complete configure, build and test pipeline of the main project, consider the functionality offered by the CDash integration features rather than using the `ctest` build and test mode. These do a better job of capturing output from the whole pipeline and providing mechanisms for customizing each step's behavior. It also has additional features that facilitate using code coverage and dynamic analysis tools such as memory checkers, sanitizers, etc. and these features can be used whether submitting results to a CDash server or not. In fact, the custom `ctest` scripting functionality that drives the whole CDash pipeline can be used without CDash, making it a potentially convenient platform independent way of scripting the whole build and test pipeline for other continuous integration systems as well. The JUnit output support available with CMake 3.21 or later can be especially useful in that scenario. A CDash server can also be used in conjunction with other CI systems to provide a richer set of features for recording and comparing build histories, test failure trends and so on.

With CMake 3.18 or later, the `ctest --stop-on-failure` option can be used to end a test run immediately upon the first error encountered. This can be a time-saving measure during development where any failure is likely to be related to the area the developer is working on at the time. It can also be used to quickly end a continuous integration run so that the failure can be reported as early as possible. This comes at the expense of only providing feedback about one of possibly many errors, so it should normally only be considered when the time to run all tests is high. The `STOP_ON_FAILURE` option to the `ctest_test()` command has a similar effect, but hard-coding this behavior into a dashboard script may be less flexible than using the `--stop-on-failure` command line option.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

Chapter 26. Installing

After all the hard work of developing the source code of a project, creating its various resources, making the build robust and implementing automated tests, the final step of making the software available for distribution is critical. It has a direct effect on the end user's first impressions of the project and if done poorly, may result in the software being rejected before it even gets a chance to be used.

Developers and users may have different expectations for how a project should be made available. For some, simply providing access to the source code repository and expecting end users to checkout and build it themselves is adequate. While this may be part of the delivery model, not all end users may want to get involved at such a low level. Instead, they will frequently expect a pre-built binary package that they can install and use on their machine, preferably via some already familiar package management system. Given the variety of package managers and delivery formats involved, this can present a daunting challenge for project maintainers. Nevertheless, there are enough common elements between most of them that with some judicious planning, it is possible to support most of the popular ones and cover all major platforms.

The earlier in a project's life cycle the delivery phase is considered, the smoother the final packaging and deployment phases are likely to be. A good starting point is to ask the following questions before development begins, or as early as possible for existing projects:

- What platforms should be supported, both initially and potentially in the future? Are there minimum platform API or SDK version requirements in order to support the features of the project?
- What are the package formats that users will be familiar with on each platform? Can the project be delivered in those formats? Are there any specific package formats that are more important than others or that are mandatory?
- Do any of the required or desirable package formats have requirements for how software must be laid out, built or delivered? Do project resources have to be provided in specific formats, resolutions, locations, etc.?
- Might end users want to install multiple versions of the software simultaneously?
- Should the software support being installed without administrative privileges?
- Can the software be made relocatable so that users can install it anywhere on their system (including on any drive, in the case of Windows)?
- Does the project expect one or more of its executables to be made available on the deployment machine through the user's `PATH` environment variable? Are there parts of the project which should not be exposed on the `PATH`?
- Does the project provide anything that other CMake projects may want to use in their own builds (libraries, executables, headers, resources, etc.)?

These questions will strongly impact how the software is laid out when installed, which in turn affects how the source code needs to access its own resources and so on. It may even impact the functionality available to the software, so understanding these things early can save wasted effort later.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

This chapter focuses on the layout aspects and how to assemble the necessary files in their required locations. It also demonstrates how to make a project easy for other CMake projects to consume by providing config package support. Developers from some backgrounds may identify with these aspects as belonging to the realm of `make install`. The next chapter completes the picture by discussing the various package formats that CMake and CPack can produce. The implementation of that support uses the install functionality described here to install to a clean staging area and then produce the final packages from those contents.

26.1. Directory Layout

Understanding the constraints imposed by the deployment platform(s) is an essential step before decisions can be made about how an installed product should be laid out. Only once those details are clear can a CMake project go about defining what to install to where. A few high level observations can be made which potentially have a strong influence on the installed layout of a project.

- Apple formats (bundles, frameworks, etc.) are heavily prescribed and offer little flexibility, but that also makes it very clear how a project needs to structure its deliverables. As covered back in Chapter 23, *Apple Features*, CMake already handles most of this automatically as part of the build phase, making the app ready for the last part of the Xcode-driven process that performs the final app signing, package creation and submission to the app store. If an install stage is used in CMake/CPack at all, it will largely be to simply package up bundles that follow the prescribed layout.
- For projects intending to support being included as part of a Linux distribution, there will almost certainly be very specific guidelines on where each type of file should be installed. The Filesystem Hierarchy Standard forms the basis of most distributions' layout and many other Unix-based systems follow a similar structure. Even if not aiming for inclusion in a distribution directly, the FHS still serves as a good guide for how to structure a package to achieve a smooth and robust installation on many Unix-based systems.
- Some projects may want to make one or more executables available on the user's `PATH` so they can be invoked easily from a terminal or command line. On Windows, if a project installation modifies the `PATH` by adding a directory that also contains some of its own DLLs, other applications may then pick up those DLLs instead of the ones that were expected (e.g. from their own private directories or one of the standard system-wide locations). DLLs from popular toolkits such as Qt regularly fall victim to this scenario as a result of packages modifying the `PATH` in ways they shouldn't. If a project wants to augment the `PATH` for its own executables, it should ensure that no DLLs are present in that directory, but this is directly at odds with the need to have the DLLs in the same directory as executables so that Windows can find them at run time. The typical solution to this is to create a directory containing only launch scripts which can then safely be added to the `PATH`.

26.1.1. Relative Layout

With the exception of deployments to Apple platforms, there is a large degree of commonality (or at least potential commonality) across all the major platforms. The install location can be thought of as consisting of a base path and a relative layout below that path. The base path may be something like `/usr/⋯`, `/opt/⋯` or `C:\Program Files` and obviously varies widely between platforms, but the

Exclusively for sbmain17@gmail.com Transaction: 0098315591

relative layout below that base point is often very similar. A common arrangement sees executables (and for Windows, also DLLs) installed to a `bin` directory, libraries to `lib` or some variant thereof and headers under an `include` directory. Other file types have somewhat more variability in where they are typically installed, but these three already cover some of the most important file types a project will install.

On Windows, another variation is for packages to put executables and DLLs at the base install location rather than under a `bin` subdirectory. While this may be a relatively common practice, it can lead to a fairly crowded base directory, making it harder for users to find other package components. Another variation is for launch scripts to be located in a subdirectory named `cmd`, which keeps them separated from DLLs in other locations such as `bin`.

Finding a directory structure that works for most platforms is desirable, since it minimizes the platform-specific logic that has to be implemented by the project's source code. If the project uses the same relative layout on all platforms, it is easier for an application to find things it needs at run time.

In the absence of any other requirements, CMake's `GNUInstallDirs` module provides a very convenient way to use a standard directory layout. It is consistent with the common cases mentioned above and it also provides various other standard locations that conform to both GNU coding standards and the FHS. Putting aside the parts that relate to the base install path (covered in the next section), the layout can even be used for Windows deployments. Starting with CMake 3.14, a number of install-related commands take their defaults from `GNUInstallDirs` or fall-back locations that are very similar.

Using the `GNUInstallDirs` module is fairly straightforward, it is included like any other module:

```
# Minimal inclusion, but see caveat further below
include(GNUInstallDirs)
```

This will create cache variables of the form `CMAKE_INSTALL_<dir>` where `<dir>` denotes a particular location. The module's documentation gives full details of all the defined locations, but some of the more commonly used ones and their intended use include:

`BINDIR`

Executables, scripts and symlinks intended for end users to run directly. Defaults to `bin`.

`SBINDIR`

Similar to `BINDIR` except intended for system admin use. Defaults to `sbin`.

`LIBDIR`

Libraries and object files. Defaults to `lib` or some variation of that depending on the host/target platform (including possibly a further architecture-specific subdirectory).

`LIBEXECDIR`

Executables not directly invoked by users, but might be run via launch scripts or symlinks located in `BINDIR` or by other means. Defaults to `libexec`

Exclusively for sbmain17@gmail.com Transaction: 0098315591

INCLUDEDIR

Header files. Defaults to `include`.

DATAROOTDIR

Root point of read-only architecture-independent data. Not typically referred to directly, except perhaps to work around caveats for `DOCDIR`.

DATADIR

Read-only architecture-independent data such as images and other resources. Defaults to the same as `DATAROOTDIR` and is the preferred way to refer to locations for arbitrary project data not covered by other defined locations.

MANDIR

Documentation in the `man` format. Defaults to `DATAROOTDIR/man`

DOCDIR

Generic documentation. Defaults to `DATAROOTDIR/doc/PROJECT_NAME` (see notes below for why relying on this default value is relatively unsafe).

Since each location is defined as a cache variable, they can be overridden if needed. Developers would not normally change them, as install locations should be under the control of the project. Even for the project though, changing the locations from the defaults is not generally advisable, but it can be useful if the project wants to mostly follow the standard layout and only needs to make a few small tweaks.

The `DOCDIR` location deserves special mention, as it defaults to a value that incorporates the `PROJECT_NAME` variable. `PROJECT_NAME` is updated by each call to `project()` and therefore can vary throughout the project hierarchy. The `GNUInstallDirs` module sets cache variables only if they are not already defined, so the value of `CMAKE_INSTALL_DOCDIR` will be determined by where the `GNUInstallDirs` module is first included. To protect against this and allow the default documentation directory to follow the project hierarchy, projects may want to explicitly set the `DOCDIR` location every time the module is included (the non-cache variable will override the cache variable):

```
# Explicitly set DOCDIR location each time
include(GNUInstallDirs)
set(CMAKE_INSTALL_DOCDIR ${CMAKE_INSTALL_DATAROOTDIR}/doc/${PROJECT_NAME})
```

For the remainder of this chapter, examples will use the `CMAKE_INSTALL_<dir>` variables for most relative install destinations.

26.1.2. Base Install Location

After the relative layout of installed files has been determined, the base install location of that layout must be decided. A number of considerations impact this decision, but perhaps the first question to answer is whether the install should be relocatable. This just means that any install base point can be used and as long as the relative layout is preserved, the installed project will still work as intended. Being relocatable is highly desirable and should be the goal of most projects, since it opens up more use cases, such as:

Exclusively for sbmain17@gmail.com Transaction: 0098315591

- Multiple versions can be installed simultaneously.
- Relocatable packages can be installed to shared drives which may have different mount points on different end users' machines.
- A set of self-contained relocatable files can be more easily packaged up by a wider range of packaging systems.
- Non-admin users can install a relocatable project locally under their own account.

Not all projects can be made relocatable, some need to place their files in very specific locations (e.g. kernel packages). Some projects can be relocatable except for a few configuration files, in which case a useful strategy can sometimes be to handle those specific files as a scripted post-install step (the next chapter discusses some aspects of this for specific packaging systems).

The choice of base install location is closely tied to the target platform, with each one having its own common practices and guidelines. On Windows, the base install location is usually a subdirectory of `C:\Program Files`, whereas on most other systems, it is `/usr/local` or a subdirectory of `/opt`. CMake provides a number of controls for managing the base install location to mostly abstract away these platform differences. Perhaps the most important is the `CMAKE_INSTALL_PREFIX` variable, which controls the base install location when the user builds the `install` target (the target may be called `INSTALL` with some generator types). The default value of `CMAKE_INSTALL_PREFIX` is `C:\Program Files\${PROJECT_NAME}` on Windows and `/usr/local` on Unix-based platforms.

When installing on Linux, the default value does not conform to the File System Hierarchy standard. The FHS requires system packages to use a base location of `/` or `/usr`, with the latter more likely to be the desired choice. For add-on packages, they should be installed to `/opt/<package>` or `/opt/<provider>`, with a recommendation to use `/opt/<provider>/<package>`. If `<provider>` is used, it is formally expected to be a LANANA-registered name or just the lowercase fully qualified domain name of the organization providing the package. This is to avoid clashes between different packages trying to use the same base install location.

For most projects, on non-Windows platforms it is advisable to explicitly set `CMAKE_INSTALL_PREFIX` to a FHS-compliant `/opt/⋯` path. This should generally be done only in the top level `CMakeLists.txt` and it should be protected by an appropriate check that the project is in fact the top level of the source tree (to support hierarchical project arrangements).

```
if(NOT WIN32 AND CMAKE_SOURCE_DIR STREQUAL CMAKE_CURRENT_SOURCE_DIR)
    set(CMAKE_INSTALL_PREFIX "/opt/mycompany.com/${PROJECT_NAME}")
endif()
```

For cross-compiling scenarios, the `CMAKE_STAGING_PREFIX` variable can be defined to provide an override for where the install rule installs to. This is to allow installing to an alternate part of the file system while still preserving all the other effects of `CMAKE_INSTALL_PREFIX`, such as embedding of paths in the installed binaries (covered in Section 26.2.2, "RPATH" later in this chapter). `CMAKE_STAGING_PREFIX` also affects the search paths of most `find_⋯()` commands.

For some packaging scenarios and to allow testing the install process in a location off to the side, CMake supports the common `DESTDIR` functionality for non-Windows platforms. `DESTDIR` is not a CMake variable, but rather it is a variable passed to the build tool or set as an environment variable

Exclusively for sbmain17@gmail.com Transaction: 0098315591

for the build tool to read. It allows the install base location to be placed under some arbitrary location rather than the root of the file system. It is typically used on a command line when invoking the build tool directly, such as:

```
make DESTDIR=/home/me/staging install
env DESTDIR=/home/me/staging ninja install
```

The DESTDIR functionality is conceptually similar to CMAKE_STAGING_PREFIX, but DESTDIR is specified only at install time and does not affect things like find_⋯() commands. CMAKE_STAGING_PREFIX is saved as a cache variable, whereas DESTDIR is an environment variable and is not saved between invocations of the build tool. See Section 26.9, "Executing An Install" for an even more flexible and more convenient method for carrying out an install when using CMake 3.15 or later.

The combination of CMAKE_INSTALL_PREFIX, CMAKE_STAGING_PREFIX and DESTDIR gives the project and the developer the flexibility to set the base install location as needed and to perform test installs without actually touching the final intended install location. Be aware, however, that the various packaging formats may have their own default base install locations and may completely ignore these three variables in preference to their own package-specific ones.

26.2. Installing Project Targets

With the structure of the install area defined, attention can now move to the installed content itself. Projects use the install() command to define what to install, where those things should be located and so on. This command has a number of different forms, each focused on a particular type of entity which is specified by the first argument to the command. One of the key forms is for installing targets provided by the project (as opposed to imported targets provided by something external to the project):

```
install(TARGETS targets...
        [EXPORT exportName]
        [CONFIGURATIONS configs...]
        # RUNTIME_... dependency options require CMake 3.21 or later
        [RUNTIME_DEPENDENCIES runtimeDepArgs... |
         RUNTIME_DEPENDENCY_SET runtimeSetName]
        # One or more blocks of the following
        [ [entityType]
               [DESTINATION dir]   # Mandatory for CMake 3.13 and earlier
               [PERMISSIONS permissions...]
               [NAMELINK_ONLY | NAMELINK_SKIP]
               [COMPONENT component]
               [NAMELINK_COMPONENT component]   # CMake 3.12 or later only
               [EXCLUDE_FROM_ALL]
               [OPTIONAL]
               [CONFIGURATIONS configs...]
        ]...
        # Special case
        [INCLUDES DESTINATION incDirs...]
)
```


Exclusively for sbmain17@gmail.com Transaction: 0098315591

One or more `targets` are provided and then the `entityType` blocks specify how to handle installing the various parts of those targets. With CMake 3.12 or earlier, each of the targets must be defined in the same directory scope as the `install()` command, but CMake 3.13 removed this restriction. For all CMake versions, the `entityType` must be one of the following:

RUNTIME

Install executable binaries. On Windows, this also installs the DLL part of library targets. Apple bundles are excluded.

LIBRARY

Install shared libraries on all platforms except Windows. Apple frameworks are excluded.

ARCHIVE

Install static libraries (all platforms). On Windows, this also installs the import library (i.e. `.lib`) part of shared library targets. Apple frameworks are excluded.

OBJECTS

Install the objects associated with object libraries (CMake 3.9 or later only).

FRAMEWORK

On Apple platforms, install frameworks (shared or static), including any content that has been copied into them (e.g. by `POST_BUILD` custom rules).

BUNDLE

On Apple platforms, install bundles, including any content that has been copied into them.

PUBLIC_HEADER

On non-Apple platforms, this installs files listed in a framework library target's `PUBLIC_HEADER` property. On Apple platforms, these header files are handled as part of the `FRAMEWORK` entity type instead, but for non-Apple platforms, such targets are treated as ordinary shared libraries and the headers need to be explicitly installed as a separate entity type.

PRIVATE_HEADER

Analogous to the `PUBLIC_HEADER` entity type, except the affected target property is `PRIVATE_HEADER`.

RESOURCE

On non-Apple platforms, this installs files listed in a framework or bundle target's `RESOURCE` property. On Apple platforms, they are installed as part of the framework or bundle instead.

After the `entityType`, various options can be listed and they only apply to that entity type. For instance, the following shows how to install libraries in a way that puts the respective parts in their expected place on all platforms (assuming they are not Apple frameworks):

```
install(TARGETS MySharedLib MyStaticLib
    RUNTIME DESTINATION ${CMAKE_INSTALL_BINDIR}
    LIBRARY DESTINATION ${CMAKE_INSTALL_LIBDIR}
    ARCHIVE DESTINATION ${CMAKE_INSTALL_LIBDIR}
)
```

Exclusively for sbmain17@gmail.com Transaction: 0098315591

The above example shows how the DESTINATION option can specify different locations for different parts of the same target. The command is flexible enough to handle multiple targets of different types all at once. For MySharedLib, on Windows the DLL would go to the RUNTIME destination and the import library to the ARCHIVE destination. On other platforms, the shared library would be installed to the LIBRARY destination. The static library of the MyStaticLib target would be installed to the ARCHIVE destination.

CMake will usually issue a warning or error if a target provides a particular entity for which there is no corresponding entityType section (e.g. one of the targets is a static library but no ARCHIVE section is provided). As an exception to this, the entityType can be omitted, in which case the options that follow the list of targets will apply to all entity types. This is usually only done when it is obvious that there can only be one entity type for the targets listed:

```
# Targets are both executables, so specifying the entity type isn't needed
install(TARGETS exe1 exe2
        DESTINATION ${CMAKE_INSTALL_BINDIR}
)
```

For CMake 3.13 and earlier, a DESTINATION must be provided. CMake 3.14 relaxed this requirement, allowing default destinations for executables, static libraries and shared libraries, but not module libraries, Apple bundles or frameworks. Public and private headers attached to library targets also have default destinations available. For the supported target types, the default destinations are given by the same CMAKE_INSTALL_⋯ variables that the GNUInstallDirs module provides for those entities. If no such variables are defined (i.e. the GNUInstallDirs module has not been included), hard-coded defaults that mostly follow the same defaults as the GNUInstallDirs module will be used instead. Note that the hard-coded defaults lack the logic for handling subtle differences across various Linux/Unix distributions, so projects should generally include the GNUInstallDirs module to obtain the broadest platform support. The hard-coded defaults are detailed in the CMake documentation for the install() command.

```
include(GNUInstallDirs)

# Only legal with CMake 3.14 or later
install(TARGETS MyExe MySharedLib MyStaticLib)
```

The above would install MyExe to CMAKE_INSTALL_BINDIR, MyStaticLib to CMAKE_INSTALL_LIBDIR and MySharedLib to one or both of those two locations depending on the platform. Any public or private headers attached to these targets would be installed to CMAKE_INSTALL_INCLUDEDIR. While this can be very convenient and make the install command very concise, it forces the project to require CMake 3.14 as its minimum version. For the remainder of this chapter, the destinations are still explicitly given in most examples so that they remain applicable to the broadest range of CMake versions.

Options following an entity type can specify more than just the destination. They can also override the default permissions with the PERMISSIONS option, specifying one or more of the same values as for the file(COPY) command described back in Section 19.2, “Copying Files”:

Exclusively for sbmain17@gmail.com Transaction: 0098315591

OWNER_READ	OWNER_WRITE	OWNER_EXECUTE
GROUP_READ	GROUP_WRITE	GROUP_EXECUTE
WORLD_READ	WORLD_WRITE	WORLD_EXECUTE
SETUID	SETGID	

As for `file(COPY)`, permissions not supported for the platform will simply be ignored. Note that CMake usually sets appropriate permissions for all targets by default. One would typically only need to explicitly provide permissions if the installed location needs more restrictive permissions than normal or if one of the `SETUID` or `SETGID` permissions needs to be added. The following example demonstrates both scenarios.

```
# Intended to only be run by an administrator,
# so only allow the owner to have access
install(TARGETS OnlyOwnerCanRunMe
        DESTINATION ${CMAKE_INSTALL_SBINDIR}
        PERMISSIONS
            OWNER_READ OWNER_WRITE OWNER_EXECUTE
)

# Install with set-group permission
install(TARGETS RunAsGroup
        DESTINATION ${CMAKE_INSTALL_BINDIR}
        PERMISSIONS
            OWNER_READ OWNER_WRITE OWNER_EXECUTE
            GROUP_READ GROUP_EXECUTE SETGID
)
```

For the `LIBRARY` entity type, some platforms support the creation of symbolic links when version details have been provided for a library target (see Section 21.3, “Shared Library Versioning”). The set of files and symlinks that might exist for a shared library typically look something like this:

```
libMyShared.so.1.3.2                      ①
libMyShared.so.1 --> libMyShared.so.1.3.2  ②
libMyShared.so   --> libMyShared.so.1      ③
```

① The actual versioned binary built by the project.

② Symbolic link whose name is the soname of the library. When following semantic versioning, this will contain only the major part of the version in its name.

③ Namelink with no version details embedded in the file name. This is required for the library to be found when a linker command line contains an option like `-lMyShared`.

When installing `LIBRARY` entities, the `NAMELINK_ONLY` or `NAMELINK_SKIP` options can be given. The `NAMELINK_ONLY` option will result in only the namelink being installed, whereas `NAMELINK_SKIP` will result in all but the namelink being installed. If a library target has no version details or the platform doesn’t support namelinks, the behavior of these two options changes. `NAMELINK_ONLY` will then install nothing and `NAMELINK_SKIP` will install the real library. These options are especially useful when creating separate runtime and development packages, with the namelink part going into the development package and the other files/links going into the runtime package.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

When a NAMELINK_ONLY option is given, CMake will not warn about missing entity type blocks for other parts of the library not mentioned in that install() command. This is needed because NAMELINK_SKIP and NAMELINK_ONLY cannot both be given in the same install() call, the two have to be split across separate calls (see example below).

Each entityType section can also specify a COMPONENT option. Components are a logical grouping used mainly for packaging and are discussed in detail in the next chapter, but for now, think of them as a way of separating out different install sets. The above mentioned scenario for separate runtime and development packages could be set up as follows:

```
install(TARGETS MyShared MyStatic
    RUNTIME
        DESTINATION ${CMAKE_INSTALL_BINDIR}
        COMPONENT MyProj_Runtime
    LIBRARY
        DESTINATION ${CMAKE_INSTALL_LIBDIR}
        NAMELINK_SKIP
        COMPONENT MyProj_Runtime
    ARCHIVE
        DESTINATION ${CMAKE_INSTALL_LIBDIR}
        COMPONENT MyProj_Development
)

# Because NAMELINK_ONLY is given, CMake won't complain
# about a missing RUNTIME block
install(TARGETS MyShared
    LIBRARY
        DESTINATION ${CMAKE_INSTALL_LIBDIR}
        NAMELINK_ONLY
        COMPONENT MyProj_Development
)
```

From CMake 3.12, a simpler way of splitting out the namelink to a different component is available using the NAMELINK_COMPONENT option. This option can be used in conjunction with COMPONENT, but only within a LIBRARY block. Using this new option, the above can be expressed more concisely:

```
install(TARGETS MyShared MyStatic
    RUNTIME
        DESTINATION ${CMAKE_INSTALL_BINDIR}
        COMPONENT MyProj_Runtime
    LIBRARY
        DESTINATION ${CMAKE_INSTALL_LIBDIR}
        COMPONENT          MyProj_Runtime
        NAMELINK_COMPONENT MyProj_Development   # Requires CMake 3.12 or later
    ARCHIVE
        DESTINATION ${CMAKE_INSTALL_LIBDIR}
        COMPONENT MyProj_Development
)
```

If no COMPONENT is given for a block, it is associated with a default component whose name is given by the variable CMAKE_INSTALL_DEFAULT_COMPONENT_NAME. If that variable is not set, Unspecified is used as

Exclusively for sbmain17@gmail.com Transaction: 0098315591

the default component name. An example where it can be helpful to change the default component name is where a third party child project doesn't use any install components. To keep that child project's install artifacts separate from the main project, the default name can be changed just before calling `add_subdirectory()` to pull the child project into the main build.

The `EXCLUDE_FROM_ALL` option can be used to restrict an entity block to only get installed for component-specific installs. By default, an install is not component-specific and all components are installed, but packaging implementations may install specific components individually. Documentation was added in CMake 3.12 to show how to do this from the command line as well. For most projects, `EXCLUDE_FROM_ALL` is unlikely to be needed.

The `OPTIONAL` keyword is also rarely used. If the entity type of a target is expected to be present but it is missing (e.g. the import library of a Windows DLL for an `ARCHIVE` entity type section), CMake will not consider it an error. Use this option with caution, as it has the ability to mask misconfiguration of the build/install logic.

An entity type block can also be made configuration-specific by adding a `CONFIGURATIONS` option to it. That entity type will only be installed if the current build type is one of those listed. An entity type cannot be listed more than once for a single `install()` command, so if different configurations need different details, multiple calls are needed. The following example shows how to install the Debug and Release versions of static libraries in different directories:

```
install(TARGETS MyStatic
    ARCHIVE
        DESTINATION ${CMAKE_INSTALL_LIBDIR}/Debug
        CONFIGURATIONS Debug
)

install(TARGETS MyStatic
    ARCHIVE
        DESTINATION ${CMAKE_INSTALL_LIBDIR}/Release
        CONFIGURATIONS Release RelWithDebInfo MinSizeRel
)
```

The `CONFIGURATIONS` keyword can also precede all entity blocks and act as a default for those that don't provide their own configuration override. In the following example, all blocks get installed only for `Release` builds, except for the `ARCHIVE` block which is installed for `Debug` and `Release`.

```
install(TARGETS MyShared MyStatic
    CONFIGURATIONS Release
    RUNTIME
        DESTINATION ${CMAKE_INSTALL_BINDIR}
    LIBRARY
        DESTINATION ${CMAKE_INSTALL_LIBDIR}
    ARCHIVE
        DESTINATION ${CMAKE_INSTALL_LIBDIR}
        CONFIGURATIONS Debug Release
)
```

Exclusively for sbmain17@gmail.com Transaction: 0098315591

See Section 26.7.1, "Runtime Dependency Sets" for how to use the RUNTIME_DEPENDENCIES and RUNTIME_DEPENDENCY_SET keywords.

26.2.1. Interface Properties

If the targets are exported (discussed in Section 26.3, "Installing Exports" further below), they have the opportunity to set interface properties to be consumed by other projects' targets. The various INTERFACE target properties are carried through to the exported details of the installed target automatically, but special handling is needed to account for the distinctly different needs of building the target versus those for consuming the installed target. Consider the following code sample:

```
add_library(Foo STATIC ...)
target_include_directories(Foo
    INTERFACE
        ${CMAKE_CURRENT_BINARY_DIR}/somewhere
        ${MyProject_BINARY_DIR}/anotherDir
)
install(TARGETS Foo DESTINATION ...)
```

Within the build, anything linking to Foo will have the absolute paths to somewhere and anotherDir added to its header search path. When Foo is installed, it may be packaged up and deployed to an entirely different machine. Clearly the path to somewhere and anotherDir would no longer make sense, but the above example would add them to consuming targets' header search path anyway. What is needed is a way to say "Use path xxx when building and path yyy when installing", which is exactly what the BUILD_INTERFACE and INSTALL_INTERFACE generator expressions do:

```
include(GNUInstallDirs)
target_include_directories(Foo
    INTERFACE
        $<BUILD_INTERFACE:${CMAKE_CURRENT_BINARY_DIR}/somewhere>
        $<BUILD_INTERFACE:${MyProject_BINARY_DIR}/anotherDir>
        $<INSTALL_INTERFACE:${CMAKE_INSTALL_INCLUDEDIR}>
)
```

$<BUILD_INTERFACE:xxx> will expand to xxx for the build tree and expand to nothing when installing, whereas $<INSTALL_INTERFACE:yyy> does the opposite, ensuring that yyy is only added for the installed target. In the case of INSTALL_INTERFACE, yyy is usually a relative path, which is treated as being relative to the base install location.

While the header search path within the build tree may vary from target to target, it is common for the targets to share the same header search path once installed. In the above example, CMAKE_INSTALL_INCLUDEDIR is likely to be repeated for every target, but specifying it individually for each one is not the most convenient approach. The INCLUDES option of the install() command can be used instead to specify the same information for a group of targets. All the directories given after INCLUDES DESTINATION are added to the INTERFACE_INCLUDE_DIRECTORIES property of each installed target listed. This leads to a more concise description of header search paths.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

```
add_library(MyStatic STATIC ...)
add_library(MyHeaderOnly INTERFACE ...)

target_include_directories(MyStatic
    PUBLIC $<BUILD_INTERFACE:${CMAKE_CURRENT_BINARY_DIR}/static_exports>
)
target_include_directories(MyHeaderOnly
    INTERFACE $<BUILD_INTERFACE:${CMAKE_CURRENT_LIST_DIR}>
)
install(TARGETS MyStatic MyHeaderOnly
    ARCHIVE  DESTINATION ${CMAKE_INSTALL_LIBDIR}
    INCLUDES DESTINATION ${CMAKE_INSTALL_INCLUDEDIR}
)
```

Unlike the other entity type blocks, multiple directories can be listed for `INCLUDES DESTINATION` if required, although this is likely to be less common in practice. Also note that an `INCLUDES` block supports none of the other details that other `entityType` blocks support, it may only specify a `DESTINATION` keyword followed by one or more locations.

26.2.2. RPATH

When a library or executable is loaded by the operating system, it has to find all the other shared libraries the binary has been linked against. Different platforms have different ways of handling this. Windows relies on finding all required libraries by searching the locations in the `PATH` environment variable as well as the directory in which the binary is located. Other platforms use different environment variables specifically intended for the purpose, such as `LD_LIBRARY_PATH` or variations thereof, in conjunction with other mechanisms such as libraries listed in `conf` files. A drawback to the dependence on environment variables is that it relies on the person or process loading the binary to have set up the environment correctly.

In many cases, the package providing the binary already knows where many of the dependent libraries can be found, since they may have been part of the same package. Most non-Windows platforms support binaries being able to encode library search paths directly into the binaries themselves. The generic name for this feature is *run path* or `RPATH` support, although the actual name may have platform-specific variations. With embedded `RPATH` details, a binary can be self-contained and not have to rely on any paths being provided by the environment or system configuration. Furthermore, an `RPATH` can contain certain placeholders that allow it to effectively define relative paths that are only resolved to absolute paths at run time. The placeholders allow that resolution to be made based on the location of the binary, so relocatable packages can define `RPATH` details that only hard-code paths based on the package's relative layout.

Just as was the case for interface properties in the previous section, there are conflicting needs for `RPATH` in the build tree and for installed binaries. In the build tree, developers need the binaries to be able to find the shared libraries they link to so that executables can be run (e.g. for debugging, test execution and so on). On platforms that support `RPATH`, CMake will embed the required paths by default, thereby giving developers the most convenient experience without requiring any further setup. These `RPATH` details are only suitable for that particular build tree though, so when the targets are installed, CMake rewrites them with replacement paths (the default replacement yields an empty `RPATH`).

Exclusively for sbmain17@gmail.com Transaction: 0098315591

The RPATH defaults are a reasonable starting point, but they are unlikely to be suitable for installed targets. Projects will want to override the default behavior to ensure that both build tree and installed scenarios are suitably catered for. CMake allows separate control of the build and install RPATH locations, so projects can implement a strategy that best fits their needs. The following target properties and variables can be useful for influencing the RPATH behavior:

BUILD_RPATH

This target property can be used to provide additional search paths to be embedded in the build tree's binary. This will be in addition to the paths automatically added by CMake for that binary's link dependencies, so only extra paths CMake cannot work out on its own should be specified. This property should only be needed if the binary loads non-linked libraries at run time using dlopen() or some equivalent mechanism, such as when loading optional plugin modules. This property is initialized by the value of the CMAKE_BUILD_RPATH variable at the time the target is created by add_library() or add_executable(). While the automatically added paths have been supported in CMake for a long time, the BUILD_RPATH property and the CMAKE_BUILD_RPATH variable were only added in CMake 3.8.

BUILD_RPATH_USE_ORIGIN

This target property is only supported for CMake 3.14 or later. For those platforms that support $ORIGIN in an RPATH (see further below), setting this property to true results in CMake embedding $ORIGIN-relative paths rather than absolute paths in the build tree's binaries. This is to facilitate being able to make reproducible, relocatable builds. The initial value for BUILD_RPATH_USE_ORIGIN is taken from the value of the CMAKE_BUILD_RPATH_USE_ORIGIN variable at the time the target is created. This property will only affect those paths CMake determines automatically, it will not affect any paths specified in the BUILD_RPATH property. It will also not have any effect on the embedded RPATH of installed binaries.

INSTALL_RPATH

This target property specifies the RPATH of the binary when it is installed. Unlike the build RPATH, CMake does not provide any install RPATH contents by default, so the project should set this property to a list of paths that reflect the installed layout. Details further below discuss how this can be done. This property is initialized by the value of the CMAKE_INSTALL_RPATH variable when the target is created.

INSTALL_RPATH_USE_LINK_PATH

When this target property is set to true, the path of each library this target links to is added to the set of install RPATH locations, but only if the path points to a location outside the project's source and binary directories. This is mainly useful for embedding absolute paths to external libraries that are not part of the project, but that are expected to be at the same location on all machines the project will be deployed to. Use this with caution, as such assumptions can reduce the robustness of the installed package (paths may change with future releases of the external libraries, system administrators may choose non-default installation configurations, etc.). This property is initialized by the value of the CMAKE_INSTALL_RPATH_USE_LINK_PATH variable when the target is created.

BUILD_WITH_INSTALL_RPATH

Some projects use a build layout that mirrors the installed layout. Targets may expect to find certain files relative to their own location, or they could be self-contained app bundles with

Exclusively for sbmain17@gmail.com Transaction: 0098315591

embedded frameworks. For these cases, the install RPATH may also be suitable for the build tree. By setting this target property to true, the build RPATH is not used and the install RPATH will be embedded in the binary at build time instead. Note that this may cause problems during linking if using placeholders supported by the loader but not the linker (discussed below). This property is initialized by the CMAKE_BUILD_WITH_INSTALL_RPATH variable when the target is created.

SKIP_BUILD_RPATH

When this target property is set to true, no build RPATH is set. BUILD_RPATH will be ignored and CMake will not automatically add RPATH entries for libraries the target links to. Note that this can cause builds to fail if dependent libraries link to other libraries, so use with caution. This property is initialized by the value of the CMAKE_SKIP_BUILD_RPATH variable when the target is created. It is also overridden by BUILD_WITH_INSTALL_RPATH if that property is set to true.

CMAKE_SKIP_INSTALL_RPATH

This variable is the install equivalent of CMAKE_SKIP_BUILD_RPATH. Setting it to true causes INSTALL_RPATH target properties to be ignored and will likely cause the installed targets to fail to find their dependent libraries at run time, so its usefulness is questionable. Note that there is no SKIP_INSTALL_RPATH target property, only the CMAKE_SKIP_INSTALL_RPATH variable.

CMAKE_SKIP_RPATH

Setting this variable to true causes all RPATH support to be disabled and all of the above properties and variables will be ignored. It is generally not desirable to do this unless the project is managing the run time library loading itself in some other way, but in general the RPATH functionality should generally be preferred.

Install RPATH locations should ideally be based on relative paths. This is achieved on most Unix-based platforms by using the $ORIGIN placeholder to represent the location of the binary in which the RPATH is embedded. For example, the following is a common way of defining install RPATH details for projects that follow a similar layout to that defined by the GNUInstallDirs module:

```
set(CMAKE_INSTALL_RPATH $ORIGIN $ORIGIN/../lib)
```

To make this more robust and account for potential changes from the default layout, a little more work is needed. One has to work out the relative path from the executables directory to the libraries directory, which can be achieved as follows:

```
include(GNUInstallDirs)
file(RELATIVE_PATH relDir
     ${CMAKE_CURRENT_BINARY_DIR}/${CMAKE_INSTALL_BINDIR}
     ${CMAKE_CURRENT_BINARY_DIR}/${CMAKE_INSTALL_LIBDIR}
)
set(CMAKE_INSTALL_RPATH $ORIGIN $ORIGIN/${relDir})
```

All targets defined after the above will have an INSTALL_RPATH that directs the loader to look in the same directory as the binary as well as something like ../lib or its platform equivalent relative to the binary's location. Thus, for executables installed to bin and shared libraries installed to lib, this will ensure both can find any other libraries provided by the project. This is highly recommended

Exclusively for sbmain17@gmail.com Transaction: 0098315591

as a starting point when first adding RPATH support to projects. Note that Apple targets work a little differently and may have a considerably different layout, so the above needs to be adapted further to cover that platform (discussed in the next section).

One weakness to be aware of is that while loaders understand $ORIGIN, the linker most likely will not. This can lead to problems when something links to a library which itself links to another library. The first level of linking does not present a problem, since the library will be listed directly on the linker command line, but the second level of library dependency has to be found by the linker. When the linker doesn't understand $ORIGIN, it can't find the second level library via RPATH details. Therefore, unless the path is also specified by some other option like -L, linking will fail, even though the first level library technically contains all the information needed. This is a known issue that is not specific to CMake, it is a weakness of popular linkers (notably the GNU ld linker).

Depending on the various properties and variables mentioned above, CMake may be required to change the embedded RPATH details of a target when it is being installed. There are two ways this can be done. If the binary is in the ELF format, then by default, CMake uses an internal tool to rewrite the RPATH directly in the installed binary. From CMake 3.20, equivalent functionality for the XCOFF format on AIX is also provided (the feature is called LIBPATH for XCOFF, but within CMake it is still referred to as RPATH for convenience). CMake ensures there will be enough space for the install RPATH by padding the build RPATH if necessary. The details of how this is done are largely hidden from the developer, other than perhaps some odd-looking options on the linker command line at build time. For other binary formats, CMake re-links the binary at install time, specifying the install RPATH details instead. Historically, this can sometimes confuse developers who wonder why something that has already been built needs to be linked again, but ultimately the re-linking is a pragmatic way to get the desired end result. The re-linking behavior can be forced for ELF or XCOFF binaries too by setting the CMAKE_NO_BUILTIN_CHRPATH variable to true, but this should not generally be used unless the internal RPATH rewriting fails for some reason.

When cross compiling, a few other variables can modify the RPATH locations embedded in binaries. Any RPATH location that starts with the CMAKE_STAGING_PREFIX will automatically have that prefix replaced with the CMAKE_INSTALL_PREFIX. This is true for both build and install RPATH locations. Any install RPATH location that begins with the CMAKE_SYSROOT will have that prefix stripped entirely.

26.2.3. Apple-specific Targets

Apple's loader and linker work a little differently to other Unix platforms. Whereas libraries on platforms like Linux encode just the library name into a shared library (i.e. the *soname*), Apple platforms encode the full path to the library. This full path is referred to as the install_name and the path part of the install_name is sometimes called the install_name_dir. Anything linking to the library also encodes the full install_name as the library to search for.

When everything is installed to the expected location, this works well, but for relocatable packages (which includes most app bundles), this is too inflexible. As a way of dealing with this, Apple supports relative base points similar to $ORIGIN, but the placeholders are different:

@loader_path

This is more or less Apple's equivalent of $ORIGIN, but the linker is able to understand it and therefore doesn't suffer the problems other linkers experience with being unable to decode $ORIGIN.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

`@executable_path`

This will be replaced by the location of the program being executed. For libraries pulled in as dependencies of other libraries, this is less helpful, since it requires the libraries to know the location of any executable that may use them. This is generally undesirable, so `@loader_path` is usually the better choice.

`@rpath`

This can be used as a placeholder for all or part of the `install_name_dir`.

The combination of `@loader_path` and `@rpath` can provide equivalent behavior to `$ORIGIN` on other platforms. CMake provides additional Apple-specific controls to help set things up appropriately:

`MACOSX_RPATH`

When this target property is set to true, CMake automatically sets the `install_name_dir` to `@rpath` when building for Apple platforms. This is the default behavior since CMake 3.0 and is almost always desirable. It can be overridden by `INSTALL_NAME_DIR`. If the `CMAKE_MACOSX_RPATH` variable is set at the time the target is created, it is used to initialize the value of the `MACOSX_RPATH` property.

`INSTALL_NAME_DIR`

This target property is used to explicitly set the `install_name_dir` part of the library's `install_name`. The default `install_name` usually has the form `@rpath/libsomename.dylib`, but for cases where `@rpath` is not appropriate, `INSTALL_NAME_DIR` can specify an alternative. The property is initialized with the value of the `CMAKE_INSTALL_NAME_DIR` variable at the time it is created. This property is ignored on non-Apple platforms.

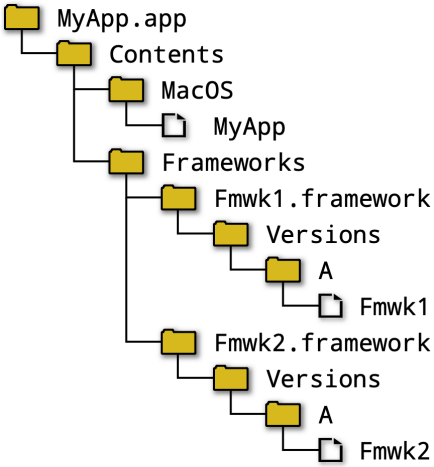
A long-standing bug in CMake versions before 3.20.1 results in mishandling of the `install_name_dir` when targeting iOS, watchOS or tvOS. The bug forces the `install_name_dir` to be the full path to the binary instead of the default `@rpath`. Furthermore, the `INSTALL_NAME_DIR` target property has no effect. If an app bundle contains embedded frameworks (see Section 23.10, "Embedding Frameworks"), this results in the app failing to run because it is unable to find its frameworks at run time. The full paths that were embedded won't exist on the device or a device simulator. Use CMake 3.20.1 or later to avoid this problem.

For non-bundle layouts, the `$ORIGIN` behavior can be extended to cover the Apple case as well:

```
if(APPLE)
    set(base @loader_path)
else()
    set(base $ORIGIN)
endif()

include(GNUInstallDirs)
file(RELATIVE_PATH relDir
     ${CMAKE_CURRENT_BINARY_DIR}/${CMAKE_INSTALL_BINDIR}
     ${CMAKE_CURRENT_BINARY_DIR}/${CMAKE_INSTALL_LIBDIR}
)
set(CMAKE_INSTALL_RPATH ${base} ${base}/${relDir})
```

Exclusively for sbmain17@gmail.com Transaction: 0098315591

Once Apple bundles or frameworks are used, the layout is completely different and alternative strategies are needed for defining the run time search paths. For example, a macOS app bundle may end up with the following structure after installing the relevant targets or as a result of embedding frameworks (discussed in Section 23.10, "Embedding Frameworks"). Only relevant parts of the bundle structure are shown:



`RPATH` details for the above arrangement could be implemented by setting the `INSTALL_RPATH` target property of `MyApp` to `@executable_path/../Frameworks` and for `Fmwk1` and `Fmwk2` it would be set to `@loader_path/../../...` For an iOS app, the paths would be `@executable_path/Frameworks` and `@loader_path/..` instead. The install `RPATH` details should also be used at build time so that embedded frameworks are handled correctly. In the following example, only the RPATH-related properties are shown. See Section 23.6, "Code Signing", Section 23.7, "Creating And Exporting Archives" and Section 23.10, "Embedding Frameworks" for additional details that would need to be set.

```
set(CMAKE_BUILD_WITH_INSTALL_RPATH YES)
add_executable(MyApp MACOSX_BUNDLE ...)
add_library(Fmwk1 SHARED ...)
add_library(Fmwk2 SHARED ...)

# Direct linking like this assumes CMake 3.19 or later
target_link_libraries(MyApp PRIVATE Fmwk1)
target_link_libraries(Fmwk1 PRIVATE Fmwk2)

set_target_properties(MyApp PROPERTIES
    INSTALL_RPATH @executable_path/../Frameworks
)
set_target_properties(Fmwk1 Fmwk2 PROPERTIES
    FRAMEWORK TRUE
    INSTALL_RPATH @loader_path/../../..
)
```

The app bundle and its embedded frameworks can then be installed like so:

```
install(TARGETS Fmwk1 Fmwk2 MyApp
    BUNDLE    DESTINATION .
    FRAMEWORK DESTINATION MyApp.app/Contents/Frameworks
)
```


Exclusively for sbmain17@gmail.com Transaction: 0098315591

Frameworks also have the ability to contain headers. As outlined in Section 23.3, "Frameworks", targets can list their public and private headers in the PUBLIC_HEADER and PRIVATE_HEADER target properties. These are then installed as part of installing the framework itself with no further configuration needed. When those same targets are built on non-Apple platforms, there won't be any framework structure to hold the headers (the targets would be treated as ordinary shared libraries), but the headers can still be installed to a nominated location:

```
install(TARGETS MyShared
    FRAMEWORK    # Apple framework case
        DESTINATION ...
    LIBRARY      # Non-Apple case
        DESTINATION ...
    PUBLIC_HEADER
        DESTINATION ...
    PRIVATE_HEADER
        DESTINATION ...
)
```

26.3. Installing Exports

When project targets are installed, they can specify the name of an export set to which they belong using the EXPORT option with install(TARGETS). That export set can then be installed using a different form of the command:

```
install(EXPORT exportName
        DESTINATION dir
        [FILE name.cmake]
        [NAMESPACE namespace]
        [PERMISSIONS permissions...]
        [EXPORT_LINK_INTERFACE_LIBRARIES]
        [COMPONENT component]
        [EXCLUDE_FROM_ALL]
        [CONFIGURATIONS configs...]
)
```

Installing an export set creates a file at the nominated destination dir with the specified name.cmake file name (it must end in .cmake). If the FILE option is not given, a default file name based on the exportName is used. The generated file will contain CMake commands that define an imported target for each target in the export set. The purpose of this file is for other projects to include it so that they can refer to this project's targets and have full information about the interface properties and inter-target relationships. With some limitations, the consuming project can then treat the imported targets just like any of its own regular targets. These export files are not usually included directly by projects, they are intended to be used by a config package, which is then found by other projects using the find_package() command (this is covered in more detail in Section 26.8, "Writing A Config Package File" later in this chapter).

When the NAMESPACE option is given, each target will have namespace prepended to its name when creating its associated imported target. Consider the following example:

Exclusively for sbmain17@gmail.com Transaction: 0098315591

```
add_library(MyShared SHARED ...)
add_library(BagOfBeans::MyShared ALIAS MyShared)

install(TARGETS MyShared
    EXPORT BagOfBeans
    DESTINATION ${CMAKE_INSTALL_LIBDIR}
)

install(EXPORT BagOfBeans
    DESTINATION ${CMAKE_INSTALL_LIBDIR}/cmake/BagOfBeans
    NAMESPACE BagOfBeans::
)
```

The above example follows the advice from Section 17.4, "Recommended Practices" where each regular target also has a namespaced `ALIAS` associated with it. When installing the export for the non-alias `MyShared` target, the same namespace is used as for the alias target (i.e. `BagOfBeans::`). This allows projects that consume the exported details to refer to the target in the same way as this project can refer to the alias (`BagOfBeans::MyShared`). Consuming projects can then elect to add this project directly via `add_subdirectory()` or pull in the export file via `find_package()`, yet still use the same `BagOfBeans::MyShared` target name regardless of which method was chosen. This important pattern is emerging as a fairly common expectation on projects among the CMake community, so it is in most projects' interests to try to follow it.

One problem that can arise when combining multiple projects into a single build via `add_subdirectory()` (a powerful technique covered in depth in Section 28.2, "FetchContent") is that different projects may define targets with the same name. CMake requires all global targets to be unique, so such projects cannot be combined in that way. To avoid this situation, projects can give their targets a project-specific name, such as `MyProj_Algo` rather than just `Algo`. Whatever is used as a namespace prefix for an export will typically also serve as a suitable prefix for a target name (replacing the `::` with an underscore or removing it completely). With this strategy, to avoid repeating the prefix in the exported name, the target's `EXPORT_NAME` target property can be set to a different name for use only when exporting the target. For example:

```
add_library(MyProj_Algo SHARED ...)
add_library(MyProj::Algo ALIAS MyProj_Algo)

set_target_properties(MyProj_Algo PROPERTIES
    EXPORT_NAME Algo
)

install(TARGETS MyProj_Algo
    EXPORT MyProj
    DESTINATION ${CMAKE_INSTALL_LIBDIR}
)

install(EXPORT MyProj
    DESTINATION ${CMAKE_INSTALL_LIBDIR}/cmake/MyProj
    NAMESPACE MyProj::
)
```


Exclusively for sbmain17@gmail.com Transaction: 0098315591

With the above example, MyProj_Algo will end up with the exported name MyProj::Algo instead of MyProj::MyProj_Algo. Projects are encouraged to make use of this feature to minimize the chances of target name clashes with other projects and still provide concise exported target names.

The name of the export set given after the EXPORT keyword does not have to be related to the NAMESPACE. The namespace is usually closely associated with the project name, but a range of different strategies can be appropriate for the naming of export sets. For example, a project could define multiple export sets with targets that share a single namespace and where the export sets might correspond to logical units that could be installed as a whole. These export sets might each correspond to a single install COMPONENT or they might collect together multiple components. The following demonstrates these cases:

```
# Single component export
install(TARGETS algo1   EXPORT    MyProj_algoFree
        DESTINATION ... COMPONENT MyProj_free
)

install(EXPORT MyProj_algoFree
        DESTINATION ... COMPONENT MyProj_free
)
```

```
# Multi component export
install(TARGETS algo2   EXPORT    MyProj_algoPaid
        DESTINATION ... COMPONENT MyProj_licensed_A
)

install(TARGETS algo3   EXPORT    MyProj_algoPaid
        DESTINATION ... COMPONENT MyProj_licensed_B
)

install(EXPORT MyProj_algoPaid
        DESTINATION ... COMPONENT MyProj_licensed_dev
)
```

In the single component example above, the export set contains just the algo1 target, which is a member of the MyProj_free component. The export file is also a member of the MyProj_free component, so when that component is installed, both the library and the export file will be installed together. For the multi component example, the export set contains algo2 from the MyProj_licensed_A component and algo3 from the MyProj_licensed_B component, but the export file is in its own separate component. Therefore, the targets can be installed with or without the export file based on whether or not the MyProj_licensed_dev component is installed.

The multi component export case above highlights an important aspect of how export sets and components need to be installed. It is an error to install the export file without also installing the actual targets that the export file points to. Thus, if the user installs the MyProj_licensed_dev component, then the MyProj_licensed_A and MyProj_licensed_B components must also be installed.

Of the remaining options of the install(EXPORT) command, a number have similar effects as they do for install(TARGETS). The PERMISSIONS, EXCLUDE_FROM_ALL and CONFIGURATIONS options apply to the installed export file rather than the targets themselves, but are otherwise equivalent. The

Exclusively for sbmain17@gmail.com Transaction: 0098315591

destination used for `install(EXPORT)` is up to the project, but there are some conventions that may be useful to follow. The motivations for these are tied to the main way the exported files are used as part of config packages, so discussion of this topic is delayed to Section 26.8, "Writing A Config Package File" further below.

The `EXPORT_LINK_INTERFACE_LIBRARIES` option is for supporting old pre-3.0 CMake behavior and relates to link interface libraries. Its use is discouraged and projects are advised to update to at least 3.0 as a minimum CMake version instead.

There is a very similar form of the `install()` command specifically for exporting targets for use with Android `ndk-build` projects:

```
install(EXPORT_ANDROID_MK exportName
        DESTINATION dir
        [FILE name.mk]
        [NAMESPACE namespace]
        [PERMISSIONS permissions...]
        [EXPORT_LINK_INTERFACE_LIBRARIES]
        [COMPONENT component]
        [EXCLUDE_FROM_ALL]
        [CONFIGURATIONS configs...]
)
```

Whereas `install(EXPORT)` creates a file for other CMake projects to consume, `install(EXPORT_ANDROID_MK)` creates an `Android.mk` file that `ndk-build` can include. The `Android.mk` file provides all the usage requirements attached to the exported targets, so the `ndk-build` project will be aware of all the compiler defines, header search paths and so on needed to link to them. The name of the exported file can be changed with the `FILE` option, but the name must end with `.mk`. All other options have the same behavior as for the `install(EXPORT)` form. `install(EXPORT_ANDROID_MK)` requires CMake 3.7 or later, but projects may want to require at least 3.11 to avoid a bug that affected static libraries with private dependencies.

In some situations, it may be desirable to have an export file without actually having to do an install. Example scenarios include sub-builds that compile for a different platform to the main build or third party projects that cannot be added to the main build directly due to clashing target names, misuse of variables like `CMAKE_SOURCE_DIR` and so on. For these sort of situations, CMake provides the `export()` command which writes an export file directly into the build tree:

```
export(EXPORT exportName
       [NAMESPACE namespace]
       [FILE fileName]
)
```

The above is essentially equivalent to a simplified `install(EXPORT)` command except the export file is written immediately. The reduced set of available options all have the same meaning as they do for `install(EXPORT)`, although the `fileName` can include a path (it must still end in `.cmake`). Some other forms of the `export()` command allow exporting individual targets instead of an export set, but if export sets are already defined, the above form is likely to be the easiest to use and maintain.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

26.4. Installing Imported Targets

With CMake 3.21 or later, the runtime artifacts of imported targets can also be installed. This is useful when the installed package needs to be fully self-contained, but some of the project's targets link to shared libraries provided by external packages.

```
install(IMPORTED_RUNTIME_ARTIFACTS targets...
        [RUNTIME_DEPENDENCY_SET runtimeSetName]
        [ [entityType]
               [DESTINATION dir]
               [PERMISSIONS permissions...]
               [COMPONENT component]
               [EXCLUDE_FROM_ALL]
               [OPTIONAL]
               [CONFIGURATIONS configs...]
        ]...
)
```

With this form, `entityType` can only be one of `LIBRARY`, `RUNTIME`, `FRAMEWORK` or `BUNDLE`. When building for Apple platforms, the entire framework or bundle will be installed, including headers, resources, etc. When building for Windows, only the DLL binary is installed, not any associated import library (because the latter is a build time rather than a run time artifact). For other platforms, only the shared library is installed.

All of the per-entity-type keywords have the same meaning as for the `install(TARGETS...)` form. See Section 26.7.1, "Runtime Dependency Sets" for details on the use of the `RUNTIME_DEPENDENCY_SET` keyword.

In the following example, the externally-provided `ExtProj::Blah` imported target is expected to be the only dependency of `MyApp`. Since the location of the `ExtProj::Blah` target is already known, it can be installed directly. No runtime dependencies need to be computed, so no runtime dependency set is required. Section 26.2.2, "RPATH" details have been omitted for brevity, but would be needed for a more complete example to allow `MyApp` to find the dependency library at run time.

```
add_executable(MyApp ...)
find_package(ExtProj REQUIRED)
target_link_libraries(MyApp PRIVATE ExtProj::Blah)

install(TARGETS MyApp)
install(IMPORTED_RUNTIME_ARTIFACTS ExtProj::Blah)
```

The `install(IMPORTED_RUNTIME_ARTIFACTS)` has an important limitation. It will halt with an error if asked to find the runtime artifacts of an imported target that is a library of unknown type. Imported targets are often created by Find modules, some of which will define a library as `UNKNOWN` rather than `SHARED` or `STATIC`. They do this because the actual library type is determined later after the library target has been created. Such imported targets cannot be used with `install(IMPORTED_RUNTIME_ARTIFACTS)`.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

26.5. Installing Files And Directories

In contrast to targets, installing individual files and directories is less complicated. Files are installed using the following form:

```
install(<FILES | PROGRAMS> files...
        DESTINATION dir | TYPE type
        [RENAME newName]
        [PERMISSIONS permissions...]
        [COMPONENT component]
        [EXCLUDE_FROM_ALL]
        [OPTIONAL]
        [CONFIGURATIONS configs...]
)
```

The main difference between `install(FILES)` and `install(PROGRAMS)` is that the latter adds execute permissions by default if `PERMISSIONS` is not given. This is intended for installing things like shell scripts which need to be executable, but are not CMake targets. Most of the options are already familiar and have the same meaning as they do for `install(TARGETS)`. The `RENAME` option can only be given if `files` is a single file and it allows that file to be given a different name when installed.

CMake 3.13 and earlier require the `DESTINATION` option to be provided, but from CMake 3.14, the `TYPE` option can be given instead. Unlike for the `install(TARGETS)` case, one of the two must be provided since CMake cannot infer the file type on its own. The set of supported file types is broader than for targets, since various non-target files can be installed with this form. The destination associated with each type is still defined the same way, taking the appropriate variable from `GNUInstallDirs` or falling back to a hard-coded default if that variable is undefined. See the CMake documentation of the `install()` command for the full set of supported types and the associated variables and fallback values. For the broadest CMake version support, projects may wish to continue to use `DESTINATION` rather than `TYPE`, but they should base the destination on the appropriate variable from `GNUInstallDirs` where possible.

In some situations, a project may want to install the binaries associated with an imported target, but the `install(TARGETS)` form does not allow imported targets to be installed directly. One way around this is to install the file(s) associated with the imported target as ordinary files. All of the usage requirements associated with the target won't be preserved, but it does at least allow the binaries to be installed. The `$<TARGET_FILE:…>` generator expression and others like it are particularly useful when employing this technique. A disadvantage of doing this is that it puts the onus back on the project to handle all the platform differences, which is particularly problematic for imported library targets.

```
# Assume MyImportedExe is an imported target for an
# executable not built by this project
install(PROGRAMS $<TARGET_FILE:MyImportedExe>
    DESTINATION ${CMAKE_INSTALL_BINDIR}
)
```

Installing directories follows a similar pattern to files, but the set of supported options is expanded:

Exclusively for sbmain17@gmail.com Transaction: 0098315591

```
install(DIRECTORY dirs...
        DESTINATION dir | TYPE type
        [FILE_PERMISSIONS permissions... |
         USE_SOURCE_PERMISSIONS]
        [DIRECTORY_PERMISSIONS permissions...]
        [COMPONENT component]
        [EXCLUDE_FROM_ALL]
        [OPTIONAL]
        [CONFIGURATIONS configs...]
        [MESSAGE_NEVER]
        [FILES_MATCHING]
        # The following block can be repeated as needed
        [ [PATTERN pattern | REGEX regex]
          [EXCLUDE]
          [PERMISSIONS permissions...] ]
)
```

Without any of the optional arguments, for each `dirs` location the entire directory tree starting at that point is installed into the destination `dir`. If the source name ends with a trailing slash, then the contents of the source directory are copied rather than the source directory itself. The same comments regarding the `DESTINATION` and `TYPE` arguments apply to this form as for installing files.

```
# Results in somewhere/foo/...
install(DIRECTORY foo DESTINATION somewhere)

# Results in somewhere/...
install(DIRECTORY foo/ DESTINATION somewhere)
```

The `COMPONENT`, `EXCLUDE_FROM_ALL`, `OPTIONAL` and `CONFIGURATIONS` options have the same meaning as for other `install()` commands. The `MESSAGE_NEVER` option prevents the log message for each file installed, but one could argue that this should not be used for consistency with messages for all other installed contents.

A few options are supported for controlling the permissions of files and directories separately. If `USE_SOURCE_PERMISSIONS` is given, each file installed will retain the same permissions as its source. `FILE_PERMISSIONS` overrides that and uses the specified permissions instead. If neither option is given, files will have the same default permissions as if the `install(FILE)` command had been used. For directories created by the install, the `DIRECTORY_PERMISSIONS` option can be used to override the defaults, which are the same as for files except execute permissions are also added.

The remaining options allow the set of files to be filtered according to one or more wildcard patterns or regular expressions. Each pattern or regex is tested against the full path to each file and directory (always specified with forward slashes, even on Windows). Wildcard patterns must match the end of the full path, not just some portion in the middle, whereas a regex can match any part of the path and is therefore more flexible. If the pattern or regex is followed by the `EXCLUDE` keyword, then all matching files and directories will not be installed. This is a useful way of excluding just a few specific things from the directory tree, but the reverse can also be implemented by giving the `FILES_MATCHING` keyword (once) before any `PATTERN` or `REGEX` blocks, which then means only those files and directories that *do* match one of the patterns or regexes will be

Exclusively for sbmain17@gmail.com Transaction: 0098315591

installed. If neither `FILES_MATCHING` nor `EXCLUDE` is given, then the only effect of the pattern or regex is to override the permissions with a `PERMISSIONS` block.

Some examples should help clarify the above points. The following example adapted slightly from the CMake documentation installs all headers from the `src` directory and below, preserving the directory structure.

```
install(DIRECTORY src/
    DESTINATION include
    FILES_MATCHING
    PATTERN *.h
)
```

The following installs documentation, skipping over some common hidden files:

```
install(DIRECTORY doc/ todo/ licenses
    DESTINATION doc
    FILES_MATCHING
    REGEX \\.(DS_Store|svn) EXCLUDE
)
```

The next example installs sample code and scripts, ensuring the latter have executable permission:

```
install(DIRECTORY src/
    DESTINATION samples
    FILES_MATCHING
    REGEX "example\\.(h|c|cpp|cxx)"
    PATTERN *.txt
    PATTERN *.sh
      PERMISSIONS OWNER_READ OWNER_WRITE OWNER_EXECUTE
                  GROUP_READ GROUP_EXECUTE
                  WORLD_READ WORLD_EXECUTE
)
```

The example below omits any `FILES_MATCHING` or `EXCLUDE` options so that patterns and regexes only modify permissions and not filter the list of files and directories:

```
install(DIRECTORY admin_scripts
    DESTINATION private
    PATTERN *.sh
      PERMISSIONS OWNER_READ OWNER_WRITE OWNER_EXECUTE
                  GROUP_READ GROUP_EXECUTE
)
```

In all cases, `install(DIRECTORY)` preserves the directory structure of the source. If the list of sources is empty, the `DESTINATION` will still be created as an empty directory.

```
install(DIRECTORY DESTINATION somewhere/emptyDir)
```


Exclusively for sbmain17@gmail.com Transaction: 0098315591

26.6. Custom Install Logic

Sometimes simply copying things into the install area isn't enough. Arbitrary processing may need to be performed as part of the install, such as rewriting parts of a file or generating content programmatically. For these cases, CMake supports adding custom logic to the install step.

```
install(SCRIPT fileName | CODE cmakeCode
        [ALL_COMPONENTS | COMPONENT component]
        [EXCLUDE_FROM_ALL]
)
```

The `CODE` form can be used to embed CMake commands directly as a single string, whereas the `SCRIPT` form will use `include()` to read in the script at install time. The `COMPONENT` and `EXCLUDE_FROM_ALL` options have their usual meanings. CMake 3.21 added support for the `ALL_COMPONENTS` option, which has the effect of executing the custom script or code for every component in a component-based install. It is an error to specify both `ALL_COMPONENTS` and `COMPONENT`.

With CMake 3.13 or earlier, it is unspecified at what point during installation the custom code runs. Generally, `install()` commands are processed in the order they appear in the directory scope, but this does not extend to `install()` calls nested within subdirectories. With CMake 3.14 or later, the order is clearly defined and is controlled by policy `CMP0082`. When this policy is set to `NEW`, install commands are executed in the order they are declared, including accounting for descending into subdirectories. Consider the following example:

```
install(CODE [[ message("Main A") ]])
add_subdirectory(subInstall)
install(CODE [[ message("Main B") ]])
```

subInstall/CMakeLists.txt:

```
install(CODE [[ message("subdir X") ]])
install(CODE [[ message("subdir Y") ]])
```

With policy `CMP0082` unset or set to `OLD`, the output during install would likely contain the following:

```
Main A
Main B
subdir X
subdir Y
```

With policy `CMP0082` set to `NEW`, the order is clearly defined and the installation output would contain:

```
Main A
subdir X
subdir Y
Main B
```

Exclusively for sbmain17@gmail.com Transaction: 0098315591

Multiple SCRIPT and/or CODE blocks can be given, in which case they will be executed in order. The rest of the options can only be specified at most once.

```
install(CODE      [[ message("Starting custom script") ]]
        SCRIPT    myCustomLogic.cmake
        CODE      [[ message("Finished custom script") ]]
        COMPONENT MyProj_Runtime
)
```

With CMake 3.14 or later, the contents given with CODE or the name of the file provided for SCRIPT can contain generator expressions (only the file name/path, not the file's contents). This requires policy CMP0087 to be set to NEW, but unlike most other policies, it is the policy's setting at the end of the directory scope that determines the behavior. The value of the CMP0087 policy when the install() command is called is not relevant. This is because CMake defers processing the contents until the end of the directory scope, not immediately as part of the install() command.

26.7. Installing Dependencies

When creating packages, a common desire is to make them self-contained. This can extend to including not just the project's own build artifacts, but also external dependencies such as compiler runtime libraries. CMake provides some features which can potentially make this task easier.

26.7.1. Runtime Dependency Sets

CMake 3.21 added direct support to install() sub-commands for automatically determining external runtime dependencies and adding them as installed artifacts. The install(TARGETS) and install(IMPORTED_RUNTIME_ARTIFACTS) sub-commands both support a RUNTIME_DEPENDENCY_SET keyword. When RUNTIME_DEPENDENCY_SET is given, both sub-commands will add their targets to the named set. The following sub-command is then used to automatically install all externally provided runtime artifacts needed by any targets in that set:

```
install(RUNTIME_DEPENDENCY_SET setName
        [ [entityType]
               [DESTINATION dir]
               [PERMISSIONS permissions...]
               [COMPONENT component]
               [NAMELINK_COMPONENT component]
               [EXCLUDE_FROM_ALL]
               [OPTIONAL]
               [CONFIGURATIONS configs...]
        ]...
        [PRE_INCLUDE_REGEXES regexes...]
        [PRE_EXCLUDE_REGEXES regexes...]
        [POST_INCLUDE_REGEXES regexes...]
        [POST_EXCLUDE_REGEXES regexes...]
        [POST_INCLUDE_FILES files...]
        [POST_EXCLUDE_FILES files...]
        [DIRECTORIES directories...]
)
```


Exclusively for sbmain17@gmail.com Transaction: 0098315591

If `entityType` is given, it can only be `LIBRARY`, `RUNTIME` or `FRAMEWORK`, since these are the only type of runtime artifacts that targets can link to. The details following an `entityType` apply to the dependencies that are found, not to the targets in the runtime dependency set. The keywords following `entityType` have the same meaning and effect as for `install(TARGETS)`. As usual, `entityType` can be omitted and the keywords that follow it will apply to all dependency entity types.

The `PRE_...`, `POST_...` and `DIRECTORIES` arguments are forwarded through to an internal call to `file(GET_RUNTIME_DEPENDENCIES)`, which is used to compute the dependencies of the targets in the set. The official documentation for the `file(GET_RUNTIME_DEPENDENCIES)` command provides a detailed explanation of these options, so it is not repeated here. Since `file(GET_RUNTIME_DEPENDENCIES)` only supports finding dependencies for Windows, Linux and macOS target platforms, the same limitations apply to `install(RUNTIME_DEPENDENCY_SET)` as well. Furthermore, the `file(GET_RUNTIME_DEPENDENCIES)` command is known to have problems that prevent it from being used in common cross-compiling scenarios.

The following is a minimal example showing the steps for installing an application and its external dependencies to the default install locations:

```
add_executable(MyApp ...)

install(TARGETS MyApp RUNTIME_DEPENDENCY_SET appDeps ...)
install(RUNTIME_DEPENDENCY_SET appDeps)
```

The `install(TARGETS)` sub-command also provides a `RUNTIME_DEPENDENCIES` option which can be used to effectively combine the two `install()` calls (a random runtime dependency set name is generated internally):

```
add_executable(MyApp ...)

install(TARGETS MyApp RUNTIME_DEPENDENCIES ...)
```

The `install(RUNTIME_DEPENDENCY_SET)` command searches for *all* dependencies of targets in the runtime dependency set. This means it also adds system libraries to the set of dependencies to install, which will often be undesirable. Even for a simple "hello world" executable, building with GCC for a typical Linux desktop system may add system libraries to satisfy the following dependencies:

- libc.so.6
- libgcc_s.so.1
- libm.so.6
- libstdc++.so.6

One can use the `PRE_EXCLUDE_REGEXES` option to skip these dependencies:

Exclusively for sbmain17@gmail.com Transaction: 0098315591

```
add_executable(MyApp main.cpp)

install(TARGETS MyApp
    RUNTIME_DEPENDENCY_SET appDeps
)
install(RUNTIME_DEPENDENCY_SET appDeps
    PRE_EXCLUDE_REGEXES
        [[libc\.so\..*]]
        [[libgcc_s\.so\..*]]
        [[libm\.so\..*]]
        [[libstdc\+\+\.so\..*]]
)
```

A similar situation exists on Windows, where the runtime API sets will be added to the dependencies. These should not be installed, they are abstractions for libraries that will be provided by the operating system. These can be excluded in a similar way (this is just an example, projects will want to tweak the set of regular expressions to suit their needs):

```
install(RUNTIME_DEPENDENCY_SET appDeps
    PRE_EXCLUDE_REGEXES
        [[api-ms-win-.*]]
        [[ext-ms-.*]]
        [[kernel32\.dll]]
    POST_EXCLUDE_REGEXES
        [[.*/system32/.*\.dll]]
)
```

In some cases, a dependency may itself require further libraries. The `file(GET_RUNTIME_DEPENDENCIES)` command is able to resolve these recursively, including following RPATH locations where appropriate. A common problem that can arise though is that an RPATH can contain special strings like `$ORIGIN`. As of CMake 3.21.1, `file(GET_RUNTIME_DEPENDENCIES)` does not handle these and will ignore them, which can lead to some dependencies not being found. A workaround on Linux is to manually specify additional places for the command to look using the `DIRECTORIES` keyword of `install(RUNTIME_DEPENDENCY_SET)`. This can lead to warnings, but will at least allow the operation to succeed.

26.7.2. InstallRequiredSystemLibraries Module

The `InstallRequiredSystemLibraries` module is intended to provide projects with the details of relevant run time libraries for the major compilers. It is much more mature than the runtime dependency set support, but also more restricted in the type of dependencies it can install.

The module provides compiler runtimes for Intel (all major platforms) and Visual Studio (Windows only). Using the module is fairly straightforward, with projects either choosing to let the module define the `install()` commands on its behalf, or it can ask for the relevant variables to be populated so it can create the necessary commands for itself. In the simplest case, projects can rely on the defaults, although setting at least the component for the `install()` commands is recommended.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

```
set(CMAKE_INSTALL_SYSTEM_RUNTIME_COMPONENT MyProj_Runtime)
include(InstallRequiredSystemLibraries)
```

The default install locations are `bin` for Windows and `lib` for all other platforms. This is likely to match the typical install layout of most projects, but it can be overridden with the `CMAKE_INSTALL_SYSTEM_RUNTIME_DESTINATION` variable:

```
include(GNUInstallDirs)

if(WIN32)
    set(CMAKE_INSTALL_SYSTEM_RUNTIME_DESTINATION ${CMAKE_INSTALL_BINDIR})
else()
    set(CMAKE_INSTALL_SYSTEM_RUNTIME_DESTINATION ${CMAKE_INSTALL_LIBDIR})
endif()

set(CMAKE_INSTALL_SYSTEM_RUNTIME_COMPONENT MyProj_Runtime)
include(InstallRequiredSystemLibraries)
```

If a project wants to define the `install()` commands itself, it needs to set `CMAKE_INSTALL_SYSTEM_RUNTIME_LIBS_SKIP` to true before including the module. The project can then access the list of runtime libraries using the `CMAKE_INSTALL_SYSTEM_RUNTIME_LIBS` variable:

```
set(CMAKE_INSTALL_SYSTEM_RUNTIME_LIBS_SKIP TRUE)
include(InstallRequiredSystemLibraries)

include(GNUInstallDirs)
if(WIN32)
    install(FILES ${CMAKE_INSTALL_SYSTEM_RUNTIME_LIBS}
        DESTINATION ${CMAKE_INSTALL_BINDIR}
    )
else()
    install(FILES ${CMAKE_INSTALL_SYSTEM_RUNTIME_LIBS}
        DESTINATION ${CMAKE_INSTALL_LIBDIR}
    )
endif()
```

When using Intel compilers, the default `install()` commands install more than just the contents of `CMAKE_INSTALL_SYSTEM_RUNTIME_LIBS`. They also install some directories not provided to the project through any documented variable. For those developers interested in exploring whether these additional contents are desirable or not, search for `CMAKE_INSTALL_SYSTEM_RUNTIME_DIRECTORIES` in the module's implementation to see how these additional contents are constructed.

Some further controls are available when using Visual Studio compilers to install various other run time components, such as Windows Universal CRT, MFC and OpenMP libraries. The installation of debug versions of runtime libraries can also be enforced. These are all described clearly in the module's documentation, so the interested reader is referred to there for further details.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

26.7.3. BundleUtilities And GetPrerequisites

Another pair of modules can also be used to install a project's run time dependencies. The `BundleUtilities` and `GetPrerequisites` modules can be thought of as forerunners to the run time dependency set approach. These two modules directly interrogate the installed binaries using platform-specific tools and recursively copy in missing libraries. These modules can be considerably more difficult to use and should be avoided where either of the approaches presented above can be used.

26.8. Writing A Config Package File

The preferred way for an installed project to make itself available for other CMake projects to consume is to provide a config package file. This file is found by consuming projects using the `find_package()` command, as introduced back in Section 24.5, "Finding Packages". The name of the config file must match either `<packageName>Config.cmake` or `<lowercasePackageName>-config.cmake`. The former is perhaps a little more common and is consistent with other functionality provided by CMake discussed further below, but both are otherwise equivalent. The file is expected to provide imported targets for all the libraries and executables the installed project wants to make available. The directory into which the config file is installed should be one of the default locations that `find_package()` will search if the base point of the install is added to the `CMAKE_PREFIX_PATH` variable. This ensures that the config file will be easy to find. From Section 24.5, "Finding Packages", the full set of locations that will be searched is:

```
<prefix>/
<prefix>/(cmake|CMake)/
<prefix>/<packageName>*/
<prefix>/<packageName>*/(cmake|CMake)/
<prefix>/(lib/<arch>|lib*|share)/cmake/<packageName>*/
<prefix>/(lib/<arch>|lib*|share)/<packageName>*/
<prefix>/(lib/<arch>|lib*|share)/<packageName>*/(cmake|CMake)/
<prefix>/<packageName>*/(lib/<arch>|lib*|share)/cmake/<packageName>*/
<prefix>/<packageName>*/(lib/<arch>|lib*|share)/<packageName>*/
<prefix>/<packageName>*/(lib/<arch>|lib*|share)/<packageName>*/(cmake|CMake)/
```

On Apple platforms, the following subdirectories may also be searched:

```
<prefix>/<packageName>.framework/Resources/
<prefix>/<packageName>.framework/Resources/CMake/
<prefix>/<packageName>.framework/Versions/*/Resources/
<prefix>/<packageName>.framework/Versions/*/Resources/CMake/
<prefix>/<packageName>.app/Contents/Resources/
<prefix>/<packageName>.app/Contents/Resources/CMake/
```

Clearly that's a large set of candidates, but the best choice depends somewhat on how the project expects to be installed. When packaging for inclusion in a Linux distribution, the distribution itself may have policies for where such files are expected to be. Rather than forcing each distribution to carry its own patches to the project to ensure the config file is installed according to its policies, projects should ideally provide a way to pass the required details into the build. A cache variable is

Exclusively for sbmain17@gmail.com Transaction: 0098315591

ideal for this purpose, since the project can specify a default, but it can be overridden without having to change the project at all. In the absence of any other constraints, two very simple and commonly used locations are `<prefix>/cmake` and `<prefix>/lib/cmake/<packageName>`, with variations on the latter being a little friendlier to multi-architecture deployments (see examples below).

For projects that provide an `Android.mk` file from an `install(EXPORT_ANDROID_MK)` command, CMake has no specific convention for its location. A reasonable arrangement would be to use a dedicated `ndk-build` directory within the package layout, but it is ultimately up to the project.

26.8.1. Config Files For CMake Projects

For simple CMake projects that use only a single export set and that have no dependencies, the `install(EXPORT)` command can be used to create a basic config file directly:

```
# Use the export file directly as the package config file.
# Not generally recommended except for trivial projects.
include(GNUInstallDirs)
install(EXPORT MyProj
    DESTINATION ${CMAKE_INSTALL_LIBDIR}/cmake/MyProj
    NAMESPACE MyProj::
    FILE MyProjConfig.cmake
)
```

Note how the destination uses the `CMAKE_INSTALL_LIBDIR` cache variable defined by the `GNUInstallDirs` module to increase the likelihood that Linux distributions won't need to make any changes. The `GNUInstallDirs` module already accounts for the common cases and by defining cache variables, it allows easy customization if required.

In practice, the config file is not normally directly generated like this. More often, a separate config file is prepared which brings in exported files via `include()` commands. A slightly expanded example using two export sets demonstrates the technique:

CMakeLists.txt

```
include(GNUInstallDirs)
install(EXPORT MyProj_Runtime
    DESTINATION ${CMAKE_INSTALL_LIBDIR}/cmake/MyProj
    NAMESPACE MyProj::
    FILE MyProj_Runtime.cmake
    COMPONENT MyProj_Runtime
)
install(EXPORT MyProj_Development
    DESTINATION ${CMAKE_INSTALL_LIBDIR}/cmake/MyProj
    NAMESPACE MyProj::
    FILE MyProj_Development.cmake
    COMPONENT MyProj_Development
)
install(FILES MyProjConfig.cmake
    DESTINATION ${CMAKE_INSTALL_LIBDIR}/cmake/MyProj
)
```

Exclusively for sbmain17@gmail.com Transaction: 0098315591

MyProjConfig.cmake

```
include(${CMAKE_CURRENT_LIST_DIR}/MyProj_Runtime.cmake)
include(${CMAKE_CURRENT_LIST_DIR}/MyProj_Development.cmake)
```

The above `MyProjConfig.cmake` is still very simple. No externally provided dependencies are needed and it is assumed that the runtime and the development components are always both installed. Consider then a scenario where the runtime component depends on some other package named `BagOfBeans`. The config file is responsible for ensuring that the required targets from `BagOfBeans` are available, which it typically does by calling `find_package()`. As a convenience, the `find_dependency()` macro from the `CMakeFindDependencyMacro` module can be used as a wrapper around `find_package()` to handle the `QUIET` and `REQUIRED` keywords. The `find_dependency()` macro also has the behavior that if it fails to find the requested package, processing of the config file ends as though a `return()` call was made immediately after the failed `find_dependency()` call. In practice, this results in simple, clean specification of dependencies with graceful handling of dependency failures.

MyProjConfig.cmake

```
include(CMakeFindDependencyMacro)
find_dependency(BagOfBeans)

include(${CMAKE_CURRENT_LIST_DIR}/MyProj_Runtime.cmake)
include(${CMAKE_CURRENT_LIST_DIR}/MyProj_Development.cmake)
```

Be aware that prior to CMake 3.15, `find_dependency()` contained an optimization that bypassed the call if it detected that the requested package had previously been found. This optimization worked fine unless later calls needed to request a different set of package components. The first time `find_dependency()` succeeds, the pre-3.15 behavior effectively locks in the set of components found. If later calls to `find_dependency()` pass a different set of components, they would be ignored. Because `find_dependency()` calls are typically made within files that are installed for other projects to use, it usually cannot be guaranteed what CMake version will ultimately be used. Therefore, if the dependency supports package components, projects should avoid `find_dependency()` and call `find_package()` directly, handling the `QUIET` and `REQUIRED` options themselves. These two options are passed to the config file as the variables `${CMAKE_FIND_PACKAGE_NAME}_FIND_QUIETLY` and `${CMAKE_FIND_PACKAGE_NAME}_FIND_REQUIRED`. Always use `${CMAKE_FIND_PACKAGE_NAME}` rather than hard-coding the package name, because there may be upper/lowercase differences.

```
unset(extraArgs)
if(${CMAKE_FIND_PACKAGE_NAME}_FIND_QUIETLY)
    list(APPEND extraArgs QUIET)
endif()
if(${CMAKE_FIND_PACKAGE_NAME}_FIND_REQUIRED)
    list(APPEND extraArgs REQUIRED)
endif()

find_package(BagOfBeans COMPONENTS Foo Bar ${extraArgs})
```


Exclusively for sbmain17@gmail.com Transaction: 0098315591

If the project wishes to support some of its own components being optional, the complexity of the config file increases significantly. A set of steps to support such functionality can be summarized as follows:

- Build up the set of components to find. Start with the required and optional components given to `find_package()` and add any that are needed to satisfy project dependencies.
- Work out the external dependencies needed by that set of components. Some will be mandatory, others may be optional, so two separate external dependency sets will need to be derived.
- Find the external dependencies and if any required dependencies fail to load, the project find operation must also fail and control should return immediately with an appropriate error message. Missing optional external dependencies should not cause failure or an error message.
- Update the set of project components to remove any that depend on a missing optional external dependency. This may require further culling of the project component set if the removed components are themselves dependencies of other components.
- Load the project components that remain.

Projects also need to decide what to do if no components are specified at all. This could be treated as though all components had been specified as optional components or even as required components. Another strategy is to load the minimal set of essential components and omit all others. The most appropriate strategy will depend on the nature of the project's components. The set of requested components will be available in the `${CMAKE_FIND_PACKAGE_NAME}_FIND_COMPONENTS` variable and if a component was specified as being required rather than optional, `${CMAKE_FIND_PACKAGE_NAME}_FIND_REQUIRED_<comp>` will be true for that component.

Config files should not report errors using `message()`, they should instead store the error message in a variable named `${CMAKE_FIND_PACKAGE_NAME}_NOT_FOUND_MESSAGE`. This will be picked up by `find_package()` which will wrap it with details about where in the project the error was raised. `${CMAKE_FIND_PACKAGE_NAME}_FOUND` should also be set to false to indicate failure. This allows `find_package()` to properly implement a call that does not use the `REQUIRED` keyword. If the package config file called `message(FATAL_ERROR …)`, the package could never be treated as optional.

Another often neglected but highly recommended practice is to avoid creating any imported targets before first checking whether the required components can be satisfied. This prevents imported targets from being created for some components but not others in the event of a failure.

The following example demonstrates how to incorporate the above points. Note the impact of using `if(…IN_LIST…)` in this case, which requires additional checking at the beginning to handle CMake versions 3.2 and older. Projects may wish to avoid using `IN_LIST` and implement equivalent logic using `list(FIND)` instead. `IN_LIST` is used in the example mostly to raise awareness of its consequences and to show how to use it safely in config files.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

```
# We use if(...IN_LIST...), make sure it is available
if(CMAKE_VERSION VERSION_LESS 3.3)
    set(${CMAKE_FIND_PACKAGE_NAME}_NOT_FOUND_MESSAGE
        "MyProj requires CMake 3.3 or later"
    )
    set(${CMAKE_FIND_PACKAGE_NAME}_FOUND FALSE)
    return()
endif()
cmake_minimum_required(VERSION 3.3...3.21)

# Work out the set of components to load
if(${CMAKE_FIND_PACKAGE_NAME}_FIND_COMPONENTS)
    set(comps ${${CMAKE_FIND_PACKAGE_NAME}_FIND_COMPONENTS})
    # Ensure Runtime is included if Development was given
    if(Development IN_LIST comps AND NOT Runtime IN_LIST comps)
        list(APPEND comps Runtime)
    endif()
else()
    # No components given, look for all components
    set(comps Runtime Development)
endif()

# Find external dependencies, storing comps in a safer variable name.
# In this example, BagOfBeans is required by the mandatory Runtime component.
set(${CMAKE_FIND_PACKAGE_NAME}_comps ${comps})
include(CMakeFindDependencyMacro)
find_dependency(BagOfBeans)

# Check all required components are available before trying to load any
foreach(comp IN LISTS ${CMAKE_FIND_PACKAGE_NAME}_comps)
    if(${CMAKE_FIND_PACKAGE_NAME}_FIND_REQUIRED_${comp} AND
       NOT EXISTS ${CMAKE_CURRENT_LIST_DIR}/MyProj_${comp}.cmake)
        set(${CMAKE_FIND_PACKAGE_NAME}_NOT_FOUND_MESSAGE
            "MyProj missing required component: ${comp}"
        )
        set(${CMAKE_FIND_PACKAGE_NAME}_FOUND FALSE)
        return()
    endif()
endforeach()

foreach(comp IN LISTS ${CMAKE_FIND_PACKAGE_NAME}_comps)
    # All required components are known to exist. The OPTIONAL keyword
    # allows the non-required components to be missing without error.
    include(${CMAKE_CURRENT_LIST_DIR}/MyProj_${comp}.cmake OPTIONAL)
endforeach()
```

A close companion to the config file is its associated version file. If a version file is provided, it is expected to have a name conforming to one of two forms and it must be in the same directory as the config file:

- `<packageName>ConfigVersion.cmake`
- `<lowercasePackageName>-config-version.cmake`

Exclusively for sbmain17@gmail.com Transaction: 0098315591

The form of the version file name generally follows the same form as its associated config file (i.e. `FooConfigVersion.cmake` would go with `FooConfig.cmake`, whereas `foo-config-version.cmake` would typically be paired with `foo-config.cmake`). The purpose of the version file is to inform `find_package()` whether the package meets the specified version requirements. `find_package()` sets a number of variables before the version file is loaded:

- `PACKAGE_FIND_NAME`
- `PACKAGE_FIND_VERSION`
- `PACKAGE_FIND_VERSION_MAJOR`
- `PACKAGE_FIND_VERSION_MINOR`
- `PACKAGE_FIND_VERSION_PATCH`
- `PACKAGE_FIND_VERSION_TWEAK`
- `PACKAGE_FIND_VERSION_COUNT`

These variables contain the version details specified as the `version` argument to `find_package()`. If no such argument was given, then `PACKAGE_FIND_VERSION` will be empty and the other `PACKAGE_FIND_VERSION_*` variables will be set to 0. `PACKAGE_FIND_VERSION_COUNT` holds the count of how many version components have been specified and the rest of the variables have their obvious meaning.

If the `find_package()` call specified a version *range* rather than just a single version, the above version-related variables refer to the lower end of the version range and the following variables will also be set:

- `PACKAGE_FIND_VERSION_RANGE`
- `PACKAGE_FIND_VERSION_RANGE_MIN`
- `PACKAGE_FIND_VERSION_RANGE_MAX`
- `PACKAGE_FIND_VERSION_MIN`
- `PACKAGE_FIND_VERSION_MIN_MAJOR`
- `PACKAGE_FIND_VERSION_MIN_MINOR`
- `PACKAGE_FIND_VERSION_MIN_PATCH`
- `PACKAGE_FIND_VERSION_MIN_TWEAK`
- `PACKAGE_FIND_VERSION_MIN_COUNT`
- `PACKAGE_FIND_VERSION_MAX`
- `PACKAGE_FIND_VERSION_MAX_MAJOR`
- `PACKAGE_FIND_VERSION_MAX_MINOR`
- `PACKAGE_FIND_VERSION_MAX_PATCH`
- `PACKAGE_FIND_VERSION_MAX_TWEAK`
- `PACKAGE_FIND_VERSION_MAX_COUNT`

Exclusively for sbmain17@gmail.com Transaction: 0098315591

CMake 3.19 or later will also set `PACKAGE_FIND_VERSION_COMPLETE` if a version or version range was given. This will be the exact version specification as provided to `find_package()`.

The version file needs to check the requested details against the actual version of the package and then set the following variables:

`PACKAGE_VERSION`

This is the actual package version, which is expected to be in the usual *major.minor.patch.tweak* format (not all components are required).

`PACKAGE_VERSION_EXACT`

Only set to true if the package version and the requested version are an exact match.

`PACKAGE_VERSION_COMPATIBLE`

Only set to true if the package version is compatible with the requested version details. It is up to the package itself how it determines compatibility. For projects that follow semantic versioning principles as covered back in Section 21.3, "Shared Library Versioning", the variable would be set according to the following rules:

- If any version component is missing, treat it as 0.
- If the *major* version components are different, the result is false.
- If the *major* version components are the same, the result is false if the *minor* version component of the package is less than the one required.
- If the *major* and *minor* version components are the same, the result is false if the *patch* version component of the package is less than the one required.
- If the *major*, *minor* and *patch* version components are the same, the result is false if the *tweak* version component of the package is less than the one required.
- For all other cases, the result is set to true.

`PACKAGE_VERSION_UNSUITABLE`

Only set to true if the version file needs to indicate that the package cannot satisfy any version requirement (basically the package doesn't have a version number, so any version requirement should be treated as a failure).

The `find_package()` command will use this information to pass back the following variables to its caller, all of which are analogous to the similar ones it passed in to the version file (the returned values here will be the actual version of the package, not the version requirements passed to the `find_package()` command):

- `<packageName>_VERSION`
- `<packageName>_VERSION_MAJOR`
- `<packageName>_VERSION_MINOR`
- `<packageName>_VERSION_PATCH`
- `<packageName>_VERSION_TWEAK`
- `<packageName>_VERSION_COUNT`

Exclusively for sbmain17@gmail.com Transaction: 0098315591

While projects are free to manually create a version file, a much simpler and most likely more robust approach is to use the `write_basic_package_version_file()` command provided by the `CMakePackageConfigHelpers` module:

```
write_basic_package_version_file(outFile
    [VERSION requiredVersion]
    COMPATIBILITY compat
    [ARCH_INDEPENDENT]   # Requires CMake 3.14 or later
)
```

If a `VERSION` argument is given, the `requiredVersion` is expected to be in the usual *major.minor.patch.tweak* form, but only the *major* part is compulsory. If the `VERSION` option is not given, the `PROJECT_VERSION` variable is used instead (as set by the `project()` command). The `COMPATIBILITY` option specifies a strategy for how the compatibility should be determined. The `compat` argument must be one of the following values (be aware that most of the names are a little misleading):

`AnyNewerVersion`

The package version must be equal to or greater than the specified version. Version ranges will be supported if CMake 3.19.0 or later is being used to write out the package version file.

`SameMajorVersion`

The package version must be equal to or greater than the specified version and the *major* part of the package version number must be the same as the one in the `requiredVersion`. This corresponds to the same compatibility requirements as semantic versioning. Note that this strategy will only support version ranges if CMake 3.19.2 or later is being used to write out the package version file.

`SameMinorVersion`

The package version must be equal to or greater than the specified version and the *major* and *minor* parts of the package version number must be the same as those in the `requiredVersion`. This choice is only supported with CMake 3.11 or later. This strategy will also only support version ranges if CMake 3.19.2 or later is being used to write out the package version file.

`ExactVersion`

The *major*, *minor* and *patch* parts of the package version number must be the same as those in the `requiredVersion`. The *tweak* part is ignored. This strategy is particularly misleading and discussions are in progress to potentially deprecate it in favor of a new, clearer strategy. It does not support version ranges.

The `ARCH_INDEPENDENT` option (available with CMake 3.14 or later) indicates that the package is architecture-independent. Normally, CMake will check that the package was built for the same architecture, typically by verifying that the package's binaries were built for the same pointer size, but specifying the `ARCH_INDEPENDENT` option disables the check.

Architecture-independent packages usually contain no binaries, since those are normally specific to a particular architecture (universal binaries supporting multiple architectures being a notable exception). Such packages often contain documentation, scripts, images and various other non-

Exclusively for sbmain17@gmail.com Transaction: 0098315591

executable files. A package consisting of a header-only library could take advantage of this option to create a single package for all architectures.

The `CMakePackageConfigHelpers` module also provides one other command that may sometimes be useful. The `configure_package_config_file()` command is intended to make it easier for projects to define a relocatable package by providing some path handling conveniences. It is not typically needed for most projects, but when the package config file needs to refer to installed files relative to the base install location rather than the location of the config file itself, it provides a simpler way to do so robustly. The command has the following form:

```
configure_package_config_file(inputFile outputFile
    INSTALL_DESTINATION path
    [INSTALL_PREFIX prefix]
    [PATH_VARS var1 [var2...] ]
    [NO_SET_AND_CHECK_MACRO]
    [NO_CHECK_REQUIRED_COMPONENTS_MACRO]
)
```

The command should be used as a replacement for `configure_file()` to copy a `<Project>Config.cmake.in` file with substitutions. It will replace variables of the form `@PACKAGE_<somevar>@` with the contents of `<somevar>` converted to an absolute path. The original contents are treated as being relative to the base install location. Each variable to be transformed in this way needs to be listed with the `PATH_VARS` option. For this functionality to work, the input file must have `@PACKAGE_INIT@` at or near the top before any use of the variables being replaced (see example further below).

The `INSTALL_DESTINATION` is the directory into which `outputFile` will be installed, relative to the `INSTALL_PREFIX`. When `INSTALL_PREFIX` is omitted, it defaults to `CMAKE_INSTALL_PREFIX`, which is usually the desired value. The `INSTALL_PREFIX` would normally only be provided if the `outputFile` will be used directly in a build tree rather than being installed (i.e. it is used in conjunction with an `export(EXPORT)` command).

The `NO_SET_AND_CHECK_MACRO` and `NO_CHECK_REQUIRED_COMPONENTS_MACRO` options prevent `@PACKAGE_INIT@` from defining some helper functions. Before imported targets became the preferred way to provide package targets, variables needed to be used. To facilitate this, a `set_and_check()` macro was provided by `configure_package_config_file()` which would only set a variable if it was not already defined. Projects providing imported targets should not need this macro and can add the `NO_SET_AND_CHECK_MACRO` to prevent it being defined.

An example should help clarify the typical usage of this command. The package config file would be generated from an input file such as the following:

MyProjConfig.cmake.in

```
@PACKAGE_INIT@

list(APPEND CMAKE_MODULE_PATH "@PACKAGE_cmakeModulesDir@")
# Include the project's export files, etc...
```


Exclusively for sbmain17@gmail.com Transaction: 0098315591

This would then be transformed by the project into the final package config file, including an appropriate replacement for the `cmakeModulesDir` variable:

CMakeLists.txt

```
include(GNUInstallDirs)
include(CMakePackageConfigHelpers)

# This will be used to replace @PACKAGE_cmakeModulesDir@
set(cmakeModulesDir cmake)

configure_package_config_file(
    MyProjConfig.cmake.in MyProjConfig.cmake
    INSTALL_DESTINATION ${CMAKE_INSTALL_LIBDIR}/cmake/MyProj
    PATH_VARS cmakeModulesDir
    NO_SET_AND_CHECK_MACRO
    NO_CHECK_REQUIRED_COMPONENTS_MACRO
)

install(FILES ${CMAKE_CURRENT_BINARY_DIR}/MyProjConfig.cmake
    DESTINATION ${CMAKE_INSTALL_LIBDIR}/cmake/MyProj
    COMPONENT ...
)
```

In the past when all details about a package were provided through variables, it was customary to check whether all required variables were set at the end of the config file before returning. A macro called `check_required_components()` was defined for this purpose, but projects that provide imported targets should perform these checks themselves. The imported targets should only be created if all required components will be found. Otherwise a failed `find_package()` call will still leave behind targets, which would interfere with any later call to `find_package()` for the same package name but with different arguments (e.g. to search in different locations). This makes the `check_required_components()` macro largely redundant.

26.8.2. Config Files For Non-CMake Projects

The config file mechanism isn't restricted to projects built by CMake, it can also be used for non-CMake projects too (although this is not yet all that common). While CMake projects can make use of various CMake features to more easily create the required files, non-CMake projects have to define them manually. For such projects, it is also important to keep the files simple, since they will likely be maintained by people not so familiar with CMake.

A good first step is to initially forgo component support and just make the package available as a simple set of imported targets. For projects that only need to provide libraries, the following example shows a fairly basic config file that should serve as a good starting point. It includes a few different types of libraries for illustration purposes.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

```
# Compute the base point of the install by getting the directory of
# this file and moving up the required number of directories
set(_IMPORT_PREFIX "${CMAKE_CURRENT_LIST_DIR}")
foreach(i RANGE 1 NumSubdirLevels)                                  ①
    get_filename_component(_IMPORT_PREFIX "${_IMPORT_PREFIX}" PATH)
    if(_IMPORT_PREFIX STREQUAL "/")
        set(_IMPORT_PREFIX "")
        break()
    endif()
endforeach()

# Use a prefix specific to this project
set(projPrefix MyProj)

# Example of defining a static library imported target
add_library(${projPrefix}::MyStatic STATIC IMPORTED)
set_target_properties(${projPrefix}::MyStatic PROPERTIES
    IMPORTED_LOCATION "${_IMPORT_PREFIX}/lib/libMyStatic.a"         ②
)

# Example of defining a shared library imported target with version details
add_library(${projPrefix}::MyShared SHARED IMPORTED)
set_target_properties(${projPrefix}::MyShared PROPERTIES
    IMPORTED_LOCATION "${_IMPORT_PREFIX}/lib/libMyShared.so.1.6.3"  ③
    IMPORTED_SONAME   "libMyShared.so.1"                            ④
)

# Another shared library example, this time for Windows
add_library(${projPrefix}::MyDLL SHARED IMPORTED)
set_target_properties(${projPrefix}::MyDLL PROPERTIES
    IMPORTED_LOCATION "${_IMPORT_PREFIX}/bin/MyShared.dll"
    IMPORTED_IMPLIB   "${_IMPORT_PREFIX}/lib/MyShared.lib"          ⑤
)
```

① `NumSubdirLevels` is how many directories this config file is below the base install point. For example, if the file is at `lib/cmake/Foo/FooConfig.cmake`, then `NumSubdirLevels` should be 3.

② The path to the library relative to the base install point (provided by `_IMPORT_PREFIX`).

③ The example shows how the shared library version number would be placed at the end of the file name for platforms such as Linux. This is obviously going to be platform specific.

④ For platforms that support sonames, `IMPORTED_SONAME` is essentially the name that will be embedded in binaries that link to this target. On Apple platforms, this would typically have a form that includes `@rpath` and potentially some subdirectory components.

⑤ For Windows, the location of the import library associated with the DLL must also be provided for anything to be able to link to it. If the intention is only to provide the DLL (e.g. so it is available at run time but not for directly linking against), the `IMPORTED_IMPLIB` can be omitted, but this would be less common.

The above is quite basic and obviously the various `IMPORTED_`⋯ properties would need to be tailored for each platform, but the non-CMake project is free to use whatever mechanisms it finds convenient to produce the installed config file's contents. For added robustness, each imported library should only be added if it does not already exist, as the following demonstrates:

Exclusively for sbmain17@gmail.com Transaction: 0098315591

```
if(NOT TARGET ${projPrefix}::MyStatic)
    add_library(${projPrefix}::MyStatic STATIC IMPORTED)
    set_target_properties(${projPrefix}::MyStatic PROPERTIES
        IMPORTED_LOCATION "${_IMPORT_PREFIX}/lib/libMyStatic.a"
    )
endif()
```

26.9. Executing An Install

An install is typically done as part of creating a final package, but it can also be run on its own. Developers may wish to check the set of files included in an install and where they get installed to, so installing to a temporary staging area can be desirable. Section 26.1.2, "Base Install Location" already discussed how to do this by building the `install` target (or `INSTALL` for some generators).

CMake 3.15 added support for invoking an install directly without going through the build tool. To make use of this capability, the `--install` option can be given as the first argument on the `cmake` command line, followed by the top of the build directory for the CMake project to install. Other command-line arguments can be given to control where to install to, which components to install, which build configuration to use (in the case of multi-configuration generators) and whether to perform stripping before carrying out the install. The ability to specify these additional pieces of information is one of the main reasons to use this method as opposed to building the `install` build target. The following examples demonstrate the convenience and flexibility:

Check development component only with a specific build config

```
cmake --install   /path/to/build/dir    \
      --prefix    /path/to/staging/area \
      --config    Debug                 \
      --component MyProj_Development
```

Install to a mounted device directory for testing, stripping binaries to reduce install size

```
cmake --install /path/to/build.dir \
      --prefix  /mnt/rpi4/staging  \
      --strip
```

It is also possible to do most of the above using `cmake -P` script mode with the `cmake_install.cmake` file at the top of the build directory, but it is less convenient and lacks a documented equivalent for `--strip`.

26.10. Recommended Practices

Installation is a non-trivial topic that requires good planning and an understanding of each intended deployment platform. It is common for a project to initially focus on only a subset of the intended set of platforms, but delaying any planning for installation and deployment can result in having to deal with unexpected complexities and platform differences late in a project's release cycle. Projects should have a clear understanding of the installed file and directory structure, as well as the full set of packaging scenarios that will eventually be supported. This can strongly

Exclusively for sbmain17@gmail.com Transaction: 0098315591

influence project structure, including such fundamental things as how functionality is split between libraries and what symbols need to be visible in the binaries as a result.

Where possible, projects should prefer to follow standard package layouts. The `GNUInstallDirs` module greatly simplifies that task, even for packages on Windows. CMake 3.14 added support for default install locations based on the type of file or target being installed and those defaults are based on the variables defined by `GNUInstallDirs`, so there is increasing momentum toward encouraging projects to use the package layout facilitated by that module. If a standard layout is not suitable, projects may still want to at least consider if a common directory structure can be used across all platforms to simplify application development.

Projects are strongly encouraged to make their packages relocatable. Unless the package needs to be installed to a very specific location, relocatable packages have significant advantages. They offer much greater flexibility to end users, they more easily support a wide range of packaging systems and they are easier to test during development.

The selection of the default install base point is platform-specific and the defaults provided by CMake are not always ideal, but package creation often overrides them anyway. Avoid including a package version number in the install base path, especially for relocatable packages. Prefer to leave that decision up to the user doing the install since different usage scenarios call for different directory structures which might not be compatible with a version-specific path. Projects should also prefer to follow appropriate standards where relevant, such as the Filesystem Hierarchy Standard for Linux (also generally appropriate for most other Unix-based platforms except Apple). If testing out installs locally, consider using the `cmake --install` feature available with CMake 3.15 or later for its improved flexibility and ease of use.

When defining target usage requirements, use the `$<BUILD_INTERFACE:…>` generator expression to properly express the header search paths to be used by the build. For any library target that will be installed, prefer to set the header search path using the `INCLUDES DESTINATION` section of the `install(TARGETS)` command rather than using `$<INSTALL_INTERFACE:…>` generator expressions on the target itself. This can be more convenient and more concise. Ensure that the `INCLUDES DESTINATION` uses a relative path that is relative to the install base point.

```
add_library(Foo ...)

# Not ideal: embeds build paths in installed export files
target_include_directories(Foo
    PUBLIC ${CMAKE_CURRENT_BINARY_DIR}
)

# Better: separate paths for build and install, with the latter added
# as part of the install() command rather than with the target
include(GNUInstallDirs)
target_include_directories(Foo
    PUBLIC $<BUILD_INTERFACE:${CMAKE_CURRENT_BINARY_DIR}>
)
install(TARGETS Foo ...
    INCLUDES DESTINATION ${CMAKE_INSTALL_INCLUDEDIR}
)
```


Exclusively for sbmain17@gmail.com Transaction: 0098315591

Always assign a COMPONENT to each installed entity and use project-specific component names. When the project is used as part of a large project hierarchy, this allows a parent project to control how child components should be treated. A good pattern to follow is <ProjectName>_<ComponentName> (e.g. MyProj_Runtime). When installing export sets, use the same project name as the namespace, with two colons appended (e.g. MyProj::). A similar pattern can be used for target names, using a real target name like MyProj_Algo with an exported name MyProj::Algo. The EXPORT_NAME target property can be used to customize the name appended to the namespace to construct the full exported name for the target. Following these naming conventions will make working with the installed project more intuitive and it will also prevent name clashes with other projects' packages.

If the project provides libraries that other projects are expected to link against, prefer to define separate components for runtime support and for development. This allows a parent hierarchical project to re-use the runtime component to package up just the shared libraries and things needed for execution and avoid packaging development-only entities like static libraries, header files and so on. It also potentially reduces the work of package maintainers (e.g. for Linux distributions) where packages are often split up into runtime and devel packages.

CMake 3.22 added the ability to use symlinks instead of copying files at install time. The feature is controlled by the CMAKE_INSTALL_MODE environment variable and affects the behavior of the file(INSTALL) and install() commands. While there are some scenarios where this functionality may be useful, it comes with a number of caveats and can easily be misused. Consider carefully the caveats mentioned in the CMAKE_INSTALL_MODE documentation before using this feature.

In package config files, always ensure no imported targets are created unless the find_package() call is going to be successful. This means all required components must be available and all required target dependencies should exist before creating any imported targets. To bring in the dependencies, use find_dependency() from the CMakeFindDependencyMacro module rather than calling find_package() from within a package config file, unless the dependency supports package components. If calling find_package() to bring in a dependency, ensure the QUIET and REQUIRED options are passed through correctly to the dependency's find_package() call. Also use the appropriate variables to define success/failure and to report an error message back to the original find_package() command rather than calling message(FATAL_ERROR …) or similar.

If a project wants to include all runtime dependencies in its packages, CMake provides a number of ways to find and install them automatically. That said, most projects will be better off spending the effort to work out their actual dependencies and install them directly. This will ensure that the build process is more predictable and reliable.

If the project does want to use one of the automatic dependency find-and-install methods, the functionality offered by the install(IMPORTED_RUNTIME_ARTIFACTS) and install(RUNTIME_DEPENDENCY_SET) commands is the most advanced and most actively developed. It is also the least mature and does have known issues (see Section 26.4, "Installing Imported Targets" and Section 26.7.1, "Runtime Dependency Sets" for details). If the project is not affected by those deficiencies or is able to use suitable workarounds, this method should still be considered the preferred choice.

The InstallRequiredSystemLibraries module is suitable if only compiler runtime dependencies need to be automatically found and installed. This module allows the project to avoid having to duplicate all the complex logic for finding the appropriate files for different Visual Studio versions, SDKs, toolkit selection, etc. If support for Intel compilers is important, understand the various libraries

Exclusively for sbmain17@gmail.com Transaction: 0098315591

that this module installs by default and decide whether or not these libraries are all needed. Projects using OpenMP in particular will most likely want to use the default install commands rather than define their own so that the required libraries do not have to be manually defined.

Avoid the `BundleUtilities` and `GetPrerequisites` modules. They are difficult to use and have known problems that are unlikely to be fixed. Neither module is being actively developed.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

Chapter 27. Packaging

The creation of release packages is an area where developers frequently feel out of their depth. The various packaging systems, platform differences and conventions can present a very steep learning curve for anyone wanting to master the art of creating robust, well presented packages across multiple platforms. Each package management system invariably uses its own unique form of input specification for what each package contains, how it should be installed, how package components relate to each other, how to integrate with the operating system and so on. Differences between platforms and even between different distributions of the same platform are not always obvious and are frequently only learned after experiencing problems from an unforeseen behavior or constraint (Windows path length restrictions and differing conventions on Linux for system library directory names are great examples of this).

Despite all these differences, there is a substantial degree of commonality in terms of the packaging concepts used. While each system or platform might implement things differently, much of their packaging functionality can be described in a fairly generic way. CMake and CPack take advantage of this and present a well defined interface for specifying these common aspects, which are then translated into the necessary package system input files and commands to produce packages in various formats. This provides a much shorter learning curve for developers, resulting in a relatively quick path to producing packages across the platforms of interest.

CMake and CPack not only abstract away the common aspects of packaging, they also simplify the use of many packager-specific features as well. By providing a simpler interface to these features, CMake and CPack enable developers to exploit more advanced packaging features in a more familiar way. For the most part, this is done by setting a few relevant variables or calling functions with the appropriate arguments, all of which are defined in the documentation for the CPack module and the various package generators.

CPack packaging is implemented internally as one or more installs to a staging area which is then used to produce the final package(s). These installs are controlled by calls to the `install()` command, which were covered in depth in the preceding chapter. This chapter now presents the second half of that process, describing the variables and commands that specify the package meta data and configuration of the packages themselves.

27.1. Packaging Basics

Setting up and executing packaging is handled in a similar way to testing. The `cpack` command line tool reads an input file and produces the appropriate package(s) based on that file's contents. If no input file is explicitly given on the command line, `cpack` will use `CPackConfig.cmake` in the current directory. This input file is most commonly produced by CMake through the inclusion of the `CPack` module, just like how including the `CTest` module generates the input file for `ctest`. Projects can customize the content of the generated packaging input file through CMake variables and commands.

The `CPack` module enables a few default package formats based on the target platform. The set of package formats to be created can be overridden on the `cpack` command line with the `-G` option. If multiple formats should be built, they can be provided as a semicolon-separated list like so:

Exclusively for sbmain17@gmail.com Transaction: 0098315591

```
cpack -G "ZIP;RPM"
```

If the CMake project was configured to use a multi configuration generator like Xcode, Visual Studio or Ninja Multi-Config, `cpack` needs to know which configuration's executables it should package up. This is done by giving a `-C` option to `cpack` (the `-C` option is ignored by single configuration CMake generators):

```
cpack -C Release
```

The `cpack` command supports a few other options, but `-G` and `-C` are two of the more commonly used. Most other details are typically provided through the input file. This is in part because instead of invoking `cpack` directly, developers can build the `package` build target which will first build the default `all` target and then invoke `cpack` with minimal options. It is therefore more convenient for the project to ensure the `cpack` input file defines all required settings. CMake will automatically create the `package` target if the top of the build tree contains a file called `CPackConfig.cmake`.

The easiest way to create the `cpack` input file is by including the `CPack` module, which should only be done once for the entire CMake project. At the point where the `CPack` module is included, the `CPackConfig.cmake` file is written to the top of the build tree (i.e. `CMAKE_BINARY_DIR`). The call to `include(CPack)` is best performed at or near the end of the top level `CMakeLists.txt` file, either directly or through a subdirectory's `CMakeLists.txt`. Making the inclusion conditional on whether the project has a parent also ensures the project only tries to define packaging if it is the top level project.

```
cmake_minimum_required(VERSION 3.0)
project(MyProj)

# ...Define targets, add subdirectories, etc...

# End of the CMakeLists.txt file
if(CMAKE_SOURCE_DIR STREQUAL CMAKE_CURRENT_SOURCE_DIR)
    # include(CPack) will happen inside the following call
    add_subdirectory(packaging)
endif()
```

While defaults are provided for most aspects of the packaging configuration, these defaults are not always appropriate. Most projects will want to set some basic details before including the `CPack` module to provide better alternatives. In particular, it is recommended that the following variables be explicitly set before calling `include(CPack)`:

`CPACK_PACKAGE_NAME`

The package name is one of the more fundamental pieces of metadata. It is used as part of the default file name for packages, it may appear in various places within UI installers and it will most likely be the name that end users will use to refer to the project. Ideally, it would not contain spaces, since spaces are replaced by other characters in some contexts. If this variable is not explicitly set, `CMAKE_PROJECT_NAME` is used as a default.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

CPACK_PACKAGE_DESCRIPTION_SUMMARY

This variable provides a short sentence of no more than a few words about the project. It should be suitable for being shown in lists of packages where space is restricted and it should give end users an idea of what the package is about. It may also be shown to the user in other situations and is only used for informational purposes. From CMake 3.9, the default value is taken from `CMAKE_PROJECT_DESCRIPTION`, whereas for earlier CMake versions the default is an empty string.

CPACK_PACKAGE_VENDOR

The vendor is usually only used for information rather than affecting package structure or behavior, but it is helpful for end users if it is set appropriately. The default value of `Humanity` is not generally suitable for anything other than acting as a placeholder until it is set properly. Prefer to use a real company or organization name rather than a domain name.

CPACK_PACKAGE_VERSION_MAJOR, CPACK_PACKAGE_VERSION_MINOR, CPACK_PACKAGE_VERSION_PATCH

These are used to construct the overall package version and may appear as part of package file names, in package metadata and in installer UIs. The version information is a critical part of packaging that projects should always explicitly set. The default values of 0, 1 and 1 respectively are only helpful as placeholders and should never be relied upon for formal release packages. A convenient pattern is to use the version details provided to the `project()` command:

```
set(CPACK_PACKAGE_VERSION_MAJOR ${PROJECT_VERSION_MAJOR})
set(CPACK_PACKAGE_VERSION_MINOR ${PROJECT_VERSION_MINOR})
set(CPACK_PACKAGE_VERSION_PATCH ${PROJECT_VERSION_PATCH})
```

`cpack` will automatically populate `CPACK_PACKAGE_VERSION` based on these three variables, but this only occurs when `cpack` runs, so `CPACK_PACKAGE_VERSION` won't yet be populated during CMake processing. From CMake 3.12, the default values for these variables are taken from the `CMAKE_PROJECT_VERSION_MAJOR`, `CMAKE_PROJECT_VERSION_MINOR` and `CMAKE_PROJECT_VERSION_PATCH` variables instead, which were only added in CMake 3.12. These variables are set by the `VERSION` details of the `project()` command in the top level `CMakeLists.txt` file, so they are much more likely to provide sensible defaults than the fairly arbitrary pre-3.12 values of 0, 1 and 1. That said, relying on these variables to provide defaults assumes that the project is always the top level project, which might not always be the case. Therefore, it is safer to always explicitly set them to what the project really wants.

CPACK_PACKAGE_INSTALL_DIRECTORY

Some packagers will append this to the base install point to create a package specific directory. Its default value can vary, but may include the package name and version. The presence of the version number in the default value is often undesirable, such as for installers that are able to upgrade a project in-place. To ensure better default behavior, projects may want to set this to the same as `CPACK_PACKAGE_NAME`.

CPACK_VERBATIM_VARIABLES

This variable should always be explicitly set to true. It ensures all contents written to the `cpack` configuration file are properly escaped. The default value is false only to preserve backward compatibility with earlier CMake versions, but the old behavior can lead to an ill-formed configuration file and should not be used.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

More variables will often be set to improve the end user experience, especially for UI installers:

CPACK_PACKAGE_DESCRIPTION_FILE

This is the name of a text file containing a slightly longer description of the project. The contents of the file may be shown in introductory screens of an installer or added to package meta data. Always use an absolute path to the file. As an alternative, the description can be provided directly as the contents of a variable named `CPACK_PACKAGE_DESCRIPTION`. While this was not documented for CMake 3.11 or earlier, it has been supported even from early versions of CMake.

CPACK_RESOURCE_FILE_WELCOME

Some installers show a welcome message in their opening screen. This variable specifies a file name whose contents should be shown for such cases. If it is not set, then for those installers that show a welcome message, CPack provides a default which acts as a placeholder, but it is a relatively poor substitute not suitable for official releases. Projects should always set this if distributing an installer that shows a welcome screen. Always use an absolute path to the file.

CPACK_RESOURCE_FILE_LICENSE

Most UI installers present a license page to the user and may ask them to accept the license before continuing. The text shown for the license is taken from the file named by this variable. Some generic placeholder text is used if the variable is not set, but projects are strongly advised to provide their own more suitable license details. Always use an absolute path to the file.

CPACK_RESOURCE_FILE_README

Some UI installers provide a separate page showing the contents of the file named by this variable. It serves as an opportunity to give the user some information before they proceed with the installation and by default has generic but typically unsuitable text. Projects should prefer to give a file with some more appropriate content via this variable if they intend to create installers which show such pages. Always use an absolute path to the file.

CPACK_PACKAGE_ICON

This variable may also be commonly set, but be aware that most of the package generators have their own different requirements for the format and use of icons within the package and associated places. Some generators ignore this variable, others use it in different ways.

An example that follows the above guidelines may look something like this:

```
set(CPACK_PACKAGE_NAME                MyProj)
set(CPACK_PACKAGE_VENDOR              MyCompany)
set(CPACK_PACKAGE_DESCRIPTION_SUMMARY "CPack example project")
set(CPACK_PACKAGE_INSTALL_DIRECTORY   ${CPACK_PACKAGE_NAME})
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_VERBATIM_VARIABLES          YES)
set(CPACK_PACKAGE_DESCRIPTION_FILE    ${CMAKE_CURRENT_LIST_DIR}/Description.txt)
set(CPACK_RESOURCE_FILE_WELCOME       ${CMAKE_CURRENT_LIST_DIR}/Welcome.txt)
set(CPACK_RESOURCE_FILE_LICENSE       ${CMAKE_CURRENT_LIST_DIR}/License.txt)
set(CPACK_RESOURCE_FILE_README        ${CMAKE_CURRENT_LIST_DIR}/Readme.txt)
include(CPack)
```


Exclusively for sbmain17@gmail.com Transaction: 0098315591

To facilitate running `cpack` with no arguments and the use of the `package` build target, the `CPACK_GENERATOR` variable should be set to the desired package formats. If not set, a fairly conservative default set of generators will be used. Since not all formats are supported or appropriate on all platforms, setting this variable requires logic to specify only those formats that make sense. The following example selects one generic archive format and one native package format for the target platform (if identified):

```
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` module also defines the necessary details that allow a source package to be produced. It creates a `CPackSourceConfig.cmake` file which can be used instead of `CPackConfig.cmake` and when the project is configured to use a Makefile or Ninja generator, a `package_source` build target is defined as well. Producing the source package is relatively straightforward, with either of the following two commands achieving the same thing.

```
# All build generators
cpack -G TGZ --config CPackSourceConfig.cmake

# Makefile and Ninja build generators only
cmake --build . --target package_source
```

The source package contains the entire source directory tree. The `CPACK_SOURCE_IGNORE_FILES` variable can be used to filter out parts of the source tree, holding a list of regular expressions that each full file path will be compared against. All matching files will be omitted from the source package. The default value of this variable ignores repository directories like `.git`, `.svn`, etc. as well as some common temporary files. If a project overrides `CPACK_SOURCE_IGNORE_FILES`, it will need to ensure it also specifies any such relevant patterns. To avoid problems with escaping and quoting in the regular expressions, it is strongly recommended to set `CPACK_VERBATIM_VARIABLES` to true.

```
set(CPACK_VERBATIM_VARIABLES YES)

set(CPACK_SOURCE_IGNORE_FILES
    /\\.git/
    \\.swp
    \\.orig
    /CMakeLists\\.txt\\.user
    /privateDir/
)
```

Exclusively for sbmain17@gmail.com Transaction: 0098315591

27.2. Components

If a project defines no components in any of its `install()` commands, then all package generators will produce a single monolithic package that contains all installed contents. When a project does define components, it provides more flexibility for how it can be packaged. Relationships can also be specified between components, allowing hierarchical component structures to be defined and dependencies between them to be enforced at install time. Each package generator makes use of these component details in different ways, with some creating separate packages for different components, while others present user selectable components in a single UI installer. Some installers even support downloading individual components on demand at install time.

The previous chapter demonstrated how to define components as part of `install()` commands. Those commands only assign content to components, they do not define any other component details. The relationships between components are specified using commands from the `CPackComponent` module, which is automatically included as part of including the `CPack` module. These commands also provide additional metadata for components which some installers use to present information to the user during installation.

The most important command from the `CPackComponent` module is `cpack_add_component()`, which describes a single component:

```
cpack_add_component(componentName
    [DISPLAY_NAME name]
    [DESCRIPTION description]
    [DEPENDS comp1 [comp2...] ]
    [GROUP group]
    [REQUIRED | DISABLED]
    [HIDDEN]
    [INSTALL_TYPES type1 [type2...] ]
    [DOWNLOADED]
    [ARCHIVE_FILE archiveFileName]
    [PLIST plistFileName]
)
```

While all keywords are optional, the `DISPLAY_NAME` and `DESCRIPTION` should at least be provided so that meaningful details are presented to the user during installation and so that non-UI installers have enough metadata for users to understand what a package is for. If the component should only be installed if one or more other components are installed, those components should be listed with the `DEPENDS` option. Note that not all package types fully enforce these dependencies. A component can be placed under a particular group with the `GROUP` option, which can be further described using the `cpack_add_component_group()` command (discussed further below).

If a component should always be installed, the `REQUIRED` keyword should be given. The user will then not be able to disable that component through an installer's UI. Without this keyword, the component can be enabled or disabled by the user, with the default initial state being enabled. To change this default to disabled, add the `DISABLED` keyword. Whether a component is required or not, it can also be hidden from installer UIs by adding the `HIDDEN` keyword. A non-required but hidden component would generally also be disabled and the installer would then only install that component if another enabled component depended on it.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

The remaining options have more specialized effects that apply to only a small number of package generators. An install type is a preset selection of components which can be used to simplify the choices a user has to make at install time. A component can be assigned to any number of install types with the `INSTALL_TYPES` option, where each `type` is a name that is defined separately by the `cpack_add_install_type()` command like so:

```
cpack_add_install_type(typeName [DISPLAY_NAME uiName])
```

The `DISPLAY_NAME` option can be omitted if `typeName` is already sufficiently descriptive, but for install types that should be shown using multiple words, `DISPLAY_NAME` must be used and `uiName` will be a quoted string. There are no predefined install types, but it is common to see packages provide install types with names like `Full`, `Minimal` or `Default`. Of the actively maintained package generators provided by CMake, only NSIS supports the install types feature.

For those generators that support downloadable components, adding the `DOWNLOADED` keyword to `cpack_add_component()` makes the component downloaded on demand rather than being included in the package directly. The `ARCHIVE_FILE` option can be used to customize the file name of the downloadable component. The only actively maintained generator provided by CMake that supports downloadable components is IFW, so discussion of this feature is deferred to Section 27.4.2, “Qt Installer Framework (IFW)”. Similarly, the `PLIST` option (only available with CMake 3.9 or later) is used exclusively by the `productbuild` package generator (see Section 27.4.6, “productbuild”).

If no components are defined with `GROUP` details, the components will act as a simple flat list in most UI installers. When grouping is used, it enables an arbitrarily deep hierarchical structure to be defined instead, where groups can contain components and other groups. A group is defined using the following command from the `CPackComponent` module:

```
cpack_add_component_group(groupName
    [DISPLAY_NAME name]
    [DESCRIPTION description]
    [PARENT_GROUP parent]
    [EXPANDED]
    [BOLD_TITLE]
)
```

This command can appear before or after `cpack_add_component()` calls that refer to the `groupName`. The `DISPLAY_NAME` and `DESCRIPTION` options serve the same purpose as their counterparts in the `cpack_add_component()` command. The `PARENT_GROUP` is the group’s equivalent of the `GROUP` option, allowing it to be placed under another group to support arbitrary group hierarchies. When the `EXPANDED` keyword is given, the group will initially be expanded in the installer UI and the presence of the `BOLD_TITLE` keyword will make that group show up as bold.

Component names should ideally be project specific to allow hierarchical project arrangements to effectively select which components to package and how to present them in installers (or in the case of non-UI installers, how to structure the component-specific packages). Group names are less restrictive, since they may contain components and groups from across different projects. A group name cannot be the same as any component name.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

The effect of both `cpack_add_component()` and `cpack_add_component_group()` is to define a range of component-specific variables in the current scope. The `CPackComponent` documentation lists some of these variables and suggests that the variables can be set directly, but this is not recommended. The commands offer a more robust and more readable way of defining component and group details and should be preferred. They should also be called in the same scope as the `include(CPack)` call, ideally immediately after it. Technically the constraint is not quite as strict as this, but defining the component details in a different scope can be more fragile.

An example should help consolidate some of the above concepts and discussions.

```
set(CPACK_PACKAGE_NAME ...)
# ... set other variables as per earlier example

include(CPack)

cpack_add_component(MyProj_Runtime
    DISPLAY_NAME  Runtime
    DESCRIPTION   "Shared libraries and executables"
    REQUIRED
    INSTALL_TYPES Full Developer Minimal
)
cpack_add_component(MyProj_Development
    DISPLAY_NAME  "Developer pre-requisites"
    DESCRIPTION   "Headers/static libs needed for building"
    DEPENDS       MyProj_Runtime
    GROUP         MyProj_SDK
    INSTALL_TYPES Full Developer
)
cpack_add_component(MyProj_Samples
    DISPLAY_NAME  "Code samples"
    GROUP         MyProj_DevHelp
    INSTALL_TYPES Full Developer
    DISABLED
)
cpack_add_component(MyProj_ApiDocs
    DISPLAY_NAME  "API documentation"
    GROUP         MyProj_DevHelp
    INSTALL_TYPES Full Developer
    DISABLED
)
cpack_add_component_group(MyProj_SDK
    DISPLAY_NAME  SDK
    DESCRIPTION   "Developer tools, libraries, etc."
)
cpack_add_component_group(MyProj_DevHelp
    DISPLAY_NAME  Documentation
    DESCRIPTION   "Code samples and API docs"
    PARENT_GROUP  MyProj_SDK
)
cpack_add_install_type(Full)
cpack_add_install_type(Minimal)
cpack_add_install_type(Developer DISPLAY_NAME "SDK Development")
```


Exclusively for sbmain17@gmail.com Transaction: 0098315591

Project generators can be asked to process components in one of three ways, the choice being controlled by the `CPACK_COMPONENTS_GROUPING` variable which can be set to one of the following values:

`ALL_COMPONENTS_IN_ONE`

A single package with all requested components will be created. Component and group structure is ignored.

`ONE_PER_GROUP`

Each top level component group should create a package. Those components that are not part of a group will also create their own package. This is the default if `CPACK_COMPONENTS_GROUPING` is not set and is usually the desirable arrangement, but for some UI installers it hides components that projects may prefer be shown.

`IGNORE`

Each component creates its own package irrespective of any component groups. This setting can be more suitable for some UI installers to ensure that no components are hidden unless explicitly configured to be so.

Two more variables also affect how generators interpret components. If `CPACK_MONOLITHIC_INSTALL` is set to true, components are disabled completely and all components are installed and bundled into a single package. This is a fairly brutal switch, so test the results carefully on all relevant platforms, paying special attention to look out for any unexpected files. For legacy reasons, each generator also has its own setting for whether or not components are supported by default. This setting can be overridden on a per-generator basis by the `CPACK_<GENNAME>_COMPONENT_INSTALL` variable, which can be set to true or false as needed.

When performing a component-based install, projects are not required to include all components in the final package(s). The set of components that will be included are controlled by the `CPACK_COMPONENTS_ALL` variable, which must be set before the call to `include(CPack)`. When not set, `cpack` packages all components, but the project can explicitly set this variable to only list the components it wants packaged. For example, if a project wanted to control whether documentation and code samples should be packaged, it could be achieved like so:

```
if(NOT MYPROJ_PACKAGE_HELP)
    set(CPACK_COMPONENTS_ALL
        MyProj_Runtime
        MyProj_Development
    )
endif()

include(CPack)
```

Rather than explicitly listing all the components to be packaged, a project may want to install all but a few specific components. The full set of components is available in the read-only pseudo property `COMPONENTS`, which can only be retrieved via the `get_cmake_property()` command. The project can start with that list of components and then remove the unwanted entries.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

```
if(NOT MYPROJ_PACKAGE_HELP)
    get_cmake_property(CPACK_COMPONENTS_ALL COMPONENTS)
    list(REMOVE_ITEM CPACK_COMPONENTS_ALL
        MyProj_Samples
        MyProj_ApiDocs
    )
endif()

include(CPack)
```

The selection of which set of components to install and how the components should be handled may seem a little complex at first. In practice, the main area that causes difficulty is understanding how each package generator handles the different values of `CPACK_COMPONENTS_GROUPING`. The later sections in this chapter explain the behavior of each generator type, but some quick experiments on a test project can often be just as instructional for coming to terms with the effects of the different settings.

27.3. Multi Configuration Packages

CPack is primarily geared towards producing packages for a single build configuration. In most cases, packages are created for the `Release` build type, but for things like SDK projects, it may be desirable to include both debug and release versions of libraries, especially for multi-config generators.

There are two main approaches to creating a multi-config package. The first method can be adapted to work for both single and multi-config generators and has been available in CMake for a long time. It is more complex to set up, but it also supports other interesting scenarios. The second method only supports multi-config generators and is only available with CMake 3.16 or later, but it is very straightforward to setup and use.

27.3.1. Multiple Build Directories

CPack provides the advanced variable `CPACK_INSTALL_CMAKE_PROJECTS` which can be used to incorporate multiple build trees into the one packaging run. It is expected to hold one or more quadruples where each quadruple consists of:

- The build directory.
- The project name (only important for multi configuration generators).
- The component to install. The special value `ALL` means to install the components listed in the `CMAKE_COMPONENTS_ALL` variable. Other values require a similar `CMAKE_COMPONENTS_XXX` variable to be defined which holds just that one component name. For example, if the component to install was called `Runtime`, then a variable `CMAKE_COMPONENTS_RUNTIME` would need to be defined and have the value `Runtime`.
- The relative location within the package to install to. The only safe value for this is a single forward slash (`/`) due to the way different package generators use it.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

The project can define sets of quadruples, one for the release build and the rest for the debug build. The build directory for the release build can simply be `CMAKE_BINARY_DIR`, but for the debug build, a second separate build directory needs to have been created and built.

The debug quadruples would only need to add those components that are different between the two build configurations, but whether using the default `ALL` component or using specific components, special care needs to be exercised to ensure installed files don't unexpectedly overwrite each other. Listing the release component last will ensure that any files that have the same name and install location will end up with the release version when packaged.

```
set(CPACK_COMPONENTS_MYPROJ_RUNTIME     MyProj_Runtime)
set(CPACK_COMPONENTS_MYPROJ_DEVELOPMENT MyProj_Development)

unset(CPACK_INSTALL_CMAKE_PROJECTS)
if(MYPROJ_DEBUG_BUILD_DIR)
    list(APPEND CPACK_INSTALL_CMAKE_PROJECTS
        ${MYPROJ_DEBUG_BUILD_DIR} ${CMAKE_PROJECT_NAME} MyProj_Runtime     /
        ${MYPROJ_DEBUG_BUILD_DIR} ${CMAKE_PROJECT_NAME} MyProj_Development /
    )
endif()

list(APPEND CPACK_INSTALL_CMAKE_PROJECTS
    ${CMAKE_BINARY_DIR} ${CMAKE_PROJECT_NAME} ALL /
)

include(CPack)
```

When using multi configuration generators like Xcode, Visual Studio or Ninja Multi-Config, the `MYPROJ_DEBUG_BUILD_DIR` directory in the above example needs to be configured to support only the `Debug` build type rather than the usual default set. This is the only way to force it to install debug build outputs. When running `cmake` in that debug build directory, explicitly set the `CMAKE_CONFIGURATION_TYPES` cache variable to `Debug` to get the necessary arrangement.

While it is possible to use just the one build directory for multi configuration generators, the techniques to do so are more fragile and complex. In contrast, the above technique works for all build and package generator types. Furthermore, it can be extended to incorporate builds for different architectures or even completely separate projects into one unified package.

27.3.2. Pass Multiple Configurations To cpack

The `cpack` tool provides a `-C` command line option which can be used to specify the configuration to package. This option only has a useful meaning for multi-config generators. From CMake 3.16, more than one configuration can be given to that option as a semicolon-separated list. `cpack` will then add all the listed configurations to the package.

This functionality is only available when invoking `cpack` directly, it is not available using the `package` build target because a build is inherently a single configuration. It should be noted that this means the user is responsible for ensuring that all configurations listed have already been built before invoking `cpack`. A typical example of the steps required looks like this:

Exclusively for sbmain17@gmail.com Transaction: 0098315591

```
cd /path/to/build/directory
cmake --build . --config Debug
cmake --build . --config Release
cpack -C "Debug;Release"
```

A significant advantage of this method is its simplicity. No additional setup is required and the steps are easily scripted. When using a multi-config generator and CMake 3.16 or later, this is likely to be the preferred approach.

27.4. Package Generators

CPack can generate a variety of package formats, each falling into one of the following categories:

Simple archives

Archives can be in a variety of formats, such as zip, tarball, bz2 and so on. They are the most basic of all the package formats, since they are just an archive of files that the user is expected to unpack somewhere on their file system. They are the most widely supported of all the package formats and are the easiest to work with when the end user wants to have multiple different versions of a project available or installed simultaneously.

UI installers

These tend to have deep integration with the target platform, providing features like adding and removing components once installed, integration with desktop menus and so on. They typically present the user with some means of selecting which components to install and are usually very intuitive, so novice users tend to prefer them. CMake supports NSIS and WIX installers on Windows, DragNDrop (i.e. DMG) and productbuild on Mac and the Qt Installer Framework (IFW) on Windows, Mac and Linux. On Mac, some older installer types are still supported, but they should be considered deprecated and are only mentioned briefly in the sections that follow.

Non-UI packages

These are aimed at a specific package manager. RPM and DEB are very popular on Linux, with FreeBSD and Cygwin packages also being supported for their respective platforms.

Niche and product-specific packages

CMake 3.12 added initial support for the NuGet package format for .NET. CMake 3.13 added a special External generator which doesn't produce packages itself, but instead creates a JSON file which some other process can consume to produce packages outside of CPack.

Regardless of which package generators are used, the same `CPackConfig.cmake` file is processed. This doesn't generally present an issue, since generator-specific configuration is normally made possible through generator-specific variables where needed. If certain logic needs to be added for only a particular generator and the existing variables offered by CMake and CPack are insufficient, the `CPACK_PROJECT_CONFIG_FILE` variable can be set to the name of a file that will be included once for each package generator being invoked. Each time it is read, the `CPACK_GENERATOR` variable will hold the name of the generator being processed rather than the whole list of generators. This allows that file to override settings made in `CPackConfig.cmake` for only those specific generators that require it. The full `cpack` run loosely follows the steps in the following pseudo code:

Exclusively for sbmain17@gmail.com Transaction: 0098315591

```
include(CPackConfig.cmake)

function(generate CPACK_GENERATOR)
    # CPACK_GENERATOR is a single generator local to this function scope
    if(CPACK_PROJECT_CONFIG_FILE)
        include(${CPACK_PROJECT_CONFIG_FILE})
    endif()

    # ...invoke package generator
endfunction()

# Here CPACK_GENERATOR is the list of generators to be processed,
# as set by CPackConfig.cmake or on the cpack command line
foreach(generator IN LISTS CPACK_GENERATOR)
    generate(${generator})
endforeach()
```

An example where the above can be useful is to set `CPACK_PACKAGE_ICON` to a generator specific value, since different generators expect this icon to be in different formats and therefore the file name needs to be generator specific.

The remainder of this chapter discusses each of the actively maintained package generators provided by CMake/CPack.

27.4.1. Simple Archives

CPack supports the creation of archives in a number of different formats. The most widely supported are `ZIP` and `TGZ`, the former being common for Windows platforms and the latter producing gzipped tarballs (`.tar.gz` or `.tgz`) that are supported essentially everywhere else. Other available archive formats include `TBZ2` (`.tar.bz2`), `TXZ` (`.tar.xz`), `TZ` (`.tar.Z`) and `7Z` (7zip archives, `.7z`). CMake 3.16 also added support for tarballs using Zstandard compression with the `TZST` (`.tar.zst`) format. For maximum portability, `ZIP` and `TGZ` should generally be preferred, but some of the other formats may produce smaller archives and may be suitable for platforms where those formats are commonly supported.

A self-extracting archive format is also supported by `cpack`. This can be requested using the generator name `STGZ`, which produces a Unix shell script with the archive embedded at the end of that script. This can be thought of as a form of console-based UI installer, but in practice it offers only very basic functionality and users may prefer a simple archive that they can unpack themselves.

For legacy reasons, archive generators have components disabled by default. To enable component-based archive creation, `CPACK_ARCHIVE_COMPONENT_INSTALL` must be set to true and then `CPACK_COMPONENTS_GROUPING` will determine the set of archive files that will be generated.

When performing a non-component install, the final package file name can be controlled using the `CPACK_ARCHIVE_FILE_NAME` variable. For component-based installs, the name of each component's package is controlled by `CPACK_ARCHIVE_<COMP>_FILE_NAME`, where `<COMP>` is the uppercased component or group name. The appropriate archive extension will be appended to the specified file name (i.e. `.tar.gz`, `.zip`, etc.).

Exclusively for sbmain17@gmail.com Transaction: 0098315591

A common convention for archive files is to make the top level of the extracted directory structure be the same as the name of the archive file without the file extension (i.e. the same as `CPACK_PACKAGE_FILE_NAME`). For non-component installs, this is already the default behavior for the archive generators, but for multi component packages, no top level directory is used by default. Projects can enforce a common top level directory for component archives by setting `CPACK_COMPONENT_INCLUDE_TOPLEVEL_DIRECTORY` to true. Since this variable is shared by all package generators, a generator specific override would be the most appropriate way to do this:

CMakeLists.txt

```
set(CPACK_PROJECT_CONFIG_FILE
    ${CMAKE_CURRENT_LIST_DIR}/cpackGeneratorOverrides.cmake
)
```

cpackGeneratorOverrides.cmake

```
if(CPACK_GENERATOR MATCHES "^(7Z|TBZ2|TGZ|TXZ|TZ|TZST|ZIP)$")
    set(CPACK_COMPONENT_INCLUDE_TOPLEVEL_DIRECTORY YES)
endif()
```

Developers should note that some archive formats, platforms and file systems have limitations on the length of file names and paths. For example, POSIX.2 requires file names to be 100 characters or less and paths to be 255 characters or less for the extended tar interchange format, while older tar formats may restrict the entire path to 100 characters or less. When unpacking an archive onto an eCryptFS file system, file names have an empirically derived limit of about 140 characters. Unpacking on Windows can have a 260 character path length limit, depending on certain settings and OS version. UTF-8 file names and paths further complicate the picture and may shorten the effective character limits even more. With these constraints in mind, projects should avoid using long paths and file names in their package contents. These restrictions are most evident with archive package types, but since other non-archive formats also use archives internally and deploy to systems with these restrictions, shorter paths and file names should be preferred in general.

27.4.2. Qt Installer Framework (IFW)

The IFW package generator offers the broadest platform support of all UI-based package formats provided by CPack. Installers can be built for Windows, Mac and Linux from the same configuration details, making it a good choice when a project wants to have a consistent UI installer across all major desktop platforms. It also has easy to use localization of component and group display names and descriptions as well as extensive customizability.

The defaults for the UI appearance and installer icons are often sufficient, but some projects may want to customize a few aspects to improve the branding, especially around the use of icons. The `CPACK_PACKAGE_ICON` variable is ignored for this generator, which relies instead on three separate IFW-specific variables to control the icons for different contexts:

- `CPACK_IFW_PACKAGE_ICON` (`.ico` for Windows, `.icns` for Mac, ignored for Linux)
- `CPACK_IFW_PACKAGE_WINDOW_ICON` (always `.png`)
- `CPACK_IFW_PACKAGE_LOGO` (preferably `.png`)

Exclusively for sbmain17@gmail.com Transaction: 0098315591

Unfortunately, these variables are not handled consistently between platforms, so it can be difficult to set them correctly. For simplicity, it may be preferable to set all three to the same image, albeit potentially in different formats and/or sizes. Testing on each platform of interest is recommended to ensure the installer presents itself as expected. The following example shows how such a configuration may be specified:

```
# Define generic setup for all generator types...

# IFW-specific configuration
if(WIN32)
    set(CPACK_IFW_PACKAGE_ICON ${CMAKE_CURRENT_LIST_DIR}/Logo.ico)
elseif(APPLE)
    set(CPACK_IFW_PACKAGE_ICON ${CMAKE_CURRENT_LIST_DIR}/Logo.icns)
endif()
set(CPACK_IFW_PACKAGE_WINDOW_ICON ${CMAKE_CURRENT_LIST_DIR}/Logo.png)
set(CPACK_IFW_PACKAGE_LOGO        ${CMAKE_CURRENT_LIST_DIR}/Logo.png)

include(CPack)
include(CPackIFW)
# Define components and component groups...
```

Component-based installation is enabled by default for the IFW generator. A single installer is always produced, but `CPACK_COMPONENTS_GROUPING` controls how much of the component hierarchy is shown to the user:

`ALL_COMPONENTS_IN_ONE`

No component hierarchy is shown, the default enabled components will always be installed.

`ONE_PER_GROUP`

Only the first level of groups is shown along with any components that do not belong to any groups. Subgroups and components under any group will be hidden.

`IGNORE`

All components that are not explicitly hidden will be shown regardless of where they are in the group hierarchy. This is likely to be the option most projects will want to use.

Components and groups can be further configured beyond what the generic commands provide:

```
cpack_ifw_configure_component(componentName
    [NAME componentNameId]
    [DISPLAY_NAME displayName...]
    [DESCRIPTION description...]
    [VERSION <version>]
    [DEPENDS compId1 [compId2...] ]   ①
    [REPLACES compId3 [compId4...] ]
    # Other options not shown
)
# The cpack_ifw_configure_component_group() command supports the same options
```

① `DEPENDENCIES` is also accepted, but prefer `DEPENDS` for consistency with other CMake commands.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

The DISPLAY_NAME and DESCRIPTION of each component or group can be given alternative contents for different languages and locales. These two options accept a list of pairs where the first value of a pair is the language or locale ID and the second value is the text for that language. The first value in the list can be given without a preceding language or locale ID and it will be used as the default text if none of the languages or locale IDs match the user's current setting at install time.

```
cpack_ifw_configure_component(MyProj_Docs
    DISPLAY_NAME Documentation
                 de Dokumentation
                 pl Dokumentacja
)
cpack_ifw_configure_component_group(MyProj_Colors
    DISPLAY_NAME en    Colors
                 en_AU Colours
    DESCRIPTION  en    "Available color palettes"
                 en_AU "Available colour palettes"
)
```

The VERSION option allows per-component and per-group version numbers to be specified. This is used by online installers to determine whether an update is available (see further below). If VERSION is not given, it defaults to CPACK_PACKAGE_VERSION.

The DEPENDS option is analogous to the same option in cpack_add_component() except that the form of the compId1⋯ entries is different. These need to follow the QtIFW style, which is a hierarchical string rather than a raw componentName. Each level of the grouped hierarchy is dot-separated, as demonstrated by the following example:

```
include(CPack)
include(CPackIFW)

cpack_add_component(foo GROUP groupA)
cpack_add_component(bar GROUP groupB)

cpack_add_component_group(groupA)
cpack_add_component_group(groupB)

cpack_ifw_configure_component(bar DEPENDS groupA.foo)
```

One can also append a version requirement to the DEPENDS value. This can be achieved by appending a separator (discussed shortly), an operator (one of =, <, <=, > or >=) and the version number. Technically the QtIFW format allows the operator to be omitted, which should result in using a default operator of =. CMake 3.20 and earlier contains a bug which causes the version number to be dropped when no operator is given. Therefore, prefer to always specify an operator if including a version constraint on an IFW component dependency.

The separator between the package name and operator can be either a colon (:) or a hyphen (-). Colons are more readable and have the advantage that component and group names can contain hyphens. A colon can only be used as the separator if using CMake 3.21 or later and QtIFW 3.1 or later. If using older CMake or QtIFW versions, a hyphen must be used as the separator and the component and group names cannot contain any hyphens.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

```
# Colons as separators, supports hyphens in names
# NOTE: Requires CMake 3.21+ and QtIFW 3.1+.
cpack_ifw_configure_component(baz DEPENDS a-b-c:=7.8.2)
cpack_ifw_configure_component(baz DEPENDS old-thing:<3)
cpack_ifw_configure_component(baz DEPENDS newbie:>=6)

# Hyphens as separators, no hyphens in names
cpack_ifw_configure_component(baz DEPENDS exacto-=7.8.2)
cpack_ifw_configure_component(baz DEPENDS oldie-<3)
cpack_ifw_configure_component(baz DEPENDS newbie->=6)
```

The name used internally within the installer for a component can be overridden with the `NAME` option. This name would be used to identify the component in `DEPENDS` arguments and also when checking if a newer version of a component is available. A top level group name can be set with the `CPACK_IFW_PACKAGE_GROUP` variable and is often set to a reverse domain name to ensure component names don't clash in large, multi-vendor installers. This top level group name must then be included when listing dependencies with the `DEPENDS` option, as the following modification of the earlier example shows:

```
set(CPACK_IFW_PACKAGE_GROUP com.examplecompany.product)

include(CPack)
include(CPackIFW)

cpack_add_component(foo GROUP groupA)
cpack_add_component(bar GROUP groupB)

cpack_add_component_group(groupA)
cpack_add_component_group(groupB)

cpack_ifw_configure_component(bar
    DEPENDS com.examplecompany.product.groupA.foo
)
```

`CPACK_IFW_PACKAGE_GROUP` is just one example of a large number of extra variables that can be set to provide IFW-specific configuration. Such variables should be set before `include(CPackIFW)` is called and can modify the appearance and behavior of the installer in a variety of ways. The `CPackIFW` module documentation provides a complete listing of all supported variables and their effects, many of which have analogous settings in the QtIFW product's native configuration settings. Most of those variables have sensible defaults and should be seen more as opportunities for customization rather than things that need to be set. One exception to this is the variables relating to the name of the maintenance tool installed along with the rest of the product, which allows the user to modify the set of installed components or remove the product completely. By default, this tool is given the name `maintenancetool`, but this gives no indication of what the tool relates to. On some platforms, the tool name can show up in desktop or application menus and the default name can be confusing for users. Therefore, projects should provide a more specific name, which can be done like so:

Exclusively for sbmain17@gmail.com Transaction: 0098315591

```
set(CPACK_IFW_PACKAGE_MAINTENANCE_TOOL_NAME
    ${PROJECT_NAME}_MaintenanceTool
)
set(CPACK_IFW_PACKAGE_MAINTENANCE_TOOL_INI_FILE
    ${CPACK_IFW_PACKAGE_MAINTENANCE_TOOL_NAME}.ini
)
include(CPackIFW)
```

The `.ini` file is used by the installer to maintain state information between invocations. Setting the name of the `.ini` file is optional, but making the name consistent with the installer itself is preferable. With the above settings, the user will see a name that relates to the project if the maintenance tool shows up in their desktop or applications menu.

A significant feature of the IFW generator is its ability to create online installers. Some or all components can be downloaded on demand instead of bundling them as part of the installer. This is particularly advantageous if some optional components are large. An added benefit of an online installer is that individual components can be upgraded if newer versions are made available from the online repositories, which provides a very convenient upgrade path. Users run the maintenance tool which contacts the set of online repositories to determine the available components and their versions. Individual components can then be added, removed or upgraded as desired.

The first step in configuring a project to support downloadable components is to specify where the installer will download them from. A primary default repository is specified with the generic `cpack_configure_downloads()` command:

```
cpack_configure_downloads(baseUrl
    [ALL]
    [ADD_REMOVE | NO_ADD_REMOVE]
    [UPLOAD_DIRECTORY dir]
)
```

The `baseUrl` is the location where the installer will look for downloadable components. The installer will expect to find a file called `Updates.xml` under that location. If the `ALL` keyword is present, all components are treated as downloadable regardless of whether they were explicitly marked as to be downloadable or not. This is a convenient way of making a fully online installer with no embedded packages, which yields the smallest possible installer.

The `ADD_REMOVE` keyword directs the installer to make the package available to Windows' Add/Remove Programs functionality, which will then run the maintenance tool when the user elects to modify the package through that part of the Windows system settings. The `ALL` keyword implies `ADD_REMOVE`, but giving `NO_ADD_REMOVE` overrides that behavior.

The `UPLOAD_DIRECTORY` option is used by other CPack generator types that support downloadable components (although none of those are actively maintained), but it is ignored by the IFW generator. When `cpack` runs, it creates downloadable packages in a separate directory so that the contents of that whole directory can be uploaded to the `baseUrl` location (which must be done manually). The `UPLOAD_DIRECTORY` option is intended to allow the project to override where this separate directory is located, but the IFW generator always creates a directory called `repository` located multiple levels deep under the base `_CPack_Packages` directory.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

The IFW generator allows projects to specify additional repositories for the maintenance tool and installer to access. This can be useful if different components are provided by different vendors or where some components have a different release schedule to others.

```
cpack_ifw_add_repository(repoName
    URL baseUrl
    [DISPLAY_NAME displayName]
    [DISABLED]
    [USERNAME username]
    [PASSWORD password]
)
```

The `repoName` is an internal tracking name and the `baseUrl` has a similar meaning as for `cpack_configure_downloads()`. The `DISPLAY_NAME` option should generally be used to give a meaningful name, otherwise the `baseUrl` is shown as the repository name, which tends to be less user friendly. If the repository needs a user name and password, it can be supplied, but keep in mind that the password will be stored unencrypted and should be considered insecure. The `DISABLED` keyword indicates that the repository should be disabled by default, but the user can enable it in the installer or maintenance tool's UI.

An example of a main repository for release packages and a secondary repository for preview packages (disabled by default) could be configured like this:

```
include(CPack)
include(CPackIFW)

cpack_configure_downloads(https://example.com/packages/product/release ALL)
cpack_ifw_add_repository(secondaryRepo
    DISPLAY_NAME "Preview features"
    URL          https://example.com/packages/product/preview
    DISABLED
)
```

Unfortunately, the `cpack_configure_downloads()` command does not currently support specifying a display name, so the main URL it supplies will always be shown as a bare URL rather than a more user friendly name.

Starting with QtIFW 4.0, the generated installers support unattended command-line installs. No extra configuration is needed on the part of the project, the developer just has to ensure that QtIFW 4.0 or later is used to generate the installer. The ability to perform scripted installs on the command line can be an important feature for some users, so it is highly recommended to use QtIFW 4.0 or later, if possible.

One drawback of this package generator is that the installer produced has extra overhead compared to most other generator types. It includes the Qt support needed for the installer's interface, networking and so on. This can make the size of even a trivial installer 18Mb or more, compared to a few hundred kB for other generator types.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

The above discussion only covers the main aspects of the IFW generator, there are considerably more capabilities available which allow projects to customize the installer and maintenance tool extensively. For many projects, the above functionality already allows flexible, robust and cross-platform installers to be created. If further tailoring is needed, the features presented will serve as a solid base on which to extend.

27.4.3. WIX

The WIX package generator produces `.msi` installers for Windows using the WiX toolset. Compared to the IFW package generator, it has a similar degree of UI customizability and offers the following advantages:

- Support for unattended, command-line installs is mature. It is provided through an option to the `msiexec` tool.
- Installers are tightly integrated into Windows' Add/Remove functionality.
- The default appearance should be familiar to most users.

On the other hand, it has the following disadvantages compared to IFW:

- No simple, direct way of providing localized component names and descriptions.
- `CPACK_WIX_COMPONENT_INSTALL` and `CPACK_COMPONENTS_GROUPING` are both ignored (see below).
- No support for downloadable components.
- Multiple versions with the same upgrade GUID cannot be installed simultaneously (see below). Each install replaces the previous one, even if in a completely different directory.

By default, the WIX generator produces a component-based package which will always be presented in the UI as though `CPACK_COMPONENTS_GROUPING` had been set to `IGNORE`. If a component-based package is undesirable, `CPACK_MONOLITHIC_INSTALL` can be set to true, but then all defined components are always installed. It is not possible to only include some components in a monolithic installer and if `CPACK_COMPONENTS_ALL` is set, CMake will issue a warning and ignore `CPACK_COMPONENTS_ALL`.

A key part of a WIX installer is that it contains a product GUID and an upgrade GUID. If any other installed package has the same upgrade GUID, that other package will be upgraded rather than installing the new package as a separate product. If the upgrade GUIDs are the same but the product GUIDs are different, then the upgrade is considered a major upgrade and the new installer will completely replace the old package. Where the product GUID is also the same, the new installer should be able to perform a minor upgrade as long as the installer reports a newer version number than the currently installed package. Service packs are an example where the same product GUID is maintained as the base version they apply to. Unless creating a fairly advanced installer or packaging strategy, projects will typically need to change the product GUID with each release, as the constraints from Windows itself for keeping the same product GUID from one package to another are fairly stringent.

CPack provides support for setting the product and upgrade GUIDs. The `CPACK_WIX_PRODUCT_GUID` and `CPACK_WIX_UPGRADE_GUID` variables can be set before calling `include(CPack)` to control them manually, or they can be left unset to allow `cpack` to generate new values each time it is invoked. For the product GUID, this automatic generation is likely to be the desired behavior, but the upgrade GUID

Exclusively for sbmain17@gmail.com Transaction: 0098315591

should ideally never change for the life of the product. Projects should obtain a GUID and set `CPACK_WIX_UPGRADE_GUID` to that value, then ideally never change it again. This will ensure all future releases are able to upgrade older releases seamlessly. The actual GUID can be obtained by a variety of means such as command line tools, web-based UUID generators or even with CMake itself using the `string(UUID)` command. For some products, it may make sense for the upgrade GUID to change with each major release to allow an older major release to co-exist with a newer one, thereby facilitating the users' migration path.

One of the criteria around when a product GUID must change is if the name of the `.msi` file changes. Since the installer's file name would typically include some version details, this means each release would be considered a major upgrade. If the user installs the new version, it would completely replace any previously installed version. The new version can be installed to a different directory and the old one would be removed. It may be tempting to then use a default installation directory (controlled by `CPACK_PACKAGE_INSTALL_DIRECTORY`) that includes a version number, but users would likely prefer the default directory to stay the same across upgrades. The default directory should ideally only change if the upgrade GUID changes, since that is the identifier that provides the continuity from one version to another.

When installing a new package and another package with the same upgrade GUID is already installed, a check is made between the versions. Only if the new package is of a later version will the upgrade be allowed to proceed. Only the first three version number components are considered in this test, so versions 2.7.4.3 and 2.7.4.9 would be considered the same version from an upgrade perspective. Projects intending to use the WIX generator should therefore avoid using more than three version number components. If allowing `CPACK_PACKAGE_VERSION` to be automatically set from the individual `CPACK_PACKAGE_VERSION_xxx` version parts, this will already be enforced.

Most of the UI defaults are acceptable for a basic WIX package. Projects may want to provide a product icon to use in place of the generic MSI installer icon for improved branding in the Add/Remove area, but the defaults are otherwise generally acceptable. The following example shows basic configuration of a WIX installer.

```
# Define generic setup for all generator types...

# WIX-specific configuration
set(CPACK_WIX_PRODUCT_ICON ${CMAKE_CURRENT_LIST_DIR}/Logo.ico)
set(CPACK_WIX_UPGRADE_GUID XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX)

include(CPack)
# Define components and component groups...
```

27.4.4. NSIS

The NSIS package generator produces installer executables for Windows using the Nullsoft Scriptable Install System. It shares a number of similar characteristics with the IFW and WIX generators, including a degree of UI customizability and support for component hierarchies. Advantages of the NSIS generator include:

- Support for unattended, command-line installs is mature and provided directly by the installer executable.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

- It is the only actively maintained CPack generator that supports install types.
- Pre/post-install and pre-uninstall commands are directly supported, although these must be implemented as NSIS commands.

The NSIS generator has a few drawbacks:

- `CPACK_NSIS_COMPONENT_INSTALL` and `CPACK_COMPONENTS_GROUPING` are both ignored. The NSIS generator has the same restrictions as the WIX generator in this regard.
- No support for downloadable components.
- Once a product is installed, users cannot change the set of installed components without redoing the install.
- Only basic UI customization is supported and there is no direct support for localization of any UI contents. These are limitations of CPack's generator, not of NSIS itself, which does offer some facilities via its own native scripting language.
- Although it is possible to install different versions to different locations, they share registry details and so are not fully isolated from each other. Only one version will show up in the Add/Remove area of the Windows settings.

By default, these installers will only perform an upgrade of an existing product installation if the new package is installed to the same directory as the old one. Projects can set the `CPACK_NSIS_ENABLE_UNINSTALL_BEFORE_INSTALL` variable to true to force the installer to check the registry for an existing installation of the package first. This check does not rely on the install location, so it is a more reliable way to check for an existing installation to be upgraded. As a result, setting this variable to true is recommended for most projects.

NSIS installers benefit from overriding the default appearance in a number of areas. The icons used for the installer, uninstaller and the product itself as shown in the Add/Remove area should be set, as the defaults are either of low quality or produce blank boxes. The name displayed for the product should also be explicitly set to avoid inappropriate default text supplied by CPack. The following example shows a basic configuration with overrides to avoid the defaults that most projects would find unsuitable.

```
# Define generic setup for all generator types...

# NSIS-specific configuration
set(CPACK_NSIS_MUI_ICON    ${CMAKE_CURRENT_LIST_DIR}/InstallerIcon.ico)   ①
set(CPACK_NSIS_MUI_UNIICON ${CMAKE_CURRENT_LIST_DIR}/UninstallerIcon.ico) ②
set(CPACK_NSIS_INSTALLED_ICON_NAME bin/MainApp.exe)     ③
set(CPACK_NSIS_DISPLAY_NAME        "My Project Suite")  ④
set(CPACK_NSIS_PACKAGE_NAME        "My Project")        ⑤
set(CPACK_NSIS_ENABLE_UNINSTALL_BEFORE_INSTALL YES)

include(CPack)
# Define components and component groups...
```

① The icon used for the installer itself. Windows may overlay further content to indicate that the installer requires administrator privileges. Use an absolute path to ensure NSIS can find the icon when creating the installer.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

② The icon used for the uninstaller that will be copied to the installation directory. Again, use an absolute path to the icon.

③ This controls the icon used for the product in the Add/Remove area. It must be a path to either an icon file (`.ico`) or an executable that has an embedded application icon of its own. The path should be to the *installed* location, relative to the base point of the install.

④ The name shown for the package in the Add/Remove area only.

⑤ The name used in many places in the installer's UI and also in the title bar during installation. The word `Setup` may be appended to it in some contexts.

CMake 3.17 added a few more customization options. The default welcome and finishing titles can be overridden using the variables `CPACK_NSIS_WELCOME_TITLE` and `CPACK_NSIS_FINISH_TITLE` respectively. The header image shown on each page of the installer can be specified with the `CPACK_NSIS_MUI_HEADERIMAGE` variable. CMake 3.20 added support for the `CPACK_NSIS_BRANDING_TEXT` and `CPACK_NSIS_BRANDING_TEXT_TRIM_POSITION` variables, which can be used to replace the generic *Nullsoft Install System* message along the bottom of the installer pages. If the package does not need to show a license page during installation, the `CPACK_NSIS_IGNORE_LICENSE_PAGE` variable can be set to true with CMake 3.22 or later to disable it.

Another customization made available by CMake 3.17 is the ability to change the name of the uninstaller from its generic default. The `CPACK_NSIS_UNINSTALL_NAME` variable can be used to specify a different name. This can improve the usability of the uninstaller by making its relationship to the package more obvious. This makes the uninstaller more identifiable in a list of running processes or when the user is switching between running applications.

An important change that also occurred in CMake 3.17 was the minimum NSIS version was raised to 3.0 (previously it was 2.09). The CMake 3.17.0 release did not check the NSIS version and would fail with a NSIS parsing error if the NSIS version was not 3.0 or later. From CMake 3.17.1, the updated minimum NSIS version is properly checked and reported.

With CMake 3.18 or later, installers can be made DPI-aware by setting `CPACK_NSIS_MANIFEST_DPI_AWARE` to true. Note that icon handling and some NSIS plugins might not yield acceptable results in all cases, so it is advisable to test the installer at different DPI settings if enabling this option.

27.4.5. DragNDrop

On Mac, products are commonly distributed as a `.dmg` file. These act like a disk image and can contain anything from a single application through to a whole suite of applications, documentation links and so on. A symlink to the `/Applications` area is frequently provided as part of the image so that users can easily drag applications onto it to install them, hence the name DragNDrop for this generator type. Configuration variables specific to this generator type use `DMG` in their name rather than `DRAGNDROP`, but note that `cpack` will only recognize `DragNDrop` as the name of the generator itself.

The `.dmg` format is closer to an archive than a UI installer. Components are used to control whether one or multiple `.dmg` files are created and what each `.dmg` file contains, but there is no install-time UI to choose components. The user is expected to open the `.dmg` file(s) and drag the contents to the desired location to install them. `CPACK_COMPONENTS_ALL` controls which components are installed and the `CPACK_COMPONENTS_GROUPING` variable controls how those components are distributed between `.dmg` file(s) as follows:

Exclusively for sbmain17@gmail.com Transaction: 0098315591

`ALL_COMPONENTS_IN_ONE`

All components will be included in a single `.dmg` file.

`ONE_PER_GROUP`

Each top level component group and each component not in a group will be put in its own separate `.dmg` file.

`IGNORE`

Each component will be put in its own separate `.dmg` file and all component groups will be ignored.

When using CMake 3.17 or later, the file name for each component's `.dmg` file can be specified using variables of the form `CPACK_DMG_<component>_FILE_NAME` if required (only relevant for the `ONE_PER_GROUP` or `IGNORE` cases).

This package generator type typically requires little customization beyond the defaults. The size and layout of the Finder window displayed when the disk image is opened can be controlled by providing a custom `.DS_Store` file. The project will need to either prepare such a file manually using an example folder containing the same things as the final disk image, or it can be created programmatically in AppleScript. The `CPACK_DMG_DS_STORE` variable can be used to name a pre-prepared `.DS_Store` file or `CPACK_DMG_DS_STORE_SETUP_SCRIPT` can point to an AppleScript file to be run at package generation time. For either case, a background image can be set with the `CPACK_DMG_BACKGROUND_IMAGE` variable if desired, but leaving the background at the blank default is relatively common. For cases where the disk image should not provide a symlink to the `/Applications` folder, the project should set `CPACK_DMG_DISABLE_APPLICATIONS_SYMLINK` to true.

An icon can be specified for the disk image by setting `CPACK_PACKAGE_ICON` to an icon in `.icns` format. This icon is only used to represent the `.dmg` file when mounted, not for the `.dmg` file itself. The specified icon may show up in the Finder title bar or certain Finder views, but it is otherwise not a prominently displayed icon.

Limited language localization is provided through the `CPACK_DMG_SLA_LANGUAGES` and `CPACK_DMG_SLA_DIR` variables. These can be used to provide specific phrases used during the license agreement phase of opening the disk image and to provide a localized version of the license agreement. See the DragNDrop generator's documentation for how these two variables are used and the requirements around the language files that need to be provided.

CMake 3.21 added the ability to specify the file system format used for the `.dmg` image. The `CPACK_DMG_FILESYSTEM` variable can be set to any value that the `hdiutil -fs` option supports, but the most typical values are either `APFS` or `HFS+`. If this variable is not set, the default is `HFS+`. Note that APFS file systems are only officially supported on macOS 10.13 or later.

The Bundle generator type is related to the DragNDrop generator. It uses the same set of `DMG` variables, plus some of its own. The Bundle generator was originally intended for producing a single app bundle potentially for submission to the Apple App Store. These days, such app bundles are better prepared during the build itself using CMake's Xcode generator, as this more closely follows the process expected by Apple. See Chapter 23, *Apple Features* for the recommended way of preparing such app bundles rather than using the CPack Bundle generator type.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

27.4.6. productbuild

An alternative to the DragNDrop generator is productbuild. Instead of producing a `.dmg` disk image, it produces a `.pkg` package for use with the macOS Installer app. `CPACK_COMPONENTS_GROUPING` is ignored and the installer always behaves as though this variable had been set to `IGNORE`. `CPACK_MONOLITHIC_INSTALL` should not be set to true with this generator, as doing so can produce broken installers. Installer types are not supported and there is very little ability to customize the UI, although the defaults are typically sufficient anyway.

Compared to the IFW generator, the main advantage of productbuild is the ability to sign the installer. This is easily configured by setting the `CPACK_PRODUCTBUILD_IDENTITY_NAME` (and also `CPACK_PRODUCTBUILD_KEYCHAIN_PATH` if required) to the signing details. Often just specifying the default identity is enough, which can be done like so:

```
set(CPACK_PRODUCTBUILD_IDENTITY_NAME "Developer ID Installer")
include(CPack)
```

The productbuild generator lacks support for downloadable components, so the creation of online installers is not possible. Upgrades are handled by replacing the previous contents of an existing install. Like for NSIS installers, the set of installed components cannot be modified without reinstalling the product. It is also not typically possible to install multiple versions simultaneously to different directories.

Installers produced by the productbuild generator are relocatable by default. What this means is that when the package is installed on an end user's machine, if the OS knows of an app bundle with the same name as one of the apps provided by the package, the installer will overwrite that existing app no matter where it is on the file system. The app will not be installed to the default `/Applications` area in these cases, which usually means it also won't show up in places where the user expects it to.

The above situation commonly arises for developers on the machine they are using to build and test packages. The app bundle produced by the build is known to the OS, so when installing the package, the build tree's app bundle is used as the install location for that app instead of the expected location in `/Applications`. There will also be another copy of the app in the `_CPack_Packages` staging directory of the build tree which can yield similar behavior. To properly test the installer, all copies of the app bundles being installed would need to be removed from the developer's machine first before running the installer.

One workaround to the above relocation problem is to mark components as not relocatable. This prevents the installer from selecting the location of an existing app bundle. The trade-off is that it also prevents the user from moving app bundles around should they so wish.

To make a component non-relocatable, a custom plist file needs to be provided for each component. This is achieved by using the `PLIST` option with the `cpack_add_component()` command:

Exclusively for sbmain17@gmail.com Transaction: 0098315591

```
cpack_add_component(MyProj_Runtime
   ...   # Other options
   PLIST runtime.plist
)
```

The plist file should be obtained by using the `--analyze` option with the `pkgbuild` command and then updating that file as needed. The verbose output of a `cpack` command for the project can also be helpful:

```
cpack -G productbuild -V
```

A typical plist file might look something like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
    "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<array>
    <dict>
        <key>BundleHasStrictIdentifier</key>
        <true/>
        <key>BundleIsRelocatable</key>
        <true/>
        <key>BundleIsVersionChecked</key>
        <true/>
        <key>BundleOverwriteAction</key>
        <string>upgrade</string>
        <key>RootRelativeBundlePath</key>
        <string>Applications/MyApp.app</string>
    </dict>
</array>
</plist>
```

Change the `BundleIsRelocatable` dictionary item to `false` to prevent the OS from relocating the app on install. There will be one `<dict></dict>` section for each app bundle in the component.

The productbuild generator should be considered a replacement for the older and no longer supported PackageMaker generator. Apple no longer provides the PackageMaker app, so developers using newer versions of macOS must use productbuild instead. The PackageMaker generator is officially deprecated as of CMake 3.17.

27.4.7. RPM

On Linux systems, RPM is one of the two dominant package management formats. RPM packages do not have UI features of their own, they are essentially just archives with a fairly extensive set of metadata and some scripting features. The system's package manager uses these to manage dependencies between packages, provide information to the user, trigger pre/post install and uninstall scripts and so on.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

Since the package itself has no UI features, there is no customization needed in that area, but the RPM generator provides extensive customizability of the metadata through a large number of variables. Many of these variables do not need to be explicitly set, since the majority of the defaults are appropriate for projects that don't need to do anything complex. For packages that do not need to invoke pre/post install or uninstall scripts and for which inter-package dependencies can be automatically determined by the underlying package creation tool, the amount of customization is similar to that of other package generators.

The RPM generator supports component installs, but components are disabled by default. When components are disabled, only a single `.rpm` is produced and the behavior is as though `CPACK_MONOLITHIC_INSTALL` was set to true. All components are included in the package in such cases. If components are enabled, then `CPACK_COMPONENTS_GROUPING` has its usual meaning and multiple `.rpm` files will be created. Components are enabled by setting `CPACK_RPM_COMPONENT_INSTALL` to true and the set of installed components is controlled by `CPACK_COMPONENTS_ALL` as usual.

```
# Define generic setup for all generator types...
set(CPACK_COMPONENTS_GROUPING ONE_PER_GROUP)

# RPM-specific configuration
set(CPACK_RPM_COMPONENT_INSTALL YES)

include(CPack)
# Define components and component groups...
```

The component or group names might not be suitable for use as package names, which are typically visible to the user as part of the `.rpm` file name, within RPM package manager UI applications, etc. These names can be set on a per-component basis with `CPACK_RPM_<COMP>_PACKAGE_NAME` where `<COMP>` is the uppercased component name. When creating a package with components disabled, the single monolithic package name can be overridden by setting `CPACK_RPM_PACKAGE_NAME` instead.

```
add_executable(sometool ...)
install(TARGETS sometool ... COMPONENT MyProjUtils)

set(CPACK_RPM_MYPROJUTILS_PACKAGE_NAME myproj-tools)
include(CPack)
```

The name of the `.rpm` files can also be customized and it is likely that projects will want to do so. The name of each component's `.rpm` file is controlled by the `CPACK_RPM_<COMP>_FILE_NAME` variable (or just `CPACK_RPM_FILE_NAME` for non-component packaging). The default value for these variables follows this pattern:

```
<CPACK_PACKAGE_FILE_NAME>[-<component>].rpm
```

The `<component>` part is the original component name (i.e. no change in upper/lowercase). One drawback to this default file name is that it does not include any version or architecture details, but such information would normally be required (or at least desirable). It is generally preferable to instruct `cpack` to let the underlying package creation tool select a better default package name,

Exclusively for sbmain17@gmail.com Transaction: 0098315591

which can be done by setting CPACK_RPM_<COMP>_FILE_NAME to the special string RPM-DEFAULT. Examples of typical file names produced by this arrangement are given below.

The RPM-DEFAULT package file name will automatically include the architecture. If the architecture needs to be explicitly specified, such as to mark a package as noarch to indicate it is not architecture specific, the per-component CPACK_RPM_<COMP>_PACKAGE_ARCHITECTURE variable can be set to the required value or CPACK_RPM_PACKAGE_ARCHITECTURE can be set to act as the default if no component specific override is set (it is also used for monolithic packages). The default value for the architecture is computed by cpack as the output of uname -m, but if building a 32-bit package on a 64-bit host, this would be wrong and so the project would need to explicitly set the architecture value.

RPM files are required to have version information. The RPM generator will use CPACK_PACKAGE_VERSION by default, but a RPM-specific version number can also be set using CPACK_RPM_PACKAGE_VERSION if required (but the need for this should be rare). Note that it is not currently possible to specify per-component versions, the CPack RPM generator is currently limited to using the same version for all components. In addition to the package version, RPM packages also have a separate release number, which is specified using CPACK_RPM_PACKAGE_RELEASE. This release number is the release of the package itself, not of the product, so the package version would normally remain constant if the release number is increased (e.g. to fix a packaging issue). If the package version changes, the release number is usually reset back to 1, which is the default value if CPACK_RPM_PACKAGE_RELEASE is not specified. An optional epoch can also be specified by CPACK_RPM_PACKAGE_EPOCH and its use may be more common on some systems or repositories than others. The full version has the format E:X.Y.Z-R where E is the epoch and must be a number if provided. When no epoch is set, the full version has the format X.Y.Z-R. Unless it is known that an epoch value is required, projects should generally leave the epoch unset.

Unless the project explicitly overrides CPACK_PACKAGE_VERSION and CPACK_RPM_PACKAGE_ARCHITECTURE, their values won't be available within CMakeLists.txt files because the defaults for these variables are only computed when cpack processes the input file, not when CMake runs. This means it is a lot more work to robustly set the package file name directly rather than using RPM-DEFAULT. The following example shows how to make use of the RPM-DEFAULT feature:

```
set(CPACK_RPM_PACKAGE_RELEASE 5)   # Optional, default of 1 is often okay
if(CMAKE_SIZEOF_VOID_P EQUAL 4)
    set(CPACK_RPM_PACKAGE_ARCHITECTURE i686)
endif()

set(CPACK_RPM_MYPROJUTILS_PACKAGE_NAME myproj-tools)
set(CPACK_RPM_MYPROJUTILS_FILE_NAME    RPM-DEFAULT)
include(CPack)
```

For the above, assuming CPACK_PACKAGE_VERSION evaluates to a string of the form X.Y.Z, the example would typically lead to package file names like:

```
myproj-tools-X.Y.Z-5.i686.rpm
myproj-tools-X.Y.Z-5.x86_64.rpm
```


Exclusively for sbmain17@gmail.com Transaction: 0098315591

As discussed in the previous chapter, the default base install point is unlikely to be desirable on Linux systems and this extends to the creation of RPM packages. In fact, for all but Windows systems, a more appropriate base point should generally be set for packaging too by explicitly setting the CPACK_PACKAGING_INSTALL_PREFIX variable. Extending the example from the previous chapter, the project may want to do something like the following:

```
if(NOT WIN32 AND CMAKE_SOURCE_DIR STREQUAL CMAKE_CURRENT_SOURCE_DIR)
    set(CMAKE_INSTALL_PREFIX "/opt/mycompany.com/${PROJECT_NAME}")
    set(CPACK_PACKAGING_INSTALL_PREFIX ${CMAKE_INSTALL_PREFIX})
endif()
```

A feature unique to RPM packages is that they can include relocation paths. Packages can specify one or more path prefixes which the user can then choose to relocate to another part of their file system at install time. To support this feature, the CPACK_RPM_PACKAGE_RELOCATABLE variable must be set to true and then CPACK_RPM_RELOCATION_PATHS can contain a list of path prefixes that the user will be allowed to relocate. If using this feature, developers should consult the RPM generator's documentation to understand how relative paths are treated and the various default fall backs that apply to both of these variables. Note also that if the project is included as part of a Linux distribution, the distribution maintainers will likely need to override both the install prefix variables and the relocation directories, so prefer to keep things simple.

The RPM package creation tool would normally be expected to strip executables and shared libraries of all debug symbols before adding them to the package. The rationale is that the size of release binaries should be minimized and they would normally hide implementation details and not provide debugging facilities. Normally, stripping is controlled by the CPACK_STRIP_FILES variable, which determines whether or not stripping is performed as part of the staged install during packaging, but in the case of the RPM generator, the RPM package creation tool often performs its own stripping by default. Therefore, even if CPACK_STRIP_FILES is false or unset, stripping may still occur. The underlying problem is that the package creation tool rpmbuild typically has a post staging install section which strips binaries and performs other tasks before creating the final .rpm package. Traditionally, the workaround offered by cpack is to override that behavior by setting the CPACK_RPM_SPEC_INSTALL_POST variable, usually to something like /bin/true. That approach is deprecated in favor of using CPACK_RPM_SPEC_MORE_DEFINE instead:

```
# Prevent stripping and other post-install steps during package creation
set(CPACK_RPM_SPEC_MORE_DEFINE "%define __spec_install_post /bin/true")
```

While the above technique for preventing stripping works, it also discards all the other operations that would normally be applied (e.g. automatic byte code compilation for python files, architecture-specific post processing). A potentially better alternative is to allow stripping of the binaries in the .rpm and produce a separate debuginfo package. Initial support for producing debuginfo packages was added in CMake 3.7 and was further improved in 3.8 and 3.9. To enable this feature, all that is usually required is to set either CPACK_RPM_DEBUGINFO_PACKAGE or the component-specific equivalent CPACK_RPM_<COMP>_DEBUGINFO_PACKAGE to true. The debuginfo packages produced will contain source files as well as the debug information. The sources are taken from CMAKE_SOURCE_DIR and CMAKE_BINARY_DIR by default, but this can be overridden with the CPACK_BUILD_SOURCE_DIRS variable if required. Parts of the source directory hierarchy can be excluded using the

Exclusively for sbmain17@gmail.com Transaction: 0098315591

CPACK_RPM_DEBUGINFO_EXCLUDE_DIRS and CPACK_RPM_DEBUGINFO_EXCLUDE_DIRS_ADDITION variables, although projects probably only want to set the latter. The former is typically used to exclude system directories and has an appropriate default value. Distribution maintainers may want to override CPACK_RPM_DEBUGINFO_EXCLUDE_DIRS independently of what the project would set in CPACK_RPM_DEBUGINFO_EXCLUDE_DIRS_ADDITION, hence the use of two separate variables.

When producing debuginfo packages, errors like the following may sometimes be encountered:

```
CPackRPM: source dir path '/path/to/source/dir' is shorter
than debuginfo sources dir path
'/usr/src/debug/SomeProject-X.Y.Z-Linux/src_0'! Source dir path must be
longer than debuginfo sources dir path.  Set
CPACK_RPM_BUILD_SOURCE_DIRS_PREFIX variable to a shorter value or make
source dir path longer.  Required for debuginfo packaging.  See
documentation of CPACK_RPM_DEBUGINFO_PACKAGE variable for details.
```

Due to the way paths are rewritten as part of the debuginfo processing, the path to the source tree needs to be longer than the intended installed location of the sources. Note that this may impact continuous integration systems where the location of the source tree is typically fixed. This need for a longer path length may be in conflict with other constraints where the path length may need to be minimized, so consider carefully whether such constraints may apply to the project.

Source RPMs can also be produced by the RPM generator. These are similar to the debuginfo packages but only contain the sources and no debugging information. They are produced in the same way as source packages for other package generators and the RPM generator's documentation includes basic instructions showing how to build a binary RPM from the source RPM, which may be a useful verification step.

```
# Create source RPM
cpack -G RPM --config CPackSourceConfig.cmake

# Verify that a binary RPM can be produced from it
mkdir -p build_dir/{BUILD,BUILDROOT,RPMS,SOURCES,SPECS,SRPMS}
rpmbuild --define "_topdir build_dir" --rebuild <source-RPM-filename>
```

The RPM generator supports many more variables than the ones discussed above. Details about what the packages provide or require can be specified or the package creation tool can be directed to automatically compute them. If the package replaces or conflicts with other packages, this can also be specified. Scripts to be run before or after package installation and uninstallation can be given, or if complete control is needed the project can provide its own custom .spec file template instead of using the default one provided by cpack (although this should be avoided if possible, since it negates much of the functionality already provided by cpack).

27.4.8. DEB

The DEB format is the other dominant package format for Linux systems and both DEB and RPM share many similar characteristics. DEB packages are also basically just archives with associated metadata, which the system's package manager uses to enforce dependencies, trigger scripts and so on.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

One difference between DEB and RPM is that the preparation of DEB packages does not require a special tool, unlike RPM packages which do. This allows DEB packages to be created on systems that do not themselves use the DEB format, which means it is possible to produce both RPM and DEB packages on RPM-based systems such as RedHat, SuSE, etc. The main caveat to this is that when creating DEB packages on non-DEB systems, tools such as `dpkg-shlibdeps` are not available, so things like automatic dependencies cannot be computed.

Components are handled in a very similar way to RPM and have analogous configuration variables. Components are enabled by setting `CPACK_DEB_COMPONENT_INSTALL` to true (this variable does not follow the naming used for all other DEB-specific variables, which have a name prefixed by `CPACK_DEBIAN_` rather than `CPACK_DEB_`). Package names have analogous `CPACK_DEBIAN_PACKAGE_NAME` and `CPACK_DEBIAN_<COMP>_PACKAGE_NAME` variables, while file names are controlled by `CPACK_DEBIAN_FILE_NAME` and `CPACK_DEBIAN_<COMP>_FILE_NAME`. The same file naming issues apply to DEB as for RPM, except the special value `DEB-DEFAULT` should be used instead of `RPM-DEFAULT`. If providing any other value, the file name must end in `.deb` or `.ipk`. Versioning for DEB is also handled in a very similar way to RPM, as is specifying an architecture. Equivalent DEB variables are provided, with `DEBIAN` replacing `RPM` in the variable names.

The DEB package generator has fewer variables to influence how dependencies are handled compared to RPM. If packaging is being performed on a DEB-based host where the `dpkg-shlibdeps` tool is available, the shared library dependencies can be automatically computed by setting `CPACK_DEBIAN_PACKAGE_SHLIBDEPS` or the component specific `CPACK_DEBIAN_<COMP>_PACKAGE_SHLIBDEPS` variables to true. Manually specified dependencies can be provided through the `CPACK_DEBIAN_PACKAGE_DEPENDS` and `CPACK_DEBIAN_<COMP>_PACKAGE_DEPENDS` variables and will be merged with the automatically determined ones if both manual and automatic dependencies are used. Note, however, that if a component-specific dependency variable is set, the non-component variable is not used for that component. If automatic dependency computation is enabled, it populates the component-specific variables, so if the project sets only `CPACK_DEBIAN_PACKAGE_DEPENDS`, it will be ignored for those components where automatic dependencies are populated. Therefore, it may be more robust to always populate `CPACK_DEBIAN_<COMP>_PACKAGE_DEPENDS` rather than `CPACK_DEBIAN_PACKAGE_DEPENDS` when automatic dependencies are enabled. Projects should also set `CPACK_DEBIAN_ENABLE_COMPONENT_DEPENDS` to true if inter-component dependencies are specified via the `DEPENDS` option to `cpack_add_component()`, which will then enforce those dependencies in the generated component packages.

Related to the above, each package can also specify the shared libraries it requires. On platforms that provide the `readelf` tool, these library dependencies can be determined automatically by setting `CPACK_DEBIAN_PACKAGE_GENERATE_SHLIBS` to true. The `readelf` tool is then used to determine the shared libraries each shared object needs and that information is added to the package. The `CPACK_DEBIAN_PACKAGE_GENERATE_SHLIBS_POLICY` variable controls whether exact (`=`) or minimum (`>=`) requirements are enforced.

The DEB generator's documentation details a number of other DEB-specific variables not mentioned above. In particular, some variables can be used to specify what the package(s) require, provide, replace and so on. Some DEB-specific metadata items can also be set, such as maintainer details, package group or category, etc. Developers should consult the DEB generator's documentation for the full set of supported variables.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

27.4.9. FreeBSD

The FreeBSD package generator was added in CMake 3.10. It does not support components and always produces a single `.pkg` file. Some FreeBSD-specific variables can be set to specify basic package metadata, with a few falling back to DEB or RPM specific variables. Much of the package configuration can be specified by the generic `CPACK_⋯` variables rather than generator specific variables, so configuration of this generator can be fairly basic. Project developers are advised to consult the FreeBSD generator's documentation for available features and limitations.

27.4.10. Cygwin

An even more basic package generator is that for Cygwin. It is essentially just a wrapper around a BZip2 archive and offers next to no configuration beyond the generic variables. Projects may wish to consider using one of the simple archive formats instead.

27.4.11. NuGet

The options supported by the NuGet generator follow a similar pattern to the other generators already discussed. Support for the NuGet package format first became available with CMake 3.12. CMake 3.20 expanded the set of available configuration variables further, providing generator-specific control of package icons, license files and the locale. See the generator's documentation for the full list of supported options.

27.4.12. External

The External generator was added in CMake 3.13 and is very different to all the other package generators. This particular generator writes out a JSON file containing package metadata, component details and other CPack information, but it does not produce a package itself. It can install the files that would normally be packaged into a temporary staging area if requested. Other tools are expected to consume the JSON file and staged install area to produce packages using their own methods. The main goal of this generator is to present the details accumulated by CPack in a way that platforms and distributions can easily use within their own policy and technical constraints. Instead of having to delegate the entire package creation process to CPack, systems can read the description CPack provides and optionally use the staged install area and feed these into their own existing methods. As such, this generator has a fairly narrow audience and is only likely to be relevant for distribution maintainers, system integrators, etc. The interested reader should consult the latest CMake documentation for a detailed description of the JSON format and supported customization options.

27.5. Recommended Practices

One of the first decisions to be made regarding packaging is which package formats the project will provide for its releases. A good starting point is to consider providing at least one simple archive format and then one native format for each target platform. The archive format is convenient when end users want to install multiple versions of the product simultaneously, since they can then just unpack the release archives to different directories. As long as the packages are fully relocatable, this is a simple and effective strategy. For the broadest compatibility, ZIP archives are recommended for Windows and TGZ for Unix-based systems.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

Different non-archive formats are appropriate depending on the target platform. If a UI installer is appropriate for all platforms, then consider using the IFW generator for a consistent end user experience regardless of platform. These installers also offer the greatest customizability, localization and options for downloadable components. If more native installers are preferred, then the choices will depend on what the project considers more important. For Windows, either WIX or NSIS may be appropriate and the capabilities are fairly similar. For Mac, a multi component project may prefer the productbuild generator for a cleaner installation experience, but the DragNDrop generator is more likely to be preferred by end users for non-component projects since it offers greater simplicity and more flexibility. On Linux, consider providing both RPM and DEB packages for the broadest adoption by end users if not using the IFW generator for cross-platform consistency.

Give particular consideration to whether end users should be able to install the product on a headless system. This directly impacts both the choice of package formats and the way components need to be defined and packaged. For a headless system, a non-UI installation method must be available and packages should not require UI-related dependencies. This means UI components need to be separated out from non-UI components. This is especially important for RPM and DEB package formats where inter-package dependencies are typically enforced by the package manager, so a component package that requires UI dependencies would potentially pull in a large number of unwanted UI-related packages for a headless system.

When defining component names, allow for the possibility that the project may be used as a child of some larger project hierarchy. Include the project name in the component name to prevent name clashes between projects. The component names shown to users in UI installers, package file names, etc. can be set to something different rather than relying on the component name used internally within the CMake project. In fact, setting custom display names and descriptions for components is encouraged, including providing localized values where the package format supports it.

When setting component details, prefer to use the commands defined by the relevant CMake modules rather than setting variables directly. Commands such as `cpack_add_component()`, `cpack_add_component_group()`, etc. use named arguments which make setting various options very readable and easier to maintain. They are also more robust, since any error in argument names will be caught by the command, whereas setting variables directly will silently go unnoticed if variable names are misspelled.

When configuring details for the various generators, a potentially large number of variables can influence the way contents are packaged. In many cases, the defaults are acceptable, but some details should always be set by the project. Projects should explicitly set all three of the `CPACK_PACKAGE_VERSION_MAJOR`, `CPACK_PACKAGE_VERSION_MINOR` and `CPACK_PACKAGE_VERSION_PATCH` variables, since the default version details are rarely suitable or might not always be reliable. The package name, description and vendor details should also always be set. To ensure robust escaping of variable values in generated input files, always explicitly set `CPACK_VERBATIM_VARIABLES` to true.

In most cases, projects will want to avoid including a version number in the name of the default installation directory. A number of installers support updating an existing install in-place, so any version number in the directory name will be inappropriate after a product upgrade. Users may also prefer the directory name to stay the same across upgrades so that they can write wrapper scripts, launchers, etc. that work across versions. Simple archive packages are the exception to this,

Exclusively for sbmain17@gmail.com Transaction: 0098315591

which is why the default behavior for non-component archive generation mostly follows the common convention of placing extracted contents under an appropriately named subdirectory that includes both the package name and the version. For component based packages, projects will want to set `CPACK_COMPONENT_INCLUDE_TOPLEVEL_DIRECTORY` to true to get similar behavior.

RPM and DEB packages should prefer to set package file names to `RPM-DEFAULT` and `DEB-DEFAULT` respectively. This ensures that package file names follow the common naming conventions and it is also a much simpler way of incorporating the package version and architecture details into the package file names. Do not rely on the default RPM or DEB package file names provided by CPack, since they omit the version and architecture details.

If debug information should be retained for release packages when using the RPM generator, consider using the debuginfo functionality rather than preventing the stripping step of package creation. Preventing stripping requires disabling other potentially desirable aspects of package generation and requires exposing debug details as part of the release package. The debuginfo functionality allows a proper release package to be provided, with debugging details captured in a separate package that can be distributed or not to end users. The DEB generator also supports creating debuginfo packages when using CMake 3.13 or later.

If multi-architecture packages need to be produced from a single `cpack` invocation, use the `CPACK_INSTALL_CMAKE_PROJECTS` variable to incorporate components from multiple build trees. This method can also be used to create debug-and-release packages regardless of the CMake version or CMake generator used. If using CMake 3.16 or later with a multi-config generator, a simpler way to produce a multi-config package is to invoke the `cpack` tool directly and specify the set of configurations to be packaged with the `-C` command line option. With either method, always list the release components last in case both debug and release configurations install artifacts to the same file name and directory. Ideally this should not occur anyway, but for cases where it may make sense to do so, the release artifact is likely to be the preferred one.

Explore and understand the UI customization options provided by each UI installer that the project will support. Defining appropriate product icons is highly recommended to ensure a professional look and feel. Projects should also always provide their own readme, welcome and license details so that the placeholder text provided by CPack is not used by any of the installers or packages' metadata.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

Chapter 28. External Content

For any project of modest complexity, it is likely that it will rely on one or more external dependencies. These could be commonly available toolkits such as zlib, OpenSSL, Boost, etc., private projects by the same organization or content to be used as resources, test data and so on. In some situations, the project can expect the operating system to supply all required dependencies. This would be appropriate if the project is being distributed as part of that operating system, for example. For standalone projects, it is more likely that the project should be in control of the exact version of its dependencies to ensure that builds are repeatable and that release packages have known origins. This is especially important when building on continuous integration systems being shared with other projects that might have different dependency requirements.

CMake provides a few choices for how to bring external content into a build. At a fairly raw level, the `file(DOWNLOAD)` command can be used to retrieve a specific file, either during the configure stage or as part of processing a CMake file in script mode (i.e. `cmake -P`). While this has its uses, it is usually well short of the level of functionality needed to incorporate whole projects. For downloading and building an entire dependency, the traditional approach in CMake has been to use the `ExternalProject` module. This has been a part of CMake for a long time and has a variety of uses apart from simply doing a download and build. The `FetchContent` module added in CMake 3.11 is built on top of `ExternalProject` and opens up a variety of new use cases, including handling dependencies shared between projects and supporting entire project hierarchies in one build. The `ExternalData` module offers another alternative for handling external content at build time, with a focus on data for test cases.

28.1. ExternalProject

The `ExternalProject` module's main purpose is to enable downloading and building external projects that cannot be easily made part of the main project directly. The external project is added as its own separate child build, effectively isolated from the main project and treated more or less as a black box. This means it can be used to build projects for a different architecture, different build settings or even to build a project with a build system other than CMake. It can also be used to handle a project that defines targets or install components that clash with those of the main project.

`ExternalProject` works by defining a set of build targets in the main project that represent the distinct stages of obtaining and building the external project. These are then collected under a single CMake target which represents the whole sequence. Timestamps are used to keep track of which stages have already been performed and do not need to be repeated unless relevant details change. The default set of stages are as follows:

Download

Various methods can be used to obtain the external project's source. These include downloading an archive from a URL and unpacking it automatically, or cloning/checking out from a source code repository such as git, subversion, mercurial or CVS. Alternatively, projects can define their own custom commands if none of the supported download options are appropriate.

Update/Patch

Once the source code has been downloaded, a patch can be applied to it (in the case of archive

Exclusively for sbmain17@gmail.com Transaction: 0098315591

downloads) or it can be brought up to date (for source code repositories). Custom commands can be provided to override the default behavior if necessary.

Configure

If the external project uses CMake as its build system, this step executes `cmake` on the downloaded source. Some information is passed through from the main build to make configuring external CMake projects fairly seamless. For non-CMake external projects, a custom command can be provided to run the equivalent steps, such as running a `configure` script with appropriate options.

Build

By default, the configured external project is built with the same build tool as the main project if CMake was used to configure the build. Custom commands can be provided for the build stage to use a different build tool or to perform some other task.

Install

The external project can be installed to a local directory, typically to somewhere within the main project's build tree. The main project then knows where to expect the external project's build artifacts to be and can incorporate them into its own build. The default behavior depends on whether or not the configure stage assumed a CMake build was being invoked.

Test

The external project may come with its own set of tests which the main project might or might not wish to run. The `ExternalProject` module provides flexibility in whether or not to run a test stage (by default it doesn't) and whether it should come before or after the install stage. If the test stage is enabled, a default `test` target will be assumed to exist in the external project, but custom commands can be specified to provide full control over what the test stage does.

The module allows other custom stages to be defined and inserted into any point in the above workflow, but the default set of stages are typically sufficient for most projects. The details for the default stages are all set by the main function provided by the module, `ExternalProject_Add()`. This function accepts many options, all of which are detailed in the module's documentation. A selection of the more commonly used ones and some typical scenarios are given below to help guide the reader on how to make the most of what `ExternalProject` offers.

28.1.1. Tour Of Main Features

The simplest case involves downloading a source archive from a URL and building it as a CMake project. The minimal information needed to achieve this is just the `URL`, which is provided like so:

```
include(ExternalProject)
ExternalProject_Add(SomeExtProj
    URL http://example.com/releases/myproj_1.2.3.tar.gz
)
```

The first argument to the function is always the name of a build target to be created in the main project. This target will be used to refer to the external project's whole build process. By default, it is added to the main project's `all` target, but this can be disabled by adding the usual

Exclusively for sbmain17@gmail.com Transaction: 0098315591

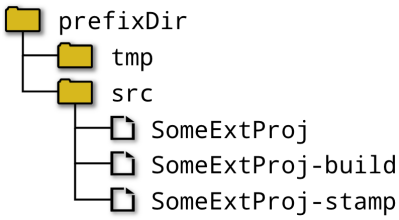
EXCLUDE_FROM_ALL option, which has the same effect as it does for commands like add_executable(), add_custom_target(), etc. In the above example, building the SomeExtProj target will result in the following being performed during the build stage of the main project:

- Download the tarball and unpack it.
- Run cmake with default options based on the main build.
- Invoke the same build tool as the main project for the default target.
- Build the external project's install target.

These steps all use a separate set of directories created in the build directory to hold the sources, build outputs, timestamps and other temporary files associated with the external project's build. The structure of these directories depends on a few different factors and the module documentation provides a detailed explanation of how the directory structure is chosen. A simpler starting point is to show how the main project can control the locations rather than relying on the defaults. The base location of the directories can be set using the PREFIX option.

```
ExternalProject_Add(SomeExtProj
    PREFIX prefixDir
    URL    http://example.com/releases/myproj_1.2.3.tar.gz
)
```

When used this way, the directory layout will be based under prefixDir, which should generally be provided as an absolute path and would normally be somewhere within the main project's build area. The default relative directory layout created under this location is shown below. The unpacked archive will be in prefixDir/src/SomeExtProj and the CMake build will use prefixDir/src/SomeExtProj-build as its build directory.



The EP_PREFIX and EP_BASE directory properties can be set to influence the above layout, see the ExternalProject documentation for details. The prefix and these directory properties only provide coarse control over the directory structure. For those cases where it is needed, ExternalProject_Add() allows some or all of the individual directories to be set directly:

```
ExternalProject_Add(SomeExtProj
    DOWNLOAD_DIR downloadDir
    SOURCE_DIR   sourceDir
    BINARY_DIR   binaryDir
    INSTALL_DIR  installDir
    TMP_DIR      tmpDir
    STAMP_DIR    stampDir
    URL          http://example.com/releases/myproj_1.2.3.tar.gz
)
```

Exclusively for sbmain17@gmail.com Transaction: 0098315591

In practice, the TMP_DIR and STAMP_DIR would rarely be used, but the others are of more direct relevance to the main project and are sometimes provided. The default install location will be up to the external project, which will typically be a system wide location, so it is very common for INSTALL_DIR to be specified to facilitate collecting all the final artifacts of external projects in one place within the build directory (further steps are required to make the external projects use the specified INSTALL_DIR, as later examples will show).

Another useful technique is to provide SOURCE_DIR and give a location of an existing directory that has already been populated. When used this way, no download method needs to be given, in which case the command will simply use the existing contents of the specified source directory. This can be a very convenient way of building a part of the main project's source tree for a different platform. For example:

```
ExternalProject_Add(Firmware
    SOURCE_DIR  ${CMAKE_CURRENT_LIST_DIR}/Firmware
    INSTALL_DIR ${CMAKE_CURRENT_BINARY_DIR}/Firmware-artifacts
    #... other options to configure differently
)
```

When the external project also uses CMake as its build system, it can be desirable to add cmake command line options to influence its configuration. The most direct way to achieve this is using the CMAKE_ARGS option, which should be followed by the arguments to be passed to the external project's cmake command. The above example can be extended to use a toolchain file, configure a release build and use the nominated install directory like so:

```
ExternalProject_Add(Firmware
    SOURCE_DIR  ${CMAKE_CURRENT_LIST_DIR}/Firmware
    INSTALL_DIR ${CMAKE_CURRENT_BINARY_DIR}/Firmware-artifacts
    CMAKE_ARGS  -D CMAKE_TOOLCHAIN_FILE=${CMAKE_CURRENT_LIST_DIR}/fwtoolchain.cmake
                -D CMAKE_BUILD_TYPE=Release
                -D CMAKE_INSTALL_PREFIX=<INSTALL_DIR>   # See further below
)
```

If more than a couple of CMake options need to be set, the length of the generated cmake command line could become a problem. An alternative is to specify cache variables to be defined using CMAKE_CACHE_ARGS rather than defining them via CMAKE_ARGS. These arguments are expected to be in the form -Dvariable:TYPE=value and will be converted to a file containing commands of the form set(variable value CACHE TYPE "" FORCE). This file is then passed to the cmake command line with a -C option. The effect is the same as if the variables had been set directly on the cmake command line via -D options. There are other options which can be used to change the CMake generator and a few other less common aspects of how CMake is invoked, but these are less frequently used. Consult the module documentation for further details.

If the external project does not use CMake as its build system, the CONFIGURE_COMMAND option can be given to provide an alternative custom command to be executed instead of running cmake. For example, many projects provide a configure script, which could be set up like so:

Exclusively for sbmain17@gmail.com Transaction: 0098315591

```
ExternalProject_Add(SomeAutotoolsProj
    URL               someUrl
    CONFIGURE_COMMAND <SOURCE_DIR>/configure
    ...
)
```

The configure command is run in the build directory, but the `configure` script will be in the source directory. Rather than explicitly having to define the directory layout to be used for the external project, the above demonstrates an alternative strategy whereby the default structure is used, but the command's *placeholder* support provides the location of the source directory. The previous example also used a placeholder for the install directory passed as the value for `CMAKE_INSTALL_PREFIX`. A placeholder is just the option name for a particular directory surrounded by angle brackets, the most commonly used being `<SOURCE_DIR>`, `<BINARY_DIR>` and `<INSTALL_DIR>`. `<DOWNLOAD_DIR>` is also available with CMake 3.11 or later. The full list of placeholders is given in the module documentation.

If the `CONFIGURE_COMMAND` option is not used, the project is assumed to be a CMake build and the external project's build step will use the same build tool as the main project. For such cases, the default behavior of the build step is suitable and no special handling is needed. When `CONFIGURE_COMMAND` is provided, the default build tool is assumed to be `make` and the default build command is to invoke `make` without any explicit target. If a non-default target should be built instead or a build tool other than `make` is needed, a custom build command must be provided. For example:

```
find_program(MAKE_EXECUTABLE NAMES nmake gmake make)
ExternalProject_Add(SomeAutotoolsProj
    URL               someUrl
    CONFIGURE_COMMAND <SOURCE_DIR>/configure
    BUILD_COMMAND     ${MAKE_EXECUTABLE} specialTool
)
```

The custom build command could do anything, it doesn't have to be a known build tool. It can even be set to an empty string to effectively bypass the build stage. Predictably, the same pattern continues for the install stage too. For CMake projects, the main project's build tool will be invoked to build a target called `install` by default, whereas for non-CMake projects the default command is simply `make install`. The `INSTALL_COMMAND` option can be used to provide a custom install command or it can be set to an empty string to disable the install stage altogether. This is often used when the main project can use the results of the build stage without needing any further install.

```
ExternalProject_Add(SomeAutotoolsProj
    URL               someUrl
    CONFIGURE_COMMAND <SOURCE_DIR>/configure
    BUILD_COMMAND     ${MAKE_EXECUTABLE} specialTool
    INSTALL_COMMAND   ""   # Disables the install stage
)
```

Care should be taken to handle the install stage properly. If the external project uses CMake as its build system, the destination of the default install rule is controlled by the `CMAKE_INSTALL_PREFIX`

Exclusively for sbmain17@gmail.com Transaction: 0098315591

cache variable. If this variable is not set, the default location will be used, which typically results in the external project being installed to a system wide location, which is not usually the desired outcome (certainly not if the project is being built within a continuous integration system). Similarly, if the external project uses a build system other than CMake, the default install command will be `make install`, which again will likely try to install to a system wide location. For the CMake case, setting the cache variable via `CMAKE_ARGS` as shown in the earlier example addresses the situation, while for a Makefile based project, something like the following is usually appropriate:

```
ExternalProject_Add(otherProj
    URL               ...
    INSTALL_DIR       ${CMAKE_CURRENT_BINARY_DIR}/otherProj-install
    CONFIGURE_COMMAND <SOURCE_DIR>/configure
    INSTALL_COMMAND   ${MAKE_EXECUTABLE} DESTDIR=<INSTALL_DIR> install
)
```

The `INSTALL_DIR` option doesn't do anything other than define a value for the `<INSTALL_DIR>` placeholder. It is up to the caller to use the `<INSTALL_DIR>` placeholder to pass that information through to wherever it is needed. Projects should use `INSTALL_DIR` to define the location and then use the `<INSTALL_DIR>` placeholder rather than embedding the path directly in options like `INSTALL_COMMAND`. This ensures that the location can be queried later if required, as covered in Section 28.1.3, "Miscellaneous Features" further below.

The test stage is handled slightly differently and does nothing by default. To enable it, one of the test-specific options must be given, such as `TEST_BEFORE_INSTALL YES` or `TEST_AFTER_INSTALL YES`. Once enabled, the pattern is the same as for the build and install stages, with the appropriate build tool invoking the `test` target by default, but `TEST_COMMAND` can be given to provide alternative behavior.

Of course, `ExternalProject` has considerably more downloading support than just a basic URL to download. For archives, it supports the main project giving a hash of the file to be downloaded. This not only has the obvious advantage of verifying the downloaded contents, it also allows the module to check a file it might have downloaded previously and avoid re-downloading it again if it knows it already has one with the correct hash. The hash value can be for any algorithm that the `file()` command supports, but it is typically either `MD5` or `SHA1`. The hash is given with the `URL_HASH` option as in the following example:

```
ExternalProject_Add(SomeAutotoolsProj
    URL      someUrl
    URL_HASH MD5=b4a78fe5c9f2ef73cd3a6b07e79f2283
    #... other options
)
```

Specifying a hash is strongly recommended. CMake will issue a warning if the `URL` option is used without an accompanying `URL_HASH` option (as a special case to maintain backward compatibility with older CMake versions, the `URL_MD5` option can be used to provide a MD5 hash, but projects should avoid it in favor of the more flexible `URL_HASH` option).

It is also possible to specify more than one URL and let the project try each in turn until one succeeds. This can be useful when the available servers to connect to might change depending on

Exclusively for sbmain17@gmail.com Transaction: 0098315591

the network connection, VPN settings, etc. or to try local servers before potentially slower remote servers. This feature cannot be used with `file://` urls.

```
ExternalProject_Add(SomeProj
    URL      http://mirrors.example.com/releases/someproj-1.2.3.tar.gz
             https://somewhereelse.com/artifacts/someproj-1.2.3.tar.gz
    URL_HASH MD5=b4a78fe5c9f2ef73cd3a6b07e79f2283
    #... other options
)
```

When downloading archives, the archive format is detected based on the file contents after download and the archive is unpacked automatically. The automatic unpacking can be disabled if needed and various aspects of how the download itself is configured can be controlled. See the module documentation for details on the relevant options for these less common scenarios.

Downloaded contents don't have to be from an archive, the module can also work directly with source code repositories for git, subversion, mercurial or CVS. Each of these require the repository to be named with a `<REPOTYPE>_REPOSITORY` option and then other repository specific options may also be given.

```
ExternalProject_Add(MyProj
    GIT_REPOSITORY git@example.com/git/myproj.git
    GIT_TAG        3a281711d1243351190bdee50a40d81694aa630a
)
```

The above example shows the typical information needed to clone a git repository and checkout a particular commit. If the `GIT_TAG` option is omitted, the latest commit on the default branch (usually `master`) will be used. The name of a tag or branch can also be given with `GIT_TAG` instead of a commit hash. While `GIT_TAG` does support these different choices, it should be noted that only a commit hash is truly unambiguous. With git, the commit referenced by a branch or tag name can move over time, so using them does not guarantee a repeatable build. Similarly, omitting `GIT_TAG` altogether is the same as giving the name of the default branch, so it too won't always point at the same commit.

There is another reason to only use commit hashes with `GIT_TAG`. Because a tag or branch can change over time, `ExternalProject_Add()` may need to contact the remote end every time CMake is run, even if it already has the named tag or branch locally. CMake cannot be sure that the tag or branch hasn't moved without fetching from the remote. This round trip every time CMake is re-run can be expensive, especially if the project is using many external projects. As a special case, if a tag is requested and that tag is already checkout out, it will be assumed that the tag hasn't moved and no fetch will be performed. This trade-off between performance and robustness is typically safe for most projects where tags never move once created, but it does open an avenue for a non-reproducible build. If a commit hash is used instead, `ExternalProject_Add()` knows robustly whether it already has the commit locally without needing to contact the remote.

Other options can be used to customize the git behavior, including specifying a different default remote name, control of git submodules, shallow clones and arbitrary git config options. Consult the module documentation for further details.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

Checking out from a subversion repository is fairly similar to git:

```
ExternalProject_Add(MyProj
    SVN_REPOSITORY svn+ssh@example.com/svn/myproj/trunk
    SVN_REVISION   -r31227
)
```

The `SVN_REVISION` option specifies a `svn` command line option that is expected to specify the commit to check out. This will frequently be a global revision number specified with the `-r` option as shown above, but could technically be any valid command line option. If `SVN_REVISION` is omitted, the latest revision is used, but projects should strive to always provide this option to ensure that the build is repeatable. A few other security-related subversion options are also supported by `ExternalProject_Add()`, such as for authenticating with the repository and specifying certificate trust settings. Consult the `ExternalProject` module documentation for details on these less frequently used options.

In comparison, the support for Mercurial and CVS is very basic. In the case of Mercurial, only the repository and tag can be specified, while for CVS the module is also required:

```
ExternalProject_Add(MyProjHg
    HG_REPOSITORY  http://example.com/hg/myproj
    HG_TAG         dd2ce38a6b8a
)
ExternalProject_Add(MyProjCVS
    CVS_REPOSITORY http://example.com/cvs/myproj
    CVS_MODULE     someModule
    CVS_TAG        -rsomeTag
)
```

The `CVS_TAG` option is analogous to the `SVN_REVISION` option in that it is placed on the `cvs` command line directly, so it must include any required command option prefix such as shown above.

28.1.2. Step Management

Sometimes it can be useful or even necessary to refer to one of the steps in the `ExternalProject` sequence, such as to add a dependency on another CMake target that provides an input to a particular step. The `STEP_TARGETS` option can be given to `ExternalProject_Add()` to tell it to create CMake targets for the specified set of steps. These targets have names of the form `mainName-step`, where `mainName` is the target name given as the first argument to `ExternalProject_Add()` and `step` is the step the target represents. For example, the following would result in targets named `MyProj-configure` and `MyProj-install` being defined:

```
ExternalProject_Add(MyProj
    GIT_REPOSITORY git@example.com/git/myproj.git
    GIT_TAG        3a281711d1243351190bdee50a40d81694aa630a
    STEP_TARGETS   configure install
)
```


Exclusively for sbmain17@gmail.com Transaction: 0098315591

Adding dependencies for these step targets requires a little more care. To make a step target depend on some other CMake target, the project should use the `ExternalProject_Add_StepDependencies()` function provided by the module rather than calling `add_dependencies()`. This ensures that things like the step timestamps are handled correctly. The form of that command is as follows:

```
ExternalProject_Add_StepDependencies(mainName step otherTarget1...)
```

The following example shows how to use this function to make the `configure` step of the previous example depend on an executable built by the main project:

```
add_executable(PreConfigure ...)
ExternalProject_Add_StepDependencies(MyProj configure PreConfigure)
```

To make an ordinary CMake target depend on a step target, `add_dependencies()` is fine:

```
add_executable(PostInstall ...)
add_dependencies(PostInstall MyProj-install)
```

If a particular step of one external project needs to depend on a step of a different external project, `ExternalProject_Add_StepDependencies()` must once again be used:

```
ExternalProject_Add(Earlier
    STEP_TARGETS build
    ...
)
ExternalProject_Add(Later
    STEP_TARGETS build
    ...
)
ExternalProject_Add_StepDependencies(Later build Earlier-build)
```

The above arrangement can be useful if `Earlier` defines tests that are time consuming to run, but in a parallel build the `Later` project doesn't need to wait for those tests, only for `Earlier` to be built.

When the same set of step targets need to be defined for multiple external projects, rather than repeating them each time, they can be made the default by setting the `EP_STEP_TARGETS` directory property instead.

```
set_property(DIRECTORY PROPERTY EP_STEP_TARGETS build)
ExternalProject_Add(Earlier ...)
ExternalProject_Add(Later ...)
ExternalProject_Add_StepDependencies(Later build Earlier-build)
```

For many projects though, such granularity of dependencies offers only limited gains and the complexity may not be worth it. The whole external project can be made dependent on another target by using the `DEPENDS` option with `ExternalProject_Add()`, which is much simpler:

Exclusively for sbmain17@gmail.com Transaction: 0098315591

```
add_executable(PreConfigure ...)
ExternalProject_Add(MyProj
    DEPENDS PreConfigure
    ...
)
```

The DEPENDS option takes care of ensuring all of the step dependencies are handled correctly just as ExternalProject_Add_StepDependencies() does when setting up more granular dependencies.

Projects are not limited to only the default steps, they can create their own custom steps and insert them into the step workflow with whatever dependency relationships they require. The ExternalProject_Add_Step() function provides this capability:

```
ExternalProject_Add_Step(mainName step
    [COMMAND           command [args...] ]
    [COMMENT           comment]
    [WORKING_DIRECTORY dir]
    [DEPENDS           filesWeDependOn...]
    [DEPENDEES         stepsWeDependOn...]
    [DEPENDERS         stepsDependOnUs...]
    [BYPRODUCTS        byproducts...]
    [INDEPENDENT       bool]  # CMake 3.19 or later
    [ALWAYS            bool]
    [EXCLUDE_FROM_MAIN bool]
    [LOG               bool]
    [USES_TERMINAL     bool]
)
```

COMMAND is used to define the action to take when the step is executed. It is analogous to the custom command that can be specified for each of the default steps. COMMENT can be supplied to provide a custom message when executing the step, but as noted back in Section 18.1, “Custom Targets”, such comments are not always shown, so consider them helpful but not essential. The WORKING_DIRECTORY option has the same meaning as for an add_custom_target() command.

Comprehensive dependency details can be provided with the custom step. If the command depends on a specific file or set of files, they should be listed with the DEPENDS option. For files generated as part of the build, they must be generated by custom commands created in the same directory scope. The DEPENDEES and DEPENDERS options define where this custom step sits in the step workflow of the external project. Care must be taken to fully specify all direct dependencies or else the custom step will potentially execute out of sequence. The BYPRODUCTS option should also be used if the custom step produces a file that something else in the external project or main project depends on. Without this, the Ninja generators will likely complain about a missing build rule.

CMake 3.19 added support for the INDEPENDENT keyword, which must be followed by a boolean true or false value. Where the step does not depend on anything outside of this external project, it can be specified as independent. It may still depend on other steps within the mainName external project as long as those other steps are also considered independent. For example, the update step depends on the download step, but both are created as independent because they don’t rely on any targets from other parts of the build. Specifying a step to be independent can improve parallelism and may

Exclusively for sbmain17@gmail.com Transaction: 0098315591

avoid unnecessarily re-executing the step when unrelated external dependencies change. If the INDEPENDENT option is given a value of false or is not present, the step target will depend on everything that the mainName target depends on. The reader is referred to the policy CMP0114 documentation, which discusses the use, limitations and relevant historical notes around this functionality in considerable depth.

A custom step can be made to always appear out of date by setting the ALWAYS option to true. Projects should generally only do this if no other step depends on it, since anything depending on it would then also be always considered out of date and this may lead to builds doing more work than they need to. If the custom step is intended to only be built on demand, then setting both ALWAYS and EXCLUDE_FROM_MAIN to true is usually the desired combination. The remaining options LOG and USES_TERMINAL are discussed in the next section.

All of the default steps are created by calls to ExternalProject_Add_Step() internally from within ExternalProject_Add(). Projects must not try to redefine them, which means custom steps cannot be named mkdir, download, update, skip-update, patch, configure, build, install or test.

The actions and inter-step dependencies are defined by ExternalProject_Add_Step(), but in order for a target to be created for a custom step, the ExternalProject_Add_StepTargets() function must be called as well. This function is also called internally by ExternalProject_Add() to create targets for steps listed in its STEP_TARGETS option or set via the EP_STEP_TARGETS directory property.

```
ExternalProject_Add_StepTargets(mainName [NO_DEPENDS] steps...)
```

The NO_DEPENDS option is rarely desirable and is not recommended for most scenarios. It is deprecated since CMake 3.19 and is not available if policy CMP0114 is set to NEW. See the discussion of this option in the module documentation for further details.

The following example demonstrates how to define a package custom step which depends on the build step, but is only executed when explicitly requested:

```
ExternalProject_Add_Step(MyProj package
    COMMAND           ${CMAKE_COMMAND} --build <BINARY_DIR> --target package
    DEPENDEES         build
    ALWAYS            YES
    EXCLUDE_FROM_MAIN YES
)
ExternalProject_Add_StepTargets(MyProj package)
```

28.1.3. Miscellaneous Features

For any of the default or custom steps, a custom command can be specified. For ExternalProject_Add(), the relevant options are those that end with _COMMAND, while for ExternalProject_Add_Step() it is the COMMAND option that provides the custom command to execute. Both of these functions allow more than one command to be given by appending further COMMAND options that immediately follow the first. Each command is then executed in order for that step.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

```
ExternalProject_Add(MyProj
    CONFIGURE_COMMAND ${CMAKE_COMMAND} -E echo "Starting custom configure"
              COMMAND ./configure
              COMMAND ${CMAKE_COMMAND} -E echo "Custom configure completed"
    BUILD_COMMAND     ${CMAKE_COMMAND} -E echo "Starting custom build"
          COMMAND     ${MAKE_EXECUTABLE} MySpecialTarget
          COMMAND     ${CMAKE_COMMAND} -E echo "Custom build completed"
)
ExternalProject_Add_Step(MyProj package
    COMMAND           ${CMAKE_COMMAND} -E echo "Starting packaging step"
    COMMAND           ${CMAKE_COMMAND} --build <BINARY_DIR> --target package
    COMMAND           ${CMAKE_COMMAND} -E echo "Packaging completed"
    DEPENDEES         build
    ALWAYS            YES
    EXCLUDE_FROM_MAIN YES
)
ExternalProject_Add_StepTargets(MyProj package)
```

Commands can also be given access to the terminal, which can be important for things like repository access that may require the user to enter a password for a private key, etc. While this is not applicable to continuous integration builds, it is sometimes useful for developers in their day to day activities. Access to the terminal is controlled on a per-step basis with `ExternalProject_Add()` options of the form `USES_TERMINAL_<STEP>`, where `<STEP>` is the uppercased step name and the value given for the option is a true or false constant. For custom steps, the `USES_TERMINAL` option for the `ExternalProject_Add_Step()` command has the same effect. If using a git or subversion repository for the download, it is advisable to give the download and update steps access to the terminal.

```
ExternalProject_Add(MyProj
    GIT_REPOSITORY         git@example.com/git/myproj.git
    GIT_TAG                3a281711d1243351190bdee50a40d81694aa630a
    USES_TERMINAL_DOWNLOAD YES
    USES_TERMINAL_UPDATE   YES
)
```

Steps should only be given access to the terminal if it is needed. The effect of doing so is mostly relevant for the Ninja generators, where the custom step will be placed into the `console` job pool. All targets allocated to the `console` pool are forced to run serially and the output of any tasks running in other job pools in parallel is buffered until the current `console` job completes. Be especially careful not to give the build step access to the terminal unless absolutely necessary, since it has the potential to have a significant negative impact on the overall build performance of the project.

In some cases, it can be useful to capture the output from individual steps to file rather than have it go to the terminal. This is especially useful where there is a large amount of output that will only be of interest if there is an error or other unexpected problem. To redirect a step's output to file, set the `LOG_<STEP>` option in `ExternalProject_Add()` or the `LOG` option in `ExternalProject_Add_Step()` to a true value. The terminal output will then only show a short message indicating whether or not the step was successful and where the log files can be found.

Prior to CMake 3.14, log files were always created in the timestamp directory (i.e. `STAMP_DIR`), which isn't always the most intuitive location. With CMake 3.14 and later, the log file directory can be

Exclusively for sbmain17@gmail.com Transaction: 0098315591

specified with the LOG_DIR keyword. If a relative path is given, it will be treated as being relative to CMAKE_CURRENT_BINARY_DIR. The LOG_MERGED_STDOUTERR option was also added in CMake 3.14 and when it is set to true, the stdout and stderr streams will be merged into a single log file for each step with LOG_<STEP> also set to true.

```
ExternalProject_Add(MyProj
    GIT_REPOSITORY       git@example.com/git/myproj.git
    GIT_TAG              3a281711d1243351190bdee50a40d81694aa630a
    LOG_BUILD            YES
    LOG_TEST             YES
    LOG_DIR              logs   # Requires CMake 3.14 or later
    LOG_MERGED_STDOUTERR YES    # Requires CMake 3.14 or later
)
```

A project may want to know the value of an ExternalProject_Add() option. Placeholders such as <SOURCE_DIR> cover many of the common scenarios where values are needed within the call to ExternalProject_Add(), but for other cases the ExternalProject_Get_Property() command is useful:

```
ExternalProject_Get_Property(mainName propertyName...)
```

Its syntax differs significantly from other property retrieval commands like get_property(). No output variable name is given, instead a variable matching the name of the property to be retrieved is created. This allows multiple properties to be retrieved in one call.

```
ExternalProject_Get_Property(MyProj SOURCE_DIR LOG_BUILD)
set(msg "MyProj source will be in ${SOURCE_DIR}")
if(LOG_BUILD)
    string(APPEND msg " and its build output will be redirected to log files")
endif()
message(STATUS "${msg}")
```

28.1.4. Common Issues

The ExternalProject module is both powerful and effective when used correctly, but it can also sometimes lead to problems that can be difficult to trace. One of the most common problems encountered is when trying to set up multiple external projects where one project wants to be able to use build outputs from another. This generally requires the main project to do two things:

- Specify the dependency relationships between the two projects.
- Give the depender project the information it needs to find the dependee.

The first point is easy enough to establish by creating a dependency for the configure step of the depender on the main target of the dependee. The second point requires understanding how the depender wants to know about the location of the dependee. For example, if it uses find_package(), find_library(), etc. to locate the dependee, then setting its CMAKE_PREFIX_PATH may be sufficient. The following example demonstrates this technique, building both zlib and libpng as external projects and installing them to the same directory. Since libpng requires zlib, giving it the common install

Exclusively for sbmain17@gmail.com Transaction: 0098315591

area for CMAKE_PREFIX_PATH allows it to find zlib. The example ensures zlib is installed before libpng runs its configure step, which is when libpng will use CMAKE_PREFIX_PATH.

```
set(installDir ${CMAKE_CURRENT_BINARY_DIR}/install)
include(ExternalProject)
ExternalProject_Add(zlib
    INSTALL_DIR ${installDir}
    URL         https://zlib.net/zlib-1.2.11.tar.gz
    URL_HASH    SHA256=c3e5e9fdd5004dcb542feda5ee4f0ff0744628baf8ed2dd5d66f8ca1197cb1a1
    CMAKE_ARGS  -DCMAKE_INSTALL_PREFIX:PATH=<INSTALL_DIR>
)
ExternalProject_Add(libpng
    INSTALL_DIR ${installDir}
    URL         ftp://ftp-osl.osuosl.org/pub/libpng/src/libpng16/libpng-1.6.34.tar.gz
    URL_HASH    MD5=03fbc5134830240104e96d3cda648e71
    CMAKE_ARGS  -DCMAKE_INSTALL_PREFIX:PATH=<INSTALL_DIR>
                -DCMAKE_PREFIX_PATH:PATH=<INSTALL_DIR>
)
ExternalProject_Add_StepDependencies(libpng configure zlib)
```

Another dependency-related issue that can arise when using the Ninja generators is Ninja complaining that it doesn't know how to build a particular file that an external project is supposed to be supplying. The following example demonstrates such a situation.

```
ExternalProject_Add(MyProj
    # Options to download and build a library "someLib"
    ...
)
ExternalProject_Get_Property(MyProj INSTALL_DIR)

add_library(MyProj::someLib STATIC IMPORTED)
set_target_properties(MyProj::someLib PROPERTIES
    # Platform-specific due to hard-coded library location and file name
    IMPORTED_LOCATION ${INSTALL_DIR}/lib/libsomeLib.a
)
add_dependencies(MyProj::someLib MyProj)
```

The Ninja generators will try to find libsomeLib.a at the expected location, but it won't yet exist before the MyProj external project is built for the first time. Ninja will then halt with an error saying it doesn't know how to build the missing dependency. Other generators may be more relaxed in their dependency checking and not complain, but that should not be considered confirmation of correctly specified dependencies. A solution to the above is to add a BUILD_BYPRODUCTS option to the ExternalProject_Add() call to specify the build outputs (available in CMake 3.2 or later). Ninja will then have all the information it needs to satisfy its dependencies.

```
ExternalProject_Add(MyProj
    BUILD_BYPRODUCTS <INSTALL_DIR>/lib/libsomeLib.a
    # Options to download and build the above library
    ...
)
```


Exclusively for sbmain17@gmail.com Transaction: 0098315591

The above situation is an example of the sort of problems that arise when mixing `ExternalProject` with targets defined in the main project. This is difficult to do robustly and usually involves having to manually specify platform specific details that CMake normally handles on the project's behalf (e.g. library names and locations). Projects should consider whether a superbuild arrangement would be more appropriate and not try to create build targets of their own when using `ExternalProject`.

Dependency problems can also arise in other situations. Consider the earlier example where `ExternalProject` was used to enable building firmware artifacts with a different toolchain to the main build.

```
ExternalProject_Add(Firmware
    SOURCE_DIR  ${CMAKE_CURRENT_LIST_DIR}/Firmware
    INSTALL_DIR ${CMAKE_CURRENT_BINARY_DIR}/Firmware-artifacts
    CMAKE_ARGS  -D CMAKE_TOOLCHAIN_FILE=${CMAKE_CURRENT_LIST_DIR}/fwtoolchain.cmake
                -D CMAKE_BUILD_TYPE=Release
                -D CMAKE_INSTALL_PREFIX=<INSTALL_DIR>
)
```

The above would build successfully and all would appear to be in order. If the developer then went and made a change to the sources in the `Firmware` source directory, the main project would not rebuild the firmware targets. This is because `ExternalProject` uses timestamps to record successful completion of the steps, so unless something changes in the way the dependencies are computed, the main project thinks the `Firmware` project is still up to date. This can be addressed by forcing the `Firmware` build target to always be considered out of date using the `BUILD_ALWAYS` option:

```
ExternalProject_Add(Firmware
    SOURCE_DIR   ${CMAKE_CURRENT_LIST_DIR}/Firmware
    INSTALL_DIR  ${CMAKE_CURRENT_BINARY_DIR}/Firmware-artifacts
    CMAKE_ARGS   -D CMAKE_TOOLCHAIN_FILE=${CMAKE_CURRENT_LIST_DIR}/fwtoolchain.cmake
                 -D CMAKE_BUILD_TYPE=Release
                 -D CMAKE_INSTALL_PREFIX=<INSTALL_DIR>
    BUILD_ALWAYS YES
)
```

This will result in the `Firmware` project's build tool being invoked every time the main project is built. If nothing has changed in the `Firmware` project, its build step will do no actual work, but if there has been a change, then anything that has become out of date will be rebuilt as expected. The main drawback to setting `BUILD_ALWAYS` to true is that it effectively makes the main external project target always appear out of date to the main build, so the main build will never be a true no-op even when nothing actually needs rebuilding.

28.2. FetchContent

Some of the strengths of `ExternalProject` are also its weaknesses. It allows external project builds to be completely isolated from the main project, so it can use a different toolchain, target a different platform, use a different build type or even an entirely different build system. The cost of these benefits is that the main project knows nothing about what the external project produces. That

Exclusively for sbmain17@gmail.com Transaction: 0098315591

information has to be provided to the main build manually if anything in the main build needs to refer to the external project's outputs. This is the sort of thing that CMake is meant to do on the project's behalf, so it can be a backward step to use `ExternalProject` in this way.

For external projects that also use CMake as their build system, the flexibility to build it with different settings to the main project is often unnecessary. In fact, the more common case is likely to be that the external project should be built with the same settings as the main project, but this is not all that easy to do using `ExternalProject`. Often a much more convenient arrangement would be to add it to the main build directly using `add_subdirectory()` as though it was part of the main project's own sources. This cannot be done with the traditional use of `ExternalProject` because the source isn't downloaded until build time. Projects may use alternative strategies such as git submodules to overcome this, but they are not without their own drawbacks too.

The `FetchContent` module was added in CMake 3.11 to solve problems like those mentioned above. It uses `ExternalProject` internally to set up a sub-build which downloads and updates the external content, but it does this during the configure stage. This means the downloaded content is available immediately, so the main project can then bring it into the main build via `add_subdirectory()`, make use of it as resources and so on.

In projects that depend on many external projects, it can sometimes be the case that those external projects in turn share some common dependencies. It would be undesirable to download and build those common dependencies multiple times, but `ExternalProject` on its own doesn't directly provide a way to handle that. The `FetchContent` module offers a solution to this scenario as well. It allows dependency details of external projects to be defined separately from the command that is used to initiate the download. The first time download details are specified for a given dependency, they are saved internally and any later attempts to define them are silently ignored. When the project is asked to populate the dependency, it uses those saved details and any other parts of the project can simply re-use that content instead of downloading them again. This "first setter wins" approach means that a parent project can override dependency details of external child projects pulled in via `add_subdirectory()`.

The canonical pattern for how the `FetchContent` module is intended to be used is demonstrated by the following example:

```
include(FetchContent)
FetchContent_Declare(googletest  ①
    GIT_REPOSITORY https://github.com/google/googletest.git
    GIT_TAG        ec44c6c1675c25b9827aacd08c02433cccde7780  # release-1.8.0
)

# NOTE: A more concise way of expressing the following is
#       available from CMake 3.14, discussed further below
FetchContent_GetProperties(googletest)  ②
if(NOT googletest_POPULATED)
    FetchContent_Populate(googletest)
    add_subdirectory(${googletest_SOURCE_DIR} ${googletest_BINARY_DIR})  ③
endif()
```

① Record details of where GoogleTest should be obtained from. If somewhere else in the project has already done this, the details declared here will be ignored.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

② Populate the GoogleTest contents, but only if some other part of the project hasn't already done so.

③ Always provide both `xxx_SOURCE_DIR` and `xxx_BINARY_DIR` to `add_subdirectory()`. When `xxx_SOURCE_DIR` points to a location not in the current binary directory (and this is typically what occurs), `add_subdirectory()` requires the associated binary directory to be given as well.

The `FetchContent_Declare()` command requires its first argument to be the name of the dependency being declared (this name is treated case insensitively). The arguments that follow the name are expected to be any of the options supported by `ExternalProject_Add()`, except those relating to the configure, build, install or test steps. In practice, the only options typically given are those that define a download method, such as the git details in the GoogleTest example above.

The `FetchContent_GetProperties()` command allows the project to check whether a particular dependency has already been populated and it also retrieves some directory details. The full form of the command is as follows:

```
FetchContent_GetProperties(name
    [SOURCE_DIR srcDirVar]
    [BINARY_DIR binDirVar]
    [POPULATED  doneVar]
)
```

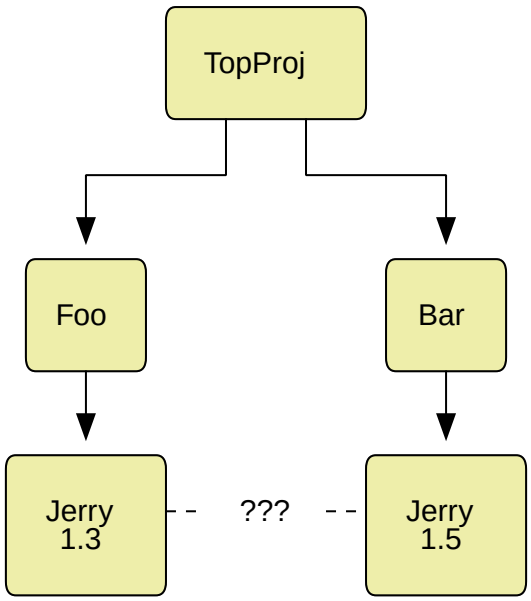
The `SOURCE_DIR`, `BINARY_DIR` and `POPULATED` options can be used to specify the name of a variable in which to store the associated property for the `name` dependency. If none of these options are given, then the command sets the variables `<lcname>_SOURCE_DIR`, `<lcname>_BINARY_DIR` and `<lcname>_POPULATED` in the caller's scope, where `<lcname>` is the `name` converted to lowercase. The optional arguments are not needed if the canonical pattern is being followed.

The `POPULATED` property will be true if `FetchContent_Populate()` has already been called somewhere in the project for the specified `name`. If it is true, then the `SOURCE_DIR` property specifies where the downloaded contents can be found. Since the downloaded contents might not be an immediate subdirectory of the place where `FetchContent_Populate()` is called, the `BINARY_DIR` property is almost always needed as well for the call to `add_subdirectory()`.

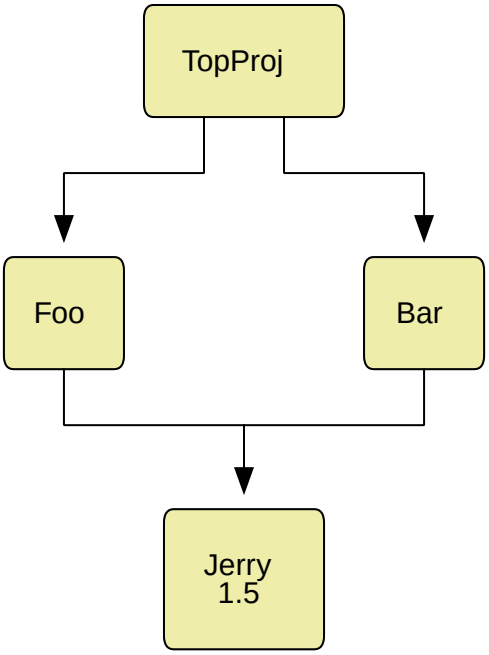
If `FetchContent_GetProperties()` has confirmed that the specified content has not yet been populated, then the `FetchContent_Populate()` command can be called to carry out the content population. When used as part of a project with the canonical form shown above, it will accept just one argument, which is the name of the dependency to populate. Using the saved details declared earlier, the content is populated if it hasn't already been populated by a previous `cmake` run. The `<lcname>_POPULATED`, `<lcname>_SOURCE_DIR` and `<lcname>_BINARY_DIR` variables will also be set in the caller's scope in exactly the same way as they are for a call to `FetchContent_GetProperties(name)`.

The following example highlights the way the `FetchContent` module allows a top level project to override the details set by the lower level dependencies. Consider a top level project `TopProj` which depends on external projects `Foo` and `Bar`. Both `Foo` and `Bar` in turn both depend on another external project, `Jerry`, but they each want slightly different versions of it.

Exclusively for sbmain17@gmail.com Transaction: 0098315591



Only one copy of `Jerry` should be downloaded and built, which `Foo` and `Bar` would then use. When these projects are combined into one build, the selected version of `Jerry` has to override the version normally used by `Foo` or `Bar`, or possibly even both. The top level project is responsible for ensuring that a valid version is selected such that `Foo` and `Bar` can build against it. This example assumes that while `Foo` uses version 1.3 when built on its own, it can safely use a later version. The desired arrangement and an example that implements it looks like this:



Exclusively for sbmain17@gmail.com Transaction: 0098315591

TopProj CMakeLists.txt

```
# Declare the direct dependencies
include(FetchContent)
FetchContent_Declare(foo GIT_REPOSITORY ... GIT_TAG ...)
FetchContent_Declare(bar GIT_REPOSITORY ... GIT_TAG ...)

# Override the Jerry dependency to force our preference
FetchContent_Declare(jerry
    URL      https://example.com/releases/jerry-1.5.tar.gz
    URL_HASH ...
)

# Populate the direct dependencies but leave Jerry to be populated by foo
FetchContent_GetProperties(foo)
if(NOT foo_POPULATED)
    FetchContent_Populate(foo)
    add_subdirectory(${foo_SOURCE_DIR} ${foo_BINARY_DIR})
endif()
FetchContent_GetProperties(bar)
if(NOT bar_POPULATED)
    FetchContent_Populate(bar)
    add_subdirectory(${bar_SOURCE_DIR} ${bar_BINARY_DIR})
endif()
```

Foo CMakeLists.txt

```
include(FetchContent)
FetchContent_Declare(jerry
    URL      https://example.com/releases/jerry-1.3.tar.gz
    URL_HASH ...
)
FetchContent_GetProperties(jerry)
if(NOT jerry_POPULATED)
    FetchContent_Populate(jerry)
    add_subdirectory(${jerry_SOURCE_DIR} ${jerry_BINARY_DIR})
endif()
```

The `CMakeLists.txt` file for `Bar` would be identical to that of `Foo` except the `URL` would specify `jerry-1.5.tar.gz` instead of `jerry-1.3.tar.gz`. The above skeleton example allows `Foo` and `Bar` to be built as standalone projects on their own, or they can be incorporated into another project like `TopProj` and still have the required flexibility to resolve the common dependencies.

The canonical pattern can be a little verbose, requiring a number of commands to test whether a dependency has already been populated, do the population if it hasn't been done yet and then add it to the main build. If many dependencies are being added, this can be quite tedious and repetitive. With CMake 3.14 or later, the canonical pattern can be implemented much more concisely using another command called `FetchContent_MakeAvailable()`. It accepts a list of dependency names which should be populated and added to the main build according to the canonical pattern, skipping over any that have already been populated. As an additional enhancement, if a dependency is populated but there is no `CMakeLists.txt` file at the top of its source tree, `add_subdirectory()` will not be called and it will not be considered an error. This allows the command to be used to populate arbitrary

Exclusively for sbmain17@gmail.com Transaction: 0098315591

projects which may simply contain a collection of files to be used as resources, scripts, toolchain files, etc. With CMake 3.18 or later, the `SOURCE_SUBDIR` option can be given with `FetchContent_Declare()` to specify an alternative location to search for a `CMakeLists.txt` file instead of the top of the dependency's source tree.

Consider the earlier example involving `TopProj`, `Foo`, `Bar` and `Jerry`. If `TopProj` wanted to completely take over the way `Foo`, `Bar` and `Jerry` were populated and added to the main build, it could be achieved very concisely like so (argument values have been omitted for brevity):

```
# Declare the direct dependencies
include(FetchContent)
FetchContent_Declare(jerry URL ... URL_HASH ...)
FetchContent_Declare(foo   GIT_REPOSITORY ... GIT_TAG ...)
FetchContent_Declare(bar   GIT_REPOSITORY ... GIT_TAG ...)

# Apply the canonical pattern to each dependency in turn. Because jerry is first
# in the list, it will be populated here rather than by either foo or bar.
FetchContent_MakeAvailable(jerry foo bar)
```

Clearly this is much simpler than manually writing out the canonical pattern for each dependency. For projects that can set CMake 3.14 or later as their minimum CMake version, `FetchContent_MakeAvailable()` should be preferred unless extra customization beyond the canonical pattern is needed (see further discussion of this in Section 29.2, "Non-superbuild Structure").

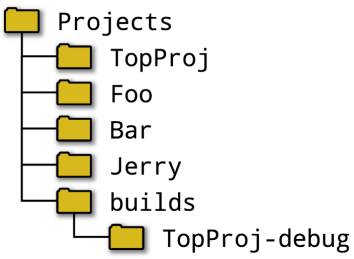
28.2.1. Developer Overrides

There may be occasions when a developer wants to work on multiple projects at once, making changes in both the main project and its dependencies or across multiple dependencies, etc. When changing parts of an external project, the developer will want to work with a local copy rather than have to update the remote location it is downloaded from each time. The `FetchContent` module offers direct support for this mode of operation by allowing the source directory of any external dependency to be overridden with a CMake cache variable. These variables have names of the form `FETCHCONTENT_SOURCE_DIR_<DEPNAME>` where `<DEPNAME>` is the dependency name in uppercase.

In the previous example, consider a situation where the developer wants to make a change to `Foo` and see how it affects the main project. They can create a separate clone of `Foo` outside of the main project and then set `FETCHCONTENT_SOURCE_DIR_FOO` to that location. The `TopProj` project would use the source of that local copy and not modify it in any way, but it would still use the same build directory for it within its own `TopProj` build area. The only difference would be where the source comes from and by setting `FETCHCONTENT_SOURCE_DIR_FOO`, the developer would take over control of the content. They would be free to change anything in their local copy, make further commits, switch branches or whatever else may be needed, then rebuild the main `TopProj` project without having to change `TopProj` at all.

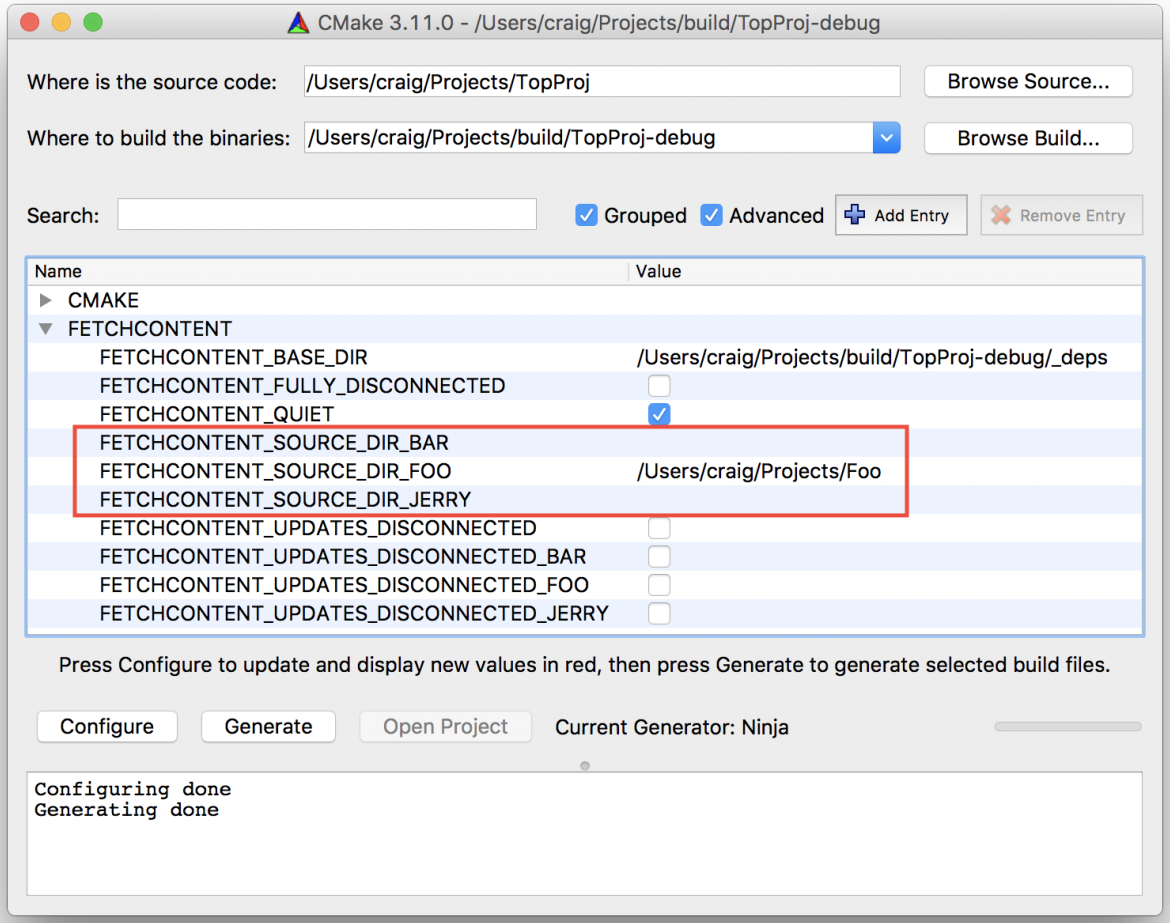
An arrangement that works well for the above usage is to have a common directory under which the developer checks out the different projects they want to work with. The main project can then be pointed at these local checkouts when needed, but still use the default downloaded contents otherwise. Such an arrangement may look like this for the above example:

Exclusively for sbmain17@gmail.com Transaction: 0098315591



If the developer wanted to make some changes to `Foo` and test it with a build of `TopProj`, they could set `FETCHCONTENT_SOURCE_DIR_FOO` to `/…/Projects/Foo`, but all of the build output from the `Foo` dependency would still be under `Projects/builds/TopProj-debug`. If `FETCHCONTENT_SOURCE_DIR_BAR` was left unset, then `Bar` would still be downloaded rather than using the local checkout in `Projects/Bar`. The developer could switch to that local checkout just as easily by setting `FETCHCONTENT_SOURCE_DIR_BAR` at any time.

Because the relevant cache variables all share the same prefix, they are easy to find in the CMake GUI or `ccmake` tool. This in turn makes it trivial to see which projects are currently using a local copy instead of the default downloaded contents.



A significant advantage of the above scenario is that it integrates well with IDE features like code refactoring tools, etc. The IDE sees the whole project, including its dependencies, so when local checkouts of those dependencies are used, refactoring can be performed transparently across multiple projects just as easily as if they were all part of the same project. Even if not using any

Exclusively for sbmain17@gmail.com Transaction: 0098315591

local checkouts of dependencies, the IDE has greater opportunity to build up a more complete code model for auto completion, following symbols and so on.

28.2.2. Other Uses For FetchContent

`FetchContent` enables more than just downloading an external project's source code and adding it to the main project via `add_subdirectory()`. Another use case is to collect commonly used CMake modules in a central repository and re-use them across many projects. Multiple collections can be pulled in via this mechanism, which makes it relatively straightforward to incorporate useful CMake scripts from other projects without having to embed copies in the main project's own sources. The following demonstrates an example where an external git repository is downloaded and its `cmake` subdirectory is added to the CMake module search path of the main project.

```
include(FetchContent)
FetchContent_Declare(JoeSmithUtils GIT_REPOSITORY ... GIT_TAG ...)
FetchContent_GetProperties(JoeSmithUtils)
if(NOT joesmithutils_POPULATED)
    FetchContent_Populate(JoeSmithUtils)
    list(APPEND CMAKE_MODULE_PATH ${joesmithutils_SOURCE_DIR}/cmake)
endif()
```

Another scenario takes advantage of the fact that the `FetchContent` module can be used even before the first `project()` call. This feature allows the module to provide toolchain files which the developer can then use for the main project.

```
cmake_minimum_required(VERSION 3.14)

include(FetchContent)
FetchContent_Declare(CompanyXToolchains
    GIT_REPOSITORY ...
    GIT_TAG        ...
    SOURCE_DIR     ${CMAKE_BINARY_DIR}/toolchains
)
FetchContent_MakeAvailable(CompanyXToolchains)

project(MyProj)
```

```
cmake -DCMAKE_TOOLCHAIN_FILE=toolchains/beta_cxx.cmake ...
```

In the above example, the directory into which the toolchains are downloaded is explicitly overridden using the `SOURCE_DIR` option. Assuming the `CompanyXToolchains` project is just a simple collection of toolchain files with no subdirectories, this makes their location both predictable and easy for developers to use. Where organizations work with very specific toolchains that are expected to always be installed to the same place, this can be a very effective way to facilitate a whole team using common build setups. The technique could even be extended to download the actual toolchains themselves.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

28.2.3. Restrictions

For the most part, the `FetchContent` module comes with some strong advantages, but there are some restrictions to be aware of. The main limitation is that CMake target names must be unique across the whole set of projects being combined, so if two external projects define a target with the same name, they cannot both be added via `add_subdirectory()`. If projects are following a naming convention that uses a project specific prefix or something similar, then this limitation is fairly easy to avoid. The difficulties tend to come from projects which never expected to be used in this way and which use fairly generic names that are likely to be commonplace. For those projects that do use project specific target names, the name of the binary that is created can still be controlled separately using the `OUTPUT_NAME` target property. For example:

```
add_library(BagOfBeans_Varieties ...)
set_target_properties(BagOfBeans_Varieties PROPERTIES
    OUTPUT_NAME BeanTypes
)

add_executable(BagOfBeans_Planter )
set_target_properties(BagOfBeans_Planter PROPERTIES
    OUTPUT_NAME Planter
)
```

`OUTPUT_NAME` and related properties are covered in more detail in Section 29.5.2, “Target Outputs”.

A similar but slightly less severe limitation applies for install components. Ideally, each project would name their install components with a project specific prefix or something equally unique. This allows a parent project to pick out just the components it wants to include in its own packaging. If two or more external project dependencies use the same install component names, then the parent project cannot separate them and has to treat them as one. Whether this matters or not will be situation dependent, but again it can be easily avoided by ensuring projects use good naming conventions for their install components.

The practice of absorbing an external dependency into a larger parent build via `add_subdirectory()` is not yet all that widespread. Many projects have never considered that use case and it is not unusual to encounter patterns that make a project hard to incorporate in this way. A common example is where a project assumes it is the top level project and it uses variables like `CMAKE_SOURCE_DIR` and `CMAKE_BINARY_DIR` where alternatives like `CMAKE_CURRENT_SOURCE_DIR` and `CMAKE_CURRENT_BINARY_DIR` may be more appropriate. Such problems are usually easy to fix, but it requires write access to the project, having changes accepted by project maintainers, vendoring a copy of the project in which the relevant fix can be made or other similar measures.

28.3. ExternalData

Another module called `ExternalData` provides an alternative way of managing files to be downloaded at build time. The focus of this module is on downloading test data when a particular target representing that data is built. In some ways, it is similar to how `ExternalProject` works, but the way the two modules define the content to be downloaded is considerably different. The `ExternalProject` module allows the download details to be explicitly defined and it supports a variety of methods. The `ExternalData` module takes a different approach, expecting individual files

Exclusively for sbmain17@gmail.com Transaction: 0098315591

to be available under one of a set of project-defined base URL locations, with paths and file names encoded using a particular hashing method. The actual file is represented in the project's source tree by a placeholder file of the same name, except with the name of a hashing algorithm appended as a file name suffix. The module provides a function to translate string arguments of a special form into their final downloaded location and name, along with a wrapper around the `add_test()` function to make it easier to pass these resolved locations to test commands.

In practice, the steps involved in setting up the necessary support for `ExternalData` tend to make it less attractive. The server from which the data is to be downloaded has to use a defined structure and treats every file separately. Every time a new file is added or an existing file is updated, it has to be manually hashed and uploaded to a path and file name that matches that hash. If the file is large but has only a small difference to the previous iteration, the file still has to be fully copied. In comparison, the `ExternalProject` module can achieve the same thing with one of its repository based download methods, but the steps involved are easy and familiar for most developers. These methods also typically handle small changes in large textual files efficiently.

One reason to consider using `ExternalData` is its support for a file series rather than just an individual file. This is more of a niche scenario that typically arises for tests that process a sequence of files. Even then, one could potentially implement similar functionality with `ExternalProject` and a `foreach()` loop, which may still be simpler to set up than the fairly rigid structure `ExternalData` requires. If the project has tests that are heavily focused on time series data or other similarly sequential data sets, then it may be worth at least evaluating whether `ExternalData` offers a preferable way to obtain that data on demand at build time. Consult the module's documentation for reference details, or for a more practical introduction, the article on this topic available from the same site as this book may be helpful.

28.4. Recommended Practices

Both `ExternalProject` and `FetchContent` provide ways to incorporate external content into a parent project. `ExternalProject` is good for bringing in external projects that are mature, have good packaging and provide well defined config files that `find_package()` can use to import the relevant targets. It also has the advantage that external dependencies are only downloaded if the build needs them and the downloading can be done in parallel with other build tasks. It can be less convenient when developers need to work across multiple projects and make changes, especially if any modest amount of refactoring is involved.

Since `ExternalProject` has been part of CMake for a long time, there is also an established body of material available for it online. Despite this, it is common to see developers struggle with getting it set up robustly. A particularly common weakness is hard-coding paths and file names of libraries in platform specific ways as a result of blending `ExternalProject` with other targets in the main project instead of a classical superbuild arrangement. Give careful thought to the maturity and quality of packaging of the external dependencies and whether the main project can use a superbuild arrangement before choosing to make use of `ExternalProject`. Prefer not to use it if the main project cannot be converted to a superbuild arrangement.

The `FetchContent` module is a good choice where other projects need to be added to the build in a way that allows them to be worked on at the same time. It affords developers the freedom to work across projects and temporarily switch to local checkouts, change branches, test with different

Exclusively for sbmain17@gmail.com Transaction: 0098315591

release versions and various other use cases in a seamless manner. It is also friendly to IDE tools, since the whole build appears as a single project. Code completion often provides greater insight and may be more reliable than if the projects had been loaded separately. Refactoring can also be performed across repositories much more robustly.

If adding dependencies to an existing mature project, `FetchContent` can be much less disruptive than `ExternalProject`, since it doesn't require any restructuring of the main project. It is also well suited to incorporating external projects that are relatively immature and which don't yet have install components and packaging implemented. A further advantage of `FetchContent` is that it inherently results in using the same compiler and settings across the whole project hierarchy. If a minimum CMake version of 3.11 or higher is acceptable, consider whether `FetchContent` is a more convenient and natural fit for the project than `ExternalProject`. It is also strongly recommended to become familiar with tools like `ccache` for speeding up the build, as the benefits are especially pronounced when using `FetchContent`. See Section 30.3, "Compiler Caches" for a detailed discussion of how to set this up for various compilers.

Whether using `ExternalProject` or `FetchContent`, if download details are being defined for a git repository, prefer to set `GIT_TAG` to the commit hash rather than a branch or tag name. This is more efficient, since it avoids making any network connection if the local clone already has that commit.

If the project wants to download test data on demand, check whether the `ExternalData` module is an appropriate choice. The `ExternalProject` module may be simpler to use and is likely to be better understood by most developers, but in specific cases such as working with file sequences, `ExternalData` could potentially be simpler. If in doubt, prefer `ExternalProject` for its easier interface and potentially more efficient handling of small changes to large data sets.

When working on a project, always assume it will some day be used as a child of some other parent project. This provides the most flexibility for how the project can be used in the future. Common problems to look out for include:

- Do not assume the project is the top level project. Use variables like `CMAKE_CURRENT_SOURCE_DIR` and `CMAKE_CURRENT_BINARY_DIR` rather than `CMAKE_SOURCE_DIR` and `CMAKE_BINARY_DIR` when referring to locations relative to the project's own directory structure.
- Use target names that are project specific. Avoid generic names, even for internal utility targets, since CMake requires globally unique target names for all but non-global imported targets across the whole project hierarchy.
- Similarly, use install component names that are project specific and avoid generic names.
- Prefer to provide some granularity in the set of install components the project defines so that parent projects can choose which parts they might want to install. Consider different ways the project might be deployed, in whole or in part, and make sure the install components allow the different combinations of installed contents to be captured.
- Always use the namespace-aliased name of a target when linking if such an alias target is available (i.e. prefer to link to `MyProj::mpfoo` rather than just `mpfoo`). This allows the project to be used in both `ExternalProject` and `FetchContent` scenarios.
- Avoid forcibly setting cache variables. Prefer instead to use regular CMake variables to override any potential cache variable in the current scope and below. Even better would be to use target or directory properties if they provide the necessary behavior.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

Chapter 29. Project Organization

The factors that contribute to an effective project structure are many and varied. What works for one project may not work for another, but there are typically some things that do tend to be common. Choosing a flexible but predictable directory structure early in the life of a project allows it to evolve with minimal friction and reorganization.

One of the most important decisions is whether a project should be structured as a superbuild or as a regular project. The two are fundamentally different and have their own strengths and weaknesses. The decision largely comes down to how the project wants to treat its dependencies and whether there is a desire and opportunity to absorb them directly or keep them isolated in their own sub-builds. For those projects without any dependencies (and importantly without any future prospect of ever having any dependencies), a regular project is the obvious choice. But when there are dependencies, the right project structure can be the difference between fighting against the build and having it work smoothly.

One of the most common topics that comes up on mailing lists, issue trackers and Q&A sites relates to problems stemming from trying to use one project structure but expecting it to have the capabilities of another. In many cases, this arises because a project is started with a particular structure, but then as dependencies are added, that structure no longer supports what the developer wants the project to be able to do. Those involved have become accustomed to working with the existing structure, so changing it will likely be very disruptive and will often meet with considerable resistance. The older a project is, the harder such a change is likely be. Therefore, decide how dependencies should be handled early in the life of the project, with due consideration for future expectations.

29.1. Superbuild Structure

Where dependencies do not use CMake as their build system, a superbuild tends to be the preferred structure. This treats each dependency as its own separate build, with the main project directing the overall sequence and the way details are passed from one dependency's build to another. Each separate build is added to the main build using `ExternalProject`. Such an arrangement allows CMake to look at what each build produces and automatically detect information that can then be passed on to other dependencies, thereby avoiding having to manually hard code that information in the main build. Even if all the dependencies use CMake, a superbuild may still be preferred for other reasons, such as to avoid target name clashes or problems with projects that assume they are always the top level project.

A superbuild allows precise control over the sequencing of the separate dependency builds. For example, one or more dependencies can be required to fully complete their own build, including their install step, before other dependencies run their own configuration phase. For such an example, the later configuration steps can see the installed artifacts and work out the appropriate file names, locations, etc. automatically. This is not possible in a regular build.

Superbuilds can be implemented with a top level `CMakeLists.txt` file that follows a fairly predictable pattern. One variation uses a common install area for all dependencies, while another installs each dependency to their own separate install area. While both are similar, using a common install area is slightly simpler to define:

Exclusively for sbmain17@gmail.com Transaction: 0098315591

```
cmake_minimum_required(VERSION 3.0)
project(SuperbuildExample)
include(ExternalProject)

set(installDir ${CMAKE_CURRENT_BINARY_DIR}/install)

ExternalProject_Add(someDep1   ①
    ...
    INSTALL_DIR ${installDir}
    CMAKE_ARGS  -DCMAKE_INSTALL_PREFIX:PATH=<INSTALL_DIR>
)
ExternalProject_Add(someDep2
    ...
    INSTALL_DIR ${installDir}
    CMAKE_ARGS  -DCMAKE_INSTALL_PREFIX:PATH=<INSTALL_DIR>
                -DCMAKE_PREFIX_PATH:PATH=<INSTALL_DIR>   ②
)
ExternalProject_Add_StepDependencies(someDep2 configure someDep1)   ③
```

① At least one dependency must require no others.

② For other dependencies that use `find_package()` to locate their dependencies, setting `CMAKE_PREFIX_PATH` to the common install directory is typically enough.

③ A step dependency is added to ensure the configure step only runs after other required dependencies have been installed.

If each dependency should be installed to its own install area, the only difference to the above is that the `CMAKE_PREFIX_PATH` given to later dependencies may need to be a list of all previous dependencies' install directories instead of just a single common install directory.

If a dependency doesn't use CMake as its build system, the overall structure doesn't change, only the way the dependency's build details are defined. For instance, a dependency that uses a build system like autotools might instead be specified like so:

```
ExternalProject_Add(someDep3
    INSTALL_DIR ${installDir}
    CONFIGURE_COMMAND <SOURCE_DIR>/configure --prefix <INSTALL_DIR>
    ...
)
```

Other options might also need to be passed to such a `configure` script to tell it in more specific ways where to find its dependencies. This will obviously vary based on the dependency's configuration capabilities.

Packaging is a little less straightforward in superbuilds. In some respects, each dependency is really in control of its own packaging, so the top level project is ultimately unlikely to be packaging anything. Instead, one or more of the `ExternalProject_Add()` calls is likely to be given a custom packaging step, if indeed packaging needs to be supported at all. The previous chapter demonstrated how to implement this with `ExternalProject_Add_Step()` function like so (a similar approach can be used for non-CMake sub-projects):

Exclusively for sbmain17@gmail.com Transaction: 0098315591

```
ExternalProject_Add_Step(MyProj package
    COMMAND           ${CMAKE_COMMAND} --build <BINARY_DIR> --target package
    DEPENDEES         build
    ALWAYS            YES
    EXCLUDE_FROM_MAIN YES
)
ExternalProject_Add_StepTargets(MyProj package)
```

In general, the key thing to keep in mind is that superbuilds work well when all they do is bring together other external projects. They usually rely on all the external projects having well defined install rules and each of the projects should ideally be able to find their own dependencies if made aware of the location of the other external projects. If any of these things are not true, then the top level project will inevitably end up having to hard code platform specific details about one or more of the projects, at which point the benefits of a superbuild start decreasing.

29.2. Non-superbuild Structure

If a project has no dependencies or if dependencies are being brought into the main build using `FetchContent` or a mechanism like git submodules, then some forward planning will help avoid difficulties later. A practice which really helps a project to remain easy to understand and work with is to think of its top level `CMakeLists.txt` as more like a table of contents. The structure can be divided up into the following sections:

Preamble

This includes the most basic setup, such as the calls to `cmake_minimum_required()` and `project()`. It may also include some use of the `FetchContent` module to bring in things like toolchain files and CMake helper repositories. This section should typically be quite short.

Project wide setup

This high level section would do things like set some global properties and default variables, perhaps define some build options in the CMake cache and may include a small amount of logic to work out some things needed by the whole build. Setting default language standards, build types and various search paths is common in this section.

Dependencies

Bring in external dependencies so that they are available to the rest of the project. Rather than defining these in the top level `CMakeLists.txt` file, putting them in a dedicated directory is cleaner and has robustness advantages.

Main build targets

This section should ideally just be one or more `add_subdirectory()` calls.

Tests

While unit tests may be embedded within the same directory structure as the main sources, integration tests may sit outside of this in their own separate area. These would be added after the main build targets.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

Packaging

This should generally be the last thing the project defines, again ideally in its own subdirectory to help keep the top level uncluttered.

The recurring pattern in the above is that apart from the preamble and project wide setup, most things are best defined in subdirectories added via `add_subdirectory()`. Not only does this make the top level `CMakeLists.txt` file easy to read and understand, it allows each subdirectory to focus on a particular area. This helps make things easier to find and it also means directory scopes can be used to minimize exposing variables from unrelated areas to things that don't need to know about them. An example of a simple top level `CMakeLists.txt` that follows the above guidelines might look like this:

```
# Preamble
cmake_minimum_required(VERSION 3.1)
project(MyProj)
enable_testing()

# Project wide setup
list(APPEND CMAKE_MODULE_PATH ${CMAKE_CURRENT_LIST_DIR}/cmake)
set(CMAKE_CXX_STANDARD 11)
set(CMAKE_CXX_STANDARD_REQUIRED YES)
set(CMAKE_CXX_EXTENSIONS NO)

# Externally provided content
add_subdirectory(dependencies)

# Main targets built by this project
add_subdirectory(src)

# Typically needed only if we are the top level project
if(CMAKE_SOURCE_DIR STREQUAL CMAKE_CURRENT_SOURCE_DIR)
    add_subdirectory(tests)
    add_subdirectory(packaging)
endif()
```

In practice, the project wide setup will likely contain more than shown above and there may be other directories for things built by the project (e.g. for documentation, adding other installable content such as scripts, images and so on).

If following the above advice about defining most things in subdirectories, the top directory of the project's source tree will typically contain mostly just administrative files. These might include a readme file of some kind, license details, contribution instructions and so on. Continuous integration systems also frequently look for a particular file name in the top level directory. Keeping source files out of this top level directory ensures that it remains focused on the high level description of the project.

Delegating the dependency handling to its own subdirectory achieves a couple of important things. Firstly, it ensures that no dependency can ever see `CMAKE_SOURCE_DIR` and `CMAKE_CURRENT_SOURCE_DIR` as being equal, so they can rely on comparing these two variables to detect if they are being incorporated into a larger project structure or they are being built standalone. The simple example above shows how this is frequently used to avoid defining tests and packaging details when the

Exclusively for sbmain17@gmail.com Transaction: 0098315591

project is not the top level project. Placing all dependency handling in its own subdirectory also ensures that no non-cache variables used to set up the dependencies can bleed out to other parts of the build accidentally. A beneficial consequence of this is that it also tends to encourage the use of CMake targets rather than variables as the means by which the rest of the project makes use of dependencies.

An example using the `FetchContent` module to incorporate the project's dependencies into the build might look like this:

dependencies/CMakeLists.txt

```
include(FetchContent)

# Declare all the dependency details first in case any
# dependency wants to pull in some of the same ones
# (this keeps us in control)
FetchContent_Declare(jerry ...)
FetchContent_Declare(foo ...)
FetchContent_Declare(bar ...)

# Add the foo and bar dependencies if not already part of
# the build (two methods are shown)

# OPTION A (CMake 3.14 or later)
FetchContent_MakeAvailable(foo bar)

# OPTION B (CMake 3.13 or earlier)
FetchContent_GetProperties(foo)
if(NOT foo_POPULATED)
    FetchContent_Populate(foo)
    add_subdirectory(${foo_SOURCE_DIR} ${foo_BINARY_DIR})
endif()
FetchContent_GetProperties(bar)
if(NOT bar_POPULATED)
    FetchContent_Populate(bar)
    add_subdirectory(${bar_SOURCE_DIR} ${bar_BINARY_DIR})
endif()
```

Sometimes, a dependency may require setting certain variables before calling `add_subdirectory()`. Ideally, this would be done in its own scope so that it can't affect other dependencies added later in the same scope. It can therefore be useful to put each dependency population in its own subdirectory too, which would leave the above example looking like this:

dependencies/CMakeLists.txt

```
include(FetchContent)

FetchContent_Declare(jerry ...)
FetchContent_Declare(foo ...)
FetchContent_Declare(bar ...)

add_subdirectory(foo)
add_subdirectory(bar)
```


Exclusively for sbmain17@gmail.com Transaction: 0098315591

For a minimum CMake version of 3.13 or earlier, the subdirectories would then look something like the following (the subdirectory for `bar` could do something similar):

dependencies/foo/CMakeLists.txt

```
FetchContent_GetProperties(foo)
if(NOT foo_POPULATED)
    FetchContent_Populate(foo)

    # Add any customizations needed before actually pulling in the dependency.
    # For example, build static libs by default and only build those targets
    # that another target depends on.
    set(BUILD_SHARED_LIBS NO)
    set_directory_properties(PROPERTIES EXCLUDE_FROM_ALL YES)

    # Now add the dependency
    add_subdirectory(${foo_SOURCE_DIR} ${foo_BINARY_DIR})
endif()
```

With CMake 3.14 or later, because the above is in its own directory scope, it could be expressed much more succinctly as:

dependencies/foo/CMakeLists.txt

```
set(BUILD_SHARED_LIBS NO)
set_directory_properties(PROPERTIES EXCLUDE_FROM_ALL YES)
FetchContent_MakeAvailable(foo)
```

Projects have the flexibility to use `FetchContent_MakeAvailable()` for the common cases or to invoke the relevant commands explicitly to extend the logic where needed. One example of extending the canonical pattern is to handle either a pre-built binary package or a source package:

```
FetchContent_GetProperties(foo)
if(NOT foo_POPULATED)
    FetchContent_Populate(foo)

    if(EXISTS ${foo_SOURCE_DIR}/CMakeLists.txt)
        # Probably source, but could still be a binary package that
        # provides itself through a top level CMakeLists.txt file
        add_subdirectory(${foo_SOURCE_DIR} ${foo_BINARY_DIR})
    else()
        # Must be a binary package, assume it provides a config file in a
        # standard location within its directory layout
        find_package(foo REQUIRED
            NO_DEFAULT_PATH
            PATHS ${foo_SOURCE_DIR}
        )
        # For this to be useful, imported targets must be promoted to global
        # so that other parts of the project can access them
        set_target_properties(foo::foo PROPERTIES IMPORTED_GLOBAL TRUE)
    endif()
endif()
```

Exclusively for sbmain17@gmail.com Transaction: 0098315591

Where the canonical pattern is sufficient, prefer to use `FetchContent_MakeAvailable()` if the minimum CMake version allows it. The command may gain further capabilities in future CMake releases and projects would then get those capabilities for free without any modification. It is also vastly simpler to write and maintain than explicitly writing out the canonical pattern for each dependency individually.

The `FetchContent` module and the `IMPORTED_GLOBAL` target property are only available from CMake 3.11 onward. Adding dependencies without these features is much harder and requires compromising on some of the recommended principles or foregoing the possibility of adding pre-built binary packages. Without being able to promote local targets to global, alternative methods generally rely on passing details back to the main build via variables or global targets have to be defined to act as proxies for local imported ones. A less desirable approach adds dependencies directly from the top level `CMakeLists.txt` file, but that makes the project hard to incorporate into a larger project hierarchy. If pre-built binary packages do not need to be supported, then the `IMPORTED_GLOBAL` target property isn't needed and these alternative methods can usually be avoided. For supporting CMake versions before 3.11, techniques like git submodules or `file(DOWNLOAD)` might be alternatives to the `FetchContent` module.

Of the other main project's top level subdirectories, adding `tests` and `packaging` doesn't require anything special, they should just follow the recommended practices already covered in the preceding chapters. The contents and structure of the `tests` subdirectory will be specific to the project, while `packaging` typically only needs a `CMakeLists.txt` file and maybe a few other files to be configured into the build directory for use by `cpack`. It may also contain resources used by some of the package generators. The structure of the `src` directory is a larger topic covered in its own section in Section 29.5, "Defining Targets" further below.

29.3. Common Top Level Subdirectories

The previous section already mentioned some directory names commonly found as subdirectories immediately below the top of the source tree. Expanding that list to cover other frequently used directories gives the following:

- `cmake`
- `dependencies`
- `doc`
- `src`
- `tests`
- `packaging`

In the absence of any other conventions, projects are encouraged to use these same directory names. Collecting CMake helper scripts in a `cmake` subdirectory makes them easy to find, allowing developers to browse through the contents of that directory and discover useful utilities they may otherwise not have known about. With just a single `list(APPEND CMAKE_MODULE_PATH …)` call in the project-wide setup section of the top level `CMakeLists.txt` file, they can be made available to the entire project. The `doc` subdirectory can be a convenient place to collect documentation, which can be useful if using formats like Markdown or Asciidoc and files contain relative links to each other.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

There are a few subdirectory names that projects should avoid. By default, calling `add_subdirectory()` with just a single argument will result in a corresponding directory of the same name in the build directory. The project should avoid using a source directory name that may result in a clash with one of the pre-defined directories created in the build area. Names to avoid include the following:

- `Testing`
- `CMakeFiles`
- `CMakeScripts`
- Any of the default build types (i.e. any of the values of `CMAKE_CONFIGURATION_TYPES`).
- Any directory name starting with an underscore.

Since some file systems may be case insensitive, all of the above names should not be used in any upper/lowercase combination. Other common directory names used as install destinations may also appear in the build directory depending on the strategy used for built binary locations (discussed further below in Section 29.5.2, "Target Outputs"). Therefore, it would also be wise to avoid source directory names like `bin`, `lib`, `share`, `man` and so on.

A few projects choose to define a top level `include` directory and collect public headers there rather than keeping them next to their associated implementation files. Be aware that some IDE tools may be unable to find headers automatically if they are split out like this, so such an arrangement may be less convenient for some developers. It also tends to make changes for a particular feature or bug fix less localized. On the other hand, a dedicated `include` directory clearly communicates which headers are intended to be public and they can have the same directory structure as they would when installed. Both approaches have their merits, but keeping headers with their associated implementation files is perhaps a little simpler for new developers.

29.4. IDE Projects

When using project generators such as Xcode or Visual Studio, a project or solution file is created at the top of the build directory. This can be opened in the IDE just like any other project file for that application, but it is still under the control of CMake. Importantly, these project files are generated as part of the build, so they should not be checked into a version control system. Also note that changes made to the project from within the IDE will be lost the next time CMake runs.

Because the Xcode or Visual Studio project files are generated by CMake, this means the way the project's targets and files are presented in the project hierarchy or file tree are also under the CMake project's control. CMake provides a number of properties that can influence how targets and files are grouped and labeled in some IDE environments. The first level of grouping is for targets, which can be enabled by setting the `USE_FOLDERS` global property to true. The location of each target can then be specified using the `FOLDER` target property, which holds a case sensitive name under which to place that target. To create a tree-like hierarchy, forward slashes can be used to separate the nesting levels. If the `FOLDER` property is empty or not set, the target is left ungrouped at the top level of the project. Both the Xcode and the Visual Studio generators honor the `FOLDER` target property.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

```
set_property(GLOBAL PROPERTY USE_FOLDERS YES)

add_executable(Foo ...)
add_executable(Bar ...)
add_executable(test_Foo ...)
add_executable(test_Bar ...)

set_target_properties(Foo Bar           PROPERTIES FOLDER "Main apps")
set_target_properties(test_Foo test_Bar PROPERTIES FOLDER "Main apps/Tests")
```

Up to CMake 3.11, the `FOLDER` target property is empty by default, whereas from CMake 3.12, it is initialized from the value of the `CMAKE_FOLDER` variable.

The name displayed for the target itself within the IDE defaults to the same target name that CMake uses. Visual Studio generators allow this display name to be overridden by setting the `PROJECT_LABEL` target property, but the Xcode generator does not honor this setting.

```
set_target_properties(Foo PROPERTIES PROJECT_LABEL "Foo Tastic")
```

Some targets are created by CMake itself, such as for installing, packaging, running tests and so on. For Xcode, most of these are not shown in the file/target tree, but for Visual Studio they are grouped under a folder called `CMakePredefinedTargets` by default. This can be overridden with the `PREDEFINED_TARGETS_FOLDER` global property, but there is usually little reason to do so.

The grouping of individual files under each target can also be controlled by the CMake project. This is done using the `source_group()` command and is independent of the target folder grouping (i.e. it is always supported, even if the `USE_FOLDERS` global property is false or unset). The command has two forms, the first of which is used to define a single group:

```
source_group(group
    [FILES src...]
    [REGULAR_EXPRESSION regex]
)
```

The `group` can be a simple name under which to group the relevant files, or it can specify a hierarchy similar to that for targets. With CMake 3.18 or later, either forward slashes or back slashes can be used as the separator between nesting levels. For CMake 3.17 or earlier, only back slashes are supported. Be aware that to get through CMake's parsing correctly, back slashes must be escaped, so a group `Foo` with a nested `Bar` underneath it would be specified like so:

```
# Backslashes require escaping
source_group(Foo\\Bar ...)

# Forward slashes don't need escaping, but their support requires CMake 3.18 or later
source_group(Foo/Bar ...)
```

Individual files can be specified with the `FILES` argument, with relative paths assumed to be relative to `CMAKE_CURRENT_SOURCE_DIR`. Because the command is not specific to a target, this option is the way to

Exclusively for sbmain17@gmail.com Transaction: 0098315591

ensure only specific files are affected by the grouping. If the project wants to define a grouping structure that should be applied more generally, the `REGULAR_EXPRESSION` option is more appropriate. It can be used to effectively set up grouping rules that will be applied to all targets in the project. Where a particular file could match more than one grouping, a `FILES` entry takes precedence over a `REGULAR_EXPRESSION`. Where a file matches multiple regular expressions, `REGULAR_EXPRESSION` groups defined later take precedence over those defined earlier.

The following example sets up general rules for all targets such that files with commonly used source and header file extensions will be grouped under `Sources`. Test sources and headers will override that grouping and be placed under a `Tests` group instead, while the special case `special.cxx` will be put in its own dedicated subgroup below `Sources`.

```
source_group(Sources REGULAR_EXPRESSION "\\.(c(xx|pp)?|hh?)$")
source_group(Tests REGULAR_EXPRESSION "test.*")   # Overrides the above
source_group(Sources\\Special FILES special.cxx)  # Overrides both of the above
```

CMake provides default groups `Source Files` for sources and `Header Files` for headers, but these are easily overridden, as the above example demonstrates. Other default groups such as `Resources` and `Object Files` are also defined.

The second form of the `source_group()` command allows the group hierarchy to follow the directory structure for specific files. It is available with CMake 3.8 or later.

```
source_group(TREE    root
             [PREFIX prefix]
             [FILES  src...]
)
```

The `TREE` option directs the command to group the specified files according to their own directory structure below `root`. The `PREFIX` option can be used to place that grouping structure under the `prefix` parent group or group hierarchy. This can be used very effectively in conjunction with the `SOURCES` target property to reproduce the directory structure of all sources that make up a target, but only if all of those sources are below a common point (e.g. no generated sources from the build directory). Many targets satisfy these conditions, so the following example pattern can often be used to quickly and easily give some structure to the way a target is presented in an IDE.

```
# Only suitable if SOURCES does not contain generated files in this example
get_target_property(sources someTarget SOURCES)

source_group(TREE   ${CMAKE_CURRENT_SOURCE_DIR}
             PREFIX "Magic\\Sources"
             FILES  ${sources}
)
```

IDEs generally only show files that are explicitly added as sources of a target. If a target is defined with only its implementation files added as sources, its headers won't usually appear in the IDE file lists. Therefore, it is common practice to explicitly list headers as well, even though they won't actually be compiled. CMake will effectively ignore them except to add them to IDE source lists. This

Exclusively for sbmain17@gmail.com Transaction: 0098315591

extends to more than just header files, it can also be used to add other non-compiled files as well, such as images, scripts and other resources. Some features such as those associated with the `MACOSX_PACKAGE_LOCATION` source property require a file to be listed as a source file to have any effect.

In certain situations, it may be desirable for a source file to appear in IDE file lists but not be compiled. Platform-specific files that should only be compiled and linked on other target platforms are an example of this. To prevent CMake from trying to compile a particular file, that source file's `HEADER_FILE_ONLY` source property can be set to true (do not be confused by the property name, it can be used for more than just headers).

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

if(NOT WIN32)
    # Don't compile this file for non-Windows platforms
    set_source_files_properties(net_win.cpp PROPERTIES
        HEADER_FILE_ONLY YES
    )
endif()
```

29.5. Defining Targets

The preceding chapters have presented a range of CMake features that allow a target to be defined in detail. This includes the sources and other files that a target is built up from, how a target should be built and how a target interacts with other targets. The focus of this section is to demonstrate how to use these techniques in a way that makes the project easy to understand, produces a robust build, provides flexibility and promotes maintainability.

For simple projects, the number of source files and targets is likely to be small, in which case it is relatively manageable for all of the relevant details to be given in a single `CMakeLists.txt` file. If following the project directory structure recommended earlier in this chapter, this would mean the `src` subdirectory would have no further subdirectories and its `CMakeLists.txt` file would define all that was needed. Initially, it may look as simple as something like this:

src/CMakeLists.txt

```
add_executable(Planter main.cpp soy.cpp coffee.cpp)
target_compile_definitions(Planter PUBLIC COFFEE_FAMILY=Robusta)

add_test(NAME NoArgs   COMMAND Planter)
add_test(NAME WithArgs COMMAND Planter beanType=soy)
```

This makes a number of assumptions about how the project will be used, but perhaps the biggest ones are that the project won't be installed or packaged and that it won't be absorbed into a larger project hierarchy by some other project. These are limitations that can and should be avoided. The specific weaknesses of the simple case above include:

- The target name is not specific to the project, so if this project was later incorporated into a larger parent project, the target name may clash with targets defined elsewhere. Using a project-specific prefix on the target name is an easy way to address this weakness.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

- There are no install rules, so the target cannot easily be installed or be included in a package.
- No namespaced alias target is defined, so even if an `install()` command was later added and packaging was implemented, other projects would have to use different target names for pre-built binary versus source inclusion.
- The test names are not very specific and a parent project cannot prevent their addition.
- Headers are not listed as sources, so they won't show up in some IDEs.

Addressing the above points and following the recommended practices of the previous chapters, the example expands out to more like the following (assuming a project name of `BagOfBeans`):

```
#=============================
# Define targets
#=============================
add_executable(BagOfBeans_Planter main.cpp soy.cpp soy.h coffee.cpp coffee.h)
add_executable(BagOfBeans::Planter ALIAS BagOfBeans_Planter)
set_target_properties(BagOfBeans_Planter PROPERTIES
    OUTPUT_NAME Planter
    EXPORT_NAME Planter
)
target_compile_definitions(BagOfBeans_Planter PUBLIC COFFEE_FAMILY=Robusta)

#=============================
# Testing
#=============================
if(TEST_BAGOFBEANS OR CMAKE_SOURCE_DIR STREQUAL BagOfBeans_SOURCE_DIR)
    add_test(NAME Planter.NoArgs   COMMAND BagOfBeans_Planter)
    add_test(NAME Planter.WithArgs COMMAND BagOfBeans_Planter beanType=soy)
endif()

#=============================
# Packaging
#=============================
include(GNUInstallDirs)
install(TARGETS     BagOfBeans_Planter
        EXPORT      BagOfBeans_Apps
        DESTINATION ${CMAKE_INSTALL_BINDIR}
        COMPONENT   BagOfBeans_Apps
)
```

That may be a surprising amount of detail for a fairly simple executable, but it highlights that for real world projects, there's more to consider than just building a binary in isolation. The added complexity is mostly for unique names to reduce the likelihood of clashes. The addition of packaging logic also tends to add a fair amount of detail that an inexperienced developer probably hasn't had much exposure to. Adding clear sections to the file as shown above can help make it easier to understand for newer developers and also keep it organized as the project evolves.

29.5.1. Target Sources

When the number of source files increases, having them all in the one directory can make them more difficult to work with. This is generally addressed by placing them under subdirectories

Exclusively for sbmain17@gmail.com Transaction: 0098315591

grouped by functionality, which brings a few other benefits too. Not only does it help keep things from becoming too cluttered, it also makes it easy to turn certain features on and off based on CMake cache options or other configure time logic. For example:

```
add_executable(BagOfBeans_Planter main.cpp)

option(BAGOFBEANS_SOY    "Enable planting soy beans"    ON)
option(BAGOFBEANS_COFFEE "Enable planting coffee beans" ON)
if(BAGOFBEANS_SOY)
    add_subdirectory(soy)
endif()
if(BAGOFBEANS_COFFEE)
    add_subdirectory(coffee)
endif()
```

In all of the preceding chapters, executables and libraries were always defined in the one directory, so the full list of files could be supplied directly to the `add_executable()` or `add_library()` call. In the above arrangement, the subdirectories add sources to the target after it has been defined using the `target_sources()` command, which is available with CMake 3.1 or later. This command works just like the other `target_…()` commands and has a very similar form:

```
target_sources(targetName
    <PRIVATE|PUBLIC|INTERFACE> src...
    # Repeat with more sections as needed
    ...
)
```

One or more `PRIVATE`, `PUBLIC` or `INTERFACE` sections is provided, each of which lists source files to be added to the relevant target. `PRIVATE` sources are added to the `SOURCES` property of `targetName`, while `INTERFACE` sources are added to the `INTERFACE_SOURCES` property. A `PUBLIC` source is added to both properties. The more practical way of thinking of this is that `PRIVATE` sources are compiled into `targetName`, `INTERFACE` sources are added to anything that links to `targetName` and `PUBLIC` sources are added to both.

In practice, anything other than `PRIVATE` would be unusual, since adding a source file to all targets linking against `targetName` would have limited usefulness. One could use it to add resources that need to be part of the same translation unit to work, or to embed something that should not be exposed through any inter-library interface, but these situations would be uncommon.

A peculiarity of the `target_sources()` command prior to CMake 3.13 is that if a source is specified with a relative path, that path is assumed to be relative to the source directory of the target it is being added to. This creates a number of problems. The first is that if it were added as an `INTERFACE` source, then the path would be treated as relative to that other target, not `targetName`. Clearly this could create incorrect paths, so any non-`PRIVATE` source would need to be specified with an absolute path. The second problem is that relative paths behave unintuitively when `target_sources()` is called from a directory other than the one in which `targetName` was defined. Consider how the `CMakeLists.txt` file for one of the directories of the earlier example might be specified:

Exclusively for sbmain17@gmail.com Transaction: 0098315591

src/coffee/CMakeLists.txt

```
# WARNING: Wrong file paths with CMake 3.12 or earlier
target_sources(BagOfBeans_Planter
    PRIVATE
        coffee.cpp
        coffee.h
)
...
```

The above is intended to add the sources from the same directory, but with CMake 3.12 or earlier, they will be interpreted as being relative to `src` rather than `src/coffee`. The most robust way to address this is to prefix them with `CMAKE_CURRENT_SOURCE_DIR` or `CMAKE_CURRENT_LIST_DIR` to ensure they always use the correct path:

src/coffee/CMakeLists.txt

```
target_sources(BagOfBeans_Planter
    PRIVATE
        ${CMAKE_CURRENT_LIST_DIR}/coffee.cpp
        ${CMAKE_CURRENT_LIST_DIR}/coffee.h
)
target_compile_definitions(BagOfBeans_Planter
    PUBLIC COFFEE_FAMILY=Robusta
)
target_include_directories(BagOfBeans_Planter
    PUBLIC $<BUILD_INTERFACE:${CMAKE_CURRENT_LIST_DIR}>
)
```

Having to prefix each source file with `${CMAKE_CURRENT_SOURCE_DIR}` or `${CMAKE_CURRENT_LIST_DIR}` in use cases such as this is inconvenient and not particularly intuitive. In recognition of this, the behavior was changed in CMake 3.13 to treat relative paths as being relative to `CMAKE_CURRENT_SOURCE_DIR` at the point where `target_sources()` is called, not the source directory in which the target was defined. Policy `CMP0076` provides backward compatibility for those projects that were relying on the old behavior.

The above also demonstrates how other `target_…()` commands can be moved into the subdirectories too, not just `target_sources()`. This helps keep things local to the code they relate to. For example, compile definitions, compiler flags and header search paths that are specific to a particular feature can be added only if that feature is enabled. If the directory structure needed to be reorganized and this directory moved elsewhere, nothing in this file would need to change and other sources in the target that had `#include "coffee.h"` would continue to work unmodified.

The one exception to this localization of details is `target_link_libraries()`. CMake 3.12 and earlier prohibited `target_link_libraries()` from operating on a target defined in a different directory. If a subdirectory needed to make the target link to something, it couldn't do it from within that subdirectory. The call to `target_link_libraries()` would have to be made in the same directory as where `add_executable()` or `add_library()` was called. If, for example, the `BagOfBeans_Planter` target needed to link against a library called `Weather`, it would have to add the call in `src/CMakeLists.txt` rather than `src/coffee/CMakeLists.txt`. This would result in something like the following:

Exclusively for sbmain17@gmail.com Transaction: 0098315591

```
option(BAGOFBEANS_COFFEE "Enable planting coffee beans" ON)

if(BAGOFBEANS_COFFEE)
    add_subdirectory(coffee)
    target_link_libraries(BagOfBeans_Planter PRIVATE Weather)
endif()
```

CMake 3.13 lifted this restriction, allowing subdirectories to be truly self-contained. For CMake versions from 3.1 to 3.12, subdirectories can be fully self-contained apart from adding libraries that a target should link to. Before CMake 3.1, a completely different approach was needed which relied on building up lists of sources in a variable and only creating the target once all subdirectories had been added. Such an arrangement might look like this:

```
# Pre-CMake 3.1 method, avoid using this approach
unset(planterSources)
unset(planterDefines)
unset(planterOptions)
unset(planterLinkLibs)

# Subdirs add to the above variables using PARENT_SCOPE
option(BAGOFBEANS_SOY    "Enable planting soy beans"    ON)
option(BAGOFBEANS_COFFEE "Enable planting coffee beans" ON)
if(BAGOFBEANS_SOY)
    add_subdirectory(soy)
endif()
if(BAGOFBEANS_COFFEE)
    add_subdirectory(coffee)
endif()

# Lastly define the target and its other details.
# All variables are assumed to name PRIVATE items.
add_executable(BagOfBeans_Planter ${planterSources})
target_compile_definitions(BagOfBeans_Planter PRIVATE ${planterDefines})
target_compile_options(BagOfBeans_Planter PRIVATE ${planterOptions})
target_link_libraries(BagOfBeans_Planter PRIVATE ${planterLinkLibs})
```

The above would get even more complicated if some items needed to be anything other than `PRIVATE`. The use of variables like this is fragile, as it relies on nothing in subdirectories using the same variables for a different target and typos in variable names would not be caught as an error by CMake. It also enforces a stronger coupling between parent and child directories, since each child subdirectory would have to pass all relevant variables back up to its parent using `set(… PARENT_SCOPE)`. For deeply nested directories, this quickly gets tedious and is error-prone.

The above discusses ways to add sources to library and executable targets. CMake 3.20 added the ability to add sources to custom targets via `target_sources()` as well. Prior to CMake 3.20, sources could only be added to custom targets in the `add_custom_target()` call. Enabling it to be done via `target_sources()` allows sources to be added to a custom target incrementally, such as from calls within subdirectories that extend the custom target in some way.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

29.5.2. Target Outputs

When a library or executable is built, its default location will be either `CMAKE_CURRENT_BINARY_DIR` or a configuration-specific subdirectory below it, depending on the generator used. For projects with many subdirectories or deeply nested hierarchies, this can be inconvenient for developers. This default can be overridden using the following set of target properties:

`RUNTIME_OUTPUT_DIRECTORY`

Used for executables on all platforms and DLLs on Windows.

`LIBRARY_OUTPUT_DIRECTORY`

Used for shared libraries on non-Windows platforms.

`ARCHIVE_OUTPUT_DIRECTORY`

Used for static libraries on all platforms and import libraries associated with DLLs on Windows.

For all three of the above, multi configuration generators like Visual Studio, Xcode and Ninja Multi-Config will automatically append a configuration-specific subdirectory to each value unless it contains a generator expression. Associated per configuration properties with `_<CONFIG>` appended are also supported for historical reasons, but those should be avoided in favor of using generator expressions where configuration specific behavior is needed.

A common use of these target properties is to collect libraries and executables together in a similar directory structure as they would have when installed. This is helpful if applications expect various resources to be located at a particular location relative to the executable's binary. On Windows, it can also simplify debugging, since executables and DLLs can be collected into the same directory, allowing the executables to find their DLL dependencies automatically (this isn't needed on other platforms, since `RPATH` support embeds the necessary locations in the binaries themselves).

Following the usual pattern, these target properties are each initialized by a CMake variable of the same name with `CMAKE_` prepended. When all targets should use the same consistent output locations, these variables can be set at the top of the project so that the properties don't have to be set for every target individually. To allow the project to be incorporated into a larger project hierarchy, these variables should only be set if they are not already set so that parent projects can override the output locations. They should also use a location relative to `CMAKE_CURRENT_BINARY_DIR` rather than `CMAKE_BINARY_DIR`. The following example shows how to safely collect binaries under a `stage` subdirectory of the current binary directory unless a parent project overrides this.

```
set(stageDir ${CMAKE_CURRENT_BINARY_DIR}/stage)
include(GNUInstallDirs)
if(NOT CMAKE_RUNTIME_OUTPUT_DIRECTORY)
    set(CMAKE_RUNTIME_OUTPUT_DIRECTORY ${stageDir}/${CMAKE_INSTALL_BINDIR})
endif()
if(NOT CMAKE_LIBRARY_OUTPUT_DIRECTORY)
    set(CMAKE_LIBRARY_OUTPUT_DIRECTORY ${stageDir}/${CMAKE_INSTALL_LIBDIR})
endif()
if(NOT CMAKE_ARCHIVE_OUTPUT_DIRECTORY)
    set(CMAKE_ARCHIVE_OUTPUT_DIRECTORY ${stageDir}/${CMAKE_INSTALL_LIBDIR})
endif()
```

Exclusively for sbmain17@gmail.com Transaction: 0098315591

Avoid creating CMAKE_⋯_OUTPUT_DIRECTORY as cache variables, they should not be under the control of the developer. They should be controlled by the project because parts of the project may make assumptions about the relative layout of the binaries. More importantly, leaving them as ordinary variables also means they can be unset within subdirectories where test executables are defined, allowing them to avoid being collected with the other main binaries and cluttering up that area.

The name of the built binary can also be controlled by the project. By default, the base name of the binary will be the same as the target name. When target names follow the convention of incorporating the project name (to help keep them unique when part of a larger project hierarchy), the target name may not be appropriate as the binary base name. The OUTPUT_NAME target property can be used to override the base name, or for more fine-grained control, the more specific RUNTIME_OUTPUT_NAME, LIBRARY_OUTPUT_NAME and ARCHIVE_OUTPUT_NAME properties can be set. In most cases, OUTPUT_NAME is both sufficient and preferred. All of these properties support generator expressions.

```
add_executable(BagOfBeans_Planter ...)
set_target_properties(BagOfBeans_Planter PROPERTIES
    OUTPUT_NAME Planter
)
```

Older projects sometimes read the LOCATION target property to try to obtain the output location of a binary and use it in places like custom target commands. As already highlighted in Section 14.4, “Recommended Practices”, this is problematic for multi-configuration generators, since the location depends on the configuration and the LOCATION target property doesn’t account for that. Projects should use generator expressions like $<TARGET_FILE:⋯> instead. CMake 3.0 and later will warn if a project tries to set this target property.

29.6. Windows-specific Issues

Windows’ lack of support for RPATH causes a number of problems for developers. When running an executable during development, any DLLs the executable requires must be either in the same directory or be located in one of the directories listed in the PATH environment variable. For the project’s main binaries, the various ⋯_OUTPUT_PATH properties can be used to place executables and libraries in the same location, but this technique is less convenient for test executables since there could be many of them and having them all in the one output directory can be problematic.

For tests executed through ctest, the ENVIRONMENT test property can be used to add the required DLL directories to the PATH like so:

```
add_executable(test_Foo ...)
target_link_libraries(test_Foo PRIVATE Algo)
add_test(NAME FooWithAlgo COMMAND test_Foo)
if(WIN32)
    set(algoDir  "$<SHELL_PATH:$<TARGET_FILE_DIR:Algo>>")
    set(execPath "PATH=${algoDir}$<SEMICOLON>$ENV{PATH}")
    set_tests_properties(FooWithAlgo PROPERTIES
        ENVIRONMENT "${execPath}"
    )
endif()
```


Exclusively for sbmain17@gmail.com Transaction: 0098315591

This won't help with trying to run the test executable within the Visual Studio IDE. CMake 3.12 introduced the VS_DEBUGGER_COMMAND and VS_DEBUGGER_WORKING_DIRECTORY target properties. CMake 3.13 then expanded the set of target properties further by adding VS_DEBUGGER_COMMAND_ARGUMENTS and VS_DEBUGGER_ENVIRONMENT, along with generator expression support for all four properties. VS_DEBUGGER_ENVIRONMENT can be used to set the PATH environment variable when running the target within the Visual Studio IDE. The above example can be extended to use this property as follows:

```
if(WIN32)
    set(algoDir  "$<SHELL_PATH:$<TARGET_FILE_DIR:Algo>>")
    set(execPath "PATH=${algoDir}$<SEMICOLON>$ENV{PATH}")
    set_tests_properties(FooWithAlgo PROPERTIES
        ENVIRONMENT "${execPath}"
    )
    set_target_properties(test_Foo PROPERTIES
        VS_DEBUGGER_ENVIRONMENT "${execPath}"
    )
endif()
```

The VS_DEBUGGER_COMMAND and VS_DEBUGGER_COMMAND_ARGUMENTS properties can be used to customize the command to be executed for the target within the IDE if needed. The VS_DEBUGGER_WORKING_DIRECTORY property can be used to override the directory from which the command is executed. All of these VS_DEBUGGER_⋯ properties are supported for Visual Studio 2010 or later.

If the project's minimum CMake version constraints prevent VS_DEBUGGER_⋯ properties from being used, more elaborate measures are needed. CMake 3.8 added support for the VS_USER_PROPS target property which can be used to override the location of the user properties file on a per-target basis. A custom properties file can be created with its LocalDebuggerEnvironment entry set to the additional PATH entries to be merged with the default PATH. If the DLLs any tests need are collected together in a small number of locations, one user properties file can be generated and re-used for each test (but it is still possible to generate and use a custom user properties file for each target if required). The configure_file() command can be used to fill in the output directory automatically. User property files can do more than just set up the debugger environment, but the following basic example provides a starting point for those wishing to explore this technique further:

```
file(TO_NATIVE_PATH ${CMAKE_RUNTIME_OUTPUT_DIRECTORY} baseDir)
configure_file(user.props.in user.props @ONLY)
```

user.props.in

```
<?xml version="1.0" encoding="utf-8"?>
<Project DefaultTargets="Build" ToolsVersion="15.0"
  xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <PropertyGroup Condition="'$(Configuration)|$(Platform)'=='Debug|Win32'">
    <LocalDebuggerEnvironment>PATH=@baseDir@\Debug</LocalDebuggerEnvironment>
  </PropertyGroup>
  <PropertyGroup Condition="'$(Configuration)|$(Platform)'=='Release|Win32'">
    <LocalDebuggerEnvironment>PATH=@baseDir@\Release</LocalDebuggerEnvironment>
  </PropertyGroup>
</Project>
```

Exclusively for sbmain17@gmail.com Transaction: 0098315591

Another aspect unique to Windows is that for executables and DLLs, it is typical for a PDB (program database) file to be generated so that debugging information is available during development. There are two kinds of PDB files and CMake provides features for both. For shared libraries and executables, the PDB_NAME and configuration specific PDB_NAME_<CONFIG> target properties can be used to override the base name of the PDB file. The default name is normally the most appropriate though, since it matches the DLL or executable name except it has a .pdb suffix instead of .dll or .exe. The PDB file is placed in the same directory as the DLL or executable by default, but this can be overridden with the PDB_OUTPUT_DIRECTORY and configuration specific PDB_OUTPUT_DIRECTORY_<CONFIG> target properties. Note that unlike the other ⋯_OUTPUT_DIRECTORY properties, PDB_OUTPUT_DIRECTORY does not support generator expressions with CMake 3.11 or earlier.

A second kind of PDB file is also created which holds information for the individual object files being built for a target. This PDB file is less useful during development, except perhaps for static libraries. For C++, this latter PDB file has a default name VCxx.pdb where xx represents the version of Visual C++ being used (e.g. VC14.pdb). Because the default name is not target-specific, it is easy to make mistakes and mix up the PDBs for different targets in some situations. CMake allows the name of each target's object PDB file to be controlled with the COMPILE_PDB target property or the associated configuration-specific COMPILE_PDB_<CONFIG> target properties. The location of these object PDB files can also be overridden with the COMPILE_PDB_OUTPUT_DIRECTORY and COMPILE_PDB_OUTPUT_DIRECTORY_<CONFIG> target properties. Note that these object PDB files are of little use for DLL and executable targets, since the main PDB already contains all the debugging information required.

29.7. Cleaning Files

Project generators usually provide some kind of clean target that can be used to remove all the generated files, build outputs, etc. This is sometimes used by IDE tools to provide a basic rebuild feature as a clean followed by a build, or by developers to simply remove build outputs to force rebuilding everything on the next build attempt. Sometimes a project defines a custom rule in such a way that it creates files that CMake doesn't know about, so they are not included in the clean step and have the potential to still affect the next build.

There are a variety of mechanisms by which files can become part of the set to be removed by a clean operation. The Ninja generators will automatically add any generated files to the clean set. Any files listed as BYPRODUCTS of a custom command or custom target are also recognized as generated and are added to the clean set as well. With CMake 3.13 and later, Makefile generators also add generated and byproduct files to the clean set.

For other files to be cleaned, the preferred approach depends on the minimum CMake version. For CMake 3.15 or later, the ADDITIONAL_CLEAN_FILES directory and target properties can be used to specify a list of files to be cleaned. Miscellaneous files created as part of building a particular target should be added to the target property. If files are not associated with a single target, they should be added to the directory property instead. Only the Ninja and Makefile generators support the ADDITIONAL_CLEAN_FILES properties. If needing to support CMake 3.14 or earlier, the ADDITIONAL_MAKE_CLEAN_FILES directory property can be used instead, but it is officially deprecated as of CMake 3.15. It works the same way as the ADDITIONAL_CLEAN_FILES directory property, but it is only supported by Makefile generators.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

29.8. Re-running CMake On File Changes

Certain more advanced techniques may require CMake to be re-run if a particular file changes. Normally, CMake does a good job of automatically tracking dependencies for things it controls, such as copying files with the `configure_file()` command, but custom commands and other tasks may rely on files for which CMake isn't aware of the dependency. Such files can be added to the `CMAKE_CONFIGURE_DEPENDS` directory property and if any of the listed files change, CMake will be re-run before the next build. If a file is specified with a relative path, it will be taken to be relative to the source directory associated with the directory property.

Most projects won't typically need to make use of the `CMAKE_CONFIGURE_DEPENDS` directory property, but it can and should be used where CMake doesn't have the opportunity to know about files which act as input to the configure or generation steps. Most file dependencies are build time dependencies, not configure or generation time, so before using this property, check whether the project really does need to re-run CMake rather than simply recompiling a source file or target as part of the regular build.

29.9. Injecting Files Into Projects

There will inevitably come a time where a project from some external source needs to be added to a build, but it has some sort of problem that prevents it from working correctly. Some common examples include not setting variables or properties that should have been set, or making the assumption it is the top level project. These things are especially common when working with projects that support very old CMake versions and have not been updated to handle newer CMake features and techniques. For some of these issues, it is possible to inject CMake code without having to actually modify the external project and work around the problem.

The `project()` command has a feature whereby it will check for a variable with the name `CMAKE_PROJECT_<PROJNAME>_INCLUDE` where `<PROJNAME>` is the project name as given to the `project()` command. If that variable is defined, it is assumed to hold the name of a file that CMake should include as the last thing the `project()` command does before returning. In effect, the `project()` command works like this:

```
project(SomeProj)

if(DEFINED CMAKE_PROJECT_SomeProj_INCLUDE)
    include(${CMAKE_PROJECT_SomeProj_INCLUDE})
endif()
```

When using CMake 3.15 or later, two more variables are available to inject files before *every* `project()` command, regardless of the project name given. These variables, `CMAKE_PROJECT_INCLUDE_BEFORE` and `CMAKE_PROJECT_INCLUDE` allow code to be inserted before or after normal `project()` command processing respectively. They are mostly useful for situations where the project name may not be readily known, such as when a build is driven by generic scripts that are re-used across multiple project builds. The expanded behavior accounting for these additional variables is equivalent to the following:

Exclusively for sbmain17@gmail.com Transaction: 0098315591

```
# CMake 3.15 or later only
if(DEFINED CMAKE_PROJECT_INCLUDE_BEFORE)
    include(${CMAKE_PROJECT_INCLUDE_BEFORE})
endif()

project(SomeProj)

# CMake 3.15 or later only
if(DEFINED CMAKE_PROJECT_INCLUDE)
    include(${CMAKE_PROJECT_INCLUDE})
endif()

# All CMake versions
if(DEFINED CMAKE_PROJECT_SomeProj_INCLUDE)
    include(${CMAKE_PROJECT_SomeProj_INCLUDE})
endif()
```

Because this behavior is supported for every `project()` call, each `project()` call therefore becomes a potential point of CMake code injection. It can be used to change the defaults for target properties within the project, add compiler or linker flags and so on. Another particularly handy use of this feature is to safely set options for continuous integration builds without having to save them in the CMake cache. This means incremental builds are less likely to be affected by old CMake cache options that are removed or no longer set after subsequent changes to the project.

For example, consider a developer working on an integration branch where extra checks should be enabled temporarily. A naive approach would be to explicitly set variables like `CMAKE_C_FLAGS` or `CMAKE_CXX_FLAGS`, but since the CI scripts shouldn't change the project itself, the only choice would be to set them as cache options. When the branch is merged, those cache options will continue to be present for incremental builds, but they should no longer be getting applied. The only course of action then is to clear the cache which would likely force a complete rebuild. A better alternative is to use `CMAKE_PROJECT_<PROJNAME>_INCLUDE` to process a CI-specific file at the end of the top-most `project()` call. This file would be under source control just like the rest of the project. Before the branch is merged, that file would be restored to its normal contents and the build would not retain the temporary flags.

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.0)
project(MyProj)
...
```

CMake would be invoked like so by the CI system:

```
cmake -D CMAKE_PROJECT_MyProj_INCLUDE:FILEPATH=path/to/ciOptions.cmake ...
```

Ordinarily, the file `ciOptions.cmake` might be empty or just contain a few common settings such as turning on optional features. For the branch, it might contain things like this:

Exclusively for sbmain17@gmail.com Transaction: 0098315591

ciOptions.cmake

```
compile_definitions(DO_EXTRA_CI_CHECKS=1)
set(ENABLE_SANITIZERS YES)
```

Injecting files into `project()` commands like this should not be part of the normal development of a project. It has specific uses for overcoming deficiencies in older projects and for very controlled situations such as in continuous integration builds. Outside of those cases, developers should generally prefer to add or modify the project's `CMakeLists.txt` files directly.

29.10. Presets

As a developer becomes more experienced with CMake, certain usage patterns and scenarios tend to be repeated. Some common examples include:

- A preferred way of placing and naming build directories relative to the source directory.
- A preference for a particular CMake generator on a given platform.
- Setting a certain combination of CMake cache variables to particular values for a given project.
- Re-using a toolchain file across projects, sometimes in conjunction with setting a few CMake cache variables or selecting a particular CMake generator.

These are all things that relate to setting up a fresh build directory for a project. Developers will sometimes create scripts to automate the process, or they might use undocumented CMake features to pre-load the CMake cache. While these measures may be an improvement, they are not always robust or convenient.

CMake 3.19 added a new feature called *presets* which allow scenarios like those above to be handled in a more reliable and convenient way. Presets can specify the build directory, CMake generator, target architecture, host toolset, CMake variables, environment variables and even vendor-specific content (e.g. for certain IDE tools). These affect the configure phase of the developer workflow. CMake 3.20 extended the presets functionality further, adding support for sections that affect the build and test phases.

CMake looks in the top source directory of a project for files named `CMakePresets.json` and `CMakeUserPresets.json`. Neither file is required to be present, but if the user tries to list or use presets, at least one of the files must exist. If both files are present, they will effectively be merged by reading the `CMakePresets.json` file first and then the `CMakeUserPresets.json` file. The two files have exactly the same format, but they serve different purposes:

`CMakePresets.json`

This file should only be provided by the project. It can be used to define presets for things like continuous integration jobs or cross-compilation setups using toolchains provided as part of the project (see Section 28.2.2, "Other Uses For FetchContent" for one way this might be done). It should not refer to any paths or files outside of the source or build directories, except perhaps well-known locations that would typically be present on most host systems. The preset file should be kept under version control, just like other parts of the project.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

`CMakeUserPresets.json`

This file provides presets defined by the developer for use on their own machine. The file contents can refer to any file or path without restriction, since they only have to make sense for that user's local machine. Projects should never provide a `CMakeUserPresets.json` file and it should ideally be excluded from source control. Developers may want to add this file name to their global git ignore configuration or other equivalent source control settings.

The CMake documentation includes a detailed explanation of the format of preset files. The following example shows the most important top level elements:

```
{
  "version": 2,
  "configurePresets": [...],
  "buildPresets": [...],
  "testPresets": [...]
}
```

The `version` field specifies the JSON schema and must always be present. Version 1 is the first release and only supports `configurePresets`. Version 2 requires CMake 3.20 or later and adds support for `buildPresets` and `testPresets`. Version 3 adds other capabilities and requires CMake 3.21 or later.

Configure presets apply to the configure phase of the developer workflow. They can be used when running CMake to set up a build directory for a project. For a preset file to be useful, it needs to provide a `configurePresets` section containing an array of one or more presets. A few key examples illustrate the most important elements and essential usage.

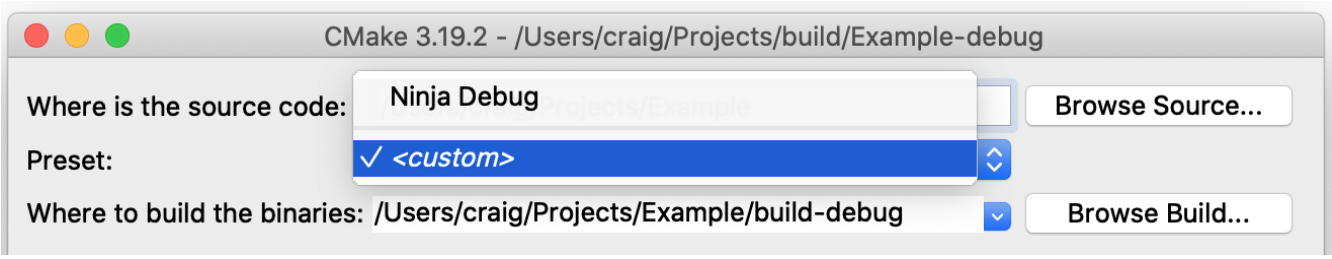
```
{
  "version": 1,
  "configurePresets": [
    {
      "name": "ninja",
      "displayName": "Ninja Debug",
      "generator": "Ninja",
      "binaryDir": "build-debug",
      "cacheVariables": { "CMAKE_BUILD_TYPE": "Debug" }
    }
  ]
}
```

Every preset must provide a `name`. It is a single word that serves as the unique name of the preset. A `displayName` is optional, but recommended. It is shown in GUI applications in combo boxes, etc. The `name` and `displayName` (if defined) of all available (non-hidden) configure presets can be obtained by running the following command from the top source directory of a project:

```
cmake --list-presets

Available configure presets:

  "ninja" - Ninja Debug
```


Exclusively for sbmain17@gmail.com Transaction: 0098315591

The CMake GUI also shows the available configure presets for the currently selected source directory. Presets are listed by their `displayName`, falling back to the `name` if no `displayName` is provided. The `ccmake` tool does not support presets.



The `name` is used to select a configure preset on the `cmake` command line. When run from the top source directory of the project, the following would select the configure preset defined above:

```
cmake --preset=ninja
```

With version 1 or 2, configure presets must also provide a `generator` and `binaryDir`, although these can be inherited (see further below). Version 3 allows the `generator` and `binaryDir` to be omitted, in which case they follow the same behavior as when presets are not used.

If the configure preset provides a `binaryDir` field (and also a `generator` field if it is a version 1 or 2 presets file), the `--preset=…` option is enough on its own to configure a build directory. Any of the usual CMake options can still be provided on the `cmake` command line too and will override the preset, where relevant. For instance, the above example defines the `CMAKE_BUILD_TYPE` cache variable. Other cache variables can be added with `-D` options on the command line, or even override the `CMAKE_BUILD_TYPE`.

Cache variables can be defined in the simple *key:value* form as shown in the above example, or they can be defined as a JSON object. The non-object form will define the variable as type `STRING` unless the value is an unquoted `true` or `false`, in which case it will be of type `BOOL`. The object form allows the variable type to be specified, which can improve the way the variable is presented in GUI applications. For example:

```
{
  "version": 1,
  "configurePresets": [
    {
      "name": "ninja",
      "generator": "Ninja",
      "binaryDir": "build",
      "cacheVariables": {
        "CMAKE_TOOLCHAIN_FILE": {
          "type": "FILEPATH",
          "value": "/path/to/toolchain.cmake"
        }
      }
    }
  ]
}
```

Exclusively for sbmain17@gmail.com Transaction: 0098315591

Presets can inherit from one or more other presets of the same type (configure, build or test) to add or override fields. This can be used to compose presets from common building blocks, thereby avoiding duplication and reducing errors. Importantly, presets defined in `CMakeUserPresets.json` can inherit from those defined in `CMakePresets.json`, but not vice versa. A preset can also be hidden, indicating it only exists so that other presets can inherit from it.

CMakePresets.json

```
{
  "version": 1,
  "configurePresets": [
    {
      "name": "ci-enable-all",
      "hidden": true,
      "binaryDir": "build",
      "cacheVariables": { "MYPROJ_FEATURE_X": true, "MYPROJ_FEATURE_Y": true }
    },
    {
      "name": "ci-linux",
      "inherits": "ci-enable-all",
      "displayName": "Linux continuous integration setup",
      "generator": "Ninja",
      "cacheVariables": { "CMAKE_BUILD_TYPE": "Release" }
    },
    {
      "name": "ci-macos",
      "inherits": "ci-enable-all",
      "displayName": "macOS continuous integration setup",
      "generator": "Xcode"
    }
  ]
}
```

CMakeUserPresets.json

```
{
  "version": 1,
  "configurePresets": [
    {
      "name": "sibling-build-dir",
      "hidden": true,
      "binaryDir": "${sourceParentDir}/build/${sourceDirName}"
    },
    {
      "name": "ninja",
      "inherits": ["sibling-build-dir", "ci-enable-all"],
      "displayName": "Ninja Debug",
      "generator": "Ninja",
      "cacheVariables": { "CMAKE_BUILD_TYPE": "Debug" }
    }
  ]
}
```


Exclusively for sbmain17@gmail.com Transaction: 0098315591

The `CMakePresets.json` file defines a base configure preset called `ci-enable-all` which the other two configure presets in that file both inherit from. This allows the common set of cache options and the `binaryDir` to be shared without having to repeat them everywhere. The `ci-linux` and `ci-macos` configure presets only need to define the information specific to their own generator and platform.

The `CMakeUserPresets.json` file shows how to inherit from multiple presets, including one defined in `CMakePresets.json`. Both `ci-enable-all` and `sibling-build-dir` define a `binaryDir`. In such cases where multiple base presets define the same field and the derived preset doesn't override it, the one that appears earlier in the `inherits` list takes precedence. In this case, for the `ninja` preset, the `binaryDir` from `sibling-build-dir` is used because it appears before `ci-enable-all`. If the `ninja` preset itself defined a `binaryDir`, that would override the others.

Another feature demonstrated by the `CMakeUserPresets.json` example is the use of macros of the form `${macroName}`. These evaluate to a computed value much like how CMake variables work. The CMake documentation provides the full list of available macro names and the fields that support them. Some of the more useful macro names include:

`presetName`

The name of the preset selected by the developer. When used in a base preset, it provides the name of the final preset that was selected, not the name of the base preset.

`generator`

The generator as seen by the final selected preset (i.e. taking inheritance into account).

`sourceDir`

The full absolute path to the project source directory.

`sourceParentDir`

The full absolute path to the parent of the project source directory.

`sourceDirName`

The name of the project source directory without any path. This is the same as the last part of `sourceDir`, i.e. the part after the last path separator.

`hostSystemName`

Provides the same value as the `CMAKE_HOST_SYSTEM_NAME` variable. Requires version 3 or higher.

The `CMakeUserPresets.json` in the above example shows how `${sourceParentDir}` and `${sourceDirName}` can be used to make a preset file more generic. This could be important if re-using it across multiple projects (e.g. by copying or symlinking it into each project's source directory). `${sourceParentDir}` allows the build directories to be placed outside the project source directory, as recommended in Section 2.2, "Out-of-source Builds". When all project sources are siblings of each other, this conveniently collects all build directories under a single location. `${sourceDirName}` provides the project-specific part that ensures each project's build directory is unique.

Note how the names given to the presets in `CMakePresets.json` all have a `ci-` prefix. Good practice is to choose preset names in `CMakePresets.json` that would be unlikely to clash with names a developer might define in their own `CMakeUserPresets.json` file. Since continuous integration setup is something that the project should define, a `ci-` prefix is typically a good choice. This also clearly

Exclusively for sbmain17@gmail.com Transaction: 0098315591

communicates the intended use case for the presets. Using a prefix that incorporates the name of the project is another potential strategy.

Configure presets can modify the configure step's environment with an `environment` map. This does not carry forward to the build step unless a build preset is used (discussed further below). Environment variables can also be evaluated with either `$env{varName}` or `$penv{varName}` in fields where macros are supported. The only difference between the two is that `$penv{varName}` always provides the value from the parent environment, whereas `$env{varName}` will take into account changes made to the environment by the preset. `$penv{varName}` is necessary when the current value of an environment variable needs to be included in the new value, as is common for variables like `PATH`. An environment variable can also be given the value `null` to cause it to be unset.

```
{
  "version": 1,
  "configurePresets": [
    {
      "name": "ninja",
      "generator": "Ninja",
      "binaryDir": "build",
      "environment": {
        "PATH": "${sourceDir}/scripts:$penv{PATH}",
        "PATH_COPY": "$env{PATH}",
        "CLEAR_ME_PLEASE": null
      }
    }
  ]
}
```

When a preset inherits from another and they both define an `environment`, the result is a merged union of all presets in the inheritance hierarchy. If multiple presets define the same environment variable, the value taken follows the same inheritance rules as described earlier (base values are overridden by derived presets, earlier inherited presets take precedence over later ones in the `inherits` list). Similar merging behavior is used for `cacheVariables` as well.

When a configure preset uses a Visual Studio generator, it may also want to specify an architecture or host toolset. Dedicated fields are provided for this and can be used as in the following example:

```
{
  "version": 1,
  "configurePresets": [
    {
      "name": "vs2019-arm64",
      "displayName": "Visual Studio 2019 - ARM64",
      "generator": "Visual Studio 16 2019",
      "architecture": "ARM64",
      "toolset": "host=x86",
      "binaryDir": "build-arm64"
    }
  ]
}
```


Exclusively for sbmain17@gmail.com Transaction: 0098315591

The version 3 schema supports a `condition` object, which can be used to enable a preset based on certain rules. For example, a preset with a Xcode `generator` would only be useful on a macOS machine. The condition can be a constant, string equality, inequality, regular expression match or a check for whether a list contains a particular value. Boolean logic can also be expressed using `allOf`, `anyOf` and `not` relationships. The official CMake documentation defines the supported conditions fairly comprehensively, but the following examples give an idea of some typical uses:

```
{
  "version": 3,
  "configurePresets": [
    {
      "name": "default-generator",
      "binaryDir": "build"
    },
    {
      "name": "xcode",
      "generator": "Xcode",
      "binaryDir": "build-xcode",
      "condition": {
        "type": "equals",
        "lhs": "${hostSystemName}",
        "rhs": "Darwin"
      }
    }
  ]
}
```

```
{
  "version": 3,
  "configurePresets": [
    {
      "name": "package-release",
      "generator": "Ninja",
      "binaryDir": "build",
      "condition": {
        "type": "allOf",
        "conditions" : [
          {
            "type": "equals",
            "lhs": "$penv{CI_COMMIT_REF_PROTECTED}",
            "rhs": "true"
          },
          {
            "type": "matches",
            "string": "$penv{CI_COMMIT_TAG}",
            "regex": "release/.*"
          }
        ]
      }
    }
  ]
}
```

Exclusively for sbmain17@gmail.com Transaction: 0098315591

The preceding discussion focused on `configurePresets`, which apply to the configure phase. The `buildPresets` provide a similar capability for when running the build tool. The developer can select a build preset to use at build time like so:

```
cmake --build --preset name
```

Build presets support many of the same fields as configure presets, such as `name`, `displayName`, `inherits`, `hidden` and `environment`. Each of these work the same way as for configure presets. In addition, a build preset must specify (or inherit) the name of a configure preset in a field called `configurePreset`, which is used to determine the location of the build directory. A build preset also inherits any environment from the configure preset by default, although this can be disabled. The following example demonstrates these capabilities:

```
{
  "version": 2,
  "configurePresets": [
    {
      "name": "ninja",
      "generator": "Ninja Multi-Config",
      "binaryDir": "build",
      "environment": {
        "PATH": "/tools/dir:$penv{PATH}"
      }
    }
  ],
  "buildPresets" : [
    {
      "name": "base",
      "hidden": true,
      "configurePreset": "ninja",
      "configuration": "Release"
    },
    {
      "name": "devtools",
      "inherits": "base",
      "inheritConfigureEnvironment": false,
      "targets": ["hexdump", "logger"]
    },
    {
      "name": "alldocs",
      "inherits": "base",
      "targets": ["manual", "api", "quickstart"]
    }
  ]
}
```

Listing the available build presets for the above example, it can be confirmed that two build presets are available from the command line:

```
cmake --build --list-presets
```


Exclusively for sbmain17@gmail.com Transaction: 0098315591

```
Available build presets:

  "devtools"
  "alldocs"
```

Both of these build presets inherit the hidden `base` build preset, which in turn is associated with the `ninja` configure preset. The result is that both `devtools` and `alldocs` will be associated with the build directory defined by the `ninja` configure preset. The `devtools` build preset also explicitly discards the environment changes defined by the configure preset.

Many other command line options can also be expressed as part of a build preset. For instance, the above example specifies the `Release` configuration as part of the `base` build preset, which is then inherited by `devtools` and `alldocs` too. The CMake documentation specifies the full list of available fields and the command line option each one corresponds to.

Test presets follow a very similar pattern to build presets, except they are applied to invocations of `ctest`:

```
ctest --preset name
ctest --list-presets
```

Test presets support the same common `name`, `displayName`, `inherits`, `hidden` and `environment` fields, plus others that map to various `ctest` command line options. They can specify commonly used combinations of test fixture settings, regular expressions for selecting or excluding tests, test output options and so on. The following illustrates how some of these features can be used:

```
{
  "version": 2,
  "configurePresets": [
    {
      "name": "ninja", ...
    }
  ],
  "testPresets" : [
    {
      "name": "mytests",
      "configurePreset": "ninja",
      "filter": {
        "include": {
          "name": "SomeFeature"
        },
        "exclude": {
          "label": "Slow"
        }
      }
    }
  ]
}
```

Exclusively for sbmain17@gmail.com Transaction: 0098315591

The material in this section only covered the basics of using presets. They are best learned by experimenting with real world projects, trying out different ideas and exploring how the various capabilities can be used. Consult the presets manual in the CMake documentation for a comprehensive coverage of all the available features.

29.11. Recommended Practices

The way projects are structured and used can vary considerably. Some things that used to be commonplace are now considered poor practice, as new features and lessons learned allow older methods to be replaced by newer ones that are more robust, more flexible and allow things that were not possible previously. Tools are upgraded, languages evolve, dependencies change - all of these things mean that projects will also need to adapt over time. For CMake projects in particular, those that continue to target older CMake versions before 3.0 will increasingly face a bumpy path. There is a strong move toward a target-centric model and much of CMake's development is geared in that direction. Therefore, prefer to set a minimum CMake version that allows the project to make use of those features. Anything less than CMake 3.1 is likely to be too restrictive, so consider at least CMake 3.7 where possible due to the updated language support and new features. If working with newer tools like CUDA or a very recent language standard, the latest CMake release is strongly advised. New releases of Visual Studio or Xcode also tend to require recent CMake versions in order to pick up fixes and additions for changes in those toolchains.

A fundamental choice that every project needs to make is whether to structure itself as a superbuild or as a regular build. If the project can set a minimum CMake version of 3.11, the non-superbuild arrangement has more powerful features available to it for dependency management which may make the need for a superbuild unnecessary. Consider whether the `FetchContent` module and the promotion of local imported targets to global scope offer more flexibility and a better experience for developers. Where all dependencies of a project are relatively mature and have well defined install rules, a superbuild may still be a suitable alternative and comes with the advantage that it can be used with much older CMake versions. Both methods have their place, but the earlier in a project's life that the decision can be made on whether or not to use a superbuild, the more likely the project can avoid large scale disruptive restructuring later on.

Irrespective of whether a project is a superbuild or not, aim to keep the top level of the project focused on the higher level details. Think of the top level `CMakeLists.txt` file as being more like a table of contents for the project. The top level directory should mostly just contain administrative files and a set of subdirectories each focused on a particular area. Avoid subdirectory names that may cause clashes with those created in the build directory automatically. Prefer instead to use fairly standard names unless there is an existing convention that must be followed. For regular projects, aim to make the top level `CMakeLists.txt` file follow the common section pattern of:

- Preamble
- Project wide setup
- Dependencies
- Main build targets
- Tests
- Packaging

Exclusively for sbmain17@gmail.com Transaction: 0098315591

Clearly delineating each section with comment blocks will help encourage developers working on the project to maintain that structure. Establishing this pattern across projects will help reinforce the focus on keeping the top level `CMakeLists.txt` file streamlined and acting as a high level overview.

When defining build targets that have sources spread across directories, prefer to create the target first, then have each subdirectory add sources to it using `target_sources()`. Where appropriate, group the subdirectories by functionality or feature so that they can be easily moved around or enabled/disabled as a unit. In many cases, the other target-focused commands (i.e. `target_compile_definitions()`, `target_compile_options()` and `target_include_directories()`) can then also be used locally within the subdirectory that they relate to. This helps keep information close to the location where it is relevant rather than spreading it across directories. Avoid using variables to build up lists of sources to be passed back up through directory hierarchies and eventually used to create a target, define compiler flags, etc. The use of variables instead of operating on targets directly is much more fragile, more verbose and less likely to result in CMake catching typos or other errors.

Following on from the above and reiterating one of the recommendations from Chapter 4, *Building Simple Targets*, avoid the all too common practice of unnecessarily using a variable to hold the name of a target or project. The following pattern in particular should be avoided:

```
set(projectName ...)
project(${projectName})
add_executable(${projectName} ...)
```

The above example ties together things that should not be so strongly related. The project name should rarely change. Specify the name of the project directly in the `project()` command and use the standard variables CMake provides if it needs to be referred to elsewhere in the project. For targets, the target name is used so widely that trying to carry it around in a variable is both cumbersome and error prone. Give the target a name and use that name consistently throughout the project. Even if there is only one target in the whole project, it doesn't necessarily have to be the same as the project name and the two should be considered distinct.

When adding tests, consider keeping the test code close to the code being tested. This helps keep logically related code together and encourages developers to keep tests up to date. Tests that are distributed to other parts of the source directory hierarchy can easily be forgotten. For tests that draw on multiple areas such as integration tests, the locality principle is not as strong, so collecting these higher level tests in a common place may be appropriate. The top level `tests` subdirectory is intended for situations such as this.

For larger projects, consider whether it is worth organizing the way the project is presented in IDE tools. If there are many targets, it can be difficult to work with the project unless some structure is added using the `FOLDER` target property. For those targets with many sources, they too can be organized using the `source_group()` command, which can be used to define group hierarchies around whatever concepts or features make sense.

Special consideration should be given to projects that are anticipated to be built on Windows, especially where developers may use the Visual Studio IDE. The lack of `RPATH` support means

Exclusively for sbmain17@gmail.com Transaction: 0098315591

executables rely on being able to find their DLL dependencies in either the same directory or via the `PATH` environment variable. This impacts both test programs run through `ctest` and the developer's ability to run executables from within the Visual Studio IDE. Forcing all executables and DLLs into the same output directory is one solution to this problem, made possible by the various `⋯OUTPUT_DIRECTORY` target properties and their associated `CMAKE_⋯OUTPUT_DIRECTORY` variables. These are frequently used to create a directory layout that mirrors that used when the project is installed. Avoid copying DLLs in post build rules or custom tasks to put them in multiple locations so that other executables can find them. This is fragile and can easily result in stale DLLs mistakenly being used.

Test programs would ideally not be collected to the same place as the main programs and DLLs. Some test code may need to find other files relative to their own location, so keeping them separate may even be a requirement. Use the `ENVIRONMENT` test property to specify an appropriate `PATH` to ensure tests can find their DLLs when run through `ctest`. Also consider using CMake 3.13 or later and populating the `VS_DEBUGGER_ENVIRONMENT` property on all executable targets too so that they can be run directly from within the Visual Studio IDE.

When using the Visual Studio generator, prefer to leave the PDB settings at their defaults. This typically results in the PDB file appearing in the location developers expect and with a name that matches the executable or library they correspond to. Trying to change the output directory of PDB files has implementation complexities when generator expressions are used and it can be difficult to get the PDB files into the desired directory in some cases.

If a project's continuous integration system does not clear the CMake cache before each job (e.g. it uses incremental builds), care must be taken to avoid having cache variables from a previous build affect later builds in unintended ways. Prefer to use the `CMAKE_PROJECT_<PROJNAME>_INCLUDE` variable to inject any CI-specific logic via a dedicated file under source control. This allows regular CMake variables to be set without having to save them in the cache, thereby reducing opportunities for them to affect later builds if those variables are subsequently removed from the CI logic.

On the other hand, if every CI build starts with an empty initial build directory, then presets may be a more convenient and more robust mechanism for defining the CI configurations. Projects should only provide a `CMakePresets.json` file and let developers create their own `CMakeUserPresets.json` file if they want to. Avoid using preset names in `CMakePresets.json` that would be likely to clash with names developers would use in their own user presets file. A good strategy is to start each preset name in `CMakePresets.json` with a prefix like `ci-` or the project name. For developers, consider creating a `CMakeUserPresets.json` file that captures common configurations which can be used across different projects, or at least that require minimal tailoring per project. Use `inherits` to compose presets in a "mix-in" fashion to avoid repeating the same details across related presets. Add `CMakeUserPresets.json` to the user-wide source control ignore list to minimize the chances of that file ever being committed to a repository.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

Chapter 30. Build Performance

For larger projects, build times can be non-trivial, extending out to hours in extreme cases. The impact of a long build time can go beyond mere annoyance, it can drive developer behavior toward undesirable practices and discourage experimentation of new ideas. Long builds can also cause difficulties with continuous integration systems, causing job delays and increasing requirements on hardware resources to cope with the high build volumes.

A variety of factors can contribute to long build times. Poor project structure can lead to the same source files being compiled multiple times with the same settings for different targets. Appropriate use of libraries can usually eliminate such problems, although this may require some refactoring to decouple parts of the code.

Specifying linking relationships between targets that don't actually need to be linked is another source of inefficiency. It reduces the ability of the build tool to execute build tasks in parallel and it expands the set of targets that have to be relinked whenever the dependee is rebuilt.

Exposing private details in public headers of C or C++ projects is another example of how code structure can cause more things to be rebuilt than necessary when a change is made. Reducing coupling between code is one of the most effective ways to reduce incremental build times when making changes during development. The material covered in Section 20.2, "Source Code Access To Version Details" is a prime example of using code structure to minimize rebuild times.

When seeking to improve build performance, addressing structural problems such as those mentioned above should be the first priority. Specifying correct relationships between entities is a pre-requisite for many other techniques to be robust or effective. With those areas addressed, attention can turn to the techniques discussed in the remainder of this chapter.

30.1. Unity Builds

A unity build (also sometimes referred to as a *jumbo* build) is where source files that would normally be compiled individually are effectively concatenated to produce a smaller number of larger files. If each of the individual source files include similar sets of headers, then the compiler would normally be processing those headers multiple times, one for each individual source file. When the individual sources are concatenated into a single combined source file, the compiler can avoid processing each header more than once. If the header processing time is non-trivial, this can lead to meaningful savings in build time.

CMake 3.16 and later supports automatically converting a target's sources to a unity build. By setting the `UNITY_BUILD` target property to true, CMake will take care of combining sources into one or more unity sources and building those instead of the originals. This process is transparent to the project, requiring nothing more than enabling the feature via the target property. Not all languages can have their sources combined in this way, but CMake supports doing so for at least C and C++ sources (other languages' sources will be left as they are and compiled individually). Sources will only be combined with other sources of the same language.

The `UNITY_BUILD` target property is initialized by the `CMAKE_UNITY_BUILD` variable when the target is created. Setting that variable to true can be an effective way to enable unity builds across the entire

Exclusively for sbmain17@gmail.com Transaction: 0098315591

project, although it should usually be a cache variable to allow developers to easily switch it off if they need to. Making it a non-cache variable could cause problems in hierarchical builds, since the developer would not be able to easily disable unity builds in dependency sub-projects.

In practice, it may take more than simply turning on the unity build feature to get a worthwhile or even buildable result. There are some restrictions to be aware of when combining source files in this way:

- Source files often define their own local file scope entities, such as types, variables, pre-processor macros and so on. When each source file is built individually, these file-scope entities are kept separate, but when combined in a unity build, they can clash between files.
- Source files may already be quite large. This can be especially common for sources produced by code generation tools. If combining sources results in an excessively large unity source file, the compiler may run out of memory or exceed other internal limits during processing.

Strategies like using a naming convention to avoid duplicating symbol names of file scope entities can help with the first of the above points. Sometimes, that may not be practical or there may be other constraints preventing such changes. For the second point, large source files should be excluded from unity builds, since there is little to gain when they are already of a substantial size.

CMake provides a few controls which may help in situations like the above. If a small number of sources are problematic, they can be excluded from being combined with other sources on a case-by-case basis. This is done by setting the `SKIP_UNITY_BUILD_INCLUSION` property on the problematic source files to true. Those sources will then never be combined with others in a unity build. Note that CMake already excludes any source file which isn't normally compiled (e.g. header files) or any source file which has any of the following source properties defined:

- `COMPILE_OPTIONS`
- `COMPILE_DEFINITIONS`
- `COMPILE_FLAGS`
- `INCLUDE_DIRECTORIES`

For resolving name clashes, especially for symbols in C++ anonymous namespaces, the `UNITY_BUILD_UNIQUE_ID` target property available with CMake 3.20 or later may be useful. It's official documentation contains a description of the types of problems it is intended to solve, details of its usage and a simple example. In general though, prefer to rework the source code to avoid the name clashes if possible, as it will be simpler and much easier to understand.

With CMake 3.18 or later, the `UNITY_BUILD_MODE` target property specifies the method for choosing which sources to combine together. If defined, this property may hold the values `BATCH` or `GROUP`. If the property is not defined or if using CMake 3.17 or earlier, the behavior is the same as `BATCH`.

30.1.1. BATCH Mode

In `BATCH` mode, CMake combines sources based on the order in which they were added to the target. The `UNITY_BUILD_BATCH_SIZE` target property controls the maximum number of sources which may be combined. The first `UNITY_BUILD_BATCH_SIZE` files for a particular language will be combined into one unity source, the next `UNITY_BUILD_BATCH_SIZE` sources will be combined into another unity source,

Exclusively for sbmain17@gmail.com Transaction: 0098315591

and so on until all original sources have been included or skipped for reasons discussed above. This property is initialized by the `CMAKE_UNITY_BUILD_BATCH_SIZE` variable if that variable is defined, otherwise it is given an initial value of 8.

This method is suitable when none of the sources are particularly big. If some sources are much larger than others, the large sources can be listed away from each other as a crude way of minimizing the likelihood that they will be combined together. This is not particularly robust, but may be good enough for some projects.

```
option(CMAKE_UNITY_BUILD "Enable unity builds")
add_executable(MyApp
    someBigSource.cpp
    little1.cpp
    little1.h             ①
    little2.cpp
    little2.h             ①
    # ---------
    anotherBigSource.cpp
    customFlags.cpp       ②
    mustBeSeparate.cpp    ③
    small1.cpp
    # ---------
    differentLang1.c
    differentLang2.c
)
set_target_properties(MyApp PROPERTIES
    UNITY_BUILD_BATCH_SIZE 3
)
set_source_files_properties(customFlags.cpp PROPERTIES
    COMPILE_DEFINITIONS
        COMPVER=${CMAKE_CXX_COMPILER_VERSION}
)
set_source_files_properties(mustBeSeparate.cpp PROPERTIES
    SKIP_UNITY_BUILD_INCLUSION YES
)
```

① Headers are not compiled, so they are not added to unity sources.

② Source files with certain source properties set are automatically excluded from being added to unity sources.

③ Source file explicitly excluded from unity sources due to having its `SKIP_UNITY_BUILD_INCLUSION` source property set to true.

The two big source files `someBigSource.cpp` and `anotherBigSource.cpp` are kept separate by limiting the unity batch size to 3 and ensuring that there are enough source files listed between these two that they are at least 3 sources apart in the source list. CMake will combine `someBigSource.cpp`, `little1.cpp` and `little2.cpp` into the first unity source, then `anotherBigSource.cpp` and `small1.cpp` into a second unity source. Note also that `differentLang1.c` and `differentLang2.c` are not combined with the other C++ sources, they are placed in their own unity source because they are a different language. The actual names of the unity sources are an internal implementation detail, projects should not attempt to refer to them directly.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

30.1.2. GROUP Mode

When more precise control is needed over how sources should be combined, `GROUP` mode is more appropriate. When `UNITY_BUILD_MODE` is set to `GROUP`, each source file in the target needs to specify the name of the unity group it belongs to with the `UNITY_GROUP` source file property. Any source that does not define this property will not be added to a unity source and will be compiled directly (i.e. as though unity builds were not enabled).

No batch size limits are applied to unity groups, the `UNITY_BUILD_BATCH_SIZE` property is ignored. The project is in full control over which sources are combined. There could be different strategies for how the project groups sources. It could be based on file size, on the set of headers that are included, on keeping related code together to maximize optimization opportunities, etc. The following shows an alternative way of grouping the sources of the previous example:

```
option(CMAKE_UNITY_BUILD "Enable unity builds")
add_executable(MyApp
    someBigSource.cpp
    anotherBigSource.cpp
    little1.cpp
    little1.h
    little2.cpp
    little2.h
    small1.cpp
    customFlags.cpp
    mustBeSeparate.cpp
    differentLang1.c
    differentLang2.c
)
set_source_files_properties(customFlags.cpp PROPERTIES
    COMPILE_DEFINITIONS
        COMPVER=${CMAKE_CXX_COMPILER_VERSION}
)
set_target_properties(MyApp PROPERTIES
    UNITY_BUILD_MODE GROUP
)
set_source_files_properties(
    little1.cpp little2.cpp small1.cpp
    PROPERTIES UNITY_GROUP small
)
set_source_files_properties(
    differentLang1.c differentLang2.c
    PROPERTIES UNITY_GROUP different
)
```

In the above, only some of the source files have a group defined. Those that don't will be compiled individually, which is convenient in this case since `someBigSource.cpp`, `anotherBigSource.cpp` and `mustBeSeparate.cpp` should all be excluded from unity builds. The `customFlags.cpp` source is also excluded (it has to be since it has its `COMPILE_DEFINITIONS` source property set).

As the above example shows, the project can sometimes provide more appropriate grouping than the `BATCH` method. The extra verbosity and maintenance burden it introduces should be weighed against any overall gain in build performance.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

30.2. Precompiled Headers

Another technique that can sometimes improve build times is to employ precompiled headers to cache some of the work done by the compiler. For cases where header file processing is a non-trivial contributor to build times, this technique can lead to a worthwhile saving.

Most major compilers support precompiled headers, but there are significant differences between the various implementations. CMake 3.16 introduced direct support for precompiled headers which takes away the burden of having to manage these differences in each project. The primary way to instruct CMake to set up precompiled headers for a target is with the `target_precompile_headers()` command. This follows the same form as all the other `target_⋯()` commands:

```
target_precompile_headers(targetName
    <PRIVATE|PUBLIC|INTERFACE> headers...
   [<PRIVATE|PUBLIC|INTERFACE> headers...] ...
)
```

The `PRIVATE`, `PUBLIC` and `INTERFACE` keywords have their usual meaning, specifying whether the `headers` that follow the keyword should be applied to the specified target and/or to things linking against the target. `PRIVATE` headers are appended to a target property called `PRECOMPILE_HEADERS`, whereas `INTERFACE` headers are appended to the `INTERFACE_PRECOMPILE_HEADERS` property. `PUBLIC` headers are appended to both properties. CMake uses these properties to build up a set of headers that will be force-included when compiling sources for the specified target or things that link against it, as appropriate. The method used to accomplish this varies between compilers, but from a project perspective, an important point is that no source changes are required. CMake will generate files and use command line arguments to make compilers include the specified headers rather than requiring sources to be modified to explicitly include any special header.

In typical scenarios, precompile headers should be specified as `PRIVATE`. The main reason for supporting non-private relationships for a precompile header is to support interface libraries whose reason for existence is to collect together commonly used compiler definitions, options and so forth. Targets opt in to having these applied to themselves by linking to the interface library. The following example demonstrates such an arrangement:

```
add_library(MyCommonPCH INTERFACE)
target_precompile_headers(MyCommonPCH INTERFACE
    <iostream>
    <vector>
    <algorithm>
    [["myAlgo.h"]]
)
target_include_directories(MyCommonPCH INTERFACE
    $<BUILD_INTERFACE:${CMAKE_CURRENT_LIST_DIR}>
)

add_library(ShipBuilder ships.cpp)
add_library(CarBuilder  cars.cpp)
target_link_libraries(ShipBuilder PRIVATE MyCommonPCH)
target_link_libraries(CarBuilder  PRIVATE MyCommonPCH)
```

Exclusively for sbmain17@gmail.com Transaction: 0098315591

In the list of headers provided to target_precompile_headers(), note how lua-style brackets have been used to ensure that the quotes are included as part of the header. The delimiters around each header name are significant, as they affect how that header is treated when CMake forms the internal files used to collect the headers to be precompiled. Where a header is surrounded by angle brackets or quotes, they are used as is. The effect is as though they were preceded by #include and used verbatim. Such headers are expected to be found on the header search path for the source file that is using them. This is why the example also adds CMAKE_CURRENT_LIST_DIR to the header search path for things that link against MyCommonPCH (see Section 26.2.1, "Interface Properties" for why the $<BUILD_INTERFACE:…> generator expression is also used here).

Where there are no angle bracket or quote delimiters at all, the target_precompile_headers() command will assume headers are paths relative to the current source directory and will convert them to an absolute path if they are not already absolute. This would make such interface libraries unsuitable for being installed, since they would have paths from the build tree embedded in their interface properties.

A potentially simpler and more efficient way to define a common set of precompile headers is to use an alternative form of the target_precompile_headers() command:

```
target_precompile_headers(targetName
    REUSE_FROM otherTarget
)
```

The REUSE_FROM keyword instructs the command to apply the precompile headers from otherTarget to targetName as well. Note that this is not copying the precompile header settings to targetName, it is literally reusing the precompile header configuration from otherTarget, including the internal files generated for it. This can lead to fewer generated files compared to using an interface library to share common settings between targets, potentially resulting in lower disk space being used. In projects with many targets sharing the same precompiled header, the savings can be significant.

An important prerequisite for the REUSE_FROM form is that both targets must use a compatible set of compiler flags, definitions, etc. Without this restriction, the precompiled header from otherTarget might be different to what should have been generated for targetName with its own unique settings. Compiler flags or definitions which do not affect the precompiled header can potentially be different between the two targets, or omitted in one but not the other. Projects should aim to minimize such cases to avoid potential compiler warnings.

30.3. Compiler Caches

In many cases, significant improvements in build performance can be achieved by employing a compiler cache. The purpose of such a cache is to keep previously built object files so that if the compiler is asked to compile the exact same source file with the same set of flags, headers and toolchain, the object file can be retrieved directly from the cache instead of having to be compiled again. A well-configured compiler cache can be one of the most effective ways to reduce build times.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

When using GCC, Clang or any compiler that claims to be compatible with either of those, `ccache` is one of the most commonly used compiler cache tools. It works in one of two ways:

- Replace the location of the default system compilers with symlinks to itself. Many Linux distributions employ this arrangement.
- Precede the normal compiler command line with `ccache`.

The first option provides compiler caching as the default system-wide behavior, but it has drawbacks. The main limitation is that it doesn't allow the developer to choose which compiler to use, instead locking in the choice of using the default system compiler. In many cases, the default compiler is quite old, so developers frequently want to use newer compilers installed in other locations. Another limitation is that it requires administrator access to set up if the system doesn't install `ccache` that way by default.

A much more flexible approach is to precede the usual compiler command line with `ccache`. This method allows any supported compiler to be used, not just the system default. No system-wide change is required, so the technique can be employed freely by any user.

Some CMake generators support a compiler launcher feature which maps very closely to this second way of using `ccache`. Starting with CMake 3.4, the `<LANG>_COMPILER_LAUNCHER` target properties can be used to specify a list of items to precede the compiler command line in a language-specific way. This is supported by the Ninja and Makefiles generators only. The default value for these properties are taken from the corresponding `CMAKE_<LANG>_COMPILER_LAUNCHER` variables when a target is created.

The following demonstrates how to enable `ccache` for the whole build, but only after first confirming that `ccache` is available. This ensures the build will work with or without `ccache`.

```
find_program(CCACHE_PROGRAM ccache)

if(CCACHE_PROGRAM)
    set(CMAKE_C_COMPILER_LAUNCHER   ${CCACHE_PROGRAM})
    set(CMAKE_CXX_COMPILER_LAUNCHER ${CCACHE_PROGRAM})
endif()
```

For most projects, the above example is too simplistic. Certain `ccache` configuration options should typically be set to ensure safe and efficient caching behavior. To set these options on a per-project basis, various `CCACHE_…` environment variables can be used. The supported variables are all detailed in the `ccache` documentation, but a few of the more important ones are:

`CCACHE_CPP2`

Set this to `true` to avoid spurious warnings with some compilers. This is the default with `ccache` 3.3 or later, but to support earlier `ccache` releases as well, the project should explicitly set it.

`CCACHE_BASEDIR`

If there could be many projects building the same dependencies or generated sources in different build directories with the same build options, setting `CCACHE_BASEDIR` to `${CMAKE_BINARY_DIR}` may result in more cache hits. This is likely to be the case for projects that use `FetchContent` to bring their dependencies into the build, since they will always place the same

Exclusively for sbmain17@gmail.com Transaction: 0098315591

dependency at the same relative location under the build directory by default. Conversely, if most of the sources tend to be from the project itself, then setting `CCACHE_BASEDIR` to `${CMAKE_SOURCE_DIR}` may be a better setting. Developers should test the effectiveness of either setting before using it in a project.

`CCACHE_SLOPPINESS`

If precompiled headers are enabled, it is important to set this option to `pch_defines,time_macros`. The *Precompiled headers* section of the `ccache` documentation explains why these settings are needed. For improved cache hit performance, it may also be desirable to add `include_file_mtime` to the list of sloppiness options. This comes with a theoretical risk of a race condition, but for typical scenarios, that race condition is highly unlikely (files included by the preprocessor would have to be updated while the source is being compiled). Consult the `ccache` manual for details before deciding whether to add `include_file_mtime` to the sloppiness options.

`CCACHE_PREFIX`

`ccache` supports being used in conjunction with another compiler command wrapper. Popular examples include tools such as `distcc`, `icecc` and `sccache-dist` which distribute compilation across multiple machines to reduce build times. When chaining multiple compiler wrappers, it is recommended to have `ccache` as the first wrapper invoked and then specify the other wrapper through the `CCACHE_PREFIX` environment variable (e.g. `CCACHE_PREFIX=distcc`). `ccache` then handles chaining to the second wrapper if the result of the compile command is not already in its cache.

Developers should also become familiar with the different modes of operation that `ccache` offers. Direct mode, preprocessor mode and depend mode all have their advantages and restrictions, with direct mode being the default. Pay special attention to the interaction with features like precompiled headers, as problems can sometimes occur. As one example, depend mode is advisable when using precompiled headers with Clang to avoid issues with timestamps when switching branches, but changing compiler definitions may not be handled correctly in depend mode.

The `<LANG>_COMPILER_LAUNCHER` properties accept a list, so CMake's command mode can be used to set the relevant environment variables when invoking `ccache`. For example:

```
find_program(CCACHE_PROGRAM ccache)
if(CCACHE_PROGRAM)
    set(ccacheEnv
        CCACHE_CPP2=true
        CCACHE_BASEDIR=${CMAKE_BINARY_DIR}
        CCACHE_SLOPPINESS=pch_defines,time_macros
    )
    foreach(lang IN ITEMS C CXX OBJC OBJCXX CUDA)   ①
        set(CMAKE_${lang}_COMPILER_LAUNCHER
            ${CMAKE_COMMAND} -E env ${ccacheEnv} ${CCACHE_PROGRAM}
            PARENT_SCOPE
        )
    endforeach()
endif()
```

① Enable `ccache` for all supported languages (a more recent `ccache` version is required for `CUDA`).

Exclusively for sbmain17@gmail.com Transaction: 0098315591

The above technique is suitable for the Ninja or Makefiles generators, but not for the Xcode generator. `ccache` does support the `AppleClang` compiler, but a different method must be used to insert `ccache` before the compiler command and set the relevant environment variables.

When using the Xcode generator, the compilers to use for C and C++ can be overridden by setting the `CMAKE_XCODE_ATTRIBUTE_CC` and `CMAKE_XCODE_ATTRIBUTE_CXX` variables. The linkers must also be set with the `CMAKE_XCODE_ATTRIBUTE_LD` and `CMAKE_XCODE_ATTRIBUTE_LDPLUSPLUS` variables to work around a bug in at least some versions of the Xcode IDE where it can select the wrong linker language if only `CMAKE_XCODE_ATTRIBUTE_CC` and `CMAKE_XCODE_ATTRIBUTE_CXX` are set. Curiously, the bug is restricted to building within the IDE, any command-line builds using `xcodebuild` are not affected. Also recall that, as mentioned back in Section 23.1, "CMake Generator Selection", the `CMAKE_XCODE_ATTRIBUTE_…` variables only have an effect if set in the top level `CMakeLists.txt` file.

Each of the Xcode project variables allow specifying only a single value, but the command line needs to have multiple options (at least the `ccache` executable and the real compiler to be invoked). Therefore, a separate launch script needs to be written out and the project variables pointed at them. Relevant environment variables can also be set in these launch scripts. For example:

```
# Write out launch scripts for C and C++ languages
foreach(lang IN ITEMS C CXX)
    set(launch${lang} ${CMAKE_BINARY_DIR}/launch-${lang})

    file(WRITE ${launch${lang}}
       "#!/bin/bash\n\n"
       "export CCACHE_CPP2=true\n"
       "export CCACHE_BASEDIR=${CMAKE_BINARY_DIR}\n"
       "export CCACHE_SLOPPINESS=pch_defines,time_macros\n"
       "exec \"${CCACHE_PROGRAM}\" \"${CMAKE_${lang}_COMPILER}\" \"$@\"\n"
    )

    execute_process(COMMAND chmod a+rx ${launch${lang}})
endforeach()

# Redirect Xcode to use our launchers
set(CMAKE_XCODE_ATTRIBUTE_CC         ${launchC})
set(CMAKE_XCODE_ATTRIBUTE_CXX        ${launchCXX})
set(CMAKE_XCODE_ATTRIBUTE_LD         ${launchC})
set(CMAKE_XCODE_ATTRIBUTE_LDPLUSPLUS ${launchCXX})
```

Projects can combine the two methods to provide support for `ccache` across the different generators. It may be useful to implement the logic in its own function and call it from the top level `CMakeLists.txt` file, typically just after the `project()` command. The function should do nothing if not called from the top level, since that is a requirement for the Xcode generator. It also ensures that the top level project is in control for hierarchical builds. The function can be implemented in its own file and potentially in its own source repository, then added to the build via `FetchContent`. This is a convenient way to share the same logic across many projects.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

```
function(useCompilerCache)
    if(NOT CMAKE_CURRENT_SOURCE_DIR STREQUAL CMAKE_SOURCE_DIR)
        return()
    endif()

    find_program(CCACHE_PROGRAM ccache)
    if(NOT CCACHE_PROGRAM)
        return()
    endif()

    set(ccacheEnv
        CCACHE_BASEDIR=${CMAKE_BINARY_DIR}
        CCACHE_CPP2=true
        CCACHE_SLOPPINESS=pch_defines,time_macros
    )

    if(CMAKE_GENERATOR MATCHES "Ninja|Makefiles")
        foreach(lang IN ITEMS C CXX OBJC OBJCXX CUDA)
            set(CMAKE_${lang}_COMPILER_LAUNCHER
                ${CMAKE_COMMAND} -E env ${ccacheEnv} ${CCACHE_PROGRAM}
                PARENT_SCOPE
            )
        endforeach()
    elseif(CMAKE_GENERATOR STREQUAL Xcode)
        foreach(lang IN ITEMS C CXX)
            set(launch${lang} ${CMAKE_BINARY_DIR}/launch-${lang})
            file(WRITE ${launch${lang}} "#!/bin/bash\n\n")
            foreach(keyVal IN LISTS ccacheEnv)
                file(APPEND ${launch${lang}} "export ${keyVal}\n")
            endforeach()
            file(APPEND ${launch${lang}}
                "exec \"${CCACHE_PROGRAM}\" \"${CMAKE_${lang}_COMPILER}\" \"$@\"\n")
            execute_process(COMMAND chmod a+rx ${launch${lang}})
        endforeach()
        set(CMAKE_XCODE_ATTRIBUTE_CC         ${launchC}   PARENT_SCOPE)
        set(CMAKE_XCODE_ATTRIBUTE_CXX        ${launchCXX} PARENT_SCOPE)
        set(CMAKE_XCODE_ATTRIBUTE_LD         ${launchC}   PARENT_SCOPE)
        set(CMAKE_XCODE_ATTRIBUTE_LDPLUSPLUS ${launchCXX} PARENT_SCOPE)
    endif()
endfunction()
```

30.4. Debug-related Improvements

A typical developer workflow is to make a code change, perform an incremental build to rebuild anything that depends on the change and re-run one or more executables, tests, etc. Some of these may be run under a debugger, which involves loading the relevant debug symbols for the executable and any shared libraries used by it. With many common toolchains, the default settings lead to a number of inefficiencies in this workflow.

One common default behavior is for compilers to store debug information directly in the generated object files. At the link step, the linker then processes all this debug information, even though it isn't actually needed to produce a working binary. When creating an executable or shared library, the

Exclusively for sbmain17@gmail.com Transaction: 0098315591

linker may also embed a copy of the relevant debug information, which requires the linker to process all the symbols across the object files to work out what to store, including handling multiple definitions of the same symbol. This embedding also results in multiple copies of debug information being stored across the various object files and binaries in the build. The size of this information is frequently very significant, often making up the majority of the size of object files and binaries. All this extra processing and duplication can have a noticeable impact on the build performance. It can also lead to very high memory use during linking.

One solution to the above is to store the debug information in a separate file instead of directly in the object file, shared library or executable. Any reasonably recent version of GCC or Clang supports a feature called *split dwarf* which does exactly this, but it is not enabled by default. It can be turned on by adding the `-gsplit-dwarf` compiler option. The compiler will then create a `.dwo` file beside the object file, then record a reference to the `.dwo` in that object file instead of the full debug information. Furthermore, any executable or shared library that links in that object file will also contain a reference to its `.dwo` file instead of embedding a copy of the debug information.

To gain the most benefit from split dwarf, it should be enabled for the whole build. This ensures that all debug information for all targets will be separated from the object and binary files. Developers can add it manually to the CMake cache or set it as part of a toolchain file. This is done by appending `-gsplit-dwarf` to variables like `CMAKE_C_FLAGS_DEBUG` and `CMAKE_CXX_FLAGS_DEBUG`, or their `…_INIT` counterparts in the case of toolchain files. This requires no changes to the project and gives the developer full discretion over whether or not to use the feature.

In some cases, it may be desirable for the project itself to add the flag, such as in a company environment where the project is built in a controlled setup. When doing this, consider the advice in Chapter 15, *Compiler And Linker Essentials* and prefer to use the `add_compile_options()` command to set the relevant directory properties rather than modifying the `CMAKE_<LANG>_FLAGS_DEBUG` variables. The following example demonstrates how this can be done, including a test for whether the flag is supported or not:

```
include(CheckCCompilerFlag)
check_c_compiler_flag("-gsplit-dwarf" HAVE_SPLIT_DWARF)

if(HAVE_SPLIT_DWARF)
    # Only add for debug builds, but could also expand the
    # generator expression to add for RelWithDebInfo too
    add_compile_options("$<$<CONFIG:Debug>:-gsplit-dwarf>")
endif()
```

Split dwarf has some subtleties around its interaction with options like `-g`, `-g1` and so on. The interaction of these flags with `ccache` in particular has caused some issues in the past, so it is advisable to use a recent `ccache` release if adding this compiler flag (at least `ccache` 3.7). Similarly, having the debug information split out to a separate file may also interfere with distributed compilation tools like `distcc`. Developers should check the documentation of the version of the tools they are using and ensure the tools support split dwarf before enabling it.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

30.5. Alternative Linkers

As discussed in the previous section, when working with debug builds, the link time can be significant due to the debug symbol handling. The default BFD linker on many Unix systems does not perform well in this regard, but alternative linkers can do a much better job. The gold linker is relatively mature and is typically much faster and uses much less memory. It is generally reliable, but may fail on certain projects (e.g. as a result of linking to a shared library that only works with the default linker). The llvm linker (`lld`) potentially offers even better performance, but it is not as mature and may not work in all cases.

Enabling an alternative linker follows a similar pattern to `-gsplit-dwarf`, except that a linker option needs to be set rather than a compiler option. The relevant option can also be set for all build types, since the linkers give better performance in general, so there's no real reason to only use them for debug builds. The alternative linkers can be enabled with `-fuse-ld=gold` or `-fuse-ld=lld`, assuming the relevant linker is installed and supported by the toolchain being used. If set by the developer through the CMake cache, the linker option should be appended to the `CMAKE_EXE_LINKER_FLAGS`, `CMAKE_MODULE_LINKER_FLAGS` and `CMAKE_SHARED_LINKER_FLAGS` variables. If being set via a toolchain file, the same variables with `_INIT` appended to their names should be set.

> If targeting Android and using the NDK, developers should be aware of a bug in older versions of `lld` which effectively breaks debugging. If using NDK r21, set the CMake variable `ANDROID_LD` to `lld` instead of adding the `-fuse-ld=lld` option directly (from NDK r22, `lld` is already the default linker). The NDK toolchain file will then apply the necessary workaround.

If the project wants to set the linker option, again it should follow the advice given in Chapter 15, *Compiler And Linker Essentials* and modify directory properties rather than variables. This is most easily done with the `add_link_options()` command available with CMake 3.13 or later:

```
# Force the gold linker
add_link_options(-fuse-ld=gold)
```

30.6. Optimizing Build Dependencies

Ordinarily, a target's dependencies must be built before the target itself, but there are cases where this constraint can be relaxed or even removed entirely. Consider the following example:

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.9)
project(simple LANGUAGES CXX)

add_library(Func func.cpp)
add_executable(App main.cpp)
target_link_libraries(App PRIVATE Func)
```


Exclusively for sbmain17@gmail.com Transaction: 0098315591

func.cpp

```
int func() { return 42; }
```

main.cpp

```
int func();
int main() { return func(); }
```

The *Unix Makefiles* generator will compile `func.cpp`, create the `libFunc.a` static archive library, compile `main.cpp` and finally link the `App` executable. None of these tasks can be performed in parallel, they must be performed in that order. Since CMake 3.9, the *Ninja* generator recognizes that `main.cpp` can be compiled without waiting for `libFunc.a` to be built. Compilation does not depend on `libFunc.a` in this case. This relaxation is only possible when there are no custom commands associated with the `Func` target. Even with this mild condition, the improved parallelism that the *Ninja* generator can achieve as a result of this relaxation can be significant for some projects.

CMake 3.19 added support for the `OPTIMIZE_DEPENDENCIES` target property. When this is set to true on a static or object library target, it allows further relaxation under certain conditions (for all CMake generator types, not just Ninja). Consider this example:

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.19)
project(moreRelaxed LANGUAGES CXX)

add_library(MyStatic STATIC func.cpp)
add_library(MyShared SHARED impl.cpp)
target_link_libraries(MyStatic PRIVATE MyShared)

set_target_properties(MyStatic PROPERTIES
    OPTIMIZE_DEPENDENCIES YES
)
```

func.cpp

```
int impl();
int func() { return impl(); }
```

impl.cpp

```
int impl() { return 42; }
```

In this case, the `MyStatic` target can technically be fully built without having to compile `impl.cpp` or produce a shared library for `MyShared` at all. This is because `MyStatic` is a static library, so it doesn't need to actually link to `MyShared`. The relationship between the two library targets still exists and will be applied to any other target that links to `MyStatic`, but it doesn't affect building `MyStatic` itself.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

This particular dependency relaxation can only be performed when CMake can be certain that building the static or object library does not require the dependency to be built. Examples of things that prevent this from occurring include:

- An explicit `add_dependencies()` command specifying the dependency.
- A custom command attached to the dependency.
- A generated source file for the static or object library whose custom command depends on the dependency.
- The dependency containing any generated source files.
- The dependency uses a language that produces side effects and that language is relevant to the static or object library. For more detail on this particular aspect, see the discussion in the official CMake documentation for the `OPTIMIZE_DEPENDENCIES` target property.

Rather than explicitly specifying the `OPTIMIZE_DEPENDENCIES` property on each target, it will usually be more convenient to set the `CMAKE_OPTIMIZE_DEPENDENCIES` variable project-wide. This variable is used to initialize the `OPTIMIZE_DEPENDENCIES` property when creating targets.

As of CMake 3.22, there is a bug that prevents this feature from working with the Ninja generator. Projects can still safely enable the feature, it's just that the dependencies won't be optimized until the bug is fixed.

30.7. Recommended Practices

Always focus on build correctness before trying to optimize build performance. Ensure that relationships between targets are properly expressed and that dependencies in custom commands fully capture *all* dependencies. Under-specified dependencies can lead to build failures, or worse still, silently using an old binary output from a previous build due to one task not waiting for that output to be regenerated in the current build. Such problems can be notoriously hard to trace, often going undetected for a very long time before being diagnosed as the underlying cause of unreliable builds. Adding certain build optimizations into the mix before addressing such problems will likely only make the build even more unreliable. Conversely, by not adding unnecessary dependencies between targets, the build tool has a greater opportunity to execute parts of the build in parallel and shorten the overall build time.

One simple and relatively safe build optimization is to enable project-wide optimization of dependencies for static and object library targets. This can be done by setting the `CMAKE_OPTIMIZE_DEPENDENCIES` variable to true in the top level `CMakeLists.txt` file, a toolchain file or via a cache variable. When using CMake 3.19 or later, this can relax dependency relationships for static and object library targets. This may improve build parallelism or reduce what has to be built during day-to-day development. Earlier CMake versions will simply ignore the variable.

When seeking to optimize build performance, be careful about assuming characteristics of the machine on which the build will be performed. Tuning values like the unity build batch size for one machine may result in poor performance on another with very different characteristics (e.g. less memory or a different number of cpus). Prefer to leave such tuning up to the developer using cache variables, except for situations where the project already knows upper limits beyond which little

Exclusively for sbmain17@gmail.com Transaction: 0098315591

gain can be expected (e.g. source files that are already very large). Use the `GROUP` unity mode sparingly and only where it provides a measurable improvement in build performance.

When considering unity builds for a project, start from the point of view that unity builds will be enabled project-wide rather than trying to turn it on for specific targets. Focus instead on identifying those targets or source files that are incompatible with a unity build and disable it on just those entities. This ensures hierarchical builds have the opportunity to enable unity builds as widely as possible throughout the dependency tree.

Targets should build successfully with or without compiler support for precompiled headers. It should be considered an optimization, not a requirement. In particular, do not explicitly include a precompile header (e.g. `stdafx.h`) in the source code, let CMake force-include an automatically generated precompile header on the compiler command line instead. This is more portable across the major compilers and is likely to be easier to maintain. It will also avoid warnings being generated from certain code checking tools like `iwyu` (include what you use).

Precompiled headers are most effective when the headers in the precompile set rarely change. This is typically true for headers provided by the system or the compiler, but may not be the case for headers provided by the project. If adding project headers to the precompile set, measure the effect and confirm that it gives a worthwhile benefit. Avoid adding headers that change often, as this will tend to have an overall negative effect on build times due to the increased scope of rebuilds when those headers change.

Where possible, use `ccache` on both developer and continuous integration machines. It is one of the most effective ways for reducing incremental build times and compiling code seen frequently by the compiler with the same settings (a common characteristic of continuous integration builds). Cache misses can result in an increase in compile time for that file of around 10-30%, but the gain for cache hits in typical development or continuous integration workloads more than makes up for that (order-of-magnitude improvements are not unusual).

Configure `ccache` to store its cache on a fast disk such as a SSD, as the latency of that storage can have a strong effect on the cache performance. Also increase the maximum cache size well above the 5Gb default, since it is typically much too low for most real-world deployments. Ensure the cache sloppiness options include `pch_defines` and `time_macros` if using precompiled headers.

For any modest size project, consider enabling split dwarf if it is supported by the toolchain. This can drastically reduce disk space consumed by debug builds, it can speed up linking and it can reduce memory used by some linkers. When using `ccache` with split dwarf, ensure that a recent version of `ccache` is used (3.7 or later). Older versions were susceptible to using out-of-date `.dwo` files or generating more cache misses. They also used to fail to handle the combination of precompiled headers and split dwarf properly.

Avoid forcing the use of a non-default linker if there is any chance that the project may become a child dependency of another project. A parent project may have restrictions which require it to use the default linker, so it must have a way to prevent the child project from trying to use something else. A good strategy is to only try to change the linker if it is the top level project in the build, or leave it for a toolchain file to specify. This follows the general recommendation that the developer should be able to enable or disable most toolchain-related features except where it is an actual requirement of a project.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

Chapter 31. Working With Qt

Qt has enjoyed direct support in CMake for some time. This support has evolved over many releases of both projects and can loosely be grouped under the following three broad areas:

- Running Qt tools like `moc`, `uic` and `rcc` as needed during the build.
- Defining variables and imported targets to simplify building Qt applications and libraries.
- Deploying Qt applications to various platforms.

For Qt 4 and earlier, support for these features was provided exclusively by CMake. Starting with Qt 5, most of the features are provided by Qt itself, with CMake only handling some of the more automated aspects of the Qt tool support. This chapter focuses mostly on Qt 5 support.

Qt 6 takes things further, offering stronger support for CMake projects with an expanded CMake API (the author has been involved in some of that work). Qt 6.2 in particular provided a step change in CMake functionality, especially for QML projects. Improvements to the CMake API continue to be developed for Qt 6.3 and beyond.

CMake 3.5 should be considered the bare minimum version required for convenient Qt 5 support, while Qt 6 needs at least CMake 3.16. It is strongly advised to use a more recent CMake release if possible, ideally the latest. For Qt 6, this is especially true. While it does support CMake 3.16, it is a much smoother experience if using 3.21 or later. More recent CMake versions allow more automated features and more efficient implementations in Qt's CMake API. Some features require at least CMake 3.19 to be used at all.

31.1. Basic Setup

Any project that uses Qt needs to find a suitable Qt installation. Given Qt's size, it is not generally practical to build Qt from source as part of the project. Instead, packages are usually obtained from the official Qt website, a package manager or built separately from sources beforehand.

The most common way to bring Qt into the project's build is to use `find_package()` to search for the `Qt5` package and specify which Qt components the project needs. The following example shows how to require the `Gui` and `Widgets` components, but leave `DBus` as an optional component. It also specifies that Qt 5.9 or later is needed.

```
find_package(Qt5 5.9 REQUIRED
    COMPONENTS          Gui Widgets
    OPTIONAL_COMPONENTS DBus
)
```

In many cases, Qt won't be installed in one of CMake's default search locations. Developers will typically need to provide that information to CMake using the `CMAKE_PREFIX_PATH` cache variable. When running CMake from an environment like Qt Creator, this has to be set in the project settings. When running CMake from the command line, it can be done in the usual way, like so:

Exclusively for sbmain17@gmail.com Transaction: 0098315591

```
cmake -D CMAKE_PREFIX_PATH=/Users/mylogin/Qt/5.12.0/clang_64 /path/to/my/project
```

If successful, the `find_package()` call will define an imported target for each component found. These imported targets will be named `Qt5::<ModuleName>`, such as `Qt5::Gui`, `Qt5::Widgets`, etc. When project targets link against these imported targets, they will automatically have the relevant compiler defines, header search paths and library dependencies added to them. As a result, defining targets that use Qt should generally be quite straightforward:

```
find_package(Qt5 5.9 REQUIRED COMPONENTS Widgets)

add_executable(SimpleGUI MACOSX_BUNDLE WIN32 main.cpp)
target_link_libraries(SimpleGUI PRIVATE Qt5::Widgets)
```

Alternatively, each Qt component could be found independently with separate calls to `find_package()`:

```
find_package(Qt5Widgets 5.9 REQUIRED)
find_package(Qt5Gui     5.9 REQUIRED)
find_package(Qt5DBus    5.9)
```

This is what the Qt documentation recommends, but it potentially allows mixing libraries from different Qt installations. This would generally be discouraged, since it opens up the possibility of using an inconsistent set of libraries and potentially hard to trace bugs. Using one `find_package()` call to find the umbrella `Qt5` package is safer and more clearly communicates the project's intent for component handling.

Regardless of whether finding the `Qt5` umbrella package or the individual components separately, the same set of imported targets and variables will be defined. In general, projects should prefer to work with the imported targets rather than the variables, but there are a couple of per-component variables which may be useful:

`Qt5<component>_FOUND`

This set of variables is most useful for checking whether an optional component was found. Another choice would be to test whether or not the relevant imported target is defined, but checking the variable may be more readable.

`Qt5<component>_VERSION`

This records the version of a found component. It can be useful for things like working around known issues or recording build details.

Some of the Qt components also define associated CMake commands. A number of these are discussed in Section 31.3, "Autogen" further below.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

31.2. Build Details

Most projects should be able to link to the relevant imported targets and not have to concern themselves too much with the compiler and linker details. That said, some important, less obvious areas deserve special attention.

31.2.1. Standard Library Implementation

Qt 6.0 and later require a compiler that supports C++17. Qt 5.7 and later requires C++11 and Qt 5.6 optionally supports C++11. This draws attention to a subtle but important issue, that the project should use the same C++ standard library as was used to build Qt. If the project does not define any compiler feature requirements or any minimum language standard, then the Qt imported targets will transitively set the relevant details via interface properties. The project would also normally specify the language standard and perhaps the standard library implementation for the whole build. Therefore, the project should be careful to ensure that its chosen language standard and standard library implementation are consistent with those used by the Qt libraries.

A common example of inconsistent requirements would be a project that uses Qt 5 but makes use of C++17 features. It might set its targets' `CXX_STANDARD` and `CXX_STANDARD_REQUIRED` properties, or it might add `cxx_std_17` as a compile feature requirement. This can result in the Qt libraries linking against a standard library for C++11 but the project linking against a standard library for C++17. This may be safe for some library implementations, but it is not guaranteed to be so.

If the project is built with a different C++ standard or even just a different compiler version than was used to build Qt, there is no guarantee that binary compatibility will be maintained. In practice, for the mainstream compilers it is typically safe if all of the following are true:

- The project's targets use a C++ standard setting that is the same as or higher than that used by Qt itself.
- The project is built with a compiler of the same type and same or later version as was used to build Qt (some restrictions apply, e.g. earlier Visual Studio versions may only be compatible within the same major release).
- The project is built with the same standard library implementation as was used to build Qt (less of an issue with more recent compilers and Qt versions).

31.2.2. Position Independent Code

As mentioned back in Section 21.6, "Mixing Static And Shared Libraries", shared libraries require sources to be compiled as position-independent code (PIC) on some platforms. This relates to symbol relocation at runtime and is handled automatically by CMake through the `POSITION_INDEPENDENT_CODE` target property. On such platforms, programs can also sometimes be built as position-independent executables (PIE). This adds a layer of security by making addresses of symbols less predictable through ASLR (Address Space Layout Randomization).

For Qt applications, PIE can present some challenges. When building with GCC on ELF platforms (e.g. x86-based Linux systems), executables must not be compiled with the `-fPIE` flag or crashes will likely occur due to the way Qt uses symbols. Following extensive discussions between GCC and Qt developers, a recommendation to use `-fPIC` instead was put forward as a workaround. The use of

Exclusively for sbmain17@gmail.com Transaction: 0098315591

-fPIC instead of -fPIE for executables is an unusual arrangement. The -fPIC flag is typically only used for libraries.

CMake follows the typical usage of these options and adds -fPIE when compiling sources of executable targets that have their POSITION_INDEPENDENT_CODE target property set to true (with some caveats — see the CheckPIESupported module and CMP0083 policy, both of which were added in CMake 3.14). This is incompatible with the unique needs of Qt applications. For those executable targets, projects should ensure the POSITION_INDEPENDENT_CODE target property is not set to true. If a project wants ASLR for a Qt application, the -fPIC compiler flag must be added to the executable target.

From Qt 5.14, adding the -fPIC flag is automatically enforced by the Qt5::Core target through its INTERFACE_COMPILE_OPTIONS property. Therefore, no special steps are required for projects building against Qt 5.14 or later. When using earlier Qt versions, projects must add -fPIC to their executable targets themselves using target_compile_options() or some other equivalent method. Note that this only applies to executable targets, the POSITION_INDEPENDENT_CODE target property will yield the correct compiler flags for library targets.

31.2.3. Windows GUI Applications

An aspect unique to Windows is that GUI and console programs have different entry points. Console applications use main(), just like other platforms, but Windows GUI applications use WinMain() as their entry point. To free projects from having to handle this difference, Qt provides a special library that defines a simple WinMain() function which forwards to main(). The Qt5::Core imported target will automatically link in this special library if the executable linking to Qt5::Core is a GUI application. The WIN32_EXECUTABLE target property controls this choice and is most commonly set by adding the WIN32 keyword to the add_executable() call.

```
find_package(Qt5 5.9 REQUIRED Core Gui)

add_executable(ConsoleApp ...)
add_executable(GuiApp WIN32 ...)

target_link_libraries(ConsoleApp PRIVATE Qt5::Core)
target_link_libraries(GuiApp PRIVATE Qt5::Gui Qt5::Core)
```

In the above example, ConsoleApp will not link in the special Qt5::WinMain library, whereas GuiApp will. As this example shows, projects shouldn't need to directly handle or even mention Qt5::WinMain. It is handled automatically. In the event that this automatic linking of the Qt5::WinMain library needs to be disabled, the Qt5_NO_LINK_QTMAIN target property can be set to false.

31.3. Autogen

Qt provides a number of command-line tools which process sources and generate other source files to be consumed by the build. The three most important and heavily used of these tools are moc, uic and rcc. CMake provides dedicated features that automate the way these tools are used within a project. CMake commands provided by Qt also allow projects to use these tools in a more manual, granular way for greater control if required.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

CMake 3.8 changed the way Qt-related auto-generated files are handled. Those files are now generated into their own separate directory by default and that directory is automatically added to things like the header search paths where needed. When using CMake 3.7 or earlier, projects must take extra steps to account for such search paths. The following sections discuss how these changes affect each of the tools.

31.3.1. Moc

Qt uses its own meta-object system to implement signals and slots, run-time type information and its own property system. It provides the `moc` tool which scans source code looking for particular macros (e.g. `Q_OBJECT`, `Q_GADGET` and so on) and generates C++ code implementing the underlying functions the macros declare. The following is a typical example of a class that requires processing by `moc`:

```
class SomeClass : public QObject
{
    Q_OBJECT
public:
    SomeClass(QObject* parent);
    ~SomeClass();
    // ... various signals, slots,
    // properties, member functions, etc....
};
```

CMake provides a fully automated way of managing how and when `moc` should be run on source files of a given target. This functionality is enabled by setting the `AUTOMOC` target property to true. The property is initialized from the value of the `CMAKE_AUTOMOC` variable when the target is created. When `AUTOMOC` is enabled, files are scanned at build time and `moc` is run on the relevant files. The generated code will be incorporated into the target in the appropriate way depending on how the project and source files are structured. If extra command line options need to be passed to the `moc` tool, they can be specified in the `AUTOMOC_MOC_OPTIONS` target property.

In the simplest scenario, a class like the one above is defined in a header and uses the `Q_OBJECT` macro. No other project source files make any assumption about the file name(s) that `moc` will generate. The project lets `AUTOMOC` take care of compiling the generated files and linking them into the target. This is very simple to set up and gives `AUTOMOC` the most control over the way the generated code is added to the build.

```
set(CMAKE_AUTOMOC YES)

add_executable(MyQtApp
    main.cpp
    myclass.cpp
    myclass.h
)

find_package(Qt5 COMPONENTS Core REQUIRED)
target_link_libraries(MyQtApp PRIVATE Qt5::Core)
```


Exclusively for sbmain17@gmail.com Transaction: 0098315591

One weakness of the above approach is that all generated sources may be combined into a single implementation file with more recent CMake versions. If the number of classes processed by AUTOMOC in this way is very large, it can put pressure on compiler resources and even cause build failure in extreme cases. At the very least, it can make larger builds less efficient, so it may be desirable to have the generated files compiled individually to take advantage of build parallelism and reduce resource requirements. One way to achieve this is for project sources to explicitly include the generated sources themselves rather than leaving AUTOMOC to add them to the build. A target can have a C++ source file which contains a line like the following (basename is just a placeholder in this example):

basename.cpp

```
#include "moc_basename.cpp"
```

If a file called basename.h is found (or with any other recognized header file extension), then AUTOMOC will run moc on that header such that it generates a file called moc_basename.cpp. Since the implementation file already pulls in that generated file via an #include statement, AUTOMOC doesn't have to do anything more to add it to the build. It will already be compiled as part of basename.cpp. In other words, adding an include for moc_basename.cpp makes that source file take over adding the generated file to the build instead of letting AUTOMOC combine it with other generated files. An added advantage is that it integrates quite naturally with CMake's unity build support (see Section 30.1, "Unity Builds"). The generated moc_basename.cpp effectively becomes part of the basename.cpp file it belongs to. Unity builds have a configurable limit to how many target sources will be merged into one batch, so this will indirectly limit how many generated moc_*.cpp files get combined into one compilation unit as well.

In some cases, a private class may be declared in a C++ implementation file rather than in a header. If such a class uses Q_OBJECT, moc must be run on that C++ file. To account for this, if a source file contains a line that includes a .moc file like shown just below, AUTOMOC will run moc on that source file even if it isn't a header. In such cases, it will use the specified file name for the generated code:

```
#include "basename.moc"
```

Again, the source file pulls in the generated code on its own, so AUTOMOC doesn't have to do anything more to add it to the build. That has the same benefit for unity builds as mentioned previously.

By default, AUTOMOC scans for the macros Q_OBJECT, Q_GADGET and Q_NAMESPACE. CMake 3.17 added Q_NAMESPACE_EXPORT to this list as well. From CMake 3.10, this list of macro names is taken from the AUTOMOC_MACRO_NAMES target property, which is initialized from the CMAKE_AUTOMOC_MACRO_NAMES variable when the target is created. For most projects, these defaults are fine. If the conventional macro names are embedded within wrapper macros, the names of those wrapper macros need to be added to AUTOMOC_MACRO_NAMES as well. The following contrived example demonstrates such a scenario:

mywrapper.h

```
#define MY_WRAPPER Q_OBJECT
```

Exclusively for sbmain17@gmail.com Transaction: 0098315591

someclass.cpp

```
#include mywrapper.h

class SomeClass : public QObject
{
    MY_WRAPPER
public:
    // ...
};
```

For such a case, the project would most conveniently handle this by adding the wrapper macro to the global defaults like so:

```
list(APPEND CMAKE_AUTOMOC_MACRO_NAMES MY_WRAPPER)
```

If the wrapper macro refers to a file name that should be considered a dependency for the `moc` generation, the `AUTOMOC_DEPEND_FILTERS` target property should be used to express that dependency. This property is not often needed, so the interested reader is referred to the official CMake documentation for that property for further details.

When using `AUTOMOC`, CMake 3.8 and later will generate the `moc_*.cpp` or `*.moc` files in a separate directory set aside for such auto-generated files. That directory will be automatically added to the target's header search path, so the project doesn't need to do anything else. When using CMake 3.7 or earlier, the generated files are created in the current build directory and that directory is *not* automatically added to the header search path. It is therefore common practice for projects to set the `CMAKE_INCLUDE_CURRENT_DIR` variable to true so that the current source and build directories are always added to the header search path. If the project already requires CMake 3.8 or later, it should avoid setting `CMAKE_INCLUDE_CURRENT_DIR` since it is not needed for `AUTOMOC` in that case.

In some situations, it may be necessary to prevent `moc` from processing a particular file. This is most easily achieved by setting that file's `SKIP_AUTOMOC` source property to true. If other autogen tools like `uic` and `rcc` should also skip the file, the `SKIP_AUTOGEN` source property can be set to true instead.

```
add_executable(MyApp noMocPlease.cpp noAutoGen.cpp ...)

set_source_files_properties(noMocPlease.cpp PROPERTIES
    SKIP_AUTOMOC TRUE
)
set_source_files_properties(noAutoGen.cpp PROPERTIES
    SKIP_AUTOGEN TRUE
)
```

While setting `AUTOMOC` to true is the generally recommended way of enabling `moc` to be run on the relevant sources, in very large projects the performance cost of scanning many sources may be non-trivial. If only some sources need `moc` processing and others don't, projects can manually specify which ones to process with the following command provided by the `Qt5::Core` component instead of using `AUTOMOC`:

Exclusively for sbmain17@gmail.com Transaction: 0098315591

```
qt5_wrap_cpp(outVar sources... [OPTIONS ...])
```

The files to be processed with `moc` should be listed as the `sources`. The resultant set of generated `moc_*.cpp` files will be provided back to the caller in the `outVar` variable. Any extra command-line options to be passed to the `moc` tool can be specified after the `OPTIONS` keyword. The project is responsible for ensuring that the generated sources are added to the build and at least one target must depend on them.

The easiest way to satisfy these requirements is to add the generated sources to an executable or library target. For example:

```
find_package(Qt5 COMPONENTS Core REQUIRED)

# Process this file manually rather than using AUTOMOC
qt5_wrap_cpp(genMocs myclass.h)

add_executable(MyQtApp
    main.cpp
    myclass.cpp
    myclass.h
    ${genMocs}
)
```

Another way to satisfy the requirements is to `#include` the generated `moc_*.cpp` file within one of the existing source files and create a custom target that depends on the files listed in `outVar`. The custom target is needed to ensure that build rules are created for the custom `moc` execution steps. Note that the `qt5_wrap_cpp()` command will generate its output files in the current build directory similar to the `AUTOMOC` behavior of CMake 3.7 and earlier, so `CMAKE_INCLUDE_CURRENT_DIR` may be needed if using this method. Such an arrangement may look something like this:

```
find_package(Qt5 COMPONENTS Core REQUIRED)

# Process this file manually rather than using AUTOMOC
qt5_wrap_cpp(genMocs myclass.h)

set(CMAKE_INCLUDE_CURRENT_DIR YES)
add_executable(MyQtApp
    main.cpp
    myclass.cpp
    myclass.h
)

add_custom_target(MyMocs DEPENDS ${genMocs})
```

myclass.cpp

```
#include "moc_myclass.cpp"
// Rest of the file as normal...
```

Exclusively for sbmain17@gmail.com Transaction: 0098315591

The first method is simpler, whereas the second method may be useful if AUTOMOC had been used previously and the source files were already modified to #include the generated moc_*.cpp files directly. Either method should integrate with unity builds in a natural way.

31.3.2. Widgets

Qt GUI applications can be based on different UI technologies. They can use the QML declarative language, widgets based on the C++ QWidget hierarchy, or sometimes a combination of both. QML requires no special handling from the build system, it is handled purely by the code (.qml files may be stored in resources for the application to use at run time though and the build system may need to process the resources). Widgets can be implemented purely in C++ code too, but they can also be defined using a XML-based UI description file. These UI files typically have a .ui file extension and are processed by the uic tool to produce C++ code, which is then compiled into the project just like any other source file.

The uic tool has many similarities to moc, both in the way the tools work and the CMake support available. CMake provides the AUTOUIC target property which fulfills a very similar role to AUTOMOC. The AUTOUIC property is initialized from the value of the CMAKE_AUTOUIC variable when the target is created. When AUTOUIC is set to true, CMake will scan that target's source files at build time looking for #include statements that pull in header files of the form ui_basename.h. When it finds such statements, it searches for a matching basename.ui file in the same directory or any directories listed in that target's AUTOUIC_SEARCH_PATHS target property. If the #include statement has a path component before the ui_basename.h, the search will be performed with and without that path. If a basename.ui file is found, it will be processed with the uic tool to generate the header.

```
find_package(Qt5 COMPONENTS Widgets Core REQUIRED)

set(CMAKE_AUTOUIC YES)
add_executable(MyQtApp
    main.cpp
    mainwindow.cpp
    mainwindow.h
)
target_link_libraries(MyQtApp PRIVATE Qt5::Widgets Qt5::Core)
```

When using AUTOUIC, CMake 3.8 and later will generate the ui_*.h files in a separate directory set aside for such auto-generated files. The behavior is exactly the same as for AUTOMOC, with that directory being automatically added to the target's header search path. When using CMake 3.7 or earlier, the generated files are created in the current build directory and that directory is *not* automatically added to the header search path. Therefore, the CMAKE_INCLUDE_CURRENT_DIR variable is typically set to true when CMake 3.7 or earlier must be supported.

In another similarity to AUTOMOC, if a source file should not be scanned by AUTOUIC for some reason, its SKIP_AUTOUIC source property can be set to true. If that source file should also be skipped by other Qt tools like moc and rcc, the SKIP_AUTOGEN source property should be set to true instead.

In some projects, it may be necessary to customize the command-line options given to uic. For example, the --tr option can be used to specify a different translation function. Such command-line options can be specified in the AUTOUIC_OPTIONS target property, which takes its initial value from the

Exclusively for sbmain17@gmail.com Transaction: 0098315591

CMAKE_AUTOUIC_OPTIONS variable when the target is created. In addition, a target can enforce uic options transitively on anything that links to it by setting the INTERFACE_AUTOUIC_OPTIONS target property. This is useful, for example, when the target itself provides the translation function and it wants to ensure that libraries that link to it will use that translation function instead of the default one. The CMake documentation contains examples of this scenario in its Qt manual.

The use of AUTOUIC on large projects can have similar performance implications as AUTOMOC. If a project wants to avoid the automatic source scanning and specify the .ui files to be processed directly, it can do so with the macro provided by the Qt5::Widgets component:

```
qt5_wrap_ui(outVar uiFiles... [OPTIONS ...])
```

Unlike qt5_wrap_cpp() which produces .cpp files, the qt5_wrap_ui() command produces headers. The list of headers provided back to the caller in the output variable should be added directly to a target to ensure that the custom build rules created by qt5_wrap_ui() are applied.

```
find_package(Qt5 COMPONENTS Widgets Core REQUIRED)

qt5_wrap_ui(genUiHeaders mainwindow.ui)

add_executable(MyQtApp
    main.cpp
    mainwindow.cpp
    mainwindow.h
    ${genUiHeaders}
)
```

Regardless of whether or not AUTOUIC is used, projects may choose to list .ui files as sources directly for a target. This will not affect how those files are processed, but it does make those files show up in the list of sources in some IDE tools. Projects may even want to define a source group to improve the organization within some IDEs. For example:

```
source_group("UI Files" REGULAR_EXPRESSION [[.*\.ui]])
```

31.3.3. Resources

Qt resources are typically defined in files with a .qrc file extension. Such files are XML descriptions of other files (images, .qml files, etc.) which the application will access using the Qt resource system. These .qrc files need to be processed by the rcc tool, which generates either a source file embedding the resources as C++ code or a binary file that can be dynamically loaded at run time.

The simplest way of handling .qrc files is to set the AUTORCC target property to true and add the .qrc files to the list of sources for the target. This will result in the resources being encoded as a generated C++ source file which AUTORCC automatically adds to the target. There is nothing more for the project to do, so this method is very straightforward. The AUTORCC property takes its initial value from the CMAKE_AUTORCC variable when the target is created.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

```
set(CMAKE_AUTORCC YES)
add_executable(MyQtApp
    main.cpp
    myclass.cpp
    myclass.h
    myApp.qrc
)
```

Command line options for `rcc` can be provided in the `AUTORCC_OPTIONS` target property, but this should not typically be required. Similar to the other autogen capabilities, `AUTORCC` can be skipped for a particular `.qrc` file by setting its `SKIP_AUTORCC` source file property, or by setting `SKIP_AUTOGEN` if it should be skipped by `moc` and `uic` as well.

If the project wants to create a binary version of the resources to be dynamically loaded at run time, `AUTORCC` cannot be used. An example scenario is where a very large resource only has to be loaded into memory if an application setting enables a feature that requires it. For such use cases, the `Qt5::Core` component provides the following commands:

```
qt5_add_resources(outVar qrcFiles... [OPTIONS ...])
qt5_add_big_resources(outVar qrcFiles... [OPTIONS ...])
qt5_add_binary_resources(target qrcFiles...
    [OPTIONS ...]
    [DESTINATION rccFile]
)
```

The `qt5_add_resources()` command provides the same capability as `AUTORCC`. It defines custom commands that take the `qrcFiles` as input and produce source files whose names will be provided in the variable named by `outVar`. Just as for `AUTOMOC`, these sources need to be added to a target to ensure the custom build rules are enforced and the sources are compiled. There's generally little need for this command though, as `AUTORCC` is usually more appropriate (but see an exception related to unity builds at the end of this section).

The `qt5_add_big_resources()` command is very similar and can generally be used as a drop-in replacement for `qt5_add_resources()`. Its main difference is that instead of generating sources that need to be compiled, it bypasses the compiler and generates object files directly. This is most useful for very big resources which would otherwise generate huge source files that are too large for compilers to process. The names of the generated object files will be provided in the variable named by `outVar`. Once again, these need to be added to a target to ensure that the relevant build rules are properly defined. Due to internal implementation details, the `qt5_add_big_resources()` command is only made available by Qt when using CMake 3.9 or later.

The `qt5_add_binary_resources()` command provides the ability to compile `.qrc` files into a single dynamically loadable `.rcc` file. It requires a `target` name which the command will create as part of the call. The name of the generated `.rcc` file can be specified using the `DESTINATION` keyword, or if omitted, the name of the `target` with `.rcc` appended will be used and it will be created in the `CMAKE_CURRENT_BINARY_DIR`.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

```
find_package(Qt5 COMPONENTS Core REQUIRED)

qt5_add_resources(resCompiledIn compiledIn.qrc)
qt5_add_binary_resources(resLoadable runtimeLoadable.qrc
    DESTINATION runtimeLoadable.rcc
)

add_executable(MyQtApp
    ${resCompiledIn}
    ${CMAKE_CURRENT_BINARY_DIR}/runtimeLoadable.rcc
    ...
)
set_target_properties(MyQtApp PROPERTIES
    RESOURCE ${CMAKE_CURRENT_BINARY_DIR}/runtimeLoadable.rcc
)
```

Note in the above how the generated `runtimeLoadable.rcc` file is still added to the `MyQtApp` target. This is so that it can be listed in the target's `RESOURCE` property, which is a requirement for ensuring the file is installed as part of the target. See Section 23.2, "Application Bundles" for a related discussion of other ways that resources can be handled on Apple platforms.

Since `AUTORCC` is not used in the above example, the `.qrc` files could also be added to the list of sources given to `add_executable()`. This may make them show up in source file lists in some IDEs but won't otherwise affect how CMake treats them.

It should be noted that the `rcc` tool generates C++ code with generic symbol names. This has implications for unity builds. If multiple sources generated by `rcc` are added to the same target, those sources should be excluded from unity builds to prevent compile-time errors due to symbols being redefined. This can be done by setting the relevant source property, but the names of the source files are not available to the project when `AUTORCC` is used. An example of setting the source properties when using the manual commands provided by Qt instead might look like this:

```
find_package(Qt5 COMPONENTS Core REQUIRED)

qt5_add_resources(resourceSet1 resourceSet1.qrc)
qt5_add_resources(resourceSet2 resourceSet2.qrc)

add_executable(MyQtApp
    ${resourceSet1}
    ${resourceSet2}
    ...
)
set_source_files_properties(${resourceSet1} ${resourceSet2}
    PROPERTIES SKIP_UNITY_BUILD_INCLUSION TRUE
)
```

31.4. Translations

A core part of Qt is its ability to provide localized text. Strings are wrapped with a call to a special translation function in the C++ code, then a separate tool named `lupdate` is run on the code to

Exclusively for sbmain17@gmail.com Transaction: 0098315591

produce a separate `.ts` file which can be given to a translator. The tool can also process other file types, such as QML, UI and resource files. The `.ts` file is updated by the translator with translations for a given language. At some point, a tool named `lrelease` is run on the `.ts` file to produce a final `.qm` translation file which is distributed with the application. The `.qm` file format is a more efficient format intended only for run time, whereas `.ts` files are intended for translators to load into the Qt Linguist application. The `lupdate` tool can be re-run at any time to rescan the sources and update a `.ts` file with any changes without losing previously translated text.

CMake does not provide any direct assistance with coordinating the above process, but the `Qt5::LinguistTools` component does provide two CMake commands which make the steps fairly straightforward. The simpler of the two commands takes the names of one or more `.ts` files and defines custom commands that run `lrelease` to generate the associated `.qm` files:

```
qt5_add_translation(outVar tsFiles...)
```

Projects need to ensure that at least one target depends on the generated `.qm` files so that the build rules are enforced. This could be a dedicated custom target or it could be an ordinary executable or library target. For example:

```
find_package(Qt5 COMPONENTS LinguistTools Core REQUIRED)

qt5_add_translation(qmFiles
    MyQtApp_de_AT.ts
    MyQtApp_de_DE.ts
    MyQtApp_ja.ts
)

# Choice 1: Use a custom target
add_custom_target(ProcessTranslations DEPENDS ${qmFiles})

# Choice 2: Add to a real target
add_executable(MyQtApp ${qmFiles} ...)
```

The `qt5_add_translation()` command never modifies the `.ts` files, it only produces `.qm` files. In order to create or update a `.ts` file, the other command provided by the `Qt5::LinguistTools` component must be used:

```
qt5_create_translation(outVar
    [directories...]
    [sources...]
    [tsFiles...]
    [OPTIONS ...]
)
```

This command will create any of the named `tsFiles` if they do not already exist. It will then scan the specific `sources` as well as any sources it finds in the `directories` (recursively by default) and update the `tsFiles` with any new translatable strings by running the `lupdate` tool. It will then call `qt5_add_translation()` internally with the list of `.ts` files, so projects do not need to call that

Exclusively for sbmain17@gmail.com Transaction: 0098315591

themselves. The `OPTIONS` keyword can be used to specify additional command-line options to pass to the `lupdate` command, if required.

When performing the source code scan, the `lupdate` tool is given an `-I` option for each path in the `INCLUDE_DIRECTORIES` directory property. Include directories defined on targets are not used, since the source scanning is not associated with any particular target. The project needs to ensure that the lack of target include directories won't change the set of translatable strings seen by `lupdate` (if needed, the `OPTIONS` keyword can be used to manually add paths with `-I` flags).

The following example sets up scans of all sources in the current directory and below, creating or updating a single `.ts` file as appropriate (all performed at build time):

```
qt5_create_translation(qmFiles ${CMAKE_CURRENT_LIST_DIR} MyQtApp_ja.ts)

add_custom_target(ProcessTranslations DEPENDS ${qmFiles})
```

The `.ts` files only need to be updated by rescanning the sources just before the `.ts` files are handed over to a translator to update the translations. There's little point rescanning the sources with every build because the `.qm` files generated won't change until the translations are updated by the translator. In very large projects, this rescanning could be annoying for developers in their everyday workflow. Most of the time, calling `qt5_add_translation()` to provide the `.qm` files for the existing `.ts` files is sufficient. The `qt5_create_translation()` command only needs to be called when the `.ts` files are first created or when the developer specifically wants to rescan the sources and update the `.ts` files. One strategy is to use a CMake cache option to switch between the two modes:

```
set(tsFiles MyQtApp_ja.ts)
option(ENABLE_TS_RESCAN "Enable rescanning sources to update .ts files" ON)
if(ENABLE_TS_RESCAN)
    qt5_create_translation(qmFiles ${CMAKE_CURRENT_LIST_DIR} ${tsFiles})
else()
    qt5_add_translation(qmFiles ${tsFiles})
endif()

add_custom_target(ProcessTranslations DEPENDS ${qmFiles})
```

This allows the developer to be in control of when the `.ts` files will be updated and when to pay the cost of rescanning the sources.

> CMake 3.17.0 introduced some undesirable behavior for the Ninja generator which affects the above technique. When switching the `ENABLE_TS_RESCAN` cache option from true to false, the `.ts` files would be deleted at the end of the CMake step if using Ninja 1.10 or later. This relates to a Ninja feature called `cleandead`, which CMake was invoking as a final step of project file generation. CMake 3.19.2 reverted that behavior and a fix was back-ported to the 3.18.6 release, but the problem remains for the 3.17 series. Projects should strongly consider making at least 3.19.2 their minimum CMake version if using this technique.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

An important observation is that the `qt5_create_translation()` command updates `.ts` files in-place, so the build rules would directly modify the files in the source tree. These files would typically be under version control just like other sources. This is a rare example where a build step updating something in the source tree is appropriate.

31.5. Deployment

Installing and packaging Qt applications is a non-trivial exercise due to the various different files that must be included with the main application executable. An incomplete list of things that may need to be installed includes:

- Qt libraries (unless Qt is linked statically).
- Qt plugins and platform files.
- A `qt.conf` file.
- Translation files from both Qt itself and the application, if not embedded directly as resources.
- Images, icons or other resources that are not compiled into the application.

The application must also be installed with the right RPATH settings to allow libraries and frameworks to be found when the application is run (see the discussions in Section 26.2.2, "RPATH" and Section 26.2.3, "Apple-specific Targets"). On Apple platforms, the unique directory layout of app bundles further complicates deployment (see Chapter 23, *Apple Features*).

31.5.1. Qt Deployment Tools

Qt provides tools which automate some of the more difficult aspects of the deployment process. These tools are not available for all platforms, but where they are available, they greatly simplify installation and packaging. They include `macdeployqt`, `windeployqt` and `androiddeployqt`. The basic syntax of these tools is:

```
macdeployqt [options...] app-bundle

windeployqt [options] files...

androiddeployqt options...
```

Each tool has its own set of options, but the general idea is the same. They copy the necessary Qt libraries, frameworks, plugins and platform files to the relevant location within the application bundle or directory structure. They may also create a simple `qt.conf` file in the appropriate location. Command line options can be used to tailor the tool's behavior, all of which are described in the Qt documentation and the tool's own `--help` output. The reader should consult the documentation for the specific tool being used to familiarize themselves with its requirements and available options.

Discussion here in this section focuses on the CMake-specific aspects of integrating these tools into a CMake project. It is assumed that the project already defines the relevant `install()` commands and any associated properties are set appropriately. Chapter 26, *Installing*, Chapter 23, *Apple Features* and Chapter 19, *Working With Files* are especially relevant.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

The Qt deploy tool needs to be invoked at install time after targets and files have been installed. The `install(CODE)` or `install(SCRIPT)` command is used to implement the logic, but as noted in Section 26.6, "Custom Install Logic", the timing of when such install-time code will be executed is only well-defined with CMake 3.14 or later. These commands should generally be at the end of the directory scope where the application and its files are installed, after all other `install()` commands.

Projects may find it more manageable to use a separate file to hold the custom install code. As the application complexity grows, the number of custom steps required may increase and placing the logic in a separate file avoids excessive escaping. An example of a fairly generic custom install script invoking the deployment tool might look like this:

deployapp.cmake.in

```
execute_process(
    COMMAND "@DEPLOYQT_EXECUTABLE@" @DEPLOY_OPTIONS@
    WORKING_DIRECTORY ${CMAKE_INSTALL_PREFIX}
    RESULT_VARIABLE result
)
if(result)
    message(FATAL_ERROR "Executing @DEPLOYQT_EXECUTABLE@ failed: ${result}")
endif()
```

This file is expected to be copied into the build directory with `configure_file()`, substituting the location of the deployment tool and its command-line options along the way. The following shows how the associated `CMakeLists.txt` file might look for installing an application bundle on the macOS platform (code for other platforms has been omitted for brevity):

CMakeLists.txt

```
add_executable(MyQtApp MACOSX_BUNDLE WIN32 ...)
set_target_properties(MyQtApp PROPERTIES
    INSTALL_RPATH @executable_path/../Frameworks
)
install(TARGETS MyQtApp
    BUNDLE DESTINATION .
    ...
)
get_target_property(mocExe Qt5::moc IMPORTED_LOCATION)
get_filename_component(qtBinDir "${mocExe}" DIRECTORY)

find_program(DEPLOYQT_EXECUTABLE macdeployqt PATHS "${qtBinDir}" NO_DEFAULT_PATH)

set(DEPLOY_OPTIONS [[MyQtApp.app -verbose=2 "-codesign=Apple Development"]])

configure_file(deployapp.cmake.in deployapp.cmake @ONLY)
install(SCRIPT ${CMAKE_CURRENT_BINARY_DIR}/deployapp.cmake)
```

It is worth noting that the `DEPLOY_OPTIONS` for `macdeployqt` can be used to do things like handle code signing (shown in the above example) and to skip plugins that do not meet app store requirements due to their use of private APIs. See the `macdeployqt` help for details.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

The above pattern would be very similar for windeployqt as well. The main differences would be:

- The INSTALL_RPATH target property would be ignored.
- find_program() would search for windeployqt instead of macdeployqt.
- The DEPLOY_OPTIONS would be specific to the windeployqt tool.

The windeployqt tool follows a different default directory layout to the one that CMake and the GNUInstallDirs module uses. In particular, the tool assumes the application binary is at the base install location instead of in a bin subdirectory below that point. Command line options can be used to coerce the tool to produce a layout more closely aligned with CMake's defaults if required though. For example, if the MyQtApp target in the earlier example is using the standard GNUInstallDirs layout described in Section 26.1, "Directory Layout", the following set of windeployqt command line options could be used:

```
set(DEPLOY_OPTIONS [[bin/MyQtApp.exe --dir . --libdir bin --plugindir plugins]])
```

The other step required is to install a qt.conf file that accounts for the above installed layout changes. It should be installed to the same directory as the application binary, which will be in the bin directory for the standard directory layout. The windeployqt tool doesn't write a qt.conf file, so the project has to create one itself, but only a very minimal file is needed:

qt.conf

```
[Paths]
Prefix = ..
```

31.5.2. Deploying Translation Files

Deploying translation files is similar to installing any other arbitrary file or resource, but the translation files that come with Qt itself require special handling. Qt 5 splits its translation files based on components, but the Qt documentation recommends merging those files the application needs into one file per translated language for deployment (see the developers section of the Qt Linguist manual). This makes development and deployment easier because only one file of the form qt_xx.qm needs to be loaded at runtime to handle Qt's translations. The windeployqt tool handles this automatically and will create these translation files by default, but it can be disabled if desired. The macdeployqt tool does not offer this functionality, so projects have to implement it themselves. The lconvert tool provided by Qt can be used to merge the necessary .qm files and this can also be done at install time in the same deployapp.cmake script used to invoke the deployment tool.

The following expands on the macOS example from the previous section. It shows how to combine the base language file with the translations for the Qt Multimedia module (see Section 31.4, "Translations" for how the qmFiles variable is populated in the CMakeLists.txt file):

Exclusively for sbmain17@gmail.com Transaction: 0098315591

deployapp.cmake.in

```
set(outDir "${CMAKE_INSTALL_PREFIX}/@TRANSLATION_DIR@")
set(inDir  "@qtBinDir@/../translations")

# Loop over the set of languages to deploy
foreach(lang IN ITEMS ja de fr)
    execute_process(
        COMMAND "@LCONVERT_EXECUTABLE@"
                -o ${outDir}/qt_${lang}.qm
                ${inDir}/qtbase_${lang}.qm
                ${inDir}/qtmultimedia_${lang}.qm
        RESULT_VARIABLE result
    )
    if(result)
        message(FATAL_ERROR "Failed to create qt_${lang}.qm: ${result}")
    endif()
endforeach()

execute_process(
    COMMAND "@DEPLOYQT_EXECUTABLE@" @DEPLOY_OPTIONS@
    WORKING_DIRECTORY ${CMAKE_INSTALL_PREFIX}
    RESULT_VARIABLE result
)
if(result)
    message(FATAL_ERROR "Executing @DEPLOYQT_EXECUTABLE@ failed: ${result}")
endif()
```

CMakeLists.txt

```
add_executable(MyQtApp MACOSX_BUNDLE WIN32 ...)
set_target_properties(MyQtApp PROPERTIES
    INSTALL_RPATH @executable_path/../Frameworks
)
set_source_files_properties(${qmFiles} PROPERTIES
    MACOSX_PACKAGE_LOCATION translations
)
install(TARGETS MyQtApp BUNDLE DESTINATION . ...)

get_target_property(mocExe Qt5::moc IMPORTED_LOCATION)
get_filename_component(qtBinDir "${mocExe}" DIRECTORY)

find_program(LCONVERT_EXECUTABLE lconvert    PATHS "${qtBinDir}" NO_DEFAULT_PATH)
find_program(DEPLOYQT_EXECUTABLE macdeployqt PATHS "${qtBinDir}" NO_DEFAULT_PATH)

set(DEPLOY_OPTIONS [[MyQtApp.app -verbose=2 "-codesign=Apple Development"]])
set(TRANSLATION_DIR MyQtApp.app/Contents/translations)

configure_file(deployapp.cmake.in deployapp.cmake @ONLY)
install(SCRIPT ${CMAKE_CURRENT_BINARY_DIR}/deployapp.cmake)
```

The above example shows how to install the translation files to the `translations` directory. Assuming the project is following the standard layout as recommended in Section 26.1, “Directory Layout”, this matches the default location used by `QLibraryInfo` and `qt.conf` files. As a result, the project does

Exclusively for sbmain17@gmail.com Transaction: 0098315591

not need to provide its own custom `qt.conf` file, the one provided by `macdeployqt` is sufficient. Note that the deploy tool doesn't typically provide a way to change the `qt.conf` file it produces. It may issue warnings if the project tries to install its own, so following the defaults is desirable.

31.5.3. Linux Considerations

Qt does not provide an official deployment tool for Linux. There is a community-supported `linuxdeployqt` tool, but it is not officially endorsed and has some philosophical and design differences compared to the official Qt tools. If `linuxdeployqt` is not suitable, projects will need to handle all aspects of deployment themselves. The appropriate way to do that depends on whether the project will rely on the Linux distribution to provide Qt runtime support or whether it will be packaged up as a self-contained, standalone application.

Relying on the Linux distribution to provide the Qt runtime support is, on the surface at least, the simplest path. There is essentially nothing for the project to do other than package up its own components. If the project is made part of the Linux distribution, the distribution maintainer would add the necessary package dependency information. The main drawback to this approach is that when not part of the Linux distribution, the project would need to create separate binary packages for each distribution due to potential differences in binary compatibility, directory layout, etc.

Supporting self-contained, standalone packaging is a very different scenario. The Qt documentation for Linux/X11 deployment details all the necessary things that need to be done, but it is a non-trivial exercise. Developers would need to identify all of the items from that page that would apply to their project and ensure those items are part of their project's install logic. That logic may also need to account for any changes in libraries, plugins, etc. between Qt versions.

If the project needs to support both standalone packaging and direct inclusion into Linux distributions, all of the Qt-related install logic should be conditional on some kind of CMake cache option to simplify the work of distribution maintainers.

31.6. Transition To Qt 6

Qt 5.15 added some features to help projects prepare for a transition to Qt 6. In addition to the versioned imported targets and CMake commands, unversioned equivalents are also available. For each target name starting with `Qt5::`, an equivalent unversioned target with the prefix `Qt::` can be used. Similarly, each of the `qt5_…()` commands have unversioned `qt_…()` equivalents as well. Projects can switch to using the unversioned targets and commands, which may allow them to build with either Qt 5 or Qt 6 during this period of transition.

```
# Use Qt 6 if available, otherwise fall back to Qt 5
set(qtComponents Core Widgets LinguistTools)
find_package(Qt6 COMPONENTS ${qtComponents} QUIET)
if (NOT Qt6_FOUND)
    find_package(Qt5 5.15 COMPONENTS ${qtComponents} REQUIRED)
endif()

add_executable(MyQtApp ...)
target_link_libraries(MyQtApp PRIVATE Qt::Widgets Qt::Core)
```


Exclusively for sbmain17@gmail.com Transaction: 0098315591

The reader should be aware that there are differing views on whether using unversioned targets and commands is advisable (see QTBUG-83774). Each project will need to consider whether these concerns are important to them before adopting the above transitional approach.

The Qt 6 documentation recommends using unversioned commands and versioned targets. The relevant discussion can be found at the following location:

https://doc.qt.io/qt-6/cmake-qt5-and-qt6-compatibility.html

31.7. Recommended Practices

Projects should already have migrated off Qt 4 and be using Qt 5, with preparations for supporting Qt 6 soon. Development of CMake support for Qt is focused on Qt 5 and more recently on Qt 6. Staying with Qt 4 will become increasingly problematic and is not recommended. For projects working with Qt 6, prefer to use CMake 3.21 or later to take advantage of a range of new CMake features and bug fixes relevant to Qt.

When bringing Qt into a project with `find_package()`, prefer to find the `Qt5` umbrella package and specify `COMPONENTS` in that one call rather than finding each component as its own separate `find_package()` call. This communicates the intent more clearly and is more robust, since it ensures all Qt components come from the same Qt installation.

Avoid referring to things Qt provides through variables. Instead, prefer to use the imported targets like `Qt5::Core`, `Qt5::Gui`, etc. The imported targets are much more robust and more convenient to use, as they come with transitive properties that greatly simplify the task of defining targets that link against Qt libraries.

Projects should prefer to use `AUTOMOC`, `AUTOUIC` and `AUTORCC` over the manual commands provided by Qt. The `AUTOxxx` functionality is considerably simpler and results in more concise project files. It is also easier to maintain as files are added, modified, renamed and moved over the course of a project's lifetime. Only where it is clear that the impact on build performance warrants the extra effort or where the desired processing isn't supported (e.g. dynamically loadable resources) should the manual commands provided by Qt be used.

When defining translation files to be part of the project, prefer to follow the same default naming convention that Qt Linguist uses. Translation files should ideally be named according to the pattern `something_xx_YY.ts`, where `xx` is a lowercase ISO639 language code and `YY` is an uppercase ISO3166 country code. The `_YY` part can be omitted if country-based localization isn't needed. Avoid using more than one period character in the name. In particular, do not use names of the form `something.xx_YY.ts`, even though some examples in the Qt documentation do this. Prior to Qt 5.12.5, such names can trigger a bug in the implementation of commands like `qt5_create_translation()` and `qt5_add_translation()` which can lead to build failures when there is more than one `.ts` file.

Prefer to use the deployment tools provided by Qt where they are available. Tools like `macdeployqt`, `windeployqt` and `androiddeployqt` handle much of the Qt-specific aspects of deploying an application, freeing the project from having to maintain logic for the different platforms and Qt versions. For projects that support localization, ensure that Qt's `.qm` files are also included in the deployment and that the application knows where to find them. Handling of these Qt-provided `.qm` files is not covered by all deployment tools, so projects may need to implement that for themselves.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

If creating packages for a Qt GUI application, consider using the IFW package generator, which uses the Qt Installer Framework. The same end user experience can be provided across all desktop platforms, the look and feel can be customized to match the application, and localization is supported.

Exclusively for sbmain17@gmail.com Transaction: 0098315591

Index

@

<CONFIG>_POSTFIX, 144
<LANG>_COMPILER_LAUNCHER, 515
<LANG>_EXTENSIONS, 165
<LANG>_STANDARD, 164
<LANG>_STANDARD_REQUIRED, 165
<LANG>_VISIBILITY_PRESET, 246
<projectName>_IS_TOP_LEVEL, 61

A

add_compile_definitions(), 154
add_compile_options(), 154
add_custom_command(), 187, 188
add_custom_target(), 184
add_definitions(), 153
add_dependencies(), 185, 459, 522
add_executable(), 13, 15, 171, 273
add_library(), 16, 172, 237
add_link_options(), 155
add_subdirectory(), 52
add_test(), 327
ADDITIONAL_CLEAN_FILES, 494
ADDITIONAL_MAKE_CLEAN_FILES, 494
ANDROID_PLATFORM, 267
androiddeployqt, 538
ARCHIVE_OUTPUT_DIRECTORY, 491
ARCHIVE_OUTPUT_NAME, 492
ARGC, 66
ARGN, 66
ARGV, 66
ATTACHED_FILES, 360
ATTACHED_FILES_ON_FAIL, 360
AUTOMOC, 528
AUTOMOC_DEPEND_FILTERS, 530
AUTOMOC_MACRO_NAMES, 529
AUTOMOC_MOC_OPTIONS, 528
AUTORCC, 533
AUTORCC_OPTIONS, 534
AUTOUIC, 532
AUTOUIC_OPTIONS, 532
AUTOUIC_SEARCH_PATHS, 532

B

Boolean true/false values, 40
break(), 50

Exclusively for sbmain17@gmail.com Transaction: 0098315591

Build types (default), 137
BUILD_INTERFACE, 382
BUILD_RPATH, 384
BUILD_RPATH_USE_ORIGIN, 384
BUILD_SHARED_LIBS, 17, 172
BUILD_TESTING, 356
BUILD_WITH_INSTALL_RPATH, 384
BUNDLE, 282
BUNDLE_EXTENSION, 282
BundleUtilities, 402

C

C_EXTENSIONS, 165
C_STANDARD, 164
C_STANDARD_REQUIRED, 165
C_VISIBILITY_PRESET, 246
ccmake, 32
check_c_compiler_flag(), 112
check_c_source_compiles(), 110
check_c_source_runs(), 112
check_compiler_flag(), 112
check_cxx_compiler_flag(), 112
check_cxx_source_compiles(), 110
check_cxx_source_runs(), 112
check_cxx_symbol_exists(), 114
check_fortran_compiler_flag(), 112
check_fortran_source_compiles(), 110
check_fortran_source_runs(), 112
check_source_compiles(), 110, 110
check_source_runs(), 112
check_symbol_exists(), 114
CheckCCompilerFlag, 112
CheckCompilerFlag, 112
CheckCSourceCompiles, 110
CheckCSourceRuns, 112
CheckCXXCompilerFlag, 112
CheckCXXSourceCompiles, 110
CheckCXXSourceRuns, 112
CheckCXXSymbolExists, 113
CheckFortranCompilerFlag, 112
CheckFortranFunctionExists, 114
CheckFortranSourceCompiles, 110
CheckFortranSourceRuns, 112
CheckFunctionExists, 114
CheckIncludeFile, 114
CheckIncludeFileCXX, 114
CheckIncludeFiles, 114
CheckLanguage, 114

Exclusively for sbmain17@gmail.com Transaction: 0098315591

CheckLibraryExists, 114
CheckLinkerFlag, 113
CheckPIESupported, 527
CheckPrototypeDefinition, 114
CheckSourceCompiles, 110
CheckSourceRuns, 112
CheckStructHasMember, 114
CheckSymbolExists, 113
CheckTypeSize, 114
CheckVariableExists, 114
cmake, 8, 28
cmake -E command mode, 196, 220
cmake-gui, 29
CMAKE_<CONFIG>_POSTFIX, 144
CMAKE_<LANG>_BYTE_ORDER, 117
CMAKE_<LANG>_COMPILE_FEATURES, 167
CMAKE_<LANG>_COMPILER, 260
CMAKE_<LANG>_COMPILER_EXTERNAL_TOOLCHAIN, 263
CMAKE_<LANG>_COMPILER_ID, 261
CMAKE_<LANG>_COMPILER_LAUNCHER, 515
CMAKE_<LANG>_COMPILER_TARGET, 263
CMAKE_<LANG>_COMPILER_VERSION, 261
CMAKE_<LANG>_EXTENSIONS, 165, 165
CMAKE_<LANG>_EXTENSIONS_DEFAULT, 165
CMAKE_<LANG>_FLAGS, 157, 262
CMAKE_<LANG>_FLAGS_<CONFIG>, 143, 157, 262
CMAKE_<LANG>_FLAGS_INIT, 262
CMAKE_<LANG>_KNOWN_FEATURES, 167
CMAKE_<LANG>_STANDARD, 165
CMAKE_<LANG>_STANDARD_REQUIRED, 165
CMAKE_<LANG>_VISIBILITY_PRESET, 246
CMAKE_<TARGETTYPE>_LINKER_FLAGS, 158, 262
CMAKE_<TARGETTYPE>_LINKER_FLAGS_<CONFIG>, 143, 158, 262
CMAKE_<TARGETTYPE>_LINKER_FLAGS_INIT, 262
CMAKE_ANDROID_ARCH_ABI, 267
CMAKE_ANDROID_ARM_MODE, 267
CMAKE_ANDROID_ARM_NEON, 267
CMAKE_ANDROID_EXCEPTIONS, 268
CMAKE_ANDROID_RTTI, 268
CMAKE_ANDROID_STL_TYPE, 267
CMAKE_APPBUNDLE_PATH, 309, 317
CMAKE_APPLE_SILICON_PROCESSOR
    Cache variable, 292
    Environment variable, 292
CMAKE_ARCHIVE_OUTPUT_DIRECTORY, 491
CMAKE_AUTOMOC, 528
CMAKE_AUTOMOC_MACRO_NAMES, 529
CMAKE_AUTORCC, 533

Exclusively for sbmain17@gmail.com Transaction: 0098315591

CMAKE_AUTOUIC, 532
CMAKE_AUTOUIC_OPTIONS, 533
CMAKE_BINARY_DIR, 53
CMAKE_BUILD_RPATH, 384
CMAKE_BUILD_RPATH_USE_ORIGIN, 384
CMAKE_BUILD_TYPE, 138, 216
CMAKE_BUILD_WITH_INSTALL_RPATH, 385
CMAKE_COMMAND, 197
CMAKE_COMPONENTS_ALL, 426
CMAKE_CONFIGURATION_TYPES, 140, 427
CMAKE_CONFIGURE_DEPENDS, 495
CMAKE_CROSSCOMPILING, 259, 264
CMAKE_CROSSCOMPILING_EMULATOR, 264, 349
CMAKE_CTEST_ARGUMENTS, 331
CMAKE_CURRENT_BINARY_DIR, 53
CMAKE_CURRENT_FUNCTION, 77
CMAKE_CURRENT_FUNCTION_LIST_DIR, 77
CMAKE_CURRENT_FUNCTION_LIST_FILE, 77
CMAKE_CURRENT_FUNCTION_LIST_LINE, 77
CMAKE_CURRENT_LIST_DIR, 58
CMAKE_CURRENT_LIST_FILE, 58
CMAKE_CURRENT_LIST_LINE, 58
CMAKE_CURRENT_SOURCE_DIR, 53
CMAKE_DEFAULT_BUILD_TYPE, 139
CMAKE_DISABLE_FIND_PACKAGE_<packageName>, 319
CMAKE_ERROR_DEPRECATED, 125
CMAKE_EXECUTE_PROCESS_COMMAND_ECHO, 195
CMAKE_EXPORT_NO_PACKAGE_REGISTRY, 320
CMAKE_EXPORT_PACKAGE_REGISTRY, 320
CMAKE_EXTRA_INCLUDE_FILES, 115
CMAKE_FIND_APPBUNDLE, 309
CMAKE_FIND_DEBUG_MODE, 324
CMAKE_FIND_FRAMEWORK, 307, 310
CMAKE_FIND_LIBRARY_CUSTOM_LIB_SUFFIX, 311
CMAKE_FIND_PACKAGE_NO_PACKAGE_REGISTRY, 320
CMAKE_FIND_PACKAGE_NO_SYSTEM_PACKAGE_REGISTRY, 320
CMAKE_FIND_PACKAGE_PREFER_CONFIG, 314
CMAKE_FIND_PACKAGE_SORT_DIRECTION, 318
CMAKE_FIND_PACKAGE_SORT_ORDER, 318
CMAKE_FIND_ROOT_PATH, 307
CMAKE_FIND_ROOT_PATH_MODE_INCLUDE, 308
CMAKE_FIND_ROOT_PATH_MODE_LIBRARY, 310
CMAKE_FIND_ROOT_PATH_MODE_PACKAGE, 318
CMAKE_FIND_ROOT_PATH_MODE_PROGRAM, 310
CMAKE_FIND_USE_CMAKE_ENVIRONMENT_PATH, 304
CMAKE_FIND_USE_CMAKE_PATH, 304
CMAKE_FIND_USE_CMAKE_SYSTEM_PATH, 305
CMAKE_FIND_USE_PACKAGE_REGISTRY, 320

Exclusively for sbmain17@gmail.com Transaction: 0098315591

CMAKE_FIND_USE_SYSTEM_ENVIRONMENT_PATH, 304
CMAKE_FIND_USE_SYSTEM_PACKAGE_REGISTRY, 320
CMAKE_FOLDER, 484
CMAKE_FRAMEWORK, 279
CMAKE_FRAMEWORK_PATH, 303, 310, 317
CMAKE_GENERATOR, 8
CMAKE_GENERATOR_INSTANCE, 261
CMAKE_GENERATOR_NO_COMPILER_ENV, 195
CMAKE_GENERATOR_PLATFORM, 260
CMAKE_GENERATOR_TOOLSET, 261
CMAKE_GNUtoMS, 238
CMAKE_GTEST_DISCOVER_TESTS_DISCOVERY_MODE, 368
CMAKE_HOST_SYSTEM_NAME, 259, 501
CMAKE_HOST_SYSTEM_PROCESSOR, 259
CMAKE_IGNORE_PATH, 308
CMAKE_INCLUDE_CURRENT_DIR, 530, 532
CMAKE_INCLUDE_DIRECTORIES_BEFORE, 153
CMAKE_INCLUDE_PATH, 303
CMAKE_INSTALL_<dir>, 373
CMAKE_INSTALL_DEFAULT_COMPONENT_NAME, 380
CMAKE_INSTALL_MODE, 415
CMAKE_INSTALL_NAME_DIR, 387
CMAKE_INSTALL_PREFIX, 375, 386
CMAKE_INSTALL_RPATH, 384
CMAKE_INSTALL_RPATH_USE_LINK_PATH, 384
CMAKE_INSTALL_SYSTEM_RUNTIME_COMPONENT, 400
CMAKE_INSTALL_SYSTEM_RUNTIME_DESTINATION, 401
CMAKE_INSTALL_SYSTEM_RUNTIME_LIBS, 401
CMAKE_INSTALL_SYSTEM_RUNTIME_LIBS_SKIP, 401
CMAKE_IOS_INSTALL_COMBINED, 293, 300
cmake_language()
    CALL, 78, 86
    DEFER, 79, 86
    EVAL, 78, 85, 86
CMAKE_LIBRARY_ARCHITECTURE, 303, 316
CMAKE_LIBRARY_OUTPUT_DIRECTORY, 491
CMAKE_LIBRARY_PATH, 310
CMAKE_MACOSX_BUNDLE, 274
CMAKE_MACOSX_RPATH, 387
CMAKE_MATCH_<n>, 44
CMAKE_MESSAGE_CONTEXT, 127
CMAKE_MESSAGE_INDENT, 126, 130
CMAKE_MESSAGE_LOG_LEVEL, 126
cmake_minimum_required(), 10, 118
CMAKE_MODULE_PATH, 108, 116, 314
CMAKE_NO_BUILTIN_CHRPATH, 386
CMAKE_OPTIMIZE_DEPENDENCIES, 522, 522
CMAKE_OSX_ARCHITECTURES, 291, 299

Exclusively for sbmain17@gmail.com Transaction: 0098315591

CMAKE_OSX_DEPLOYMENT_TARGET, 276, 283
CMAKE_OSX_SYSROOT, 282
cmake_parse_arguments(), 68
cmake_path(), 201
cmake_policy(), 118
CMAKE_POLICY_DEFAULT_CMP<NNNN>, 121
CMAKE_POLICY_WARNING_CMP<NNNN>, 121
cmake_pop_check_state(), 114
CMAKE_POSITION_INDEPENDENT_CODE, 255
CMAKE_PREFIX_PATH, 303, 309, 310, 314, 317, 402, 463, 477
cmake_print_properties(), 131
cmake_print_variables(), 131
CMAKE_PROGRAM_PATH, 309
CMAKE_PROJECT_<PROJNAME>_INCLUDE, 495
CMAKE_PROJECT_DESCRIPTION, 419
CMAKE_PROJECT_INCLUDE, 495
CMAKE_PROJECT_INCLUDE_BEFORE, 495
CMAKE_PROJECT_NAME, 418
CMAKE_PROJECT_VERSION, 228
CMAKE_PROJECT_VERSION_MAJOR, 228, 419
CMAKE_PROJECT_VERSION_MINOR, 228, 419
CMAKE_PROJECT_VERSION_PATCH, 228, 419
CMAKE_PROJECT_VERSION_TWEAK, 228
cmake_push_check_state(), 114
CMAKE_REQUIRE_FIND_PACKAGE_<packageName>, 319
CMAKE_REQUIRED_DEFINITIONS, 111
CMAKE_REQUIRED_FLAGS, 111
CMAKE_REQUIRED_INCLUDES, 111
CMAKE_REQUIRED_LIBRARIES, 111, 114
CMAKE_REQUIRED_LINK_OPTIONS, 111
CMAKE_REQUIRED_QUIET, 111
cmake_reset_check_state(), 114
CMAKE_RUNTIME_OUTPUT_DIRECTORY, 491
CMAKE_SIZEOF_VOID_P, 266
CMAKE_SKIP_BUILD_RPATH, 385
CMAKE_SKIP_INSTALL_RPATH, 385
CMAKE_SKIP_RPATH, 385
CMAKE_SOURCE_DIR, 53
CMAKE_STAGING_PREFIX, 263, 308, 375, 386
CMAKE_SYSROOT, 263, 307, 386
CMAKE_SYSROOT_COMPILE, 263, 307
CMAKE_SYSROOT_LINK, 263, 307
CMAKE_SYSTEM_APPBUNDLE_PATH, 317
CMAKE_SYSTEM_FRAMEWORK_PATH, 305, 317
CMAKE_SYSTEM_IGNORE_PATH, 308
CMAKE_SYSTEM_INCLUDE_PATH, 305
CMAKE_SYSTEM_NAME, 259, 282
CMAKE_SYSTEM_PREFIX_PATH, 305, 317

Exclusively for sbmain17@gmail.com Transaction: 0098315591

CMAKE_SYSTEM_PROCESSOR, 259
CMAKE_SYSTEM_VERSION, 259
CMAKE_TOOLCHAIN_FILE, 258
CMAKE_TRY_COMPILE_PLATFORM_VARIABLES, 264
CMAKE_TRY_COMPILE_TARGET_TYPE, 264
CMAKE_UNITY_BUILD, 509
CMAKE_UNITY_BUILD_BATCH_SIZE, 511
CMAKE_VISIBILITY_INLINES_HIDDEN, 246
CMAKE_WARN_DEPRECATED, 125
CMAKE_WINDOWS_EXPORT_ALL_SYMBOLS, 246
CMAKE_XCODE_ATTRIBUTE_CODE_SIGN_IDENTITY, 286
CMAKE_XCODE_ATTRIBUTE_CODE_SIGNING_ALLOWED, 284
CMAKE_XCODE_ATTRIBUTE_DEVELOPMENT_TEAM, 286
CMAKE_XCODE_ATTRIBUTE_IPHONEOS_DEPLOYMENT_TARGET, 283
CMAKE_XCODE_ATTRIBUTE_ONLY_ACTIVE_ARCH, 291, 299
CMAKE_XCODE_ATTRIBUTE_XXX, 272
CMAKE_XCODE_GENERATE_SCHEME, 288
CMAKE_XCODE_GENERATE_TOP_LEVEL_PROJECT_ONLY, 298
CMakeCache.txt, 6
CMakeFindDependencyMacro, 404
CMakeForceCompiler, 264
CMakeLists.txt, 6
CMakePackageConfigHelpers, 409
CMakePrintHelpers, 131
CMakePushCheckState, 115
Commenting, 13
COMPATIBLE_INTERFACE_BOOL, 240
COMPATIBLE_INTERFACE_NUMBER_MAX, 242
COMPATIBLE_INTERFACE_NUMBER_MIN, 242
COMPATIBLE_INTERFACE_STRING, 241
COMPILE_DEFINITIONS
    Directory property, 154
    Source property, 147
    Target property, 146
COMPILE_FEATURES, 167
COMPILE_FLAGS
    Source property, 147
    Target property, 147
COMPILE_OPTIONS
    Directory property, 154
    Source property, 147
    Target property, 147
COMPILE_PDB, 494
COMPILE_PDB_<CONFIG>, 494
COMPILE_PDB_OUTPUT_DIRECTORY, 494
COMPILE_PDB_OUTPUT_DIRECTORY_<CONFIG>, 494
configure_file(), 208, 229
configure_package_config_file(), 410

Exclusively for sbmain17@gmail.com Transaction: 0098315591

continue(), 50
COVERAGE_COMMAND, 357
COVERAGE_EXTRA_FLAGS, 357
cpack command, 417
CPACK_<GENNAME>_COMPONENT_INSTALL, 425
cpack_add_component(), 422
cpack_add_component_group(), 423
cpack_add_install_type(), 423
CPACK_ARCHIVE_<COMP>_FILE_NAME, 429
CPACK_ARCHIVE_COMPONENT_INSTALL, 429
CPACK_ARCHIVE_FILE_NAME, 429
CPACK_BUILD_SOURCE_DIRS, 445
CPACK_COMPONENT_INCLUDE_TOPLEVEL_DIRECTORY, 430
CPACK_COMPONENTS_ALL, 425, 443
CPACK_COMPONENTS_GROUPING, 425, 443
cpack_configure_downloads(), 434
CPACK_DEB_COMPONENT_INSTALL, 447
CPACK_DEBIAN_<COMP>_FILE_NAME, 447
CPACK_DEBIAN_<COMP>_PACKAGE_DEPENDS, 447
CPACK_DEBIAN_<COMP>_PACKAGE_NAME, 447
CPACK_DEBIAN_<COMP>_PACKAGE_SHLIBDEPS, 447
CPACK_DEBIAN_ENABLE_COMPONENT_DEPENDS, 447
CPACK_DEBIAN_FILE_NAME, 447
CPACK_DEBIAN_PACKAGE_DEPENDS, 447
CPACK_DEBIAN_PACKAGE_GENERATE_SHLIBS, 447
CPACK_DEBIAN_PACKAGE_GENERATE_SHLIBS_POLICY, 447
CPACK_DEBIAN_PACKAGE_NAME, 447
CPACK_DEBIAN_PACKAGE_SHLIBDEPS, 447
CPACK_DMG_<component>_FILE_NAME, 440
CPACK_DMG_BACKGROUND_IMAGE, 440
CPACK_DMG_DISABLE_APPLICATIONS_SYMLINK, 440
CPACK_DMG_DS_STORE, 440
CPACK_DMG_DS_STORE_SETUP_SCRIPT, 440
CPACK_DMG_FILESYSTEM, 440
CPACK_DMG_SLA_DIR, 440
CPACK_DMG_SLA_LANGUAGES, 440
CPACK_GENERATOR, 421, 428
cpack_ifw_add_repository(), 435
cpack_ifw_configure_component(), 431
cpack_ifw_configure_component_group(), 431
CPACK_IFW_PACKAGE_GROUP, 433
CPACK_IFW_PACKAGE_ICON, 430
CPACK_IFW_PACKAGE_LOGO, 430
CPACK_IFW_PACKAGE_MAINTENANCE_TOOL_INI_FILE, 433
CPACK_IFW_PACKAGE_MAINTENANCE_TOOL_NAME, 433
CPACK_IFW_PACKAGE_WINDOW_ICON, 430
CPACK_INSTALL_CMAKE_PROJECTS, 426
CPACK_MONOLITHIC_INSTALL, 425, 443

Exclusively for sbmain17@gmail.com Transaction: 0098315591

CPACK_NSIS_BRANDING_TEXT, 439
CPACK_NSIS_BRANDING_TEXT_TRIM_POSITION, 439
CPACK_NSIS_ENABLE_UNINSTALL_BEFORE_INSTALL, 438
CPACK_NSIS_FINISH_TITLE, 439
CPACK_NSIS_IGNORE_LICENSE_PAGE, 439
CPACK_NSIS_MANIFEST_DPI_AWARE, 439
CPACK_NSIS_MUI_HEADERIMAGE, 439
CPACK_NSIS_UNINSTALL_NAME, 439
CPACK_NSIS_WELCOME_TITLE, 439
CPACK_PACKAGE_DESCRIPTION, 420
CPACK_PACKAGE_DESCRIPTION_FILE, 420
CPACK_PACKAGE_DESCRIPTION_SUMMARY, 419
CPACK_PACKAGE_ICON, 420, 429, 440
CPACK_PACKAGE_INSTALL_DIRECTORY, 419
CPACK_PACKAGE_NAME, 418
CPACK_PACKAGE_VENDOR, 419
CPACK_PACKAGE_VERSION, 419
CPACK_PACKAGE_VERSION_MAJOR, 419
CPACK_PACKAGE_VERSION_MINOR, 419
CPACK_PACKAGE_VERSION_PATCH, 419
CPACK_PACKAGING_INSTALL_PREFIX, 445
CPACK_PRODUCTBUILD_IDENTITY_NAME, 441
CPACK_PRODUCTBUILD_KEYCHAIN_PATH, 441
CPACK_PROJECT_CONFIG_FILE, 428
CPACK_RESOURCE_FILE_LICENSE, 420
CPACK_RESOURCE_FILE_README, 420
CPACK_RESOURCE_FILE_WELCOME, 420
CPACK_RPM_<COMP>_DEBUGINFO_PACKAGE, 445
CPACK_RPM_<COMP>_FILE_NAME, 443
CPACK_RPM_<COMP>_PACKAGE_ARCHITECTURE, 444
CPACK_RPM_<COMP>_PACKAGE_NAME, 443
CPACK_RPM_COMPONENT_INSTALL, 443
CPACK_RPM_DEBUGINFO_EXCLUDE_DIRS, 446
CPACK_RPM_DEBUGINFO_EXCLUDE_DIRS_ADDITION, 446
CPACK_RPM_DEBUGINFO_PACKAGE, 445
CPACK_RPM_FILE_NAME, 443
CPACK_RPM_PACKAGE_ARCHITECTURE, 444
CPACK_RPM_PACKAGE_EPOCH, 444
CPACK_RPM_PACKAGE_NAME, 443
CPACK_RPM_PACKAGE_RELEASE, 444
CPACK_RPM_PACKAGE_RELOCATABLE, 445
CPACK_RPM_RELOCATION_PATHS, 445
CPACK_RPM_SPEC_INSTALL_POST, 445
CPACK_RPM_SPEC_MORE_DEFINE, 445
CPACK_SOURCE_IGNORE_FILES, 421
CPACK_STRIP_FILES, 445
CPACK_VERBATIM_VARIABLES, 419
CPACK_WIX_PRODUCT_GUID, 436

Exclusively for sbmain17@gmail.com Transaction: 0098315591

CPACK_WIX_UPGRADE_GUID, 436
CPackComponent, 422
CROSSCOMPILING_EMULATOR, 349
ctest command, 329
CTEST_CUSTOM_MAXIMUM_FAILED_TEST_OUTPUT_SIZE, 363
CTEST_CUSTOM_MAXIMUM_PASSED_TEST_OUTPUT_SIZE, 363
CTEST_OUTPUT_ON_FAILURE, 330
CTEST_PARALLEL_LEVEL, 339
CTEST_RESOURCE_GROUP_<num>, 345
CTEST_RESOURCE_GROUP_<num>_<resourceType>, 345
CTEST_RESOURCE_GROUP_COUNT, 345
CTEST_RESOURCE_SPEC_FILE, 343
CUDA_EXTENSIONS, 165
CUDA_STANDARD, 164
CUDA_STANDARD_REQUIRED, 165
CXX_EXTENSIONS, 165
CXX_STANDARD, 164
CXX_STANDARD_REQUIRED, 165
CXX_VISIBILITY_PRESET, 246

D

DEBUG_CONFIGURATIONS, 144, 155
define_property(), 92
DEPENDS, 346
DESTDIR, 375
DISABLED, 333
DRMEMORY_COMMAND_OPTIONS, 358

E

enable_language(), 257
enable_testing(), 327, 356
ENVIRONMENT, 330, 492
ENVIRONMENT_MODIFICATION, 330
EP_STEP_TARGETS, 459
execute_process(), 192
export(), 269, 319, 392
EXPORT_NAME, 390
ExternalData, 473
ExternalProject, 451, 476
ExternalProject_Add(), 452
ExternalProject_Add_Step(), 460
ExternalProject_Add_StepDependencies(), 459
ExternalProject_Add_StepTargets(), 461

F

FAIL_REGULAR_EXPRESSION, 331
FetchContent, 465, 478
FetchContent_Declare(), 467

Exclusively for sbmain17@gmail.com Transaction: 0098315591

FetchContent_GetProperties(), 467
FetchContent_Populate(), 467
file()
    APPEND, 215
    CHMOD, 221
    CHMOD_RECURSE, 221
    CONFIGURE, 210
    COPY, 211
    COPY_FILE, 213
    CREATE_LINK, 220
    DOWNLOAD, 223
    GENERATE, 216
    GET_RUNTIME_DEPENDENCIES, 399
    GLOB, 221
    GLOB_RECURSE, 221
    INSTALL, 211
    MAKE_DIRECTORY, 219
    READ, 218
    READ_SYMLINK, 220
    REAL_PATH, 206
    RELATIVE_PATH, 206
    REMOVE, 219
    REMOVE_RECURSE, 219
    RENAME, 219
    SIZE, 221
    STRINGS, 218
    TO_CMAKE_PATH, 207
    TO_NATIVE_PATH, 207
    UPLOAD, 223
    WRITE, 215
find_dependency(), 404
find_file(), 302
find_library(), 310
FIND_LIBRARY_USE_LIB32_PATHS, 311
FIND_LIBRARY_USE_LIB64_PATHS, 311
FIND_LIBRARY_USE_LIBX32_PATHS, 311
find_package(), 109, 312
find_path(), 309
find_program(), 309
FindPkgConfig, 321
FIXTURES_CLEANUP, 347
FIXTURES_REQUIRED, 347
FIXTURES_SETUP, 347
FOLDER, 483
foreach(), 48
FRAMEWORK, 279
FRAMEWORK_VERSION, 279
function(), 64

Exclusively for sbmain17@gmail.com Transaction: 0098315591

G

generate_export_header(), 247
GenerateExportHeader, 247
GENERATOR_IS_MULTI_CONFIG, 141
Generators
    Package, 428
    Project, 7
get_cmake_property(), 93, 425
get_directory_property(), 94
get_filename_component(), 204
get_property(), 91, 97
get_source_file_property(), 96
get_target_property(), 95
get_test_property(), 98
GetPrerequisites, 402
GNUInstallDirs, 373, 491
GNUtoMS, 238
gtest_add_test(), 364
gtest_discover_tests(), 365

H

Header-only libraries, 175, 409
HEADER_FILE_ONLY, 486

I

if(), 40
IMPORTED_GLOBAL, 180, 482
IMPORTED_IMPLIB, 174
IMPORTED_LOCATION, 171, 174
IMPORTED_LOCATION_<CONFIG>, 171, 174
IMPORTED_OBJECTS, 174
INCLUDE, 304
include(), 57, 108, 161
INCLUDE_DIRECTORIES
    Directory property, 153
    Target property, 146
include_directories(), 153
include_guard(), 62
install()
    CODE, 397
    DIRECTORY, 394
    EXPORT, 389
    EXPORT_ANDROID_MK, 392
    FILES, 394
    IMPORTED_RUNTIME_ARTIFACTS, 393
    PROGRAMS, 394
    RUNTIME_DEPENDENCY_SET, 398

Exclusively for sbmain17@gmail.com Transaction: 0098315591

SCRIPT, 397
TARGETS, 376
INSTALL_INTERFACE, 382
INSTALL_NAME_DIR, 387
INSTALL_RPATH, 384
INSTALL_RPATH_USE_LINK_PATH, 384
InstallRequiredSystemLibraries, 400
INTERFACE_AUTOUIC_OPTIONS, 533
INTERFACE_COMPILE_DEFINITIONS, 146
INTERFACE_COMPILE_FEATURES, 167
INTERFACE_COMPILE_OPTIONS, 147
INTERFACE_INCLUDE_DIRECTORIES, 146, 382
INTERFACE_LINK_LIBRARIES, 148
INTERFACE_LINK_OPTIONS, 148
INTERFACE_PRECOMPILE_HEADERS, 513
IOS_INSTALL_COMBINED, 293, 300

L

LABELS, 336
lconvert, 540
LIBRARY_OUTPUT_DIRECTORY, 491
LIBRARY_OUTPUT_NAME, 492
LINK_FLAGS, 148
LINK_INTERFACE_LIBRARIES, 149
LINK_INTERFACE_MULTIPLICITY, 238
LINK_LIBRARIES, 148
link_libraries(), 155
LINK_OPTIONS
Directory property, 155
Target property, 148
LINKER: prefix, 149, 156
list(), 35
APPEND, 35
FIND, 36
GET, 35
INSERT, 35
LENGTH, 35
POP_BACK, 36
POP_FRONT, 36
PREPEND, 35
REMOVE_AT, 36
REMOVE_DUPLICATES, 36
REMOVE_ITEM, 36
REVERSE, 37
SORT, 37
LOCATION, 145, 492
lrelease, 535
lupdate, 535

Exclusively for sbmain17@gmail.com Transaction: 0098315591

M

macdeployqt, 538
MACHO_COMPATIBILITY_VERSION, 240
MACHO_CURRENT_VERSION, 240
MACOSX_BUNDLE, 274
MACOSX_BUNDLE_BUNDLE_NAME, 274
MACOSX_BUNDLE_BUNDLE_VERSION, 274
MACOSX_BUNDLE_COPYRIGHT, 274
MACOSX_BUNDLE_GUI_IDENTIFIER, 274
MACOSX_BUNDLE_ICON_FILE, 274
MACOSX_BUNDLE_INFO_PLIST, 274
MACOSX_BUNDLE_INFO_STRING, 274
MACOSX_BUNDLE_LONG_VERSION_STRING, 274
MACOSX_BUNDLE_SHORT_VERSION_STRING, 274
MACOSX_FRAMEWORK_BUNDLE_VERSION, 280
MACOSX_FRAMEWORK_ICON_FILE, 280
MACOSX_FRAMEWORK_IDENTIFIER, 280
MACOSX_FRAMEWORK_INFO_PLIST, 280
MACOSX_FRAMEWORK_SHORT_VERSION_STRING, 280
MACOSX_PACKAGE_LOCATION, 277, 486
MACOSX_RPATH, 387
macro(), 64
mark_as_advanced(), 30
math(), 38
MEMORYCHECK_COMMAND, 358
MEMORYCHECK_TYPE, 358
message(), 32, 124
moc, 528
MODULE, 282

N

NO_SYSTEM_FROM_IMPORTED, 152

O

OBJC_EXTENSIONS, 165
OBJC_STANDARD, 164
OBJC_STANDARD_REQUIRED, 165
OBJCXX_EXTENSIONS, 165
OBJCXX_STANDARD, 164
OBJCXX_STANDARD_REQUIRED, 165
ONLY_ACTIVE_ARCH, 290
OPTIMIZE_DEPENDENCIES, 521
option(), 25, 47
OSX_DEPLOYMENT_TARGET, 283
OUTPUT_NAME, 473, 492

Exclusively for sbmain17@gmail.com Transaction: 0098315591

P

PARENT_SCOPE, 22, 55, 73
PASS_REGULAR_EXPRESSION, 331
PATH, 304
PDB_NAME, 494
PDB_NAME_<CONFIG>, 494
PDB_OUTPUT_DIRECTORY, 494
PDB_OUTPUT_DIRECTORY_<CONFIG>, 494
pkg-config, 321
pkg_check_modules(), 321
PKG_CONFIG_USE_CMAKE_PREFIX_PATH, 322
pkg_get_variable(), 323
pkg_search_module(), 321
POSITION_INDEPENDENT_CODE, 254, 526
PRECOMPILE_HEADERS, 513
PREDEFINED_TARGETS_FOLDER, 484
PRIVATE_HEADER, 280, 377, 389
PROCESSORS, 339
project(), 12, 59, 227, 257
PROJECT_BINARY_DIR, 59
PROJECT_IS_TOP_LEVEL, 61
PROJECT_LABEL, 484
PROJECT_SOURCE_DIR, 59
PROJECT_VERSION, 227, 409
PROJECT_VERSION_MAJOR, 227
PROJECT_VERSION_MINOR, 227
PROJECT_VERSION_PATCH, 227
PROJECT_VERSION_TWEAK, 227
PUBLIC_HEADER, 280, 377, 389

Q

qt5_add_big_resources(), 534
qt5_add_binary_resources(), 534
qt5_add_resources(), 534
qt5_add_translation(), 536
qt5_create_translation(), 536
Qt5_NO_LINK_QTMAIN, 527
qt5_wrap_cpp(), 530
qt5_wrap_ui(), 533

R

rcc, 533
remove_definitions(), 153
RESOURCE, 277, 377
RESOURCE_GROUPS, 341
RESOURCE_LOCK, 340
return(), 61

Exclusively for sbmain17@gmail.com Transaction: 0098315591

RPATH, 383, 492
RUN_SERIAL, 340
RUNTIME_OUTPUT_DIRECTORY, 491
RUNTIME_OUTPUT_NAME, 492

S

set(), 22, 24
set_directory_properties(), 94
set_property(), 90, 96
set_source_files_properties(), 96
set_target_properties(), 95
set_tests_properties(), 98
SHELL: prefix, 156
SKIP_AUTOGEN, 530, 532, 534
SKIP_AUTOMOC, 530
SKIP_AUTORCC, 534
SKIP_AUTOUIC, 532
SKIP_BUILD_RPATH, 385
SKIP_REGULAR_EXPRESSION, 333, 345
SKIP_RETURN_CODE, 332, 345
SKIP_UNITY_BUILD_INCLUSION, 510
source_group(), 484
SOVERSION, 239
STATIC_LIBRARY_FLAGS, 149
STATIC_LIBRARY_OPTIONS, 149
STOP_ON_FAILURE, 370
string(), 33
    CONFIGURE, 210
    FIND, 33
    LENGTH, 34
    REGEX, 33
    REPLACE, 33
    STRIP, 34
    SUBSTRING, 34
    TOLOWER, 34
    TOUPPER, 34
Superbuilds, 476
Syntax
    Generator expressions, 101
    Lua-style brackets, 23, 215

T

target_compile_definitions(), 152
target_compile_features(), 167
target_compile_options(), 152
TARGET_FILE, 394
target_include_directories(), 151
target_link_libraries(), 17, 144, 150

Exclusively for sbmain17@gmail.com Transaction: 0098315591

target_link_options(), 150
target_precompile_headers(), 513
target_sources(), 176, 488
TEST_INCLUDE_FILES, 367
TIMEOUT, 333
TIMEOUT_AFTER_MATCH, 334
Toolchain files, 258
try_compile(), 111, 264
try_run(), 112, 264

U

uic, 532
UNITY_BUILD, 509
UNITY_BUILD_BATCH_SIZE, 510
UNITY_BUILD_MODE, 510
UNITY_BUILD_UNIQUE_ID, 510
unset(), 24
USE_FOLDERS, 483

V

VALGRIND_COMMAND_OPTIONS, 358
variable_watch(), 132
Variables
    Cache, 24
    Environment, 24
    Regular, 22
VERSION
    project() command, 12, 227
    Target property, 239
Versioning
    Comparisons, 43
    Projects, 227
    Shared libraries, 238
VISIBILITY_INLINES_HIDDEN, 246
VS_DEBUGGER_COMMAND, 493
VS_DEBUGGER_COMMAND_ARGUMENTS, 493
VS_DEBUGGER_ENVIRONMENT, 493
VS_DEBUGGER_WORKING_DIRECTORY, 493
VS_USER_PROPS, 493

W

while(), 50
WILL_FAIL, 333
WIN32_EXECUTABLE, 527
windeployqt, 538
WINDOWS_EXPORT_ALL_SYMBOLS, 246
write_basic_package_version_file(), 409
WriteCompilerDetectionHeader, 169

Exclusively for sbmain17@gmail.com Transaction: 0098315591

X

XCODE_ATTRIBUTE_CODE_SIGN_ENTITLEMENTS, 288
XCODE_ATTRIBUTE_CODE_SIGN_IDENTITY, 286
XCODE_ATTRIBUTE_DEVELOPMENT_TEAM, 286
XCODE_ATTRIBUTE_IPHONEOS_DEPLOYMENT_TARGET, 283
XCODE_ATTRIBUTE_PROVISIONING_PROFILE_SPECIFIER, 287
XCODE_ATTRIBUTE_TARGETED_DEVICE_FAMILY, 284
XCODE_ATTRIBUTE_XXX, 272
XCODE_EMBED_FRAMEWORKS, 295
XCODE_EMBED_FRAMEWORKS_CODE_SIGN_ON_COPY, 295
XCODE_EMBED_FRAMEWORKS_PATH, 295
XCODE_EMBED_FRAMEWORKS_REMOVE_HEADERS_ON_COPY, 295
XCODE_LINK_BUILD_PHASE_MODE, 294
xcodebuild, 288

Z

ZERO_CHECK, 298

Exclusively for sbmain17@gmail.com Transaction: 0098315591