PROFESSIONAL CMAKE

A PRACTICAL GUIDE

CRAIG SCOTT

Professional CMake: A Practical Guide

Version: 1.0.3

© 2018 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

Table of Contents

Preface . . . 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. Manipulating Cache Variables . . . 26
     - 5.4.1. Setting Cache Values On The Command Line . . . 27
     - 5.4.2. CMake GUI Tools . . . 28
   - 5.5. Debugging Variables And Diagnostics . . . 31
   - 5.6. String Handling . . . 33
   - 5.7. Lists . . . 35
   - 5.8. Math . . . 37
   - 5.9. Recommended Practices . . . 37
6. Flow Control . . . 39
   - 6.1. The if() Command . . . 39
     - 6.1.1. Basic Expressions . . . 39
     - 6.1.2. Logic Operators . . . 41
     - 6.1.3. Comparison Tests . . . 41
     - 6.1.4. File System Tests . . . 43
     - 6.1.5. Existence Tests . . . 43

6.1.6. Common Examples . . . 45
6.2. Looping . . . 46
6.2.1. foreach() . . . 46
6.2.2. while() . . . 47
6.2.3. Interrupting Loops . . . 48
6.3. Recommended Practices . . . 49
7. Using Subdirectories . . . 50
7.1. add_subdirectory() . . . 50
7.1.1. Source And Binary Directory Variables . . . 51
7.1.2. Scope . . . 52
7.2. include() . . . 54
7.3. Ending Processing Early . . . 56
7.4. Recommended Practices . . . 57
8. Functions And Macros . . . 59
8.1. The Basics . . . 59
8.2. Argument Handling Essentials . . . 60
8.3. Keyword Arguments . . . 62
8.4. Scope . . . 65
8.5. Overriding Commands . . . 67
8.6. Recommended Practices . . . 68
9. Properties . . . 70
9.1. General Property Commands . . . 70
9.2. Global Properties . . . 73
9.3. Directory Properties . . . 74
9.4. Target Properties . . . 75
9.5. Source Properties . . . 75
9.6. Cache Variable Properties . . . 76
9.7. Other Property Types . . . 77
9.8. Recommended Practices . . . 77
10. Generator Expressions . . . 79
10.1. Simple Boolean Logic . . . 80
10.2. Target Details . . . 82
10.3. General Information . . . 83
10.4. Recommended Practices . . . 84
11. Modules . . . 86
11.1. Useful Development Aids . . . 87
11.2. Endianness . . . 89
11.3. Checking Existence And Support . . . 89
11.4. Other Modules . . . 94
11.5. Recommended Practices . . . 94
12. Policies . . . 96
12.1. Policy Control . . . 96
12.2. Policy Scope . . . 99
12.3. Recommended Practices . . . 100

Part II: Builds In Depth 102
13. Build Type 103
13.1. Build Type Basics 103
13.1.1. Single Configuration Generators 104
13.1.2. Multiple Configuration Generators 104
13.2. Common Errors 105
13.3. Custom Build Types 106
13.4. Recommended Practices 110
14. Compiler And Linker Essentials 111
14.1. Target Properties 111
14.1.1. Compiler Flags 111
14.1.2. Linker Flags 112
14.1.3. Target Property Commands 113
14.2. Directory Properties And Commands 116
14.3. Compiler And Linker Variables 118
14.4. Recommended Practices 122
15. Language Requirements 124
15.1. Setting The Language Standard Directly 124
15.2. Setting The Language Standard By Feature Requirements 126
15.2.1. Detection And Use Of Optional Language Features 128
15.3. Recommended Practices 130
16. Target Types 132
16.1. Executables 132
16.2. Libraries 133
16.3. Promoting Imported Targets 139
16.4. Recommended Practices 139
17. Custom Tasks 142
17.1. Custom Targets 142
17.2. Adding Build Steps To An Existing Target 144
17.3. Commands That Generate Files 146
17.4. Configure Time Tasks 149
17.5. Platform Independent Commands 151
17.6. Combining The Different Approaches 153
17.7. Recommended Practices 155
18. Working With Files 156
18.1. Manipulating Paths 156
18.2. Copying Files 159
18.3. Reading And Writing Files Directly 165
18.4. File System Manipulation 169
18.5. Downloading And Uploading 171
18.6. Recommended Practices 173
19. Specifying Version Details 175
19.1. Project Version 175

19.2. Source Code Access To Version Details . . . 177
19.3. Source Control Commits . . . 180
19.4. Recommended Practices . . . 183
20. Libraries . . . 185
20.1. Build Basics . . . 185
20.2. Linking Static Libraries . . . 186
20.3. Shared Library Versioning . . . 186
20.4. Interface Compatibility . . . 188
20.5. Symbol Visibility . . . 193
20.5.1. Specifying Default Visibility . . . 194
20.5.2. Specifying Individual Symbol Visibilities . . . 195
20.6. Mixing Static And Shared Libraries . . . 200
20.7. Recommended Practices . . . 203
21. Toolchains And Cross Compiling . . . 205
21.1. Toolchain Files . . . 206
21.2. Defining The Target System . . . 207
21.3. Tool Selection . . . 208
21.4. System Roots . . . 210
21.5. Compiler Checks . . . 211
21.6. Examples . . . 212
21.6.1. Raspberry Pi . . . 212
21.6.2. GCC With 32-bit Target On 64-bit Host . . . 212
21.6.3. Android . . . 213
21.7. Recommended Practices . . . 218
22. Apple Features . . . 219
22.1. CMake Generator Selection . . . 219
22.2. Application Bundles . . . 220
22.3. Frameworks . . . 225
22.4. Loadable Bundles . . . 228
22.5. Build Settings . . . 229
22.6. Code Signing . . . 231
22.7. Creating And Exporting Archives . . . 233
22.8. Limitations . . . 235
22.9. Recommended Practices . . . 236
Part III: The Bigger Picture . . . 239
23. Finding Things . . . 240
23.1. Finding Files And Paths . . . 240
23.1.1. Apple-specific Behavior . . . 244
23.1.2. Cross-compilation Controls . . . 244
23.2. Finding Paths . . . 246
23.3. Finding Programs . . . 246
23.4. Finding Libraries . . . 247
23.5. Finding Packages . . . 249

23.5.1. Package Registries . . . 255
23.5.2. FindPkgConfig . . . 257
23.6. Recommended Practices . . . 259
24. Testing . . . 263
24.1. Defining And Executing A Simple Test . . . 263
24.2. Pass / Fail Criteria And Other Result Types . . . 266
24.3. Test Grouping And Selection . . . 269
24.4. Parallel Execution . . . 273
24.5. Test Dependencies . . . 275
24.6. Cross-compiling And Emulators . . . 278
24.7. Build And Test Mode . . . 278
24.8. CDash Integration . . . 281
24.8.1. Key CDash Concepts . . . 281
24.8.2. Executing Pipelines And Actions . . . 282
24.8.3. CTest Configuration . . . 284
24.8.4. Test Measurements And Results . . . 288
24.9. GoogleTest . . . 289
24.10. Recommended Practices . . . 294
25. Installing . . . 297
25.1. Directory Layout . . . 298
25.1.1. Relative Layout . . . 298
25.1.2. Base Install Location . . . 300
25.2. Installing Targets . . . 302
25.2.1. Interface Properties . . . 307
25.2.2. RPATH . . . 308
25.2.3. Apple-specific Targets . . . 311
25.3. Installing Exports . . . 314
25.4. Installing Files And Directories . . . 317
25.5. Custom Install Logic . . . 320
25.6. Installing Dependencies . . . 321
25.7. Writing A Config Package File . . . 322
25.7.1. Config Files For CMake Projects . . . 323
25.7.2. Config Files For Non-CMake Projects . . . 330
25.8. Recommended Practices . . . 331
26. Packaging . . . 334
26.1. Packaging Basics . . . 334
26.2. Components . . . 339
26.3. Multi Configuration Packages . . . 343
26.4. Package Generators . . . 344
26.4.1. Simple Archives . . . 345
26.4.2. Qt Installer Framework (IFW) . . . 347
26.4.3. WIX . . . 352
26.4.4. NSIS . . . 353
26.4.5. DragNDrop . . . 355

26.4.6. productbuild . . . 356
26.4.7. RPM . . . 357
26.4.8. DEB . . . 362
26.4.9. FreeBSD . . . 363
26.4.10. Cygwin . . . 363
26.4.11. NuGet . . . 363
26.5. Recommended Practices . . . 363
27. External Content . . . 366
27.1. ExternalProject . . . 366
27.1.1. Tour Of Main Features . . . 367
27.1.2. Step Management . . . 373
27.1.3. Miscellaneous Features . . . 376
27.1.4. Common Issues . . . 378
27.2. FetchContent . . . 380
27.2.1. Developer Overrides . . . 384
27.2.2. Other Uses For FetchContent . . . 385
27.2.3. Restrictions . . . 386
27.3. ExternalData . . . 387
27.4. Recommended Practices . . . 388
28. Project Organization . . . 390
28.1. Superbuild Structure . . . 390
28.2. Non-superbuild Structure . . . 392
28.3. Common Top Level Subdirectories . . . 396
28.4. IDE Projects . . . 397
28.5. Defining Targets . . . 400
28.5.1. Target Sources . . . 401
28.5.2. Target Outputs . . . 404
28.5.3. Windows Specific Issues . . . 406
28.6. Miscellaneous Project Features . . . 407
28.7. Recommended Practices . . . 409
Index . . . 413

Preface

A few years ago, 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-half years and the result is the book you are reading now.

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 have been valuable contributors throughout the review process and I am truly grateful for their suggestions and insights. Thanks also to Nils Gladitz for his input, especially at such short notice. I'd also like to thank my past colleagues, Matt Bolger and Lachlan Hetherton, who both provided constructive feedback and reminded me of the importance of a fresh set of eyes.

A special mention is deserved for my colleague Mike Wake. Much of the material in this book was thrashed out and tested in real, actively developed 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 even just a few years ago, which brings it within reach of more potential writers and delivers benefits to the wider community. 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 and published. I am indeed a very fortunate human being.

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.

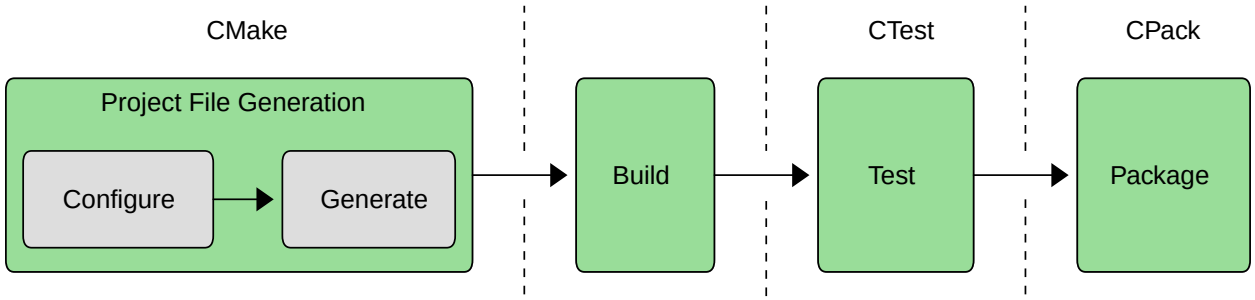
The initial focus in the first few chapters is on building a basic executable or library, covering just enough to give a new developer a quick introduction to CMake. Subsequent chapters expand that knowledge to demonstrate how to get the most out of what CMake has to offer. The techniques presented are 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.

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.

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 users mailing list is also a great source of advice and a recommended forum for asking CMake-related questions where the documentation doesn't provide sufficient guidance.

Chapter 2. Setting Up A Project

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

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

The binary directory is where everything produced by the build is created. It is often also called the build directory. For reasons that will become clear in later chapters, CMake generally uses the term *binary* directory, but among developers, the term build directory tends to be in more common use. This book tends to prefer the latter term since it is generally more intuitive. CMake, the chosen build tool (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.

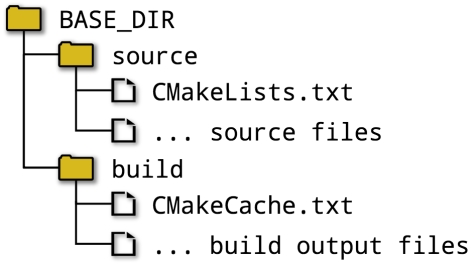
2.1. In-source Builds

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.

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 15 2017	Yes
	Visual Studio 14 2015	
	⋮	
Xcode	Xcode	Yes
Ninja	Ninja	No
Makefiles	Unix Makefiles	No
	MSYS Makefiles	
	MinGW Makefiles	
	NMake Makefiles	

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. For all generator types, CMake will carry out a series of tests and interrogate the system 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 re-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:

```
cmake --build /some/path/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 13, *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.

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 can be relatively common for them to need to support additional platforms and therefore different 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 would be to ensure the project builds with the default generator type on each platform of interest, plus one other type. The Ninja generator is an excellent choice for the latter, since it has the broadest platform support of all the generators and it also creates very efficient builds. If the project is being scripted, invoke the build tool via `cmake --build` instead of invoking the build tool directly. This allows the script to easily switch between generator types without having to be modified.

Chapter 3. A Minimal Project

All CMake projects start with a file called `CMakeLists.txt` 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

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

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 19, *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`, `Java` 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.

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. If the `CMakeLists.txt` file does not call `project()` or does not call it early enough, CMake will implicitly call it internally for the default languages `C` and `CXX` to ensure compilers and linkers are properly set up for other commands which rely on them. Later chapters give further details on setting up the toolchain and demonstrate how to query and modify things like compiler flags, compiler locations, etc.

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:

```
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.

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

EXCLUDE_FROM_ALL
: Sometimes, a project defines a number of targets, but by default only some of them should be built. When no target is specified at build time, the default ALL target is built (depending on the CMake generator 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 16, *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 22.3, “Frameworks”.

MODULE
: Specifies a library that is somewhat like a shared library, but is intended to be loaded dynamically at run-time rather than being linked directly to a library or executable. These are

typically plugins or optional components the user may choose to 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 16, *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.

`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 Chapter 16, *Target Types*).

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 16.2, "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, 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.

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 13, *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

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.

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.

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 with no error or warning, much like Unix shell scripts.

```
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 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 18, *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 normal 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 the same way as an ordinary 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])
```

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. Neither the nominal type nor the documentation string affect how CMake treats the variable, they are only used by GUI tools to present the variable to the user in a more suitable form. CMake will always treat the variable as a string during processing, the type is just to improve the user experience in GUI tools. 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.

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.). The FORCE keyword is discussed further below.

Setting a boolean cache variable is such a common need that CMake provides a separate command expressly for that purpose. 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 is equivalent to:

```
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.

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.

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}`, or put another way, normal variables take precedence over cache variables. The exception to this is that when setting a cache variable's value, if that cache variable did not exist before the call to `set()` or if the `FORCE` option was used, then any normal variable in the current scope is effectively removed. Note that this unfortunately means it is possible to get different behavior between the first and subsequent CMake runs, since in the first run, the cache variable won't exist, 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
```

Loosely speaking, the resultant behavior is that `${myVar}` will retrieve the last value that was assigned to `myVar`, regardless of whether it was a normal variable or a cache variable. The discussions in Chapter 7, *Using Subdirectories* and Chapter 8, *Functions And Macros* further clarify this behavior, explaining how a variable's scope can influence what value `${myVar}` would return.

5.4. 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.

5.4.1. 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 is given a special type that is similar to `INTERNAL`. 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.

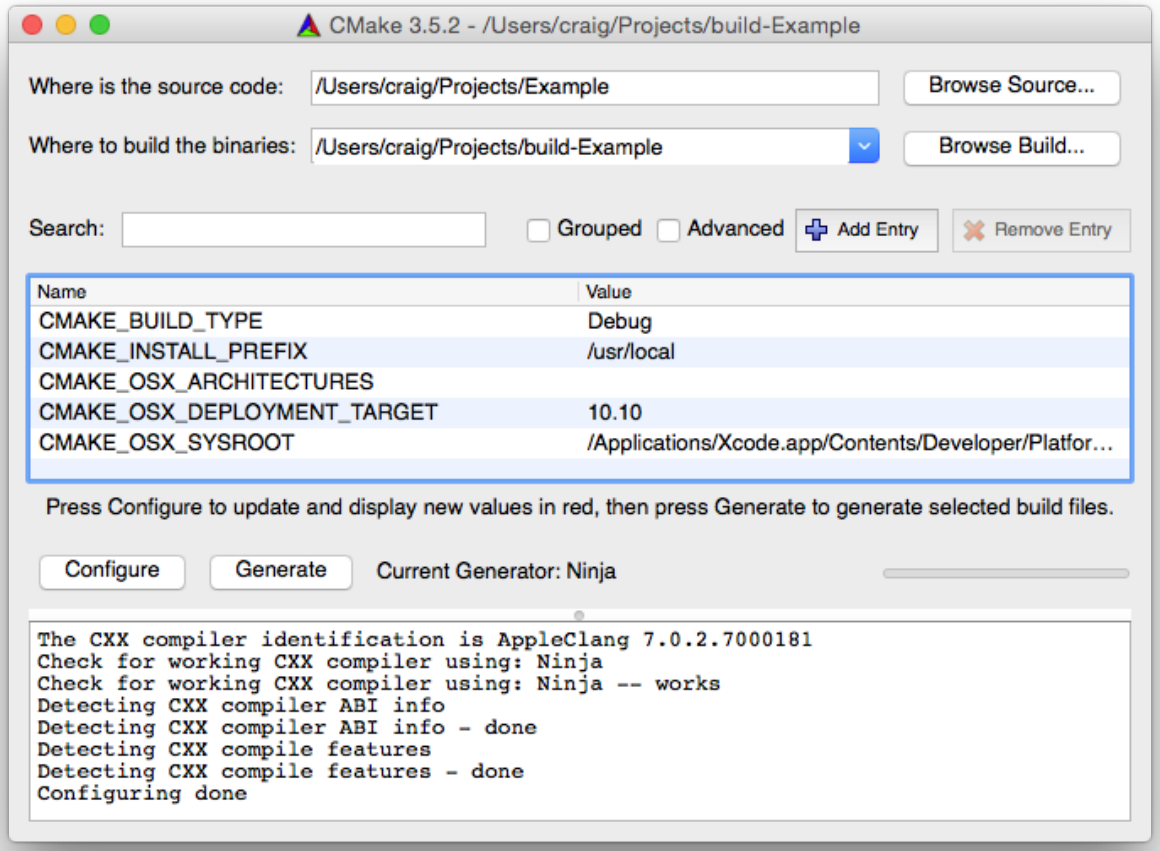
It is also possible to remove variables from the cache with the `-U` option, which can be repeated as necessary to remove more than one variable. Note that the `-U` option supports * and ? wildcards, but care needs to be taken to avoid deleting more than was intended and leaving the cache in an unbuildable 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 ...
```

5.4.2. 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. Both are included in the official CMake release packages on all platforms. 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, while at the bottom are the `Configure` and `Generate` buttons and their associated log area.



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. 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 of the cache variables will show a tooltip containing the `docstring` for that variable. New cache variables can also be added manually 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.

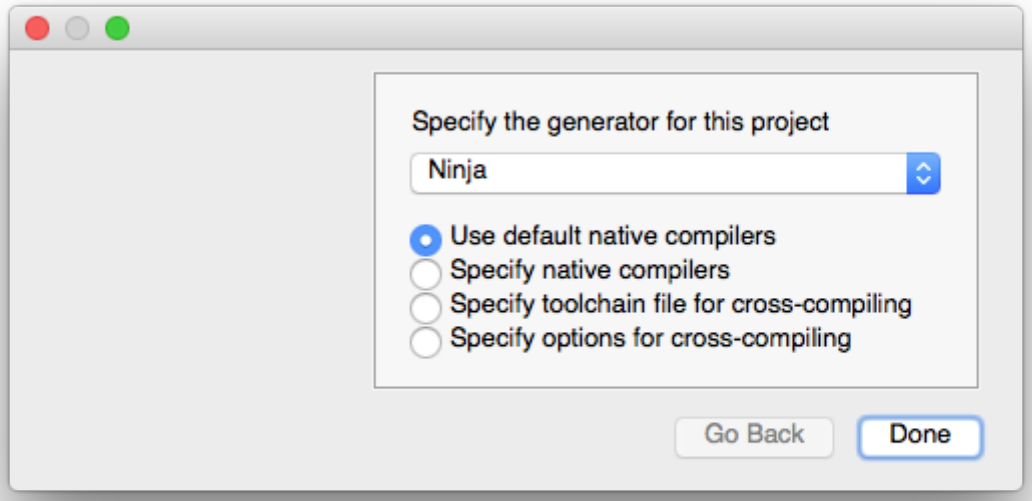
Cache variables can also have a property marking them as advanced or not. This too only affects the way the variable is displayed in `cmake-gui`, it does not in any way affect how CMake uses the variable during processing. By default, `cmake-gui` only shows non-advanced variables, which typically presents 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...])
```

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:

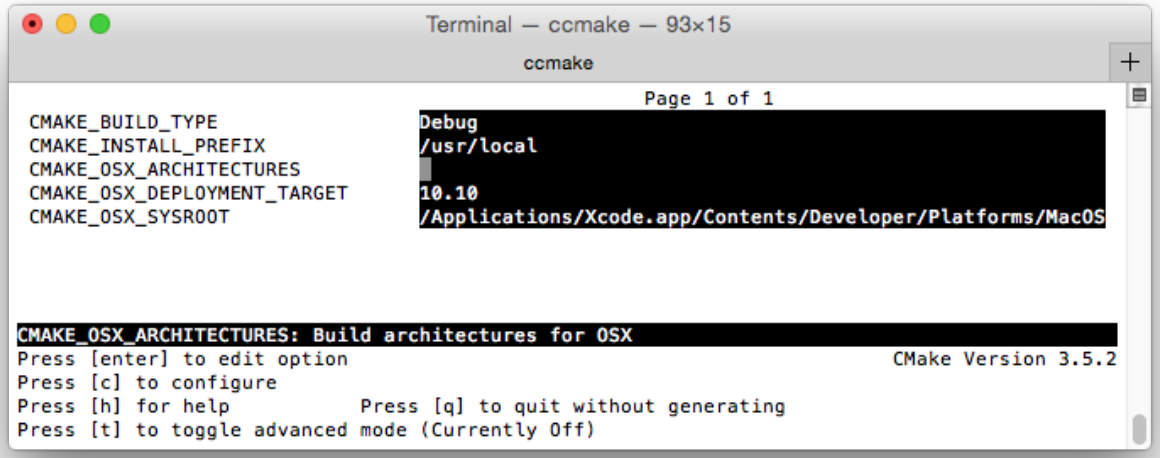


This dialog is where the CMake generator and toolchain are specified. The choice of generator is usually up to the developer's personal preference, with available options provided in the combobox. Depending on the project, the choice of generator may be more restricted than what the combobox options 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 21, *Toolchains And Cross*

Compiling. Lastly, for ultimate control, developers can specify the full set of options for cross-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 all 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.

One main drawback of the `ccmake` interface is that the log output is not captured as conveniently as with the `cmake-gui` version. The ability to filter the variables shown is also not provided and the methods for editing a variable are not as rich as with `cmake-gui`. Other than that, 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.

5.5. Debugging Variables And Diagnostics

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.

```
message([mode] msg1 [msg2]...)
```

If more than one `msg` is specified, they will be joined together into a single string without any separators. This is typically not what the developer intended, so the more common usage is a single `msg` with the message surrounded by quotes to preserve spaces. Variable values can be used and will be evaluated before printing the result. For example:

```
set(myVar HiThere)
message("The value of myVar = ${myVar}")
```

This will give the following output:

```
The value of myVar = HiThere
```

The `message()` command accepts an optional `mode` keyword which affects how the message is output and in some cases halts the build with an error. Recognized mode values are:

`STATUS`
: Incidental information. Messages will normally be preceded by two hyphens.

`WARNING`
: CMake warning, usually shown highlighted in red where supported (`cmake` command line console or the `cmake-gui` log area). Processing will continue.

`AUTHOR_WARNING`
: Like `WARNING`, but only shown if developer warnings are enabled (requires the `-Wdev` option on the `cmake` command line). Projects do not often use this particular type of message.

`SEND_ERROR`
: Indicates an error message which will be shown highlighted in red, where supported. Processing will continue until the configure stage completes, but generation will not be performed. This is like an error that allows further processing to be attempted, but ultimately still indicates a failure.

`FATAL_ERROR`
: Denotes a hard error. The message will be printed and processing will stop immediately. The log will also normally record the location of the fatal `message()` command.

`DEPRECATION`
: Special category used to log a deprecation message. If the `CMAKE_ERROR_DEPRECATED` variable is defined to a boolean true value, the message will be treated as an error. If `CMAKE_WARN_DEPRECATED` is defined to a boolean true, the message will be treated as a warning. If neither variable is defined, the message will not be shown.

If no `mode` keyword is provided, then the message is considered to be important information and is logged without any modification. It should be noted, however, that logging with a `STATUS` mode is not the same as logging a message with no mode keyword at all. When using a `STATUS` mode, the message will be printed correctly ordered with other CMake messages and will be preceded by two hyphens, whereas without any mode keyword, no leading hyphens are prepended and it is not unusual for the message to appear out of order relative to other messages which did include a `mode` keyword.

The other mechanism CMake provides for helping debug usage of variables 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, simply listing the variable to be watched without the optional `command` is sufficient, as it logs all accesses to the nominated variable. For ultimate control, however, 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 (see Chapter 8, *Functions And Macros*) and it will be passed the following arguments: variable name, type of access, the variable's value, the name of the current list file and the list file stack (list files are discussed in Chapter 7, *Using Subdirectories*). Specifying a command with `variable_watch()` would be very uncommon though.

5.6. 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 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.

```
set(longStr  abcdefabcdef)
set(shortBit def)

string(FIND ${longStr} ${shortBit} fwdIndex)
string(FIND ${longStr} ${shortBit} revIndex REVERSE)

message("fwdIndex = ${fwdIndex}, 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 [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:

```
string(REGEX MATCH    regex outVar input [input...])
string(REGEX MATCHALL regex outVar input [input...])
string(REGEX REPLACE  regex replaceWith outVar input [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 in CMake. The following examples demonstrate the required syntax:

```
set(longStr abcdefabcdef)

string(REGEX MATCHALL "[ace]" matchVar ${longStr})
string(REGEX REPLACE "([de])" "X\\1Y" replVar ${longStr})

message("matchVar = ${matchVar}")
message("replVar = ${replVar}")
```

The resultant output of the above is:

```
matchVar = a;c;e;a;c;e
replVar = 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)
```

CMake provides other operations, such as string comparison, hashing, timestamps 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.

5.7. Lists

Lists are used heavily in CMake. Ultimately, lists are just a single string with list items separated by semicolons, which 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
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
```

Appending and inserting items is also a common task:

```
list(APPEND listVar item [item...])
list(INSERT listVar index item [item...])
```

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

```
set(myList a b c)
list(APPEND myList d e f)
message("myList (first)  = ${myList}")

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

Which gives the following output:

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

Expected 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 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.

List items can also be reordered with `REVERSE` or `SORT` operations (sorting is alphabetical):

```
list(REVERSE myList)
list(SORT myList)
```

For all list operations which take an index as input, the index can be negative to indicate counting starts from the end of the list instead of the start. 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()` subcommands. Those mentioned are all supported since at least CMake 3.0, so projects should generally be able to expect them to be available. For the full list of supported subcommands, including those added in later CMake versions, the reader should consult the CMake documentation.

5.8. 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)
```

The first argument must by 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.

```
set(x 3)
set(y 7)
math(EXPR z "(${x}+${y}) / 2")
message("result = ${z}")
```

Expected output:

```
result = 5
```

5.9. 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 parse `CMakeLists.txt` files (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.

For log messages intended to remain as part of the build, aim to always specify a mode with the `message()` command. If the message is of a general informational nature, prefer to use `STATUS` rather than no mode keyword at all so that message output does not appear out of order in the build log. Temporary debugging messages frequently use no mode keyword for convenience, but if they are likely to remain part of the project for any length of time, it is better that they too specifying a mode (typically `STATUS`).

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.

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:

```
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 constant, the rules are as follows:

- If `value` is an unquoted constant with value 1, `ON`, `YES`, `TRUE`, `Y` or a non-zero number, it is treated as true. The test is case-insensitive.
- If `value` is an unquoted constant with value 0, `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.

- If neither of the above two 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 unquoted constants
if(YES)
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, so bypass the true/false constant matching
# and 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()
```

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

- A quoted string always evaluates to false in CMake 3.1 or later, regardless of the string's value (but this can be overridden with a policy setting, 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 matched a variable name would be replaced by the value of that variable. 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.

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. The following expressions are supported:

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

The above should largely be self-explanatory, but there are some points to be aware of. In particular, the `IS_NEWER_THAN` operator returns true if either file is missing or if both files have the same timestamp (which includes if the same file is given for both `file1` and `file2`). Thus, it would not be unusual to test for the existence of `file1` and `file2` before performing the actual `IS_NEWER_THAN` test, since the result of `IS_NEWER_THAN` where either file is missing will often not be what the developer intuitively expects. Full paths should also be given when using `IS_NEWER_THAN`, since the behavior for relative paths is not well defined.

The other point to note is that, unlike most other `if` expressions, none of the file system operators perform any variable/string substitution without `${}`, regardless of any quoting.

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. This can also be used to check if a specific environment variable is defined:

```
if(DEFINED SOMEVAR)         # Checks for a CMake 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 is most useful for checking whether something is defined before trying to use it. For CMake-provided commands, it would be better to test the CMake version, but for project-supplied functions and macros (see Chapter 8, *Functions And Macros*), testing for their existence with a `COMMAND` test may be useful.

`POLICY`
: Tests whether a particular policy is known to CMake. Policy names are usually of the form `CMPxxxx`, where the `xxxx` part is always 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 24, *Testing*).

One last existence test is available in CMake 3.5 and later:

```
if(value IN_LIST listVar)
```

This expression will return true if the variable `listVar` contains the specified `value`, where `value` follows the usual variable-or-string rules but `listVar` must be the name of a list variable.

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()
```

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()
```

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. One or both of `ITEMS` and/or `LISTS` 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:

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

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.

```
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)
            break()       ①
        endif()

        # Do no more processing if outer var is "b"
        if(outerVar STREQUAL "b")
            continue()    ②
        endif()
        message("Processing ${outerVar}-${innerVar}")
    endforeach()

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

① Ends the `innerVar` for loop early.

② Ends the current `innerVar` *iteration* and moves on to the next `innerVar` item.

The output from the above example would be:

```
Processing a-1
Processing a-2
Processing a-3
Accumulated list: a1;a2;a3
Accumulated list: b1;b2;b3
Processing c-1
Processing c-2
Processing c-3
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.

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.

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}")
```

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}")
```

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:

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*.

7.2. 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 a number of 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.

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:

```
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. 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. It's 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 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.

7.4. 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 28.5.1, “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. As a project evolves, it may start to use `include()` for some directories where this makes sense. 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. There are also efforts underway to remove some of the restrictions associated with calling some commands from different directories, so `add_subdirectory()` is likely to become more flexible and be the more preferred of the two methods.

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

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.

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.

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)
```

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 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 variable 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 `ARG#` where # is the number of the argument (e.g. `ARG1`, `ARG2`, etc.). This includes the named arguments, so the first named argument could also be referenced via `ARG1`, etc.

Typical situations where the `ARG…` variables 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 24, *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 remaining
# (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 the `ARGN` variable in particular. It allows a function or macro to take a variable number of arguments, yet still specify a set of named arguments which must be provided. 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.

8.3. Keyword Arguments

The previous section illustrated how the ARG… variables can be used to handle a variable set of arguments. That functionality is sufficient for the simple case where only one set of variable 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 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 the same flexibility by using the `cmake_parse_arguments()` command:

```
include(CMakeParseArguments)  # Needed only for CMake 3.4 and earlier
cmake_parse_arguments(prefix
                      noValueKeywords
                      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 above form ensures the command is available regardless of what CMake version is being used.

`cmake_parse_arguments()` takes the arguments supplied as the `argsToParse` parameter and processes them according to the specified sets of keywords. Typically, `argsToParse` is given as `${ARGN}`, which is the set of unnamed arguments passed to the enclosing function or macro. Each of the keyword arguments is a list of keyword names supported by that function or macro, so they should each be surrounded by quotes to ensure they are parsed correctly.

The `noValueKeywords` define standalone keyword arguments which act like boolean switches. The keyword being present specifies 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, however, that keywords should not be too long or they can be cumbersome to use.

When `cmake_parse_arguments()` returns, for every keyword, a corresponding variable will be available whose name consists of the specified `prefix`, an underscore and the keyword name. For example, with a prefix of `ARG`, the variable corresponding to a keyword named `FOO` would be `ARG_FOO`. If a particular keyword isn't present in the `argsToParse`, its corresponding variable will be empty. 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()

    foreach(arg IN LISTS singleValues multiValues)
        # Single argument values will print as a simple 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
```

Compared to basic argument handling using named arguments and/or 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.

8.4. Scope

A fundamental difference between functions and macros is that functions introduce a new variable scope, whereas macros do not. 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. 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 demonstrates how to implement the technique:

```
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}")
```

The output of the above would be:

```
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 7.3, "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()
```

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()
```

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.

8.6. 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.

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.

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.

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 propName [value1 [value2 [...]]])
```

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

```
GLOBAL
DIRECTORY [dir]
TARGET    [target1 [target2 [...]]]
SOURCE    [source1 [source2 [...]]]
INSTALL   [file1   [file2   [...]]]
TEST      [test1   [test2   [...]]]
CACHE     [var1    [var2    [...]]]
```

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 `PROPERTY` keyword marks all remaining arguments as defining the property name and its value(s). The `propName` would normally match one of the properties defined in the CMake documentation, a number of which are discussed in later chapters. The meaning of the value(s) are property specific. It is also permitted for a project to create new properties apart from those

already defined by CMake. It would be up to the project what such project-specific properties mean and how they might affect the build. If choosing to do this, it would be wise for projects to use some project-specific prefix on the property name to avoid potential name clashes with properties defined by CMake or other third party packages.

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 propName
             [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
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 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.

The `VARIABLE` type is a bit different, with the variable name being specified as the `propName` 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 `propName`. 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. This is a stronger test than `DEFINED` in that it tests whether the named property has actually been given a value (the value itself is irrelevant). 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 brief 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. This rarely used command has the following form:

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

Importantly, this command does not set the property's value, only its documentation and whether or not 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 `propName` 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 `propName`. 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, its parent directory scope's property 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.

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 25, *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. The setter command is defined as follows:

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

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.

The directory-specific getter command has two forms:

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

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 CMake 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 against it. Chapter 14, *Compiler And Linker Essentials* covers those commands and how they relate to target properties in depth.

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, such as indicating whether it 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.

Projects should keep in mind that 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. Also keep in mind that it is possible for one source file to be compiled into multiple targets, so any source properties that are set should make sense for all targets the source is added to.

Before rushing to start using source properties, developers should be aware of an implementation detail which may present a strong deterrent to their use in some situations. When using some CMake generators (notably the Unix Makefiles generator), the dependencies between sources and source properties are stronger than one might expect. In particular, 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 setup, where testing whether each individual source's compiler flags have changed brings with it a prohibitively big performance hit, so the CMake developers chose to implement the dependency at the target level instead. A typical scenario where projects may be tempted to go down this path is to pass version details to just one or two sources as compiler definitions, but as discussed in Section 19.2, "Source Code Access To Version Details", there are better alternatives to source properties which do not suffer from the same side effects.

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 in either the CMake GUI or ccmake.
- 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)
```

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 24, *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 14, *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. Projects should consider using other alternatives to source properties where available, such as the techniques given in Section 19.2, "Source Code Access To Version Details".

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 or Visual Studio. 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. Some kind of 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.

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 which enable this are the following:

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

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. The $<BOOL:···> expression can be used to convert anything CMake recognizes as a boolean false value into 0 and everything else to 1 (for details on what CMake considers a false value, see the discussion in Section 6.1.1, "Basic Expressions"). Together, these generator expressions provide a simple yet powerful way to optionally include content. 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>
```

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	Notes
$<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>	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>
)
```

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 or Visual Studio. This particular topic is covered in more detail in Section 13.2, “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. It’s value is different for different build configurations when using a multi configuration generator like Xcode or Visual Studio.

The above three `TARGET_FILE` expressions are especially useful when defining custom build rules for copying files around in post build steps (see Section 17.2, “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.

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. Because they are object files, they cannot be linked to as a single unit (although CMake 3.12 relaxes this restriction). They instead 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.

```
# 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.

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 and a couple of utility expressions are listed here as a starting point for further reading:

`$<CONFIG>`
: Evaluates to the build type. Use this in preference to the `CMAKE_BUILD_TYPE` variable since that variable is not used on multi configuration project generators like Xcode or Visual Studio. Earlier versions of CMake used the now deprecated `$<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_???:⋯>` generator expressions. For example, to produce the string `OLD_COMPILER` 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>:OLD_COMPILER>
```

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.

`$<LOWER_CASE:⋯>`, `$<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>
```

10.4. 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, which 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, 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 which 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. 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.

Chapter 11. Modules

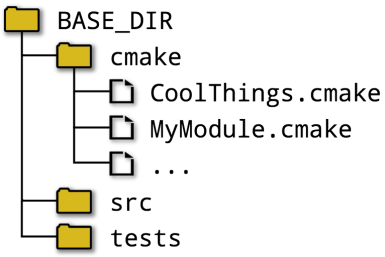
The preceding chapters have focused mostly on the core aspects of CMake. Variables, properties, flow control, generator expressions, functions, etc. are all part of what could be considered the CMake language. In contrast, modules are pre-built chunks of CMake code built on top of the core language features. 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` somwhere near the beginning of the top level `CMakeLists.txt` file. The following directory structure shows such an arrangement:



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_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 23.5, "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 23, *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. Useful Development Aids

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 macro 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.

```
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"
```

11.2. Endianness

When working with embedded platforms or projects intended for a wide variety of architectures, it can be desirable for the project to be aware of the endianness of the target system. The `TestBigEndian` module provides the `test_big_endian()` macro which compiles a small test program to determine the endianness of the target. This result is then cached so subsequent CMake runs do not have to redo the test. The macro takes only one argument, that being the name of a variable in which to store the boolean result (true means the system is big endian):

```
include(TestBigEndian)
test_big_endian(isBigEndian)
message("Is target system big endian: ${isBigEndian}")
```

11.3. 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 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. The names of these modules have the form `Check<LANG>SourceCompiles` and each provides an associated macro to perform the test:

```
include(CheckCSourceCompiles)
check_c_source_compiles(code resultVar [FAIL_REGEX regex])

include(CheckCXXSourceCompiles)
check_cxx_source_compiles(code resultVar [FAIL_REGEX regex])

include(CheckFortranSourceCompiles)
check_fortran_source_compiles(code resultVar [FAIL_REGEX regex] [SRC_EXT extension])
```

For each of the macros, the `code` argument is expected to be a string containing source code that should produce an executable for the selected 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, so 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 the `regex` regular expression, the check will be deemed to have failed, even if the code compiles and links successfully.

```
include(CheckCSourceCompiles)
check_c_source_compiles("
    int main(int argc, char* argv[])
    {
        int myVar;
        return 0;
    }" noWarnUnused FAIL_REGEX "[Ww]arn")
if(noWarnUnused)
    message("Unused variables do not generate warnings by default")
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.

A number of variables of the form `CMAKE_REQUIRED_…` can be set before calling any of the compilation test macros 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 14.3, "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 16, *Target Types*).

`CMAKE_REQUIRED_QUIET`

If this option is present, no status messages will be printed by the macro.

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 are covered in Section 21.5, "Compiler Checks".

In addition to checking whether code can be built, CMake also provides modules that test whether C or C++ code can be executed 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. The modules follow a similar structure to the compilation case, each providing a single macro implementing the check:

```
include(CheckCSourceRuns)
check_c_source_runs(code resultVar)

include(CheckCXXSourceRuns)
check_cxx_source_runs(code resultVar)
```

There is no FAIL_REGEX option for these macros, as success or failure is determined purely by the test process' exit code. 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_c_source_compiles() and check_cxx_source_compiles() also have the same effect for these two modules' macros as well.

For builds that are cross-compiling to a different target platform, the check_c_source_runs() and check_cxx_source_runs() macros 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 macros 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 macros 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 set of information for the check. These are typically just wrappers around the macros provided by one of the Check<LANG>SourceCompiles modules, 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.

Support for specific compiler flags can be checked using the Check<LANG>CompilerFlag modules, each of which provide a single macro with a name following a predictable pattern:

```
include(CheckCCompilerFlag)
check_c_compiler_flag(flag resultVar)

include(CheckCXXCompilerFlag)
check_cxx_compiler_flag(flag resultVar)

include(CheckFortranCompilerFlag)
check_fortran_compiler_flag(flag resultVar)
```

The flag-checking macros update the CMAKE_REQUIRED_DEFINITIONS variable internally to include flag in a call to the appropriate check_<LANG>_source_compiles() macro 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 macros assume that any flags already present in the relevant CMAKE_<LANG>_FLAGS variables (see Section 14.3, "Compiler And Linker Variables") do not themselves generate any compiler warnings. If they do, then the logic for each of these flag-testing macros will be defeated and the result of all such checks will be failure.

Two other notable modules are `CheckSymbolExists` and `CheckCXXSymbolExists`. The former provides a macro which builds a test C executable and the latter does the same as a C++ executable. Both 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 macros. 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. 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 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 following three macros:

```
cmake_push_check_state([RESET])
cmake_pop_check_state()
cmake_reset_check_state()
```

These macros 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() macro 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.

```
include(CheckSymbolExists)
include(CMakePushCheckState)

# Start with a known state we can modify and undo later
cmake_push_check_state()   # Could use RESET option, but needs CMake >= 3.10
cmake_reset_check_state()  # Separate call, safe for all CMake versions
set(CMAKE_REQUIRED_FLAGS -Wall)
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 all the 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()
```

11.4. 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 functions, macros, variables and properties. Many of these modules are provided as part of the support for `find_package()` calls (see Section 23.5, "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 flavor 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`, `FindPythonInterp`
- `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 27, *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 24, *Testing* and Chapter 26, *Packaging*.

11.5. 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 appending 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` appropriate 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. 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.

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]]])
```

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. 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.

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)
```

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

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.

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.

Chapter 13. 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.

13.1. 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.

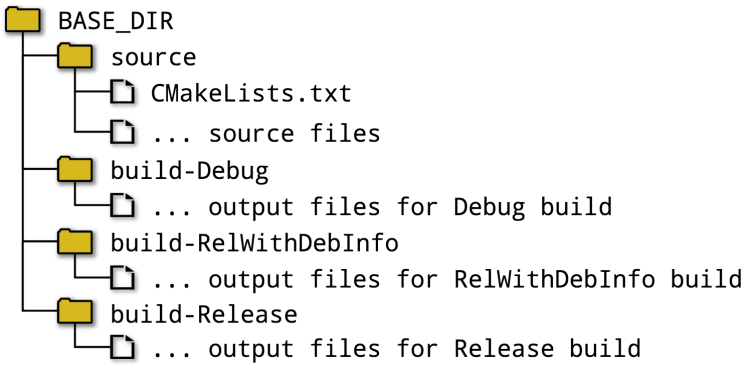
Each build type results in a different set of compiler and linker flags. It may also change other behaviors, such as altering which source files get compiled or what libraries to link to. These details are covered in the next few sections, but before launching into those discussions, it is essential to understand how to select the build type and how to avoid some common problems.

13.1.1. 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 has to be 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. Rather than switching between different build types, however, 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, with IDE environments like Qt Creator supporting switching between build directories just as easily as Xcode or Visual Studio allow switching between build schemes or configurations.

13.1.2. Multiple Configuration Generators

Some generators, notably Xcode and Visual Studio, support multiple configurations in a single build directory. These 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 look something like this:

```
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.

13.2. 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 or Visual Studio. 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:

```
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 is empty. This occasionally leads to a misunderstanding by some developers that an empty build type is equivalent to Debug, but this is not the case. 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 simply results in invoking the

compiler and linker with minimal flags and hence the behavior is determined by the compiler's and linker's own default behavior. While this may often be similar to the Debug build type's behavior, it is by no means guaranteed.

13.3. 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 multi configuration generators like Xcode and Visual Studio, the IDE environment 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 with a default list if it has not already been defined, but projects may modify the non-cache variable of the same name after that point (modifying the cache variable is unsafe since it may discard changes made by the developer). Custom build types can be defined by adding them to `CMAKE_CONFIGURATION_TYPES` and unwanted build types can be removed from that list.

Care needs to be taken, however, 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 technically 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 still recommended to use at least CMake 3.11 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 some of the ones 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)

if(CMAKE_CONFIGURATION_TYPES)
    if(NOT "Profile" IN_LIST CMAKE_CONFIGURATION_TYPES)
        list(APPEND CMAKE_CONFIGURATION_TYPES Profile)
    endif()
endif()

# Set relevant Profile-specific flag variables if not already set...
```

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)
```

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(allowableBuildTypes Debug Release Profile)

# WARNING: This logic is not sufficient
if(NOT CMAKE_BUILD_TYPE IN_LIST allowableBuildTypes)
    message(FATAL_ERROR "${CMAKE_BUILD_TYPE} is not a defined 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)

if(CMAKE_CONFIGURATION_TYPES)
    if(NOT "Profile" IN_LIST CMAKE_CONFIGURATION_TYPES)
        list(APPEND CMAKE_CONFIGURATION_TYPES Profile)
    endif()
else()
    set(allowableBuildTypes Debug Release Profile)
    set_property(CACHE CMAKE_BUILD_TYPE PROPERTY
                 STRINGS "${allowableBuildTypes}")
    if(NOT CMAKE_BUILD_TYPE)
        set(CMAKE_BUILD_TYPE Debug CACHE STRING "" FORCE)
    elseif(NOT CMAKE_BUILD_TYPE IN_LIST allowableBuildTypes)
        message(FATAL_ERROR "Invalid build type: ${CMAKE_BUILD_TYPE}")
    endif()
endif()

# Set relevant Profile-specific flag variables if not already set...
```

All of 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 and it also affects any generator expressions 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 "-p -g -O2" 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
    "${CMAKE_STATIC_LINKER_FLAGS_RELWITHDEBINFO} -p" CACHE STRING "")
set(CMAKE_MODULE_LINKER_FLAGS_PROFILE
    "${CMAKE_MODULE_LINKER_FLAGS_RELWITHDEBINFO} -p" CACHE STRING "")
```

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 an unoptimized debug build type, it can (and should) make this clear like so:

```
set_property(GLOBAL PROPERTY APPEND 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.

13.4. 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.

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 17, *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.

Chapter 14. 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 delving into the details, it is important to 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 carried across to any 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 generally be preferred instead, since they greatly improve the robustness of the build and offer much greater precision of control over compiler and linker behavior. The newer features also tend to be more consistent in their behavior and in the way they are meant to be used.

14.1. 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.

14.1.1. 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 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.

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, only 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 16, *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.

All of the above target properties also support generator expressions. This is particularly useful for the `COMPILE_OPTIONS` property, since it enables only adding a particular flag if some condition is met, such as only for a particular compiler. Another common use is to obtain a path related to some other target and use it as part of an include directory.

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.

14.1.2. Linker Flags

The target properties associated with linker flags have similarities to those for the compiler flags, but there are fewer properties involved:

`LINK_LIBRARIES`

This 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. 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.

`LINK_FLAGS`

This 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. Generator expressions are not documented as being supported. This property will be empty when a target is created.

`STATIC_LIBRARY_FLAGS`

This is the counterpart to `LINK_FLAGS`, applying only to targets being built as a static library. It will be used for the librarian or archiver tool.

Unlike the compiler properties, only `LINK_LIBRARIES` has an equivalent interface property, `INTERFACE_LINK_LIBRARIES`. There is no interface equivalent of `LINK_FLAGS` or `STATIC_LIBRARY_FLAGS`.

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.

14.1.3. Target Property Commands

The above target properties are not normally manipulated directly. CMake provides dedicated functions for modifying them in a more convenient and robust manner which also encourages

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. Only the non-INTERFACE_⋯ target properties are modified (i.e. LINK_LIBRARIES, LINK_FLAGS and STATIC_LIBRARY_FLAGS).

INTERFACE

This is the complement to PRIVATE, with items following the INTERFACE keyword only having an effect on targets that link to targetName. Only INTERFACE_⋯ target properties of targetName are modified (i.e. INTERFACE_LINK_LIBRARIES).

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 flags. In fact, they all follow the same pattern and apply the PRIVATE, PUBLIC and INTERFACE keywords in the same way.

```
target_include_directories(targetName [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.

Normally, each time target_include_directories() is called, the specified directories are appended to the relevant target properties. This makes it easy to add multiple paths in a natural, progressive manner. If required, the BEFORE keyword can be used to prepend the listed directories to existing contents of the target properties instead.

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 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 16, *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 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 25.2.1, "Interface Properties" covers this aspect of specifying header search paths in more detail.

```
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.

14.2. 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 ...)
```

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 `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(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. If a relative path is given, it will be treated as relative to the current source directory (very early versions of CMake had different behavior, see the documentation for policy CMP0015 for details). Normally, a full path or the name of a CMake target should be preferred since it is more robust. 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. For these reasons, adding linker search directories should be avoided where possible.

14.3. 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>`

In this family of variables, `<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`, while 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 21, *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:

`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.

Compiler/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.

```
# 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")
```

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 variable is left untouched, so every CMake run starts 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")
```

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. Replacing existing flags like this forces developers to go digging into the project files to find where and how to modify any lines which modify the relevant flags. For a complex project with many subdirectories, this can be quite tedious. Where possible, projects should instead 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 21, *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.

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 ...)
```

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. Because of this delayed nature of the compiler and linker variables, they can be 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 the opportunity for misuse and developer surprise.

14.4. 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, but 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. The following general guide may be useful:

- 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.

- 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 21, *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 20.5, "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. For all of the `target_⋯()` commands except `target_link_libraries()`, the specified target can be defined anywhere in the project, with the only requirement being that the target has been created at some point before the `target_⋯()` command is called. Thus, attaching additional compiler interface dependencies to a target can be done from any part of the project, but linker interface dependencies can only be done from the same directory scope in which a target is created. This restriction is being actively discussed by the CMake developers and may be removed in a future release.

Chapter 15. 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 are also supported.

15.1. 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.

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 (`<LANG>` must be one of `C` or `CXX`, with `CUDA` also an option for more recent CMake versions):

`<LANG>_STANDARD`

Specifies the language standard the project wants to use for the specified target. From the first CMake version supporting this feature, valid values for `C_STANDARD` are 90, 99 or 11, while for `CXX_STANDARD` the valid values are 98, 11 or 14. From CMake 3.8, the value 17 is also supported and from CMake 3.12, the value 20 can be used. One would reasonably presume that later CMake versions would add support for other language standards as they evolve over time. CMake 3.8 also supports `CUDA_STANDARD` with values of 98 or 11, which is a CUDA-specific version of what `CXX_STANDARD` would normally control. When a target is created, the initial value of this 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 and 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. For some compilers/linkers, this setting can change the standard library the target is linked with (see examples below). Be aware that for many compilers/linkers, the same flag is used to control both the language standard and whether or not extensions are enabled. One consequence of this is that if a project sets the <LANG>_EXTENSIONS property, it should also set the <LANG>_STANDARD property or else <LANG>_EXTENSIONS may effectively be ignored. When a target is created, the initial value of the <LANG>_EXTENSIONS property is taken from the CMAKE_<LANG>_EXTENSIONS variable.

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. Furthermore, it is strongly recommended that projects set all three properties/variables rather than just some of them. The defaults for <LANG>_STANDARD_REQUIRED and <LANG>_EXTENSIONS have proven to be relatively unintuitive for many developers, so by explicitly setting them, a project makes clear what standard behavior it expects. A few examples help demonstrate typical usage.

```
# 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. It may also add a linker flag like -stdlib=libc++ depending on the platform. 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. Note also that from Visual Studio 15 Update 3, the compiler supports specifying a C++ standard, but only for C++14 and later and C++14 is the default setting.

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 situation for C is very similar. The following example shows how to set the C standard details, this time only for a specific target:

```
# Build target foo with C99, no compiler extensions
set_target_properties(foo PROPERTIES
    C_STANDARD          99
    C_STANDARD_REQUIRED ON
    C_EXTENSIONS        OFF
)
```

It should be noted that `<LANG>_STANDARD` technically specifies a minimum standard, not necessarily an exact requirement. In some situations, CMake may select a more recent standard due to compile feature requirements (discussed next).

15.2. 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 added 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:

```
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.

15.2.1. 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 compiler 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 compiler 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:

```
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;
    ...
};
```

In addition to the `override` keyword, a number of other features can also have a similar conditionally defined symbol used in much the same way. C++ keywords like `final`, `constexpr`, `noexcept` and more can all potentially be used if available or omitted if not supported by the compiler and still produce valid and correct code. Other keywords such as `nullptr` and `static_assert` have alternative implementations which can be used if the keyword is not supported by the compiler. Specifying generator expressions for each feature to cover the supported and unsupported cases would be tedious and potentially more fragile, but a more convenient mechanism is provided by CMake through its module system. The `WriteCompilerDetectionHeader` module defines a function called `write_compiler_detection_header()` which automates such handling. It produces a header file that the project's sources can `#include` to pick up appropriately specified compiler defines. A simplified version of that function showing only the compulsory options can be described as follows.

```
write_compiler_detection_header(
    FILE      fileName
    PREFIX    prefix
    COMPILERS compiler1 [compiler2 ...]
    FEATURES  feature1 [feature2 ...]
)
```

The function will write out a C/C++ header to the specified `fileName`, the contents of which will have appropriate macros defined for each listed feature. Every feature will have a macro of the form `prefix_COMPILER_UPPERCASEFEATURE` whose value will be 1 or 0 depending on whether the feature is supported or not for the compiler being used. Some features may also have a macro of the form `prefix_UPPERCASEFEATURE` which provides the most appropriate implementation for that feature for each of the named compilers, including different versions of a compiler, where relevant.

This is best demonstrated by an example. Consider a C++ project which can make use of the `override`, `final` and `nullptr` keywords if available and which aims to support the GNU, Clang, Visual Studio and Intel compilers on any of the platforms those compilers support. The project will also define move constructors if the compiler supports rvalue references. The following would write out a single header called `foo_compiler_detection.h` in the build directory and start each macro name with the string `foo_`:

```
include(WriteCompilerDetectionHeader)
write_compiler_detection_header(
    FILE      foo_compiler_detection.h
    PREFIX    foo
    COMPILERS GNU Clang MSVC Intel
    FEATURES  cxx_override
              cxx_final
              cxx_nullptr
              cxx_rvalue_references
)
```

Example C++ code making use of the macros defined by the above might look like this:

```
#include "foo_compiler_detection.h"

class MyClass foo_FINAL : public Base
{
public:
#if foo_COMPILER_CXX_RVALUE_REFERENCES
    MyClass(MyClass&& c);
#endif
    void  func1() foo_OVERRIDE;
    void  func2(int* p = foo_NULLPTR);
};
```

The target consuming the above source file would still need to have the appropriate language standard selected, but in this case, since fall back implementations are available, the desired standard can be specified but not required:

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

add_library(foo MyClass.cpp)

# The header is written to the build directory
# so ensure we add that to the header search path
target_include_directories(foo
    PUBLIC "${CMAKE_CURRENT_BINARY_DIR}"
)
```

CMake provides fall back implementations for quite a few features, all of which are described in the `WriteCompilerDetectionHeader` module documentation. The `write_compiler_detection_header()` command also accepts a number of optional arguments not mentioned here which enable control over the structure and location of the generated header files and adding arbitrary content at the start and end of the generated header. The interested reader should consult the CMake module documentation for full details.

Projects should carefully consider whether the use of a compiler detection header is worth the complexity before diving in and making use of the `WriteCompilerDetectionHeader` module. It can be an excellent tool for expanding the range of compilers a project can support. In particular, long-lived projects may find it a useful stepping stone in updating their code base to more recent language features while they still need to support older compilers on some platforms. One of the main drawbacks to the use of the module is a potential reduction in source code readability. It may also be hard to enforce that all source files use the alternative (generated) symbol names instead of the standard language keywords, since it may not feel as natural for some developers.

15.3. 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. This can help avoid problems with linking inconsistent standard libraries and other linkage issues. 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 permitted. Omitting `CMAKE_<LANG>_STANDARD_REQUIRED` or `CMAKE_<LANG>_EXTENSIONS` can often 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 therefore 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 has a need for 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 25, *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 or not compiler extensions are allowed. These `⋯EXTENSIONS` properties/variables often only take effect if the corresponding `<LANG>_STANDARD` is also set due to how compilers and linkers frequently combine the two into a single flag, so ultimately it is difficult to escape having to specify `<LANG>_STANDARD` even when compile features are used. As a result, projects may still find it easier and more robust to prefer using the project-wide variables instead.

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. Working with very recent feature additions to a language, such as when using proposed features for an upcoming language release, is one scenario where compile features may be a useful approach where CMake supports such features. In general though, projects should prefer to use the variables or properties to set the language requirements at a higher level for better maintainability and robustness. Alternatively, setting the standard via a compile meta feature like `cxx_std_11` also avoids many of the problems of setting individual features. For the more recent language standards, CMake is moving away from defining individual features and is instead only providing the meta feature.

Projects can detect available compile features and provide implementations for whether a feature is available or not. CMake even provides some convenience macros through the `WriteCompilerDetectionHeader` module which make this task easier. Projects should generally only consider using these features as a transition path when updating an older code base to use newer language features, as they tend to feel less natural for developers and they can reduce code readability. One notable exception to this is projects intended for a wide range of compilers where language standard support can vary. For this scenario, optional support for specific language features may help reduce compiler warnings and catch coding errors when using more modern compilers. The benefits should be weighed against the increased complexity and potential reduction in readability and less natural style for most developers.

Chapter 16. 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.

16.1. 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 25, *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.

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 16.3, "Promoting Imported Targets" further below.

An ALIAS target is just a read-only way to refer to another target within CMake. It does not create a new build target with the alias name. Aliases can only point to real targets (i.e. an alias of an alias is not supported) and they cannot be installed or exported (both covered in Chapter 25, *Installing*). Prior to CMake 3.11, imported targets could not be aliased either, but CMake 3.11 relaxed some of the restrictions to allow aliasing imported targets, but only those imported targets that have global visibility.

16.2. Libraries

The add_library() command also has a number of different forms. The basic form introduced back in Chapter 4, *Building Simple Targets* can be used to define the usual types of libraries most developers are familiar with, or it can also be used to define object libraries which are just a collection of object files that are not combined into a single archive or shared library. The expanded basic form of the command is therefore:

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

Prior to CMake 3.12, 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 can't be linked, they therefore don't 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 introduces 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.

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 3.12, consider whether an ordinary static library is more appropriate and ultimately easier to use.

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 23,

Finding Things and Chapter 25, *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]
)
```

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 16.3, "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}
)
```

```
# Imported object library, Windows example shown
add_library(myObjLib OBJECT IMPORTED)
set_target_properties(myObjLib PROPERTIES
    IMPORTED_OBJECTS /some/path/obj1.obj    # These .obj files would be .o
                     /some/path/obj2.obj    # on most other platforms
)

# Regular executable target using imported object library.
# Platform differences are already handled by myObjLib.
add_executable(myExe $<TARGET_SOURCES:myObjLib>)
```

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 [IMPORTED [GLOBAL]])
```

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.

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:

```
# 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.

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.

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 only if using CMake 3.11 or later. `INTERFACE_⋯` properties can be set with `set_property()` or `set_target_properties()` 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 25, *Installing* covers much of the motivation and mechanics of the other combinations.

The last form of the `add_library()` command is for defining an alias library:

```
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, alias libraries 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.

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 alias library with a name of the form `projNamespace::originalTargetName`. 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)
```

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. The consuming project would then link against the namespaced names. For example:

```
# Pull in imported targets from an installed package.
# See details in Chapter 23: Finding Things
find_package(BagOfBeans REQUIRED)

# 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
)
```

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)
```

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 other projects consuming it as a pre-built package or child project.

16.3. 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. It should also be noted that once an imported target has been promoted to have global visibility, it is able to support the creation of an alias referring to it.

16.4. 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, as 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 packages built externally or they refer to tools from the file system that are found through Find modules. Developers should be comfortable with using imported targets, but understanding all the ins and outs of how they are defined is not usually necessary unless actually writing Find modules or manually creating config files for a package. Some specific cases are discussed in Chapter 25, *Installing* where developers may come up against certain limitations of imported targets, but such scenarios are not all that common.

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 (prior to CMake 3.12) and therefore don't support transitive dependencies. This forces projects to manually apply such dependencies themselves, which 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, but developers sometimes expect it to behave like a real library. The changes with 3.12 blur that distinction, but the remaining differences still leave room for unexpected results.

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 25.3, "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 alias 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 the library has been split in two, so define
# an interface library with the old name to effectively
# forward on the link dependency to the new libraries
add_library(computeAlgoA SHARED ...)
add_library(computeAlgoB SHARED ...)

add_library(deepCompute INTERFACE)
target_link_libraries(deepCompute INTERFACE
    computeAlgoA
    computeAlgoB
)
```

Chapter 17. 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.

17.1. 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]
                  [SOURCES source1 [source2...]]
)
```

A new target with the specified `targetName` will be available to the build. 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 ever built if it is explicitly requested or if building some other target that depends on it. The custom target is always considered out of date, so bringing any target that depends on it up to date will result in the commands being executed.

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.

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 a generator expression anywhere in the command or its arguments, a dependency is automatically created on that target too. If a dependency on any other target needs to be specified, the `add_dependencies()` command can be used 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 17.3, "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 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 17.3, “Commands That Generate Files”).

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 17.5, “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.

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.

17.2. 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]
)
```

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.

`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.

`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 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
)
```

17.3. 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]
                   [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.

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. That target 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:

- 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.

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 supported by most project generators. `IMPLICIT_DEPENDS` is ignored for all but Makefile generators, while the use of `DEPFILE` results in an error if anything other than the Ninja generator is used. `IMPLICIT_DEPENDS`

directs CMake to invoke a C or C++ scanner to determine dependencies of the listed files, while DEPFILE can be used to provide a Ninja-specific .d dependency file. Projects should generally try to avoid these two options due to the severely limited number of project generators that support them.

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.

17.4. 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, such as:

- Executing external commands to obtain information to be used during configuration.
- 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:

```
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]
)
```

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. If the `OUTPUT_STRIP_TRAILING_WHITESPACE` option is present, any trailing whitespace will be omitted from the content stored in the output variable, while 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 a variable, 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 and 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.

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.

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. The result of the command must still be captured using `RESULT_VARIABLE` and that variable must then be checked, as in the preceding example. If the command runs longer than the timeout threshold, the result variable will hold an error string indicating that the command was terminated due to timeout, which is why printing the result variable is recommended.

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.

17.5. 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.

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 the `cmake` command itself 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`
- `remove_directory`
- `rename`
- `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 "${CMAKE_COMMAND}" -E remove_directory "${discardDir}"
)
```

The platform specific implementation shows how projects typically try to implement a scenario such as this, 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 platform's command being used. The platform independent version, however, 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 ${CMAKE_COMMAND} -E remove_directory "${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
```

17.6. Combining The Different Approaches

The following example demonstrates many of the features introduced in this chapter. In particular, it shows how the different ways of specifying custom tasks can be used together to accomplish non-trivial things without having to resort to platform specific commands or functionality.

CMakeLists.txt

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

# Define an executable which generates various files in a
# directory passed as a command line argument
add_program(generateFiles generateFiles.cpp)

# Create a custom target which invokes the above executable
# after creating an empty output directory for it to populate,
# then invoke a script to archive that directory's contents
# and print the MD5 checksum of that archive
set(outDir "foo")
add_custom_target(archiver
    COMMAND ${CMAKE_COMMAND} -E echo "Archiving generated files"
    COMMAND ${CMAKE_COMMAND} -E remove_directory "${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}")
```

17.7. 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.

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 those features which do not have universal support across all platforms.

- 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 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.

Chapter 18. 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.

18.1. 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. The primary method for performing such operations is the `get_filename_component()` command, which has three different forms. The first form allows for the extraction of the different parts of a path or file name:

```
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:

`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.

EXT
: This is the complement to NAME_WE. It extracts just the extension part of the file name from the first "." onward.

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(baseName  ${input} NAME_WE)      # foo
get_filename_component(extension ${input} EXT)          # .bar.txt
```

The second form of get_filename_component() is used to obtain an absolute path:

```
get_filename_component(outVar input component [BASE_DIR dir] [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 dir instead of the current source directory (i.e. CMAKE_CURRENT_SOURCE_DIR). BASE_DIR will be ignored if input is already an absolute path.

component determines how symbolic links should be handled when computing the path to be 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)
```

The following example demonstrates its usage:

```
set(basePath   /base)
set(fooBarPath /base/foo/bar)
set(otherPath  /other/place)

file(RELATIVE_PATH fooBar ${basePath} ${fooBarPath})
file(RELATIVE_PATH other  ${basePath} ${otherPath})

# The variables now have the following values:
#   fooBar = foo/bar
#   other  = ../other/place
```

The third form of the `get_filename_component()` command is a convenience for extracting parts of a full command line:

```
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()`.

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, such as when working with custom commands and needing to pass a path to a script which requires native paths. For these situations, 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
```

18.2. Copying Files

The need to copy a file during the configure stage or during the build itself is a relatively common one. 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 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 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])
```

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` cannot simply be a directory to copy the file into, it must be a file name, optionally with 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.

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 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"
```

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 `${someVar}` form preserved without substitution. A classic example of this is where the file being copied is a Unix shell script where `${someVar}` is a valid and 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 with either the `${someVar}` or `@someVar@` form can also be performed on strings, not just files. The `string()` command has a `CONFIGURE` form which provides the same functionality:

```
string(CONFIGURE input outVar [@ONLY] [ESCAPE_QUOTES])
```

The options have the same meaning as they do for `configure_file()`. This form can be useful if the content to be copied requires more complex steps than just a simple substitution, an example of which is given in the next section.

Where no substitution is needed, another alternative is to use the `file()` command with either the `COPY` or `INSTALL` form, 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...]]
    [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:

```
file(COPY whoami.sh
    DESTINATION .
    FILE_PERMISSIONS OWNER_READ OWNER_WRITE OWNER_EXECUTE
                     GROUP_READ GROUP_EXECUTE
                     WORLD_READ WORLD_WRITE
)
```

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.

Both `COPY` and `INSTALL` also 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`. The directory structure of the source is preserved. Headers are given the same permissions as their source, whereas scripts are given owner read, write and execute permissions.

```
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
)
```

CMake offers a third option for copying files and directories. Whereas both `configure_file()` and `file()` are intended for use at configure time or possibly as part of a CMake script at install time, CMake's command mode can be 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 17.5, "Platform Independent Commands"). There are three commands related to copying, the first of which is used to copy individual files:

```
cmake -E copy file1 [file2...] destination
```

If only one source file is provided, then `destination` is interpreted 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. For this reason, it is more robust to always 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, however, 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.

Instead of copying individual files, command mode can also copy entire directories:

```
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
```

The result of this command 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.

18.3. 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 being 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 example would result in the file `hello.txt` containing a single line of text `Hello world from CMake`. Note that newlines are not automatically added, so the text from the `APPEND` line in the above example 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 lua-inspired 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"
]=])
```

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 or Visual Studio. Instead, the `file(GENERATE…)` command can be used:

```
file(GENERATE
    OUTPUT outFile
    INPUT inFile | CONTENT content
    [CONDITION expression]
)
```

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. All of the arguments 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 and having an `expression` which evaluates to 0 for those build types to be skipped and 1 for those to be generated.

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>".
]])
```

In the first case above, any generator expressions in the content of 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 combining bracket syntax with embedded content 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.

The other main 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 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 of 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` form may be more convenient. Instead of storing the entire file's contents as a single string, this form stores it as a list

with each line being one list item. The following reduced form shows the more commonly useful options:

```
file(STRINGS fileName outVar
    [LENGTH_MAXIMUM maxBytesPerLine]
    [LENGTH_MINIMUM minBytesPerLine]
    [LIMIT_INPUT maxReadBytes]
    [LIMIT_OUTPUT maxStoredBytes]
    [LIMIT_COUNT maxStoredLines]
    [REGEX regex]
)
```

Options not shown above relate to encoding, conversion of special file types or treatment of newline characters and would not be needed in most situations. Consult the CMake documentation for details on those areas.

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 particularly useful way to extract only specific lines of interest 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.

```
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`.

18.4. File System Manipulation

In addition to reading and writing files, CMake also supports other common file system operations.

```
file(RENAME source destination)
file(REMOVE files...)
file(REMOVE_RECURSE filesOrDirs...)
file(MAKE_DIRECTORY dirs...)
```

The `RENAME` form renames a file or directory, silently replacing the destination if it already exists. 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.

The `REMOVE` form 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` form will have no effect. To delete directories and all of their contents, use the `REMOVE_RECURSE` form instead.

The `MAKE_DIRECTORY` form will ensure the listed directories exist, creating intermediate paths as necessary and reporting no error if a directory already exists.

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 rename source destination
cmake -E remove [-f] files...
cmake -E remove_directory dir
cmake -E make_directory dirs...
```

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 changes the behavior when an attempt is made to remove a file that does not exist. 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.

CMake also supports listing the contents of one or more directories with either a recursive or non-recursive form of globbing:

```
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 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
)
```

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. A continuous integration system performing incremental builds is a prime candidate for being caught out by such use. The `CONFIGURE_DEPENDS` option added in CMake 3.12 tries to address this deficiency, but it comes with performance penalties and only works for some 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. 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 such problems. Section 28.5.1, “Target Sources” presents some alternative strategies which not only avoid these problems, they also encourage a more modular and self-contained directory structure.

18.5. Downloading And Uploading

The `file()` command has a number of other forms which carry out different tasks. A surprisingly powerful pair of subcommands provide the ability to download files from and upload files to a 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. 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. Both `DOWNLOAD` and `UPLOAD` share a number of common options:

`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.

The `DOWNLOAD` form also supports a few more options:

`EXPECTED_HASH ALGO=value`
: 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.

`TLS_VERIFY value`
: This option accepts a boolean value indicating whether to perform server certificate verification when downloading from 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.

`TLS_CAINFO fileName`
: A custom Certificate Authority file can be specified with this option. It only affects `https://` urls.

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.

`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.

18.6. 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 habits and patterns and applying them consistently throughout a project will help ensure new developers are exposed to better practices.

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 to it. Some IDE environments understand this convention and will still 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/copied before use, it also prevents it from being accidentally picked up instead of the destination if CMake or the compiler look for files in multiple directories. This is especially relevant when the destination file 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.
- `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.

Chapter 19. 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.

19.1. 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 of which 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 28.1, "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:

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.

19.2. 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:

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;
    return 0;
}
```

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}")
```

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;
}
```

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)

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, so it would generally be recommended to make these sort of version libraries static.

19.3. 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:

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 27.2, "FetchContent").

foobar_version.cpp.in

```
unsigned getFooBarCommitsSinceVersionChange()
{
    return @FooBar_COMMITS_SINCE_VERSION_CHANGE@;
}
// Other functions as before...
```

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 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.

19.4. 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 26, *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 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.

Chapter 20. 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.

20.1. 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 22.3, "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

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.

20.2. 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 certainly is.

20.3. 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.

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 in a fairly seamless manner. 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 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.

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, 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.

20.4. Interface Compatibility

The `VERSION` and `SOVERSION` target properties allow API versioning to be specified in a more or less platform independent manner at the operating system level. 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:

```
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, since the same constraints could effectively 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:

```
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 key difference can be exploited to allow targets to detect things like a minimum protocol version it could support or to work out the largest temporary buffer size needed among the different libraries it links to.

```
add_library(bigAndFast strategy1.cpp)
set_target_properties(bigAndFast PROPERTIES
    COMPATIBLE_INTERFACE_NUMBER_MIN PROTOCOL_VER
    COMPATIBLE_INTERFACE_NUMBER_MAX TMP_BUFFERS
    INTERFACE_PROTOCOL_VER 3
    INTERFACE_TMP_BUFFERS 200
)

add_library(smallAndSlow strategy2.cpp)
set_target_properties(smallAndSlow 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 bigAndFast smallAndSlow)
target_compile_definitions(myApp PRIVATE
    MIN_API=$<TARGET_PROPERTY:PROTOCOL_VER>
    TMP_BUFFERS=$<TARGET_PROPERTY:TMP_BUFFERS>
)
```

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.

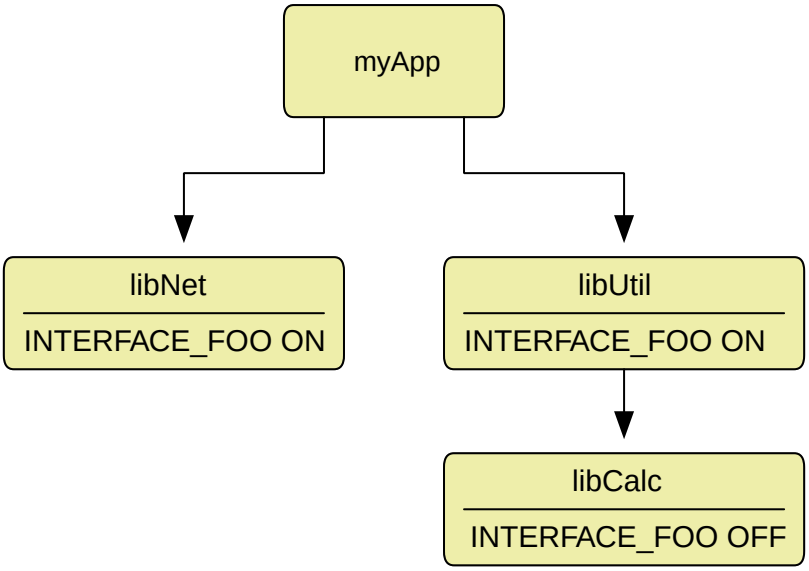
At this point, it would be natural to think about using the same property for both a minimum and maximum numeric interface compatibility to allow both the smallest and largest value to be detected in the parent. This is not possible because CMake does not (and cannot) allow the same property to be used 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(bigAndFast strategy1.cpp)
set_target_properties(bigAndFast 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(smallAndSlow strategy2.cpp)
set_target_properties(smallAndSlow 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 bigAndFast smallAndSlow)
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.

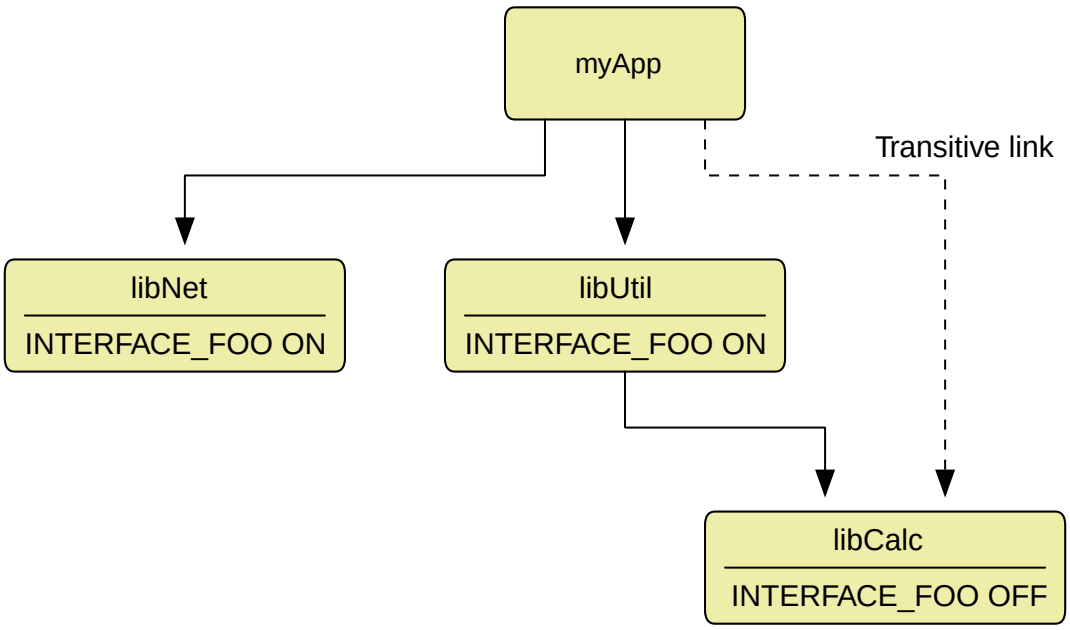
If one target defines an interface compatibility of any particular type, other targets are not required to define it too. Any target which does not define a matching interface compatibility is simply ignored for that particular property. This ensures libraries only need to define interface compatibilities 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.



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:



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.

20.5. 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.

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.

20.5.1. 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.

20.5.2. 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:

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 out 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;
```

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

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 "Leave out 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.

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.

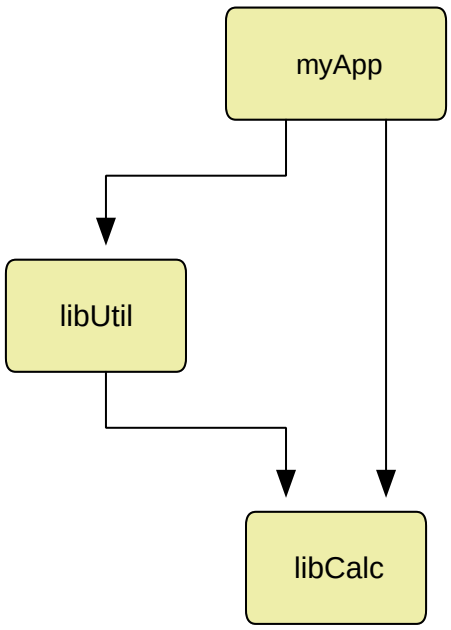
```
string(TIMESTAMP now)
set(customContents "/* Generated: ${now} */")
generate_export_header(myTools
    CUSTOM_CONTENT_FROM_VARIABLE customContents
)
```

20.6. 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.

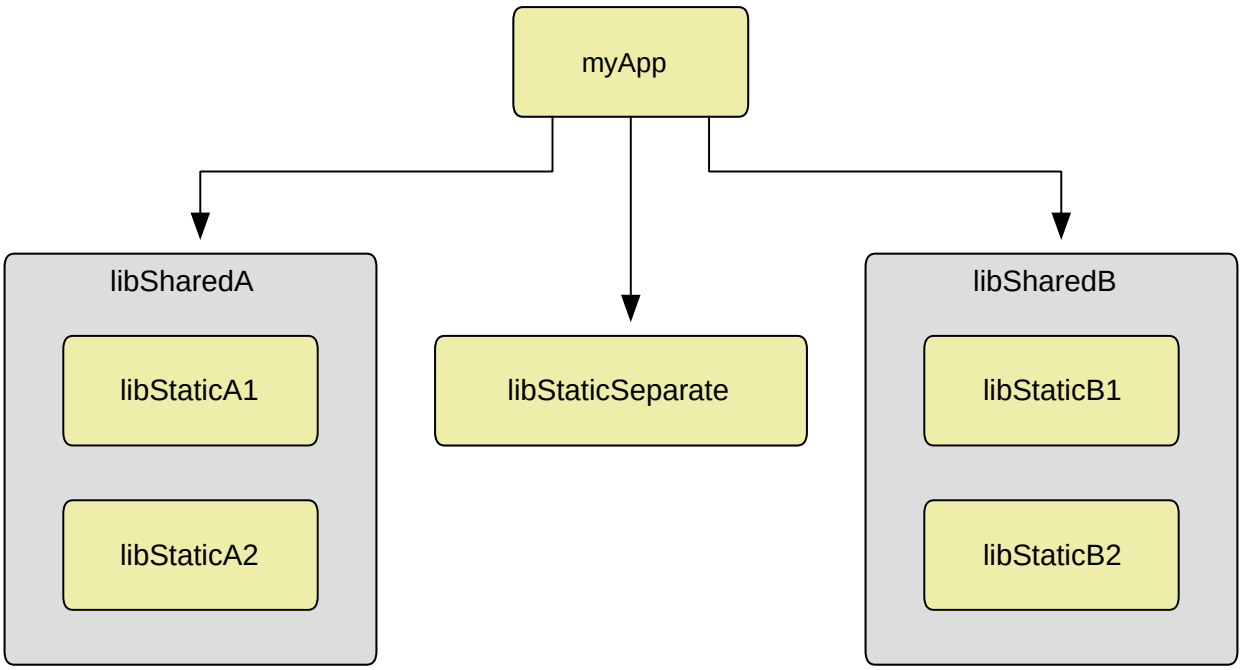
When library targets are defined as a mix of shared and static, the correctness of link dependencies becomes much more important. Consider the following set of targets:



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

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.

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
# 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`

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)
```

20.7. 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

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 28.5.1, "Target Sources" demonstrate how to add sources to an existing target progressively, allowing the target sources to be conveniently accumulated across subdirectories.

Chapter 21. 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, with an example for macOS looking something like the following (the C and CXX compiler paths shown have been collapsed for brevity):

```
-- The C compiler identification is AppleClang 9.0.0.9000039
-- The CXX compiler identification is AppleClang 9.0.0.9000039
-- Check for working C compiler: /Applications/Xcode.app/.../cc
-- Check for working C compiler: /Applications/Xcode.app/.../cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /Applications/Xcode.app/.../c++
-- Check for working CXX compiler: /Applications/Xcode.app/.../c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- 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 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 27.1, "ExternalProject" and Section 28.1, "Superbuild Structure").

21.1. 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
```

A full absolute path can be used, or for a relative path like in the above example, 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 every call to the `project()` command, not just the first one. This is normally a transparent implementation detail that the developer doesn't have to think much about, but it can lead to some subtle unexpected behavior. If the toolchain file sets or modifies variables that the project itself manipulates, or if the toolchain file incorrectly assumes it is only processed once for the whole project, then it may appear to the developer that `project()` commands are corrupting the toolchain settings or that variables are mysteriously changing without any obvious code making such changes. Developers should therefore ensure that toolchains are minimal, setting only the things they need to 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.

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 (typically 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 23, *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.

21.2. 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`, whereas `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 were able to be 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 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:

- All Apple platforms use Darwin for the CMAKE_SYSTEM_NAME, even for iOS, tvOS or watchOS. CMAKE_SYSTEM_PROCESSOR and CMAKE_SYSTEM_VERSION are not particularly meaningful for Apple platforms either and usually remain unset. Specifying the target system is done using a different variable, CMAKE_OSX_SYSROOT, which selects the base SDK to be used for the build. The target device is then 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 22.5, "Build Settings". There are also active discussions among the CMake developers around improving this area.
- The CMAKE_SYSTEM_PROCESSOR variable is typically not set when targeting Android platforms. This is discussed further in Section 21.6.3, "Android" below.

21.3. 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 23.3, "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.

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.

Once the compiler is known, CMake then identifies it and tries to determine its version. This compiler information is 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, Intel and so on. The CMake documentation for CMAKE_<LANG>_COMPILER_ID gives the full list of supported IDs. If the compiler version was able to 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:

```
add_library(foo ...)

# Conditionally add -fcode-hoisting option using variables
if(CXX_COMPILER_ID STREQUAL GNU AND
   NOT CXX_COMPILER_VERSION VERSION_LESS 7)
    target_compile_options(foo PRIVATE -fcode-hoisting)
endif()

# Same thing using generator expressions instead
target_compile_options(foo PRIVATE
    $<$<AND:$<CXX_COMPILER_ID:GNU>,
            $<VERSION_GREATER_EQUAL:$<CXX_COMPILER_VERSION>,7>>:-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 visible to the project as the `CMAKE_<LANG>_FLAGS`, `CMAKE_<LANG>_FLAGS_<CONFIG>`, `CMAKE_<TARGETTYPE>_LINKER_FLAGS` and `CMAKE_<TARGETTYPE>_LINKER_FLAGS_<CONFIG>` variables, which were covered back in Section 14.3, "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. Because the toolchain file is also re-read on every `project()` call, it can also discard 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})
```

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, meaning they expect a command line argument to specify the target platform/architecture to compile for. Clang and QNX qcc are examples of compilers that use this arrangement. For those 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.

21.4. 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 23.1.2, "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 25.1.2, "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 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 23.1.2, “Cross-compilation Controls” also discusses how CMAKE_STAGING_PREFIX affects the way CMake searches for libraries, headers and so on.

21.5. 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 24.6, “Cross-compiling And Emulators”.

21.6. 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.

21.6.1. 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, the most 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 25.1.2, "Base Install Location").

21.6.2. 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.

```
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")
```

21.6.3. Android

Cross-compiling for Android can be a bit more involved than the basic cases presented thus far and there are some differences in how the target system is described. `CMAKE_SYSTEM_NAME` must be set to `Android`, but `CMAKE_SYSTEM_PROCESSOR` is not typically set and the value of `CMAKE_SYSTEM_VERSION` is often left up to CMake to determine. Rather than setting paths to individual compilers and tools, a number of Android-specific variables control the toolchain configuration. The type of CMake generator used also affects the available options, with different generators supporting different development environments. For instance, when using a Visual Studio generator, CMake requires the Nvidia Nsight Tegra Visual Studio Edition to be installed. On the other hand, using Ninja or one of the Makefile generators allows the developer to choose between using the Android NDK or a standalone toolchain.

NDK And Standalone Toolchains

When Ninja or a Makefile generator is used, CMake uses a sequence of steps to determine whether it should use an NDK or a standalone toolchain. These steps are clearly detailed in the CMake toolchain documentation, but it can be helpful to break the steps down a little further (the first match is used):

Directly specify the development environment

- If the `CMAKE_ANDROID_NDK` variable is set, the NDK at that location will be used.
- If the `CMAKE_ANDROID_STANDALONE_TOOLCHAIN` variable is set, the standalone toolchain at that location will be used. This location must have a `sysroot` subdirectory.

Set `CMAKE_SYSROOT`

- If `CMAKE_SYSROOT` is set to a directory of the form `<ndk>/platforms/android-<api>/arch-<arch>`, then it will be as though `CMAKE_ANDROID_NDK` had been set to the `<ndk>` part of the path. The

default Android API level will be set to the <api> part of the path if not explicitly provided by the toolchain file (see below).

- If CMAKE_SYSROOT is set to a directory of the form <someDir>/sysroot, then it will be as though CMAKE_ANDROID_STANDALONE_TOOLCHAIN had been set to <someDir>.

Alternative CMake variables

- If ANDROID_NDK is set, it will be treated as though CMAKE_ANDROID_NDK had been set. New projects should prefer not to rely on this and should instead use the more canonical CMAKE_ANDROID_NDK variable directly.
- Analogously, if ANDROID_STANDALONE_TOOLCHAIN is set, it will be treated as though CMAKE_ANDROID_STANDALONE_TOOLCHAIN had been set. New projects should prefer not to rely on this and should instead use the more canonical CMAKE_ANDROID_STANDALONE_TOOLCHAIN variable directly.

Environment variables

- If either ANDROID_NDK_ROOT or ANDROID_NDK environment variables are set, they will be used as the value for the CMAKE_ANDROID_NDK CMake variable.
- If an ANDROID_STANDALONE_TOOLCHAIN environment variable is set, it will be used as the value for the CMAKE_ANDROID_STANDALONE_TOOLCHAIN CMake variable.

The NDK allows the developer a little more flexibility than a standalone toolchain. Whereas a standalone toolchain targets a single architecture and API level, the NDK may contain support for multiple toolchains and hence a range of architectures, API levels, etc. Note that the NDK roadmap shows standalone toolchains being obsoleted somewhere around the r19 release, pending removal of all but the Clang toolchain and a single STL implementation. The following is a selection of the more relevant variables for NDK and standalone toolchain arrangements:

CMAKE_SYSTEM_VERSION

When using the NDK, this can be set to the Android API level, or it can be left up to CMake to populate. When not set, CMake first checks if a CMAKE_ANDROID_API variable has been set and uses that if available. Otherwise, if CMAKE_SYSROOT has been set, CMake will try to detect the API level from the NDK directory structure. If that also fails, the latest API level supported by the NDK will be used. For a standalone toolchain, the value of CMAKE_SYSTEM_VERSION is always determined automatically from the toolchain.

CMAKE_ANDROID_ARCH_ABI

This variable specifies the Android ABI. For NDK builds, if it is not set, it will default to armeabi for NDK releases up to r16, or the oldest arm ABI available for later releases. CMAKE_ANDROID_ARCH_ABI can be given other values where the NDK has the necessary architecture support (e.g. `arm64-v8a, armeabi-v7a, armeabi-v6, mips, mips64, x86 or x86_64). This variable is set automatically when using a standalone toolchain. The value of CMAKE_ANDROID_ARCH will be derived from CMAKE_ANDROID_ARCH_ABI to provide the corresponding more general architecture value, which will be one of arm, arm64, mips, mips64, x86 or x86_64.

CMAKE_ANDROID_ARM_MODE

When CMAKE_ANDROID_ARCH_ABI is set to one of the armeabi* architectures, developers can choose between 32-bit ARM or 16-bit Thumb processors. If CMAKE_ANDROID_ARM_MODE is set to a boolean

true value, the ARM processor will be selected, otherwise if set to false or not set at all, Thumb will be the target processor. This can be set whether using the NDK or a standalone toolchain.

`CMAKE_ANDROID_ARM_NEON`

When `CMAKE_ANDROID_ARCH_ABI` is set to `armeabi-v7a`, `CMAKE_ANDROID_ARM_NEON` can be set to a boolean true value to enable NEON support. This can be set whether using the NDK or a standalone toolchain.

`CMAKE_ANDROID_NDK_TOOLCHAIN_VERSION`

This NDK-specific variable can be used to specify the toolchain to select from the NDK. If given, values must take one of the following forms:

- `X.Y` - GCC version X.Y
- `clangX.Y` - Clang version X.Y
- `clang` - Latest available Clang version

If this variable is not set, the latest GCC version available in the NDK will be used. Note that the NDK documentation (r16) states that GCC is no longer supported in the NDK and the NDK roadmap has scheduled GCC for complete removal in r18, so requesting a Clang toolchain is highly recommended.

`CMAKE_ANDROID_STL_TYPE`

Except when using a standalone toolchain, a variety of STL implementations can be selected by giving one of the supported values:

- `none`
- `system`
- `gabi++_static`
- `gabi++_shared`
- `gnustl_static`
- `gnustl_shared`
- `c++_static`
- `c++_shared`
- `stlport_static`
- `stlport_shared`

If not given, the default is `gnustl_static`. Note, however, that the GCC toolchain to which the `gnustl_*` STL implementations are closely tied will not be available as of NDK r18 and it only supports up to C++11 in older NDKs anyway. The `stlport_*` implementations are even older and more primitive and do not even support C++11. The `none` option has no support for C++ at all and the `system` option has only `new` and `delete` but no STL.

The NDK r16 documentation states that the `c++_static` and `c++_shared` STL types will be the only available types in a future NDK release, with the NDK roadmap showing this will occur in r18. It is therefore recommended that projects request one of the `c++_*` STL implementations (these are the LLVM C++ standard library implementations) and also use a Clang toolchain.

Each CMake target has its own `ANDROID_STL_TYPE` property and the `CMAKE_ANDROID_STL_TYPE` variable is used to provide the initial value of that property. In most cases, it will be desirable to use the same STL type throughout the build, so using the variable rather than setting individual target properties is likely to be simpler and more robust.

A minimal example of a toolchain file for a NDK build would look something like this:

```
set(CMAKE_SYSTEM_NAME Android)
set(CMAKE_ANDROID_NDK /path/to/android-ndk)
```

This would use the latest API level in the NDK with the latest GCC toolchain. It would target the `armeabi` architecture (Thumb processors) without neon support and would use the `gnustl_static` STL implementation. A more realistic example sets a few more of these quantities:

```
set(CMAKE_SYSTEM_NAME Android)
set(CMAKE_SYSTEM_VERSION 26) # API level
set(CMAKE_ANDROID_NDK /path/to/android-ndk)
set(CMAKE_ANDROID_ARCH_ABI arm64-v8a)
set(CMAKE_ANDROID_NDK_TOOLCHAIN_VERSION clang)
set(CMAKE_ANDROID_STL_TYPE c++_shared)
```

The above uses the latest Clang toolchain and a shared STL runtime with support for more recent C++ standards.

In comparison, a standalone toolchain file is typically going to be very simple, since many of the configuration decisions are predetermined by the toolchain itself:

```
set(CMAKE_SYSTEM_NAME Android)
set(CMAKE_ANDROID_STANDALONE_TOOLCHAIN /path/to/android-toolchain)
```

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, providing its own 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 determine if/how different CMake versions may be used and how to influence the behavior of the CMake build. This appears to be an active area of ongoing development in Android Studio.

For developers using `ndk-build` (which is essentially just a wrapper around GNU `make`) rather than gradle, CMake 3.7 introduced the ability to export an `Android.mk` file either as part of the CMake build using `export()` or as part of the install step with `install()`. The export during build form is very 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 25.3, “Installing Exports”).

Visual Studio Generators

When using one of the Visual Studio generators, CMake requires the Nvidia Nsight Tegra Visual Studio Edition to be installed. The resultant project will drive the whole build rather than forming part of a larger gradle or `ndk-build` structure. Support was first added in CMake 3.1, but many of the options were not added until CMake 3.4. The generator would typically be set on the CMake command line something like the following:

```
cmake -G "Visual Studio 12 2013 Tegra-Android" \
      -DCMAKE_TOOLCHAIN_FILE=/some/path/toolchain.cmake \
      /path/to/source
```

A minimal toolchain file would only need to set the `CMAKE_SYSTEM_NAME` to `Android`, but just like the NDK and standalone toolchain cases, further variables can be set to influence the target architecture, etc. In a number of cases, the variables to be set for Visual Studio builds are different to the NDK case, but are often related.

Whereas NDK and standalone toolchain builds would set `CMAKE_ANDROID_ARCH_ABI` and allow `CMAKE_ANDROID_ARCH` to be derived from it, toolchain files for Visual Studio builds set `CMAKE_ANDROID_ARCH` directly. Allowable values are also different for the Visual Studio case: `armv7-a`, `armv7-a-hard`, `arm64-v8a`, `x86` and `x86_64`.

Similarly, toolchain files for Visual Studio builds would set `CMAKE_ANDROID_API` rather than `CMAKE_SYSTEM_VERSION` to specify the Android API level of the target device, with `CMAKE_ANDROID_API` acting as a default value for the `ANDROID_API` target property. Furthermore, `CMAKE_ANDROID_API_MIN` can be set to specify the API version to be used to build the native code (it follows the same pattern and acts as the default value for the `ANDROID_API_MIN` target property). This is somewhat analogous to the situation for Apple platforms where the SDK used for the build can be specified separately to the minimum OS level of the target device (see Section 22.5, “Build Settings”).

The `CMAKE_ANDROID_STL_TYPE` variable can be set and accepts similar values to the NDK case, but the `c++_static` and `c++_shared` values are not supported. It is used as a default value for the `ANDROID_STL_TYPE` target property.

Since this arrangement drives the whole build, it has to set up more than just the native code built by CMake. There are a number of other target properties that relate to the parts of the build not associated with building the native code, such as settings for JAR dependencies, java sources, etc. Some of these target properties also have associated CMake variables that define defaults. These target properties all have names of the form `ANDROID_⋯` and the CMake default variables have the form `CMAKE_ANDROID_⋯`. These details are beyond the scope of the material covered here, so interested readers should consult the CMake documentation for details on the supported properties and variables, then set them as appropriate for the non-native parts of their project.

21.7. 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.

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 a simple toolchain file with the NDK and a Ninja or Makefile generator. This combination has the best CMake support and is the easiest to use. Toolchain files can be very simple and recent versions of IDE tools like Android Studio are moving to using this approach. Where developers are using their own toolchain files, avoid the popular *taka-no-me* toolchain file frequently referred to by online examples, since it is overly complicated and has known issues. The newer CMake versions support vastly simpler toolchain files which work smoothly with minimal effort.

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 built 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 23, *Finding Things* for recommended practices related to this area.

Chapter 22. 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 while CMake does formally support macOS and iOS, support for tvOS and watchOS should be considered incomplete.

22.1. 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 a fairly recent CMake release to keep up with the changes.

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 click on a build setting of interest. The *Quick Help* assistant editor shows the setting name along with a description. Defaults for all targets can be set by corresponding `CMAKE_XCODE_ATTRIBUTE_XXX` variables, which will be used to initialize the corresponding target property when the target is defined. An example which demonstrates setting some of the more commonly used attributes might look like this:

```
# Set the default signing identity and team ID to use for all targets
set(CMAKE_XCODE_ATTRIBUTE_CODE_SIGN_IDENTITY "iPhone Developer"
set(CMAKE_XCODE_ATTRIBUTE_DEVELOPMENT_TEAM   XYZ123ABCD)

# Some target-specific settings
set_target_properties(myiOSApp PROPERTIES
    XCODE_ATTRIBUTE_TARGETED_DEVICE_FAMILY     1,2
    XCODE_ATTRIBUTE_IPHONEOS_DEPLOYMENT_TARGET 10.0
)
```

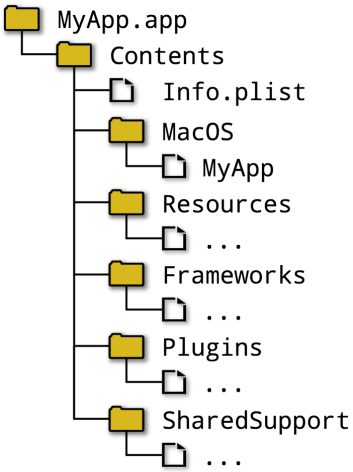
This feature can also be used to set an Xcode attribute for only one particular build type by appending `[variant=ConfigName]` to the property name. Other suffix types can be appended to the property name too for even more specific attribute settings, but their use would be unusual. Even `[variant=…]` 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(myiOSApp PROPERTIES
    XCODE_ATTRIBUTE_GCC_UNROLL_LOOPS[variant=Release] YES
    XCODE_ATTRIBUTE_ENABLE_TESTABILITY[variant=Debug] YES
)
```

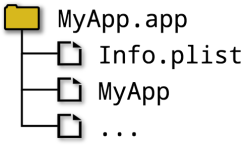
Some projects may require setting quite a few attributes in order to get the desired Xcode behavior and features, whereas other projects may be quite simple and require only a minimal number of additional settings. Some attributes are only needed in very specific circumstances, whereas others are so 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.

22.2. Application Bundles

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 very 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 ...)
```

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 arguably 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.

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`	`2.4.7rc1`
`MACOSX_BUNDLE_COPYRIGHT`	`NSHumanReadableCopyright`	`© 2018 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*

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 the *major.minor.patch* format now essentially mandated (with some exceptions). The following shows one potential mapping to provide version numbers that meet Apple's requirements:

```
add_executable(myApp MACOSX_BUNDLE ...)
set_target_properties(myApp PROPERTIES
    MACOSX_BUNDLE_BUNDLE_VERSION       "${PROJECT_VERSION}${BUILD_SUFFIX}"
    MACOSX_BUNDLE_SHORT_VERSION_STRING "${PROJECT_VERSION}"
)
```

In the above, BUILD_SUFFIX would be an empty string for final releases, or it could be one or more letters followed by a number in the range 1-255. Example suffixes might be a17 for an alpha release or rc2 for the second release candidate and so on. An example of an Info.plist file that uses these properties is included further below.

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 sources in add_executable() and target_sources() commands just like ordinary C/C++ files (see Section 28.5, "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.

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. For example:

```
set(uiFiles
    Base.lproj/Main.storyboard
    Base.lproj/LaunchScreen.storyboard
)

add_executable(MyApp MACOSX_BUNDLE
    AppDelegate.m
    AppDelegate.h
    ViewController.m
    ViewController.h
    main.m
    ${uiFiles}
)

set_target_properties(MyApp PROPERTIES
    RESOURCE "${uiFiles}"
    MACOSX_BUNDLE_INFO_PLIST "${CMAKE_CURRENT_SOURCE_DIR}/Info.plist"
)
```

Note the way the interface builder files are handled using the uiFiles variable. The value of this variable is used unquoted in the list of sources given to add_executable(). This makes the interface builder files appear as just another few items in the source file list. The RESOURCE target property, on

the other hand, holds a single value and that value is expected to be a semicolon-separated list. Therefore, the RESOURCE property requires the value of the uiFiles variable to be quoted, whereas the add_executable() call requires that it not be quoted.

In the above example, the Info.plist file would 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. A simple Info.plist for the above 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">
<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 above example, the NSMainStoryboardFile field has the value Main, which specifies that the Base.lproj/Main.storyboard UI will be used when the app starts. Note also 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 22.5, “Build

Settings" further below). Projects can instead hard-code a value directly in the `Info.plist` file, if that is more convenient.

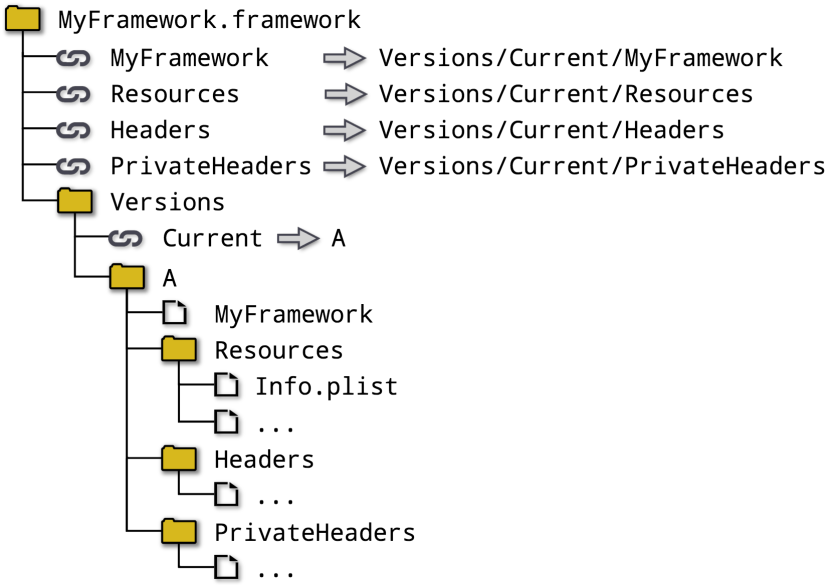
For files that need to be included in the application bundle but which are not resources in the usual sense, a different mechanism is available. Such files must still be listed as sources for the target, but instead of including them in the `RESOURCE` target property, each of the sources has its `MACOSX_PACKAGE_LOCATION` source property set to the location it should be copied to in the bundle. These paths are expected to be relative to the top of the bundle contents. This can be used to copy files to non-resource locations or to have full control over the target directory of resource files that do not need to be compiled. It is also possible to list a directory as a source and to set its `MACOSX_PACKAGE_LOCATION` to copy the directory and its contents into the bundle, but it is not documented whether this is formally supported by CMake (directories cannot normally be listed as sources). Some examples help demonstrate these behaviors.

```
add_executable(MyApp MACOSX_BUNDLE
    AppDelegate.m
    AppDelegate.h
    ViewController.m
    ViewController.h
    main.m
    sharedConfig.xml
    nestedResource.dat
    someDir   # Directory, CMake may not formally support this
)

set_source_files_properties(sharedConfig.xml PROPERTIES
    MACOSX_PACKAGE_LOCATION SharedSupport/config
)
set_source_files_properties(nestedResource.dat PROPERTIES
    MACOSX_PACKAGE_LOCATION Resources/private/other
)

# Works, but might not be formally supported
set_source_files_properties(someDir PROPERTIES
    MACOSX_PACKAGE_LOCATION Resources/lotsOfThings
)
```

A special case applies when setting the `MACOSX_PACKAGE_LOCATION` to a path starting with `Resources` and the target is being built for iOS. Because iOS app bundles 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.

22.3. Frameworks

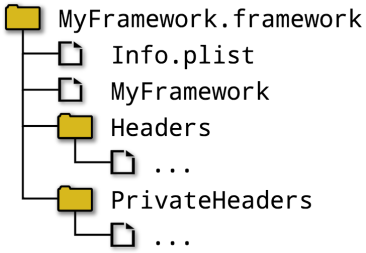
Frameworks share some similarities with application bundles, but they also have a number of unique features. A framework contains a main library, but unlike an application bundle, on macOS there may be multiple versions of the library. In addition to the usual resources, frameworks also support headers and in the case of macOS, both the resources and headers are version-specific. A typical example of the macOS framework structure looks something like this.



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. 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
)
```

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):

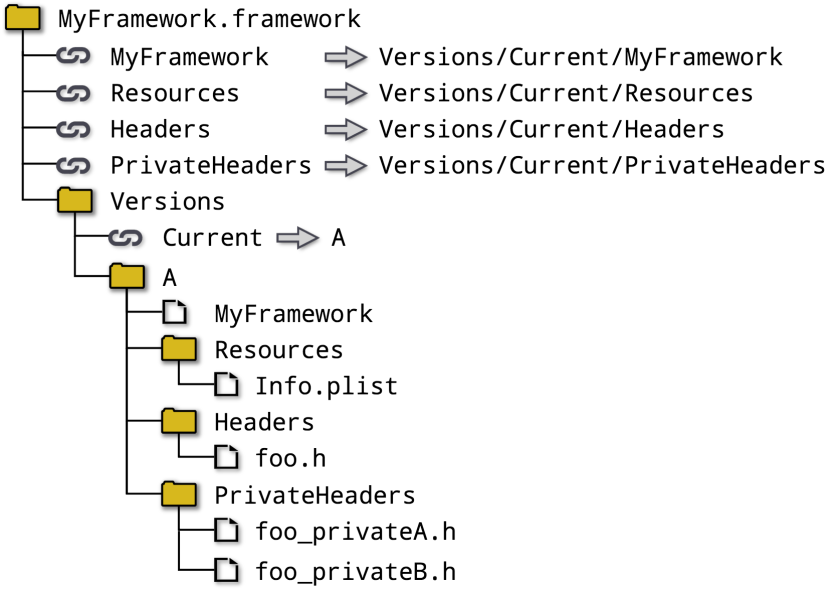
Property	Info.plist Key
`MACOSX_FRAMEWORK_BUNDLE_VERSION`	`CFBundleVersion`
`MACOSX_FRAMEWORK_ICON_FILE`	`CFBundleIconFile`
`MACOSX_FRAMEWORK_IDENTIFIER`	`CFBundleIdentifier`
`MACOSX_FRAMEWORK_SHORT_VERSION_STRING`	`CFBundleShortVersionString`

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.

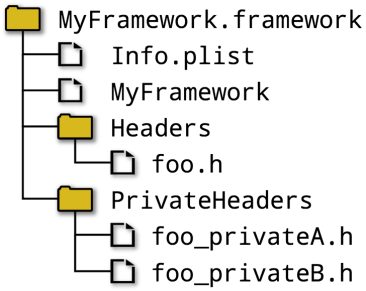
Frameworks often contain the headers associated with the framework's library. This allows the framework to be treated as a self-contained bundle which other software can build against. Framework headers are separated into public and private groups, with only public headers intended to be directly included 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. 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 the previous section for arbitrary resources.

```
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. This is covered in more detail in Section 25.2.3, “Apple-specific Targets”.

22.4. 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

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.

22.5. 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. Different versions of Xcode also come with different versions of the iOS SDK and some developers may even carry forward older SDKs to newer Xcode versions and switch between them. 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.

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 a very trivial step, as simple as setting the `CMAKE_OSX_SYSROOT` variable to one of `iphoneos`, `appletvos` or `watchos`. Xcode will then 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 will automatically populate the set of supported architectures based on the chosen SDK, so the project shouldn't need to add any extra logic for specifying architectures. This gives the developer the most control over what they want to build without having to re-run CMake. The available SDKs can be obtained by running the following command:

```
xcodebuild -showsdks
```

Due to how CMake performs its compiler tests, a couple more cache variables need to be set when targeting Apple platforms other than macOS. At least up to CMake 3.12, code signing can interfere with the compiler tests and these tests don't always use the correct target type (e.g. they don't try to create a bundle when they otherwise should). To address these problems, the `CMAKE_MACOSX_BUNDLE` and `CMAKE_XCODE_ATTRIBUTE_CODE_SIGNING_REQUIRED` variables also need to be set. In order for the

compiler tests to pick up the correct details, CMAKE_OSX_SYSROOT, CMAKE_MACOSX_BUNDLE and CMAKE_XCODE_ATTRIBUTE_CODE_SIGNING_REQUIRED need to be set very early in the configure phase. This is best done using a toolchain file, a minimal version of which would look like this:

```
set(CMAKE_MACOSX_BUNDLE YES)
set(CMAKE_XCODE_ATTRIBUTE_CODE_SIGNING_REQUIRED NO)
set(CMAKE_OSX_SYSROOT iphoneos)
```

By default, the project will have its deployment target set to the most recent one the SDK or host system supports. 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 has 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, the 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.

```
# Set the deployment target for macOS with any CMake version, or all Apple
# platforms when using CMake 3.11 or later
cmake_minimum_required(VERSION 3.9)

# Must be before first call to project()
set(CMAKE_OSX_DEPLOYMENT_TARGET 10.11)
project(AppleProject)
```

```
# Set the deployment target for iOS with any CMake version.

# Set defaults for all targets added hereafter within this directory scope or below
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)
```

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 app that supports only iPad
add_executable(MyiPadApp MACOSX_BUNDLE ...)
set_target_properties(MyiPadApp PROPERTIES
    XCODE_ATTRIBUTE_TARGETED_DEVICE_FAMILY 2
)

# An app that supports both iPhone and iPad
add_executable(RunEverywhereApp MACOSX_BUNDLE ...)
set_target_properties(RunEverywhereApp PROPERTIES
    XCODE_ATTRIBUTE_TARGETED_DEVICE_FAMILY 1,2
)
```

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.

22.6. Code Signing

Xcode functionality related to code signing has evolved considerably over the last few major Xcode releases. In recent times, 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 the automatic signing and provisioning works changed significantly, leaving many examples which demonstrate methods for Xcode 7 and earlier no longer reflecting best practice. This chapter focuses on the current automatic signing and provisioning process.

For automatic signing and provisioning to work, the app must have a valid bundle ID and two other key pieces of information need to be supplied: 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 defaults for the corresponding target properties. 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. As of Xcode 8, this should always be the string `Mac Developer` for macOS applications or `iPhone Developer` for iOS, tvOS or watchOS applications. These values will direct Xcode to select the most appropriate signing identity for the specified development team. In unusual circumstances, the signing identity can be set to a string which specifically 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 3.9)
project(macOSexample)

set(CMAKE_XCODE_ATTRIBUTE_DEVELOPMENT_TEAM   "ABC12345DE"    CACHE STRING "")
set(CMAKE_XCODE_ATTRIBUTE_CODE_SIGN_IDENTITY "Mac Developer" CACHE STRING "")
```

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 3.9)
project(iOSexample)

set(CMAKE_XCODE_ATTRIBUTE_DEVELOPMENT_TEAM   "ABC12345DE")
set(CMAKE_XCODE_ATTRIBUTE_CODE_SIGN_IDENTITY "iPhone Developer" CACHE STRING "")
```

When configured as described above, Xcode will automatically select an appropriate provisioning profile. If an appropriate profile doesn't exist, the Xcode IDE can automatically create one (this functionality is a feature of the IDE and is not available for command line builds). Such automatic provisioning is a significant improvement over earlier Xcode versions where provisioning profiles had to be created manually through the online developer portal.

In the previous section, `CMAKE_XCODE_ATTRIBUTE_CODE_SIGNING_REQUIRED` was set to `NO` in the toolchain file for iOS, but that variable is ignored when `CMAKE_XCODE_ATTRIBUTE_CODE_SIGN_IDENTITY` is set. One might be tempted to move the code signing details into the toolchain file to avoid having to set `CMAKE_XCODE_ATTRIBUTE_CODE_SIGN_IDENTITY` altogether, but note that this would mean 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.

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 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 directly if the default entitlements are not sufficient. The project must 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 the linking of frameworks 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>
```

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 through Xcode's *Embed Frameworks* build phase with the *Code sign on copy* option enabled, but unfortunately CMake does not directly support this (see Section 22.8, "Limitations" for a discussion of the restrictions on CMake's support for frameworks).

22.7. 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:

```
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 as an experimental feature. 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. 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 to be ready for distribution. This is achieved with another invocation of the `xcodebuild` tool, 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 as follows:

```
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:

- `app-store`
- `ad-hoc`
- `enterprise`
- `development`
- `developer-id`
- `package`

The default is `development`, but it is more likely that the main methods of interest will be `app-store`, `enterprise` or `ad-hoc`. When exporting an archive, the tool will re-sign the app and it will select an appropriate distribution signing identity based on the chosen method. The developer is expected to have already created/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.).

22.8. Limitations

Up to at least version 3.11, CMake's support for frameworks has a few shortcomings. Much of this results from the way frameworks (and even regular libraries) are incorporated into Xcode projects, where instead of defining an appropriate *Link Binary With Libraries* build phase, CMake hard-codes the linking directly into the *Other Linker Flags* target attribute. This matches how CMake links libraries and frameworks when using other generator types, but those generators do not have the additional framework-handling and code signing features that Xcode does. CMake does try to detect if a library it wants to link is a framework, using `-framework someLib` instead of `-lsomeLib` or `/path/to/someLib.dylib` on the linker command line for those it identifies as frameworks, but this does not make Xcode aware of the framework for anything other than linking.

For static frameworks, CMake's implementation still largely works, but for shared frameworks, there are problems. By embedding the details directly into the linker flags, Xcode isn't made fully aware of the framework and won't handle it properly when creating an application archive or performing code signing. In particular, the framework won't be installed along with the target linking to it, since there is no associated *Copy Files* build phase defined and it is during this copying that code signing of an embedded framework would normally be performed.

The choices available to projects are limited with the current CMake behavior. One could avoid using any non-system shared frameworks altogether, but this has obvious drawbacks. A project may require the developer to perform some manual project changes after running CMake to add

frameworks manually, but this is clearly fragile and precludes building in a headless environment, such as in a continuous integration system. A more viable path would be to define a script to modify the Xcode project file after CMake runs, or to define custom commands or post-build steps within the CMake project to simulate the things the Xcode project would normally do if it were aware of the embedded framework(s). None of these options are particularly satisfying and all of them go against the very nature of what CMake is meant to do on its own. Even the custom script or post-build steps approach is reasonably likely to conflict with improvements to CMake in the future where these shortcomings may eventually be addressed.

CMake's handling of entitlements is also 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, something which, as already discussed, is not handled well by CMake. 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, so most of the framework handling deficiencies are avoided.

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.

22.9. Recommended Practices

CMake is able to handle projects targeting Apple platforms, but the limitations need to be considered carefully. If applications must be signed, then the use of any non-system shared frameworks will require manual scripting and custom build steps to get the desired end result. If shared frameworks are not needed, then CMake's functionality should be sufficient and will generally automate the process without too much effort as long as the Xcode generator is used. Other generators such as Makefiles or Ninja are fine for building an unsigned macOS 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 now unnecessary or should be moved to the project itself. With Xcode 8 or later, projects should be aiming to make use of automatic signing and provisioning if at all possible, since this greatly simplifies the signing process. It also means a minimal toolchain file only needs to set `CMAKE_MACOSX_BUNDLE`, `CMAKE_XCODE_ATTRIBUTE_CODE_SIGNING_REQUIRED` and `CMAKE_OSX_SYSROOT` to get a working build that supports code signing and distribution. Other 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 is undesirable because it prevents 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. Therefore, projects should generally avoid setting `CMAKE_OSX_ARCHITECTURES` and instead let Xcode supply the standard set of architectures based on the selected SDK. The SDK is determined by `CMAKE_OSX_SYSROOT`, but importantly Xcode is able to recognize a matching simulator when a device SDK is chosen. By setting `CMAKE_OSX_SYSROOT` to something like `iphoneos`, for example, both the device and simulator builds will be available to the developer. Furthermore, 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.

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 forced by CMake. It is also sometimes used for scripted builds where it is known that only one specific platform should be built, but even then the architecture can be specified as a command line option instead, so there isn't much to be gained.

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 fat 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 the next chapter), 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 fat binary during the install step. This may be an alternative path if

using the Xcode generator is undesirable for some reason or if the project is structured to follow CMake's more platform-agnostic build-and-install model. See Section 25.2.3, "Apple-specific Targets" for a discussion of this topic.

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.

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.

The last chapter in this part of the book brings 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 chapters before it, it shows how to structure and define a project to be flexible, robust and easier for developers to work with.

Chapter 23. Finding Things

A project of at least modest size will quite likely rely on things provided by something outside of 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 file or a header for a library it uses. At a higher level, the project may want to find a complete package that potentially defines a whole range of things including targets, functions, variables and just about 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 even 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 27, *External Content* and preparing a project for being found by other projects is addressed in Section 25.7, "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.

23.1. 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"]
)
```

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 within a Find module (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 23.5, “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.

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 23.1.1, "Apple-specific Behavior" further below for additional details.

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.

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 systems, 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. This base path is then used to add one or two paths to the search locations. 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 the PATH environment variable was set to C:\foo\bin;D:\bar and CMAKE_LIBRARY_ARCHITECTURE set to somearch, the following set of search paths would be added in the order shown:

- C:\foo\include\somearch
- C:\foo\include
- C:\foo\bin
- D:\bar\include\somearch
- D:\bar\include
- D:\bar

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.

`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.

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. 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 cache 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 cache variable name that is self-documenting should make the need for explicit documentation unnecessary. By convention, these cache variables are usually all uppercase and use underscores to separate words.

23.1.1. 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 22.3, "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.

23.1.2. 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, so 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, but 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 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. The behavior specified by that variable 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 23.4, "Finding Libraries" further below.

23.2. 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.

23.3. Finding Programs

Finding programs is only slightly different to finding files, with the `find_program()` command taking exactly the same set of arguments as `find_file()`, plus one more optional argument, `NAMES_PER_DIR`. The short form of the command is also supported. 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` which provides similar control between app bundle and non-bundle paths. 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.
- `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.

23.4. Finding Libraries

Finding libraries is also similar to finding files, with the find_library() command supporting 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() and was also only added in CMake 3.4.
- 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 what 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 and DLLs on Windows can also 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, 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 and 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`, whereas 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 expanded set of search locations `/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

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.

23.5. 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.

When a module or config file is loaded, it typically defines 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. Packages can also define functions and macros, although only modules tend to do this. 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 config to understand what it provides. 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. The short form has only a few options:

```
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, but if the `EXACT` option is also given, then the package version must match exactly. A package may be optional, meaning the project can use it if available or work without it if the package cannot be found or is not of an appropriate version. Where a package is mandatory, the `REQUIRED` option should be provided to cause the command to halt with an error if the package could not be found or if the version requirements could not be met. Normally, `find_package()` will log messages if it is unable to find a package, but the `QUIET` option can be given to suppress them, which is particularly helpful for optional packages where the lack of the package should not result in warnings that may confuse the developer. `QUIET` also prevents the status messages that are normally printed when a package is found for the first time.

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 for a large package such as 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 to be found and the `Gui` component must be available. The `DBus` module, however, is optional.

```
find_package(Qt5 5.9 REQUIRED
    COMPONENTS Gui
    OPTIONAL_COMPONENTS DBus
)
```

When the `REQUIRED` option is present, the `COMPONENTS` keyword can be omitted and the mandatory components placed after `REQUIRED`. This is particularly common when there are no optional components. For example:

```
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.

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:

```
find_package(packageName
    [version [EXACT]]
    [QUIET | REQUIRED]
    [[COMPONENTS] component1 [component2...]]
    [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 the less common <lowercasePackageName>-config.cmake by default. The CONFIGS option can be used to specify a different set of file names to search for instead, but use of this option should be rare. Non-standard file names would require every project wanting to find that package to be aware of the non-standard 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)/

# 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 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 essentially the same way as they do 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 23.5.1, “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. 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`, `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.

One further control is 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.

23.5.1. Package Registries

Packages tend to be found either 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 the `HKEY_LOCAL_MACHINE` key:

```
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 adds the specified package to the user package registry and points it to the current binary directory associated with wherever `export()` was called. It is then up to the project to ensure that an appropriate config file 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 build slave. One way to prevent this is to set the `CMAKE_EXPORT_NO_PACKAGE_REGISTRY` variable to `ON`, which has the effect of disabling all calls to `export(PACKAGE)`. This prevents projects from adding their own build trees to the user package registry. Complementary to this, projects can set `CMAKE_FIND_PACKAGE_NO_PACKAGE_REGISTRY` or `CMAKE_FIND_PACKAGE_NO_SYSTEM_PACKAGE_REGISTRY` to `ON` to make all of their `find_package()` calls ignore the user and system package registries respectively.

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.

23.5.2. 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 import 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_EXECUTABLE` variable to the location of the tool. The `PKG_CONFIG_VERSION_STRING` is also set to the tool's version (except for CMake versions before 2.8.8).

In practice, projects should rarely need to use the `PKG_CONFIG_EXECUTABLE` variable, since the module also 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]
    [NO_CMAKE_PATH]
    [NO_CMAKE_ENVIRONMENT_PATH]
    moduleSpec1 [moduleSpec2...]
)

pkg_search_module(prefix
    [REQUIRED] [QUIET]
    [IMPORTED_TARGET]
    [NO_CMAKE_PATH]
    [NO_CMAKE_ENVIRONMENT_PATH]
    moduleSpec1 [moduleSpec2...]
)
```

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.

The functions expect one or more moduleSpec arguments to define what to search for. These combine the name of the package/module to find with any version requirements. The supported forms are:

- moduleName
- moduleName=version
- moduleName>=version
- moduleName<=version

Note the lack of simple > or < comparison operators, the only inequality operators supported are >= and <=. 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 just the relevant part of the package details:

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

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.

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 `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`.

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.

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.

23.6. 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.

All of the `find_…()` commands except `find_package()` work in a similar way, caching 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 where there are many such `find_…()` invocations throughout the project. There are, however, at least two consequences of this that 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 simply 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 to try to minimize the likelihood and number of unsuccessful searches.

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 of the above concerns will also apply to the package, since the logic is likely to be built upon the other `find_…()` commands. If, however, the package is 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 the location, the command proceeds with its normal search logic. This unique behavior for config mode is much more robust.

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).

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, since 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 are 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. If toolchain authors do not include such a block, at least the project still gets sensible defaults, but if a toolchain file changes the defaults to something else, at least 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. For everything else, including searching for config files, `CMAKE_PREFIX_PATH` is used.

Chapter 24. 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.

24.1. 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]
)
```

This command adds a new test called `testName` which runs the specified `command` with the given arguments. 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.

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 will substitute the real path to the executable automatically. This is particularly useful when using multi configuration generators like Xcode or Visual Studio where the location of the executable will be configuration-specific. The following shows a minimal example of a top level project that takes advantage of this behavior:

```
cmake_minimum_required(VERSION 3.0)
project(CTestExample)
enable_testing()

add_executable(testapp testapp.cpp)
add_test(NAME noArgs COMMAND testapp)
```

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`.

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.

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 and it would normally be run from the top of the build directory. When run with no command line arguments, it 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 or Visual Studio, `ctest` needs to be told which configuration it should test. This is done by including the `-C configType` option where `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.

It is possible to tell `ctest` to show all test output and various other details about the run with the `-V` option. `-VV` and `-VVV` show an increasing level of verbosity, but these are 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 tests that fail are of interest. `ctest` can be told to only show the output of failed tests by passing the `--output-on-failure` option. Alternatively, developers can set the `CTEST_OUTPUT_ON_FAILURE` environment variable to any value to avoid having to specify it every time (the value isn't used, `ctest` merely checks if `CTEST_OUTPUT_ON_FAILURE` has been set).

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"
)
```

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. A good example of this is when augmenting the `PATH` environment variable to ensure a test can find the shared libraries it links against on Windows:

```
# 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 fooTest
add_executable(fooTest ...)
target_link_libraries(fooTest PRIVATE algo)

add_test(NAME fooWithAlgo COMMAND fooTest)

if(WIN32)
    set_tests_properties(fooWithAlgo PROPERTIES ENVIRONMENT
        "PATH=$<SHELL_PATH:$<TARGET_FILE_DIR:algo>>$<SEMICOLON>$ENV{PATH}"
    )
endif()
```

Modifying the environment based on the actual environment being used to invoke `ctest` rather than CMake is more involved and is usually not strictly required. 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.

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 multi configuration generators like Xcode and Visual Studio, this target will be called `RUN_TESTS` and it will pass the currently selected build type as the configuration to `ctest`. For single configuration generators, the target is simply called `test` and it does not specify any configuration when invoking `ctest`. 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.

24.2. 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` 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(fooTest PROPERTIES
PASS_REGULAR_EXPRESSION
"Checking some condition for fooTest: passed
+.*
All checks passed"
FAIL_REGULAR_EXPRESSION "warning|Warning|WARNING"
)
```

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.

fooTest.cpp

```
int main(int argc, char* argv[])
{
    if (shouldSkip())
        return 2; // Skipped

    if (runTest())
        return 0; // Passed

    return 1; // Failed
}
```

CMakeLists.txt

```
add_executable(fooTest fooTest.cpp ...)
add_test(NAME foo COMMAND fooTest)

set_tests_properties(foo PROPERTIES
    SKIP_RETURN_CODE 2
)
```

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)
```

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 not considered a failure and is still counted in the total number of tests. A test may be skipped for other reasons which could be deemed a failure, such as a test dependency failing to be met (discussed in Section 24.5, “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 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.

Where possible, use of `TIMEOUT_AFTER_MATCH` should generally be avoided in favor of other ways to handle preconditions. Section 24.5, "Test Dependencies" and Section 24.4, "Parallel Execution" further below discuss better alternative methods.

24.3. 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. 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. `-R` 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
```

As each test is added, it is given a test number which 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
```

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:

```
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 24.4, "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.

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.

```
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.

24.4. 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 build slaves, since `CTEST_PARALLEL_LEVEL` can be set to the number of CPU cores on each slave, 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 example outputs, which use the same set of tests as defined earlier.

```
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
                                                                    continued...
```

```
100% tests passed, 0 tests failed out of 5

Label Time Summary:
bar      =   2.53 sec*proc (2 tests)
foo      =   1.65 sec*proc (3 tests)
multi    =   1.01 sec*proc (1 test)

Total Test time (real) =   2.03 sec
```

Five tests have been 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 the order in which they were started. Reducing the job limit to 2 shows output 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

100% tests passed, 0 tests failed out of 5

Label Time Summary:
bar      =   2.54 sec*proc (2 tests)
foo      =   1.65 sec*proc (3 tests)
multi    =   1.02 sec*proc (1 test)

Total Test time (real) =   2.66 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.

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. This is a fairly brutal constraint that can have a strong impact on test throughput, so it should be used sparingly. Quite often, a better alternative is the `RESOURCE_LOCK` test property, which is used to provide 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 that no other test which has 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")
```

The following sample output 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

100% tests passed, 0 tests failed out of 5

Label Time Summary:
bar      =   2.53 sec*proc (2 tests)
foo      =   1.65 sec*proc (3 tests)
multi    =   1.01 sec*proc (1 test)

Total Test time (real) =   3.67 sec
```

24.5. 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.

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
```

When using the `--rerun-failed` option, this same mechanism ensures that setup and cleanup tests are automatically added to the test set in order to satisfy the 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 (.). The following demonstrates various examples of these 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`

24.6. 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. If this property is set, CMake will prepend it before the target binary and use that as the command to run instead (i.e. the real target binary becomes the first argument to the emulator command provided by `CROSSCOMPILING_EMULATOR`). This enables tests to be run even when cross-compiling.

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 21.5, "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.

24.7. Build And Test Mode

`ctest` can be used to 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 has its own expected 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.

```
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.

```
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). 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.

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 generate_code COMMAND codegen)
add_test(NAME build_generated_code
    COMMAND ${CMAKE_CTEST_COMMAND}
            --build-and-test  ${CMAKE_CURRENT_LIST_DIR}/test_generation
                              ${CMAKE_CURRENT_BINARY_DIR}/test_generation
            --build-generator ${CMAKE_GENERATOR}
            --test-command    ${CMAKE_CTEST_COMMAND}
)

set_tests_properties(generate_code        PROPERTIES FIXTURES_SETUP    generator)
set_tests_properties(build_generated_code 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.

24.8. 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. 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

24.8.1. 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 *tracks*. 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*.

The track controls which group the pipeline results will be shown under in the dashboard results. Track names can be anything the project or developer wishes to use, but if no track 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 track that does this. The *Coverage* and *MemCheck* actions are a special case, they effectively ignore the track and their dashboard results are shown in their own dedicated groups (*Coverage* and *Dynamic Analysis* respectively).

24.8.2. 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] [--track Track] [otherOptions...]
```

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] [--track Track] [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 an attempt to set a variable instead. It may therefore be safer to use the `-M` and `-T` options rather than `-D`.

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
```

The same thing but with results reported under a different group called *Nightly Master* would be done like so:

```
ctest -M Nightly --track "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 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 --track "Simple Tests"
ctest -T Configure
ctest -T Build
ctest -T Test
ctest -T Submit

# One command
ctest -M Experimental --track "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 track 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 track 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). 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 and 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. The Jenkins plugin then only needs to be told where to find the `Test.xml` file and it is able to read the test results. When used this way, even the *Start* action can be omitted, since `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.

24.8.3. CTest Configuration

Preparing a project for CDash integration is mostly handled by a `CTest` module provided by CMake. This module should be included by the top level `CMakeLists.txt` file soon after the `project()` command.

```
cmake_minimum_required(VERSION 3.0)
project(CDashExample)

# ... set any variables to customize CTest behavior

include(CTest)

# ... Define targets and tests as usual
```

It is important that the `CTest` module is included by the top level `CMakeLists.txt` file, since it writes various files in the associated build directory and those generated files are generally expected to be at the top of the build tree. If the project is later incorporated into a parent project via `add_subdirectory()`, the parent project should also put `include(CTest)` in its top level `CMakeLists.txt` so that the necessary files are generated in the right location.

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()
```

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 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 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)
```

The `DartConfiguration.tcl` file written out by the `CTest` module contains certain configurable options for each of the dashboard actions. Most of these are already set to appropriate values by default, but the *Coverage* and *MemCheck* steps have options that may be of interest to developers. These are controlled by CMake variables which the developer can inspect and modify in the CMake cache or can be set directly 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.

The *MemCheck* step is more interesting. A number of different memory checkers are supported, including Valgrind, Purify, BoundsChecker and various sanitizers. For the first three, 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. 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 the project targets 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 or they may be read directly by the CI system, or possibly both.

To be able to get a complete pipeline captured, including the initial clone or update of an existing source tree and first configure step, one has to write a custom `ctest` script to establish all the required setup details and call the relevant `ctest` functions. This can be a much more involved process and isn't typically necessary when 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 using the `-S` or `-SP` option (they are the same except the latter creates a new process, whereas the former does not). The following demonstrates a fairly straightforward example.

```
ctest -S MyCustomCTestJob.cmake
```

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()
```

While the above custom script is fairly straightforward, the following more interesting example shows how custom scripts allow more flexible pipeline behavior to be defined. Rather than waiting to the very end of the run before submitting results to the dashboard, they 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.

```
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 TRACK 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)
```

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).

24.8.4. 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 be uploaded and attached to the build in the CDash results, with the upload being 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 only get uploaded if the test fails. This is a very useful way to record additional information about the failure to allow further investigation.

```
add_executable(doGen ...)
add_test(NAME generateFile COMMAND doGen)
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 with each test submission. 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:

```
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 the undocumented feature used heavily by projects such as Vtk and those built around it where measurements can be embedded in the test output itself 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` 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` value should be the full 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.

`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.

24.9. 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, 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 27.2, "FetchContent" and has few drawbacks (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`, in which case the list of sources is 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 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 `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 then applies different timeout limits to them. The names of the tests 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 when `ctest` is run instead of scanning the source code at CMake time. Because of this, CMake does not need to be re-run whenever the test source is changed, parameterized tests are handled robustly and there is no restriction on the formatting or way that the tests are defined. The only trade-off is that the list of tests is not available during the CMake run because it isn't obtained until actually running `ctest`.

```
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]
    [DISCOVERY_TIMEOUT seconds]  # See notes below
)
```

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.

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 particular 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 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 with the
# gtest_discover_tests() call, works with 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.

24.10. Recommended Practices

Aim to make the name of each test short, yet sufficiently specific to the nature of the test that it is 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 only serves to add 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. It may be difficult to keep test names unique among all projects, but rather than including a project-specific string in every test name, consider using the `LABELS` test property to include a project-specific label for each test. These per-project labels should allow the 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 may make the tests more convenient to work with.

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 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.

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 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. 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.

Chapter 25. 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.

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.

25.1. 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 22, *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`.

25.1.1. 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

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.

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`

`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.

25.1.2. 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:

- 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, explicitly setting `CMAKE_INSTALL_PREFIX` for non-Windows platforms to a FHS-compliant `/opt/⋯` path is advisable, but this should generally only be done in the top level `CMakeLists.txt` with 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 25.2.2, “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 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 somewhat 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.

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.

25.2. Installing 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:

```
install(TARGETS targets...
        [EXPORT exportName]
        [CONFIGURATIONS configs...]
        # One or more blocks of the following
        [ [entityType]
                DESTINATION dir
                [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...]
)
```

One or more `targets` are provided and then the `entityType` blocks specify how to handle installing the various parts of those targets. Each of the targets must be defined in the same directory scope as the `install()` command and 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 target's `RESOURCE` property of a framework or bundle target. On Apple platforms, such files are included as part of the `FRAMEWORK` or `BUNDLE` entity type 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}
)
```

The above example shows how the `DESTINATION` option can specify different locations for different parts of the same target. The command is also 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}
)
```

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 18.2, "Copying Files":

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, so 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. For instance:

```
# 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 20.3, "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. 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 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
)
```

25.2.1. Interface Properties

If the targets are exported (discussed in Section 25.3, “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 itself, 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 very common for the targets to all share the same header search path once installed. In the above example, CMAKE_INSTALL_INCLUDEDIR is used and is likely to be repeated for every installable target, but specifying it individually for each target 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 target listed. This leads to a more concise description of header search path details.

```
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.

25.2.2. 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).

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.

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, in which case 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 build problems during linking when using placeholders supported by the loader but not the linker (discussed further 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 the a similar layout to that defined by the default 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 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 in general that is not specific to CMake, it is simply 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. The `RPATH` value in the ELF headers are of fixed size, but 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 non-ELF platforms, 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 platforms 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.

25.2.3. 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`.

`@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 part of the `install_name_dir` or it can replace the `install_name_dir` completely.

The combination of `@loader_path` and `@rpath` can be used to achieve the same behavior as other Unix platforms that support `$ORIGIN`. 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.

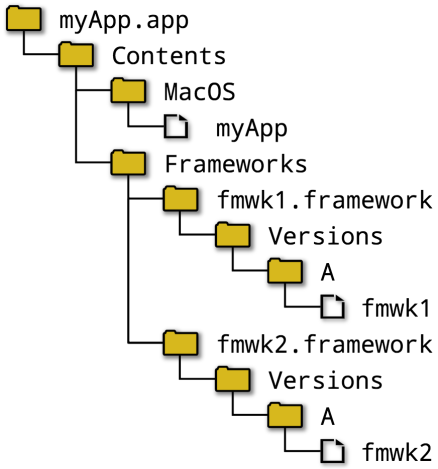
`INSTALL_NAME_DIR`

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.

For non-bundle layouts, the `$ORIGIN` behavior can be extended to cover the Apple case as well:

```
if(APPLE)
    set(basePoint @loader_path)
else()
    set(basePoint $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 ${basePoint} ${basePoint}/${relDir})
```

Once Apple bundles or frameworks are used, the Apple layout is completely different to other platforms and the above strategy is not useful. For such cases, there are different strategies 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 copying frameworks as post-build steps (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/../../... To support the post-build framework copy scenario as well, use the install RPATH details at build time. Omitting details for post-build framework copying and code signing, such an arrangement might look something like this:

```
set(CMAKE_BUILD_WITH_INSTALL_RPATH YES)
set(CMAKE_BUILD_WITH_INSTALL_NAME_DIR YES)

add_executable(myApp MACOSX_BUNDLE ...)
add_library(fmwk1 SHARED ...)
add_library(fmwk2 SHARED ...)
target_link_libraries(myApp PRIVATE fmwk1) # Only needs fmwk1 directly...
target_link_libraries(fmwk1 PRIVATE fmwk2) # ... but fmwk1 needs fmwk2

set_target_properties(myApp PROPERTIES
    INSTALL_RPATH @executable_path/../Frameworks
)
set_target_properties(fmwk1 fmwk2 PROPERTIES
    FRAMEWORK TRUE
    INSTALL_RPATH @loader_path/../../..
)
```

If the project's strategy is to only embed the framework at install time, the following is then sufficient:

```
install(TARGETS fmwk1 fmwk2 myApp
    BUNDLE    DESTINATION .
    FRAMEWORK DESTINATION myApp.app/Contents/Frameworks
)
```

On the other hand, if the project wishes to embed the framework at build time, a post-build step can be implemented relatively easily, as shown in the next example. Note, however, that the TARGET_BUNDLE_DIR and TARGET_BUNDLE_CONTENT_DIR generator expressions are only available in CMake 3.9 or later.

```
add_custom_command(TARGET myApp POST_BUILD
    COMMAND rsync -a
            $<TARGET_BUNDLE_DIR:fmwk1>
            $<TARGET_BUNDLE_DIR:fmwk2>
            $<TARGET_BUNDLE_CONTENT_DIR:myApp>/Frameworks/
)
```

The above copying step has robustness issues like not removing old contents, but for some situations it is good enough, or at least is a good starting point.

If the bundle needs to be signed, then embedding frameworks in general is not well supported by CMake. As highlighted back in Chapter 22, *Apple Features*, Apple assumes code signing is handled by Xcode as part of the build process rather than as some post install step, and CMake offers very

little assistance with the signing process. Projects currently have to implement their own logic if they wish to sign bundles with embedded frameworks.

Another complication arises if a project wishes to create universal binaries for iOS (sometimes also referred to as *fat* binaries). A build may be for the device or it may be for its simulator. Normally, an install only installs one architecture, but CMake 3.5 and later offers some assistance in the form of the `IOS_INSTALL_COMBINED` target property. If this property is true, then when the target is installed for a device build, it also builds the simulator architecture, installs it and combines the two into a single binary. The reverse is also true, such that installing a simulator build results in the device platform being built and installed as well. This feature still relies on the project implementing its own code signing logic, if relevant.

When it comes to embedding headers in frameworks, CMake provides a little more help. As outlined in Section 22.3, “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 ...
)
```

25.3. Installing Exports

When 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 25.7, "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:

```
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 16.4, "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.

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 above examples, 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. The situation is different for the multi component export where 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 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 25.7, “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. It’s 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.

25.4. 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
    [RENAME newName]
    [PERMISSIONS permissions...]
    [COMPONENT component]
    [EXCLUDE_FROM_ALL]
    [OPTIONAL]
    [CONFIGURATIONS configs...]
)
```

Most of the options are already familiar and have the same meaning as they do for `install(TARGETS)`. The only 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. The `RENAME` option can only be given if `files` is a single file. It allows that file to be given a different name when installed.

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:

```
install(DIRECTORY dirs...
        DESTINATION dir
        [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 many times as needed
        [ [PATTERN pattern | REGEX regex]
          [EXCLUDE]
          [PERMISSIONS permissions...] ]
)
```

In the absence of 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.

```
# 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 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 next example copies sample code and some scripts, overriding the permissions of the latter to ensure they are executable:

```
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 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 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. To create a single empty directory in the install area, the list of sources can be empty and the DESTINATION will still be created.

```
install(DIRECTORY DESTINATION somewhere/emptyDir)
```

25.5. Custom Install Logic

There can be situations where simply copying things into the install area isn't enough. There may be a need for arbitrary processing to be performed as part of the install, such as to rewrite parts of a file or to generate content programmatically. For these cases, CMake provides the ability to add custom logic to the install step.

```
install(SCRIPT fileName | CODE cmakeCode
        [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. Note that it is unspecified at what point in the installation process the custom code is invoked, but the current behavior is such that install() commands are generally processed in the order they appear in the directory scope (but this does not extend to install() calls nested within subdirectories).

Multiple SCRIPT and/or CODE blocks can be combined in the one command and they will be executed in the order specified. The COMPONENT and EXCLUDE_FROM_ALL options have their usual meanings but cannot be given more than once.

```
install(CODE      [[ message("Starting custom script") ]]
        SCRIPT    myCustomLogic.cmake
        CODE      [[ message("Finished custom script") ]]
        COMPONENT MyProj_Runtime
)
```

25.6. 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 modules which can potentially make this task easier.

The `InstallRequiredSystemLibraries` module is intended to provide projects with the details of relevant run time libraries for the major compilers. This coverage includes 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 itself. In the simplest case, projects can rely on the defaults, although setting at least the component for the `install()` commands is recommended.

```
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.

Another pair of modules can also be used to install a project's run time dependencies. The `BundleUtilities` and `GetPrerequisites` modules take a different approach, directly interrogating the installed binaries using platform-specific tools and recursively copying in missing libraries. These modules can be considerably more difficult to use and are not generally suitable for handling compiler run time dependencies. They can sometimes be effective in finding and installing dependencies that may not be all that predictable, such as for complex cross-platform toolkits like Qt (the `DeployQt4` module uses both modules extensively). Most projects will be better off spending the effort to work out their actual dependencies and install them directly to ensure the build process is more predictable and reliable, optionally using `InstallRequiredSystemLibraries` to take care of the compiler runtime dependencies.

25.7. 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 23.5, "Finding Packages". The name of the config file must match one of two forms:

- `<packageName>Config.cmake`
- `<lowercasePackageName>-config.cmake`

The first of the above forms 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 23.5, "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 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.

25.7.1. 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:

```
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:

MyProjConfig.cmake

```
include("${CMAKE_CURRENT_LIST_DIR}/MyProj_Runtime.cmake")
include("${CMAKE_CURRENT_LIST_DIR}/MyProj_Development.cmake")
```

CMakeLists.txt

```
# Define targets, etc...

# Create two separate export sets installed to the same place
# and a manually written config file that will include them
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
)
```

The above `MyProjConfig.cmake` is still very simple. No externally provided dependencies are needed and the config file assumes that both 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 sometimes be used as a wrapper around `find_package()` to handle the `QUIET` and `REQUIRED` keywords transparently. The `find_dependency()` macro also has the additional behavior that if it fails to find the requested package, processing of the config file stops immediately and control returns to the caller. It is 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")
```

Project authors should be aware that `find_dependency()` contains an optimization that bypasses the call if it detects that the requested package has already been found previously. This works fine unless later calls need to request a different set of package components. The first time `find_dependency()` succeeds, it effectively locks in the set of components found. If later calls to `find_dependency()` pass a different set of components, they are ignored. Therefore, if the dependency supports package components, projects should instead call `find_package()` directly and handle the `QUIET` and `REQUIRED` options themselves. These 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})
```

If the project wishes to support some of its own components being optional, then the complexity of the config file increases fairly significantly. The steps involved to fully support such functionality can be summarized as follows:

- Build up the set of project components that need to be found. Start with the set of required and optional components from the find_package() call and add any that are needed to satisfy project dependencies.
- Work out the set of external dependencies needed by that set of project 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 then 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 …), then the package could never be treated as optional by the caller.

```
# 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 specified
    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 only needed by the Development component.
set(${CMAKE_FIND_PACKAGE_NAME}_comps ${comps})
if(Development IN_LIST ${comps})
    find_dependency(BagOfBeans)
endif()

# 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 dependency: ${comp}")
        set(${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()
```

The above example demonstrates the recommended practice of not creating any imported targets before first checking whether the required components can be satisfied. This prevents imported targets being created for some components but not others in the event of a failure.

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`

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. 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. It is up to the package itself how it determines compatibility. For projects that follow semantic versioning principles as covered back in Section 20.3, “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

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
)
```

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.

`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.

`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.

`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.

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.

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. Similarly, in the past when all details were provided through variables, it was customary to check whether all required variables were set at the end before returning. A macro called `check_required_components()` was defined for this purpose, but projects providing imported targets should perform these checks themselves and only define the imported targets if all components will be found. This makes the `check_required_components()` macro redundant.

An example should help clarify the typical usage of this command:

CMakeLists.txt

```
include(GNUInstallDirs)
include(CMakePackageConfigHelpers)
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 ...
)
```

MyProjConfig.cmake.in

```
@PACKAGE_INIT@

list(APPEND CMAKE_MODULE_PATH "@PACKAGE_cmakeModulesDir@")

# Include the project's export files, etc...
```

25.7.2. 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 shows a fairly minimal config file that should serve as a good starting point:

```
# 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 example of a shared library, 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 the number of subdirectory levels this config file is below the base install point. For example, if the file is found at `lib/cmake/Foo/FooConfig.cmake`, then `NumSubdirLevels` should be 3.

② Specify the path to the library relative to the base install point, which was previously found and stored in `_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 would only be added if it did not already exist, as the following demonstrates:

```
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()
```

25.8. 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 single platform or just a subset of the final 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 affect how a project is structured, including such fundamental things as how functionality is split up between libraries and what symbols need to be visible in the binaries as a result.

Projects should prefer to follow standard package layouts where possible. Using a module like `GNUInstallDirs` can greatly simplify that task, even for packages on Windows. If that is not possible or is undesirable, projects may still want to at least consider if the same directory structure can be used on the different 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).

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}
)
```

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. An example of a good pattern to follow would be `<ProjectName>_<ComponentName>`, such as `MyProj_Runtime`. When installing export sets, use the same project name as the namespace, with two colons appended (i.e. `MyProj::`). Following these naming conventions will make working with the installed project more intuitive, but more importantly 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.

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.

Prefer to use the `InstallRequiredSystemLibraries` module for handling the installation of compiler runtime dependencies. This 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 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.

Chapter 26. 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 each packager's CMake module.

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.

26.1. 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:

```
cpack -G "ZIP;RPM"
```

If the CMake project was configured to use a multi configuration generator like Xcode or Visual Studio, `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 can only be done once for the entire CMake project. This inclusion is usually done 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. For example:

```
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)
    add_subdirectory(packaging)   # include(CPack) will happen in here
endif()
```

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`). Since CMake checks for this file only after it finishes processing `CMakeLists.txt` files, it will therefore always create the `package` build target if the `CPack` module is included.

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.

`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. It's 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.

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)
```

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/
)
```

26.2. 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.

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 26.4.2, “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 26.4.6, “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.

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   "Static libraries and headers needed for building apps"
    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")
```

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.

```
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.

26.3. Multi Configuration Packages

CPack is inherently 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. It takes a little more work to be able to build and package up both configurations into a single set of packages.

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.

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 or Visual Studio, the `MYPROJ_DEBUG_BUILD_DIR` directory in the above example needs to be configured to only support the `Debug` build type rather than the usual default set. This is the only way to force it to install debug build outputs instead of the same configuration as the main project. 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 or set of packages.

26.4. Package Generators

CPack supports the creation of packages in a variety of formats, all of which loosely fall 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 they support, 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 effectively 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.

Product-specific packages

CMake 3.12 added initial support for the NuGet package format for .NET.

Regardless of which type of package is to be generated, the same `CPackConfig.cmake` file is used in each case. This generally doesn't present an issue, since generator-specific configuration is normally made possible through generator-specific variables where needed. If certain logic needs to be added only for 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 generators that need it. The full `cpack` run loosely follows the steps in the following pseudo code:

```
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.

26.4.1. 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`). 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.).

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|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 installed 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.

26.4.2. 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`)

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 configured in further detail beyond what the generic commands provide:

```
cpack_ifw_configure_component(componentName
    [NAME componentNameId]
    [DISPLAY_NAME displayName...]
    [DESCRIPTION description...]
    [VERSION <version>]
    [DEPENDS compId1 [compId2...] ]
    [REPLACES compId3 [compId4...] ]
    # Other options not shown
)

# The cpack_ifw_configure_component_group() command supports all
# of the above options too
```

The DISPLAY_NAME and DESCRIPTION of each component or component 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 very 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)
```

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 above 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:

```
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 itself. This is particularly advantageous if some optional components are quite 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.

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.

One drawback of this package generator is that the installer produced doesn't provide an easy way for users to trigger an unattended command line install. This is a limitation of the Qt Installer Framework itself, not of CMake or CPack. The installer also has extra overhead compared to most other generator types because 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.

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.

26.4.3. 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:

- Command line (i.e. unattended) installs are directly supported 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 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...
```

26.4.4. 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:

- The installer executable directly supports unattended installs through a dedicated command line option.
- 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.

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.

26.4.5. 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:

`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.

This generator type 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_DIR and CPACK_DMG_SLA_LANGUAGES 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 itself. See the CPackDMG module's documentation for how these two variables are used and the requirements around the language files that need to be provided.

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 22, *Apple Features* for the recommended way of preparing such app bundles rather than using the CPack Bundle generator type.

26.4.6. 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. This 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, but 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 using the `PLIST` option to the `cpack_add_component()` command. The plist file should be obtained by using the `--analyze` option to the `pkgbuild` command, the other options for which can be found by looking at the verbose output of a `cpack` command for the project:

```
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. Once a plist file has been generated and updated, it can be used like so:

```
cpack_add_component(MyProj_Runtime
    ...   # Other options
    PLIST runtime.plist
)
```

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.

26.4.7. 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.

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, 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
```

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 `CPackRPM` module 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. Enabling this feature usually just requires setting `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 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, an error such as 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 CPackRPM module 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 CPackRPM module provides 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).

26.4.8. 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.

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 `CPackDeb` module 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 module documentation for the full set of supported variables.

26.4.9. FreeBSD

The FreeBSD package generator is relatively immature, being added only 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 `CPackFreeBSD` module documentation for available features and limitations.

26.4.10. 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.

26.4.11. NuGet

Support for the NuGet package format has been added in CMake 3.12. The new module supports a number of similar options as other package generators, all of which are listed in the module's documentation. These options should mostly be fairly self-explanatory and they follow similar patterns to other generators already discussed above. Developers should be aware that as this package generator is new to CMake, it will not yet have wide public testing, so checking for fixes and updates in later CMake releases is advisable.

26.5. 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.

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 more simplicity and 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, 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.

If multi architecture or debug-and-release packages need to be produced from a single `cpack` invocation, use the `CPACK_INSTALL_CMAKE_PROJECTS` variable to incorporate components from multiple build trees. When using such an arrangement, always install the release components last in case both debug and release components 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.

Chapter 27. 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.

27.1. 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

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.

27.1.1. 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://somecompany.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

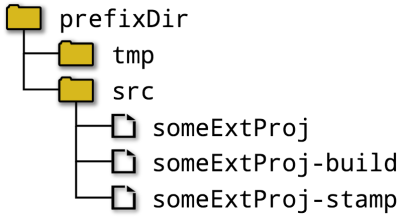
EXCLUDE_FROM_ALL option, which has the same effect as it does for commands like add_executable(), add_custom_target(), etc. In the above example, building the someExtProj target will result in the following being performed during the build stage of the main project:

- 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://somecompany.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://somecompany.com/releases/myproj_1.2.3.tar.gz
)
```

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:

```
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   ""   # Effectively disable 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`

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 27.1.3, "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 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.mycompany.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@somecompany.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 name can change over time, `ExternalProject_Add()` will need to contact the remote end every time CMake is run, even if it already has the named tag or branch cloned and checked out. It does this because it 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. If a commit hash is used instead, then `ExternalProject_Add()` can determine whether it already has the commit locally without needing to contact the remote. Therefore, once the commit has been successfully fetched, no further network connection is needed for any subsequent CMake runs.

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.

Checking out from a subversion repository is fairly similar to git:

```
ExternalProject_Add(myProj
    SVN_REPOSITORY svn+ssh@somecompany.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://somecompany.com/hg/myproj
    HG_TAG         dd2ce38a6b8a
)
ExternalProject_Add(myProjCVS
    CVS_REPOSITORY http://somecompany.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.

27.1.2. 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@somecompany.com/git/myproj.git
    GIT_TAG        3a281711d1243351190bdee50a40d81694aa630a
    STEP_TARGETS   configure install
)
```

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:

```
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...]
    [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 17.1, “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 generator will likely complain about a missing build rule.

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 not 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 (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)
```

27.1.3. 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 `External_Project_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.

```
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)
```

Another feature for commands is the ability to give them access to the terminal, which can be important for things like repository access that may require the user to provide a password for a private key, etc. While this is not suitable for continuous integration builds with no terminal available, it is sometimes useful for developers in their day to day activities. For the default steps, access to the terminal is controlled on a per step basis with options to `ExternalProject_Add()` 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, then giving the download and update steps access to the terminal is advisable.

```
ExternalProject_Add(myProj
    GIT_REPOSITORY         git@somecompany.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 generator, 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 (or to wherever that has been redirected). 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 to `ExternalProject_Add()` or the `LOG` option to `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, which will be in the timestamp directory (i.e. `STAMP_DIR`).

```
ExternalProject_Add(myProj
    GIT_REPOSITORY git@somecompany.com/git/myproj.git
    GIT_TAG        3a281711d1243351190bdee50a40d81694aa630a
    LOG_BUILD      YES
    LOG_TEST       YES
)
```

In some situations, a project may find itself wanting to know whether a particular option was given to `ExternalProject_Add()` or what the effective value of a particular option is. Placeholders such as `<SOURCE_DIR>` and so on cover many of the common scenarios where the details need to be referred to within the call to `ExternalProject_Add()`, but for other cases the module provides the `ExternalProject_Get_Property()` command. It's syntax differs significantly from other property retrieval commands like `get_property()`:

```
ExternalProject_Get_Property(mainName propertyName...)
```

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}")
```

27.1.4. 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 working 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 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`.

```
cmake_minimum_required(VERSION 3.0)
project(ExtProjDeps)
include(ExternalProject)

set(installDir ${CMAKE_CURRENT_BINARY_DIR}/install)

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)
```

The above arrangement where the main project does nothing more than define a set of external projects is often referred to as a *superbuild*, a topic discussed further in the next chapter.

Another dependency-related issue that can arise when using the Ninja generator 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
    # Relevant 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 generator 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
    # Relevant options to download and build the above library
    ...
)
```

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 work, but if there has been a change, then anything that has become out of date will be rebuilt as expected.

27.2. 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 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
)

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.

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 also retrieve some directory details. The full form of the command is as follows:

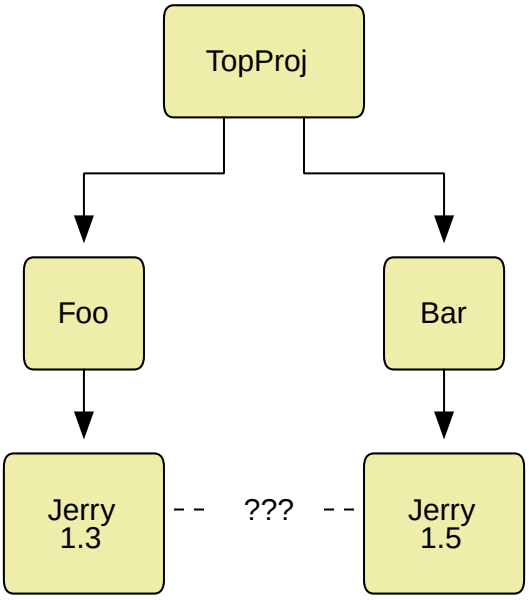
```
FetchContent_GetProperties(name
    [SOURCE_DIR srcDirVar]
    [BINARY_DIR binDirVar]
    [POPULATED  doneVar]
)
```

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.

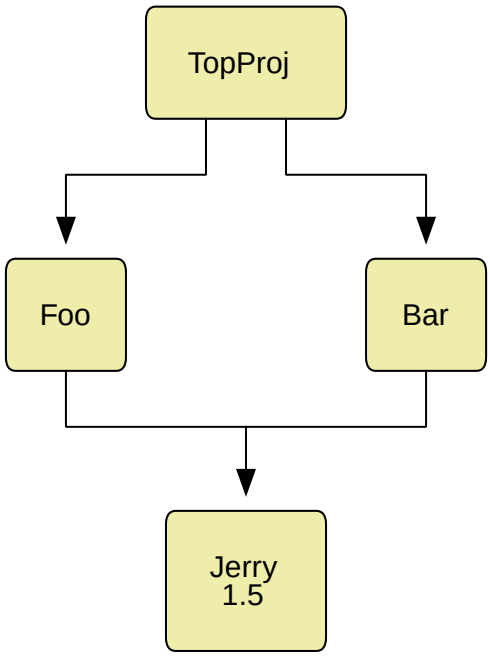
The POPULATED property will be true if FetchContent_Populate() has already been called somewhere in the project for the specified name. If it is true, then the SOURCE_DIR property specifies where the downloaded contents can be found. Since the downloaded contents might not be an immediate subdirectory 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.



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:



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 ensure we get what we want
FetchContent_Declare(jerry
    URL      https://somecompany.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://somecompany.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()
```

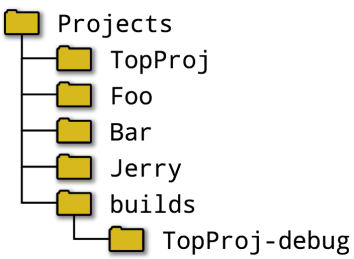
The `CMakeLists.txt` file for `Bar` would be identical to that of `Foo` except the `URL` would specify `jerry-1.5.tar.gz` instead of `jerry-1.3.tar.gz`. The above skeleton example allows `Foo` and `Bar` to be built as standalone projects on their own, or they can be incorporated into another project like `TopProj` and still have the required flexibility to resolve the common dependencies.

27.2.1. Developer Overrides

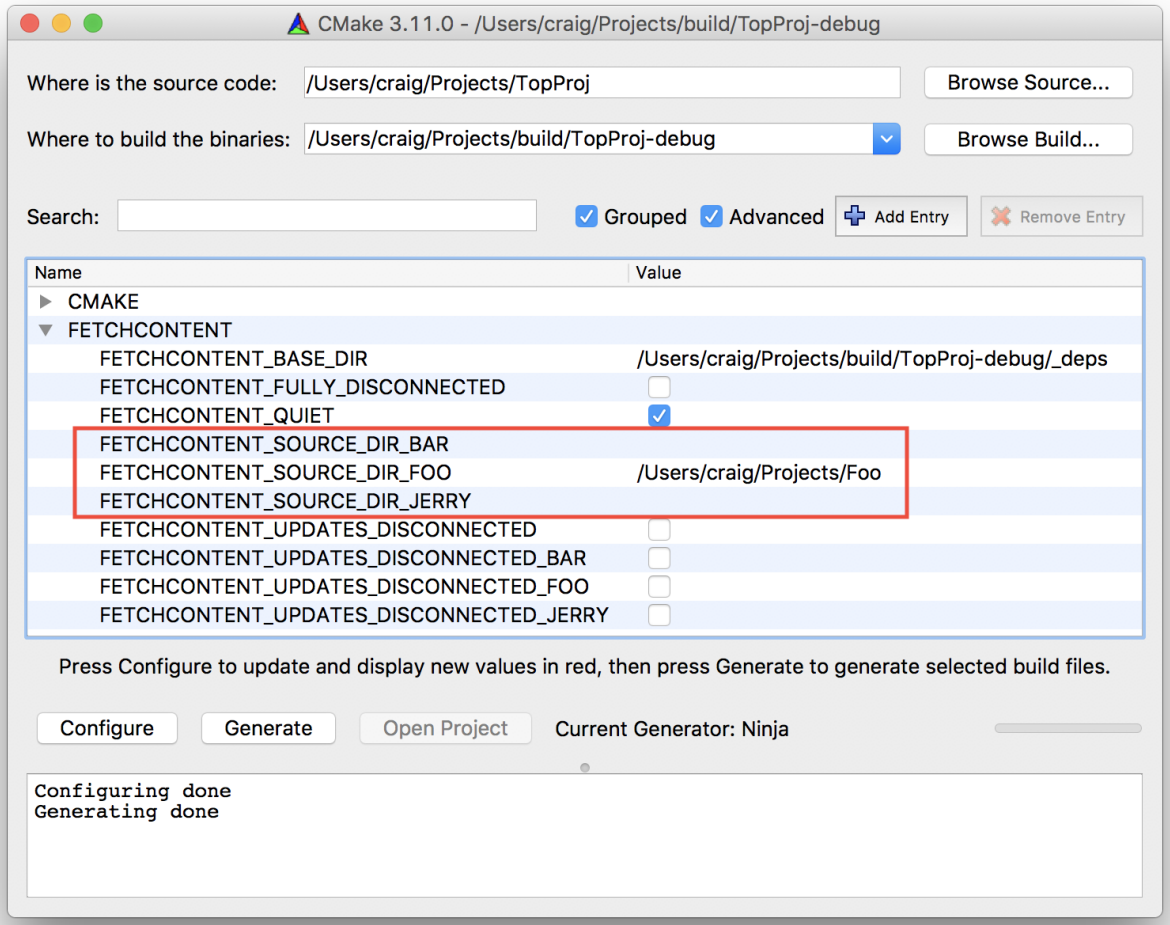
There may be occasions when a developer wants to work on multiple projects at once, making changes in both the main project and its dependencies or across multiple dependencies, etc. When changing parts of an external project, the developer will want to work with a local copy rather than have to update 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:



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 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.

27.2.2. 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 3.11)

include(FetchContent)
FetchContent_Declare(CompanyXToolchains
    GIT_REPOSITORY ...
    GIT_TAG        ...
    SOURCE_DIR     ${CMAKE_BINARY_DIR}/toolchains
)
FetchContent_GetProperties(CompanyXToolchains)
if(NOT companyxtoolchains_POPULATED)
    FetchContent_Populate(CompanyXToolchains)
endif()

project(MyProj)
```

```
cmake -DCMAKE_TOOLCHAIN_FILE=toolchains/toolchain_betacxx.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.

27.2.3. 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
)
```

The OUTPUT_NAME property and other related properties are covered in more detail in Section 28.5.2, “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 of 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.

27.3. 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 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. Choosing an appropriate repository based method also allows small changes in large files to be handled 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.

27.4. 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, but 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 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, so things like code completion and so on often provide greater insight and may be more reliable than if the projects had been loaded separately. 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`.

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.

Chapter 28. 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.

28.1. 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:

```
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):

```
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.

28.2. 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.

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 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)

# Things typically only needed 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

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 each dependency if not already part of the build
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)
```

The subdirectories would then look something like this:

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()
```

The `bar` subdirectory would be similar in structure. The above can even be extended 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()
```

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 28.5, "Defining Targets" further below.

28.3. 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 existing 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. A single `list(APPEND CMAKE_MODULE_PATH ⋯)` call in the project wide setup section of the top level `CMakeLists.txt` also makes them 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.

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 28.5.2, "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.

28.4. 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.

```
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. Be aware, however, that for historical reasons, nesting levels are defined by back slashes rather than forward slashes. 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:

```
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 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` and `REGULAR_EXPRESSION` groups defined later take precedence over those defined earlier where a file matches multiple regular expressions.

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 just ignore them other than to add them to IDE source file lists. This 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 need to compile this file for non-Windows platforms
    set_source_files_properties(net_win.cpp PROPERTIES
        HEADER_FILE_ONLY YES
    )
endif()
```

28.5. 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.
- 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 project specific, so again they may clash with test names of other projects if this one is absorbed into a larger project hierarchy. Again, incorporating the project name or some other equally unique string into the test names would address this.
- The tests are always added, even if this is not the top level project. For large projects with many tests, this can noticeably and unnecessarily increase build times.
- 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 something more like this:

src/CMakeLists.txt

```
#============================
# Define targets
#============================
add_executable(BagOfBeans_planter main.cpp soy.cpp soy.h coffee.cpp coffee.h)
add_executable(BagOfBeans::BagOfBeans_planter ALIAS BagOfBeans_planter)
set_target_properties(BagOfBeans_planter PROPERTIES OUTPUT_NAME planter)
target_compile_definitions(BagOfBeans_planter PUBLIC COFFEE_FAMILY=Robusta)

#============================
# Testing
#============================
add_test(NAME BagOfBeans.planter.NoArgs   COMMAND BagOfBeans_planter)
add_test(NAME BagOfBeans.planter.WithArgs COMMAND BagOfBeans_planter beanType=soy)

#============================
# 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 longer 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.

28.5.1. 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 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    "Support planting soy beans"    ON)
option(BAGOFBEANS_COFFEE "Support 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 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 must 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:

src/coffee/CMakeLists.txt

```
target_sources(BagOfBeans_planter
    PRIVATE
        # WARNING: These will be wrong
        coffee.cpp
        coffee.h
)
...
```

The above is intended to add the sources from the same directory, but 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 `${CMAKE_CURRENT_LIST_DIR}` or `${CMAKE_CURRENT_LIST_DIR}` in use cases such as this is inconvenient and not particularly intuitive. Improvement in this area is likely in an upcoming release (the necessary changes have already been made).

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()`, which can only be used on a target defined in the same directory. This means 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:

```
option(BAGOFBEANS_COFFEE "Support planting coffee beans" ON)

if(BAGOFBEANS_COFFEE)
    add_subdirectory(coffee)
    target_link_libraries(BagOfBeans_planter PRIVATE weather)
endif()
```

This restriction is under active discussion among the CMake developers and may be removed or at least significantly relaxed in a future CMake version. 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 are expected to add to the above variables using PARENT_SCOPE
option(BAGOFBEANS_SOY    "Support planting soy beans"    ON)
option(BAGOFBEANS_COFFEE "Support 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.

28.5.2. 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 type of project generator being used. For projects with many subdirectories or deeply nested hierarchies, this can be inconvenient for a developer to work with. For such cases, CMake provides a number of target properties that give the project a degree of control over the output location of each target's built binary:

`RUNTIME_OUTPUT_DIRECTORY`

Used for executables on all platforms and for 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 for import libraries associated with DLL libraries on Windows.

For all three of the above, multi configuration generators like Visual Studio and Xcode 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.

```
include(GNUInstallDirs)
if(NOT CMAKE_RUNTIME_OUTPUT_DIRECTORY)
    set(CMAKE_RUNTIME_OUTPUT_DIRECTORY
        ${CMAKE_CURRENT_BINARY_DIR}/stage/${CMAKE_INSTALL_BINDIR})
endif()
if(NOT CMAKE_LIBRARY_OUTPUT_DIRECTORY)
    set(CMAKE_LIBRARY_OUTPUT_DIRECTORY
        ${CMAKE_CURRENT_BINARY_DIR}/stage/${CMAKE_INSTALL_LIBDIR})
endif()
if(NOT CMAKE_ARCHIVE_OUTPUT_DIRECTORY)
    set(CMAKE_ARCHIVE_OUTPUT_DIRECTORY
        ${CMAKE_CURRENT_BINARY_DIR}/stage/${CMAKE_INSTALL_LIBDIR})
endif()
```

Avoid creating CMAKE_⋯_OUTPUT_DIRECTORY as cache variables, as they should not be under the control of the developer. They should be controlled by the project because various 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 are following 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, so this default may need to be overridden. The OUTPUT_NAME target property can be set to the base name to use for the binary, or for less typical situations, 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.

```
add_executable(BagOfBeans_planter ...)
set_target_properties(BagOfBeans_planter PROPERTIES OUTPUT_NAME planter)
```

Configuration specific variants such as OUTPUT_NAME_<CONFIG> are also supported for historical reasons, but projects should prefer to use generator expressions instead.

Older projects sometimes try to read the LOCATION target property to determine the output location and name of a binary and use it in places like custom target commands or other similar logic. As already highlighted back in Section 13.4, "Recommended Practices", this is problematic for multi configuration generators, since the location depends on the configuration, but this is not accounted for by the LOCATION target property. CMake 3.0 and later will warn if a project tries to set this target property. Projects should use generator expressions like $<TARGET_FILE:⋯> instead.

28.5.3. 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 more difficult to work with.

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(fooTest ...)
target_link_libraries(fooTest PRIVATE algo)
add_test(NAME fooWithAlgo COMMAND fooTest)
if(WIN32)
    set_tests_properties(fooWithAlgo PROPERTIES ENVIRONMENT
        "PATH=$<SHELL_PATH:$<TARGET_FILE_DIR:algo>>$<SEMICOLON>$ENV{PATH}"
    )
endif()
```

This won't help with allowing the test executable to run within the Visual Studio IDE under the debugger. For that, 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 all the DLLs any tests need will be collected together in a small number of locations, then 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.

```
file(TO_NATIVE_PATH ${CMAKE_RUNTIME_OUTPUT_DIRECTORY} baseDir)
configure_file(user.props.in user.props @ONLY)
```

User property files can be a little complex, but an example of a fairly basic one that can be used with the above might look like this:

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>
```

User property files can be used to set more than just the debugger environment, but the above at least provides a starting point for those wishing to explore this technique further.

For Windows 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` and 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.

28.6. Miscellaneous Project Features

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. Projects can tell CMake about these files by

adding them to the ADDITIONAL_MAKE_CLEAN_FILES directory property, which holds a list of files for that directory scope which should be part of a clean target. This is only supported by the Makefile family of generators. The Ninja generator does not support that property, but it does provide a more robust alternative through the BYPRODUCTS option given to commands like add_custom_command() and add_custom_target(). By listing such files as byproducts, Ninja knows to remove them when the clean target is built. The other project generators have no equivalent functionality.

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.

There will inevitably come a time where a project from some external source needs to be added to a build, but it has some problem that prevents it from working properly. Some common examples include not setting variables or properties that should have been set. This is especially common when working with projects that support very old CMake versions and have not been updated to handle newer CMake features and checks. 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(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, or it can do things like add additional 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:

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, but outside of those cases, developers should generally prefer to add or modify the project's `CMakeLists.txt` files directly.

28.7. 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

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 separate rather than being tied together.

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 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 3.8 or later and defining a user properties file which test targets can then be made aware of using the `VS_USER_PROPS` target property. This can be used to augment the debugger environment so that the tests 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.

Index

@

<CONFIG>_POSTFIX, 109
<LANG>_EXTENSIONS, 125
<LANG>_STANDARD, 124
<LANG>_STANDARD_REQUIRED, 124
<LANG>_VISIBILITY_PRESET, 194

A

ADDITIONAL_MAKE_CLEAN_FILES, 408
ANDROID_API, 217
ANDROID_API_MIN, 217
ANDROID_NDK, 214, 214
ANDROID_NDK_ROOT, 214
ANDROID_STANDALONE_TOOLCHAIN, 214, 214
ANDROID_STL_TYPE, 217
ARCHIVE_OUTPUT_DIRECTORY, 404
ARCHIVE_OUTPUT_NAME, 405
ATTACHED_FILES, 288
ATTACHED_FILES_ON_FAIL, 288
add_compile_definitions(), 117
add_compile_options(), 117
add_custom_command(), 144, 146
add_custom_target(), 142
add_definitions(), 117
add_dependencies(), 143, 374
add_executable(), 13, 15, 132, 221
add_library(), 16, 133, 185
add_subdirectory(), 50
add_test(), 263

B

BUILD_INTERFACE, 307
BUILD_RPATH, 309
BUILD_SHARED_LIBS, 17
BUILD_TESTING, 284
BUILD_WITH_INSTALL_RPATH, 309
BUNDLE, 229
BUNDLE_EXTENSION, 229
Boolean true/false values, 39
Build types (default), 103
BundleUtilities, 322
break(), 48

C

CMAKE_<CONFIG>_POSTFIX, 109
CMAKE_<LANG>_COMPILER, 208
CMAKE_<LANG>_COMPILER_EXTERNAL_TOOLCHAIN, 210
CMAKE_<LANG>_COMPILER_ID, 208
CMAKE_<LANG>_COMPILER_TARGET, 210
CMAKE_<LANG>_COMPILER_VERSION, 208
CMAKE_<LANG>_COMPILE_FEATURES, 127
CMAKE_<LANG>_EXTENSIONS, 125
CMAKE_<LANG>_FLAGS, 119, 209
CMAKE_<LANG>_FLAGS_<CONFIG>, 108, 119, 209
CMAKE_<LANG>_FLAGS_INIT, 209
CMAKE_<LANG>_KNOWN_FEATURES, 127
CMAKE_<LANG>_STANDARD, 124
CMAKE_<LANG>_STANDARD_REQUIRED, 125
CMAKE_<LANG>_VISIBILITY_PRESET, 194
CMAKE_<TARGETTYPE>_LINKER_FLAGS, 119, 209
CMAKE_<TARGETTYPE>_LINKER_FLAGS_<CONFIG>, 108, 119, 209
CMAKE_<TARGETTYPE>_LINKER_FLAGS_INIT, 209
CMAKE_ANDROID_API, 214, 217
CMAKE_ANDROID_API_MIN, 217
CMAKE_ANDROID_ARCH, 214, 217
CMAKE_ANDROID_ARCH_ABI, 214
CMAKE_ANDROID_ARM_MODE, 214
CMAKE_ANDROID_ARM_NEON, 215
CMAKE_ANDROID_NDK, 213
CMAKE_ANDROID_NDK_TOOLCHAIN_VERSION, 215
CMAKE_ANDROID_STANDALONE_TOOLCHAIN, 213
CMAKE_ANDROID_STL_TYPE, 215, 217
CMAKE_APPBUNDLE_PATH, 246, 253
CMAKE_ARCHIVE_OUTPUT_DIRECTORY, 405
CMAKE_BINARY_DIR, 51
CMAKE_BUILD_RPATH, 309
CMAKE_BUILD_TYPE, 104, 166
CMAKE_BUILD_WITH_INSTALL_RPATH, 309
CMAKE_COMMAND, 152
CMAKE_COMPONENTS_ALL, 343
CMAKE_CONFIGURATION_TYPES, 106, 344
CMAKE_CONFIGURE_DEPENDS, 408
CMAKE_CROSSCOMPILING, 207, 211

CMAKE_CROSSCOMPILING_EMULATOR, 211, 278
CMAKE_CURRENT_BINARY_DIR, 51
CMAKE_CURRENT_LIST_DIR, 55
CMAKE_CURRENT_LIST_FILE, 55
CMAKE_CURRENT_LIST_LINE, 55
CMAKE_CURRENT_SOURCE_DIR, 51
CMAKE_DISABLE_FIND_PACKAGE_<packageName>, 255
CMAKE_EXPORT_NO_PACKAGE_REGISTRY, 256
CMAKE_EXTRA_INCLUDE_FILES, 93
CMAKE_FIND_APPBUNDLE, 246
CMAKE_FIND_FRAMEWORK, 244, 247
CMAKE_FIND_LIBRARY_CUSTOM_LIB_SUFFIX, 248
CMAKE_FIND_PACKAGE_NO_PACKAGE_REGISTRY, 256
CMAKE_FIND_PACKAGE_NO_SYSTEM_PACKAGE_REGISTRY, 256
CMAKE_FIND_PACKAGE_SORT_DIRECTION, 254
CMAKE_FIND_PACKAGE_SORT_ORDER, 254
CMAKE_FIND_ROOT_PATH, 244
CMAKE_FIND_ROOT_PATH_MODE_INCLUDE, 245
CMAKE_FIND_ROOT_PATH_MODE_LIBRARY, 247
CMAKE_FIND_ROOT_PATH_MODE_PACKAGE, 254
CMAKE_FIND_ROOT_PATH_MODE_PROGRAM, 247
CMAKE_FOLDER, 397
CMAKE_FRAMEWORK_PATH, 241, 247, 253
CMAKE_GENERATOR_NO_COMPILER_ENV, 151
CMAKE_GNUtoMS, 186
CMAKE_HOST_SYSTEM_NAME, 207
CMAKE_HOST_SYSTEM_PROCESSOR, 207
CMAKE_IGNORE_PATH, 245
CMAKE_INCLUDE_DIRECTORIES_BEFORE, 116
CMAKE_INCLUDE_PATH, 241
CMAKE_INSTALL_<dir>, 299
CMAKE_INSTALL_DEFAULT_COMPONENT_NAME, 306
CMAKE_INSTALL_NAME_DIR, 312
CMAKE_INSTALL_PREFIX, 301, 311
CMAKE_INSTALL_RPATH, 309
CMAKE_INSTALL_RPATH_USE_LINK_PATH, 309
CMAKE_INSTALL_SYSTEM_RUNTIME_COMPONENT, 321
CMAKE_INSTALL_SYSTEM_RUNTIME_DESTINATION, 321
CMAKE_INSTALL_SYSTEM_RUNTIME_LIBS, 321
CMAKE_INSTALL_SYSTEM_RUNTIME_LIBS_SKIP, 321
CMAKE_LIBRARY_ARCHITECTURE, 241, 252
CMAKE_LIBRARY_OUTPUT_DIRECTORY, 405
CMAKE_LIBRARY_PATH, 247
CMAKE_MACOSX_BUNDLE, 221, 229
CMAKE_MACOSX_RPATH, 312
CMAKE_MATCH_<n>, 43
CMAKE_MODULE_PATH, 86, 94, 251
CMAKE_NO_BUILTIN_CHRPATH, 311
CMAKE_OSX_ARCHITECTURES, 237
CMAKE_OSX_DEPLOYMENT_TARGET, 224, 230
CMAKE_OSX_SYSROOT, 229
CMAKE_POLICY_DEFAULT_CMP<NNNN>, 99
CMAKE_POLICY_WARNING_CMP<NNNN>, 99
CMAKE_POSITION_INDEPENDENT_CODE, 203
CMAKE_PREFIX_PATH, 241, 246, 247, 251, 253, 322, 378, 391
CMAKE_PROGRAM_PATH, 246
CMAKE_PROJECT_<PROJNAME>_INCLUDE, 408
CMAKE_PROJECT_DESCRIPTION, 336
CMAKE_PROJECT_NAME, 335
CMAKE_PROJECT_VERSION, 176
CMAKE_PROJECT_VERSION_MAJOR, 176, 336
CMAKE_PROJECT_VERSION_MINOR, 176, 336
CMAKE_PROJECT_VERSION_PATCH, 176, 336
CMAKE_PROJECT_VERSION_TWEAK, 176
CMAKE_REQUIRED_DEFINITIONS, 90
CMAKE_REQUIRED_FLAGS, 90
CMAKE_REQUIRED_INCLUDES, 90
CMAKE_REQUIRED_LIBRARIES, 90, 92
CMAKE_REQUIRED_QUIET, 90
CMAKE_RUNTIME_OUTPUT_DIRECTORY, 405
CMAKE_SIZEOF_VOID_P, 213
CMAKE_SKIP_BUILD_RPATH, 310
CMAKE_SKIP_INSTALL_RPATH, 310
CMAKE_SKIP_RPATH, 310
CMAKE_SOURCE_DIR, 51
CMAKE_STAGING_PREFIX, 210, 245, 301, 311
CMAKE_SYSROOT, 210, 244, 311
CMAKE_SYSROOT_COMPILE, 210, 244

CMAKE_SYSROOT_LINK, 210, 244
CMAKE_SYSTEM_APPBUNDLE_PATH, 254
CMAKE_SYSTEM_FRAMEWORK_PATH, 242, 254
CMAKE_SYSTEM_IGNORE_PATH, 245
CMAKE_SYSTEM_INCLUDE_PATH, 242
CMAKE_SYSTEM_NAME, 207
CMAKE_SYSTEM_PREFIX_PATH, 242, 254
CMAKE_SYSTEM_PROCESSOR, 207
CMAKE_SYSTEM_VERSION, 207, 214
CMAKE_TOOLCHAIN_FILE, 206
CMAKE_TRY_COMPILE_PLATFORM_VARIABLES, 211
CMAKE_TRY_COMPILE_TARGET_TYPE, 211
CMAKE_VISIBILITY_INLINES_HIDDEN, 194
CMAKE_WINDOWS_EXPORT_ALL_SYMBOLS, 194
CMAKE_XCODE_ATTRIBUTE_CODE_SIGNING_REQUIRED, 229
CMAKE_XCODE_ATTRIBUTE_CODE_SIGN_IDENTITY, 232
CMAKE_XCODE_ATTRIBUTE_DEVELOPMENT_TEAM, 231
CMAKE_XCODE_ATTRIBUTE_IPHONEOS_DEPLOYMENT_TARGET, 230
CMAKE_XCODE_ATTRIBUTE_ONLY_ACTIVE_ARCH, 237
CMAKE_XCODE_ATTRIBUTE_XXX, 220
CMAKE_XCODE_GENERATE_SCHEME, 234
CMAKE_XCODE_GENERATE_TOP_LEVEL_PROJECT_ONLY, 236
CMakeCache.txt, 6
CMakeFindDependencyMacro, 324
CMakeForceCompiler, 211
CMakeLists.txt, 6
CMakePackageConfigHelpers, 328
CMakePrintHelpers, 87
CMakePushCheckState, 93
COMPATIBLE_INTERFACE_BOOL, 188
COMPATIBLE_INTERFACE_NUMBER_MAX, 190
COMPATIBLE_INTERFACE_NUMBER_MIN, 190
COMPATIBLE_INTERFACE_STRING, 189
COMPILE_DEFINITIONS
  Directory property, 117
  Source property, 112
  Target property, 111
COMPILE_FEATURES, 126
COMPILE_FLAGS
  Source property, 112
  Target property, 112
COMPILE_OPTIONS
  Directory property, 118
  Source property, 112
  Target property, 111
COMPILE_PDB, 407
COMPILE_PDB_<CONFIG>, 407
COMPILE_PDB_OUTPUT_DIRECTORY, 407
COMPILE_PDB_OUTPUT_DIRECTORY_<CONFIG>, 407
COVERAGE_COMMAND, 286
COVERAGE_EXTRA_FLAGS, 286
CPACK_<GENNAME>_COMPONENT_INSTALL, 342
CPACK_ARCHIVE_<COMP>_FILE_NAME, 346
CPACK_ARCHIVE_COMPONENT_INSTALL, 346
CPACK_ARCHIVE_FILE_NAME, 346
CPACK_BUILD_SOURCE_DIRS, 361
CPACK_COMPONENTS_ALL, 342, 358
CPACK_COMPONENTS_GROUPING, 342, 358
CPACK_COMPONENT_INCLUDE_TOPLEVEL_DIRECTORY, 346
CPACK_DEBIAN_<COMP>_FILE_NAME, 362
CPACK_DEBIAN_<COMP>_PACKAGE_DEPENDS, 362
CPACK_DEBIAN_<COMP>_PACKAGE_NAME, 362
CPACK_DEBIAN_<COMP>_PACKAGE_SHLIBDEPS, 362
CPACK_DEBIAN_ENABLE_COMPONENT_DEPENDS, 362
CPACK_DEBIAN_FILE_NAME, 362
CPACK_DEBIAN_PACKAGE_DEPENDS, 362
CPACK_DEBIAN_PACKAGE_GENERATE_SHLIBS, 362
CPACK_DEBIAN_PACKAGE_GENERATE_SHLIBS_POLICY, 362
CPACK_DEBIAN_PACKAGE_NAME, 362
CPACK_DEBIAN_PACKAGE_SHLIBDEPS, 362
CPACK_DEB_COMPONENT_INSTALL, 362
CPACK_DMG_BACKGROUND_IMAGE, 355
CPACK_DMG_DISABLE_APPLICATIONS_SYMLINK, 355

CPACK_DMG_DS_STORE, 355
CPACK_DMG_DS_STORE_SETUP_SCRIPT, 355
CPACK_DMG_SLA_DIR, 356
CPACK_DMG_SLA_LANGUAGES, 356
CPACK_GENERATOR, 338, 345
CPACK_IFW_PACKAGE_GROUP, 349
CPACK_IFW_PACKAGE_ICON, 347
CPACK_IFW_PACKAGE_LOGO, 347
CPACK_IFW_PACKAGE_MAINTENANCE_TOOL_INI_FILE, 349
CPACK_IFW_PACKAGE_MAINTENANCE_TOOL_NAME, 349
CPACK_IFW_PACKAGE_WINDOW_ICON, 347
CPACK_INSTALL_CMAKE_PROJECTS, 343
CPACK_MONOLITHIC_INSTALL, 342, 358
CPACK_NSIS_ENABLE_UNINSTALL_BEFORE_INSTALL, 354
CPACK_PACKAGE_DESCRIPTION, 337
CPACK_PACKAGE_DESCRIPTION_FILE, 337
CPACK_PACKAGE_DESCRIPTION_SUMMARY, 336
CPACK_PACKAGE_ICON, 337, 345, 355
CPACK_PACKAGE_INSTALL_DIRECTORY, 336
CPACK_PACKAGE_NAME, 335
CPACK_PACKAGE_VENDOR, 336
CPACK_PACKAGE_VERSION, 336
CPACK_PACKAGE_VERSION_MAJOR, 336
CPACK_PACKAGE_VERSION_MINOR, 336
CPACK_PACKAGE_VERSION_PATCH, 336
CPACK_PACKAGING_INSTALL_PREFIX, 360
CPACK_PRODUCTBUILD_IDENTITY_NAME, 356
CPACK_PRODUCTBUILD_KEYCHAIN_PATH, 356
CPACK_PROJECT_CONFIG_FILE, 345
CPACK_RESOURCE_FILE_LICENSE, 337
CPACK_RESOURCE_FILE_README, 337
CPACK_RESOURCE_FILE_WELCOME, 337
CPACK_RPM_<COMP>_DEBUGINFO_PACKAGE, 361
CPACK_RPM_<COMP>_FILE_NAME, 358
CPACK_RPM_<COMP>_PACKAGE_ARCHITECTURE, 359
CPACK_RPM_<COMP>_PACKAGE_NAME, 358
CPACK_RPM_COMPONENT_INSTALL, 358
CPACK_RPM_DEBUGINFO_EXCLUDE_DIRS, 361
CPACK_RPM_DEBUGINFO_EXCLUDE_DIRS_ADDITION, 361
CPACK_RPM_DEBUGINFO_PACKAGE, 360
CPACK_RPM_FILE_NAME, 358
CPACK_RPM_PACKAGE_ARCHITECTURE, 359
CPACK_RPM_PACKAGE_EPOCH, 359
CPACK_RPM_PACKAGE_NAME, 358
CPACK_RPM_PACKAGE_RELEASE, 359
CPACK_RPM_PACKAGE_RELOCATABLE, 360
CPACK_RPM_RELOCATION_PATHS, 360
CPACK_RPM_SPEC_INSTALL_POST, 360
CPACK_RPM_SPEC_MORE_DEFINE, 360
CPACK_SOURCE_IGNORE_FILES, 338
CPACK_STRIP_FILES, 360
CPACK_VERBATIM_VARIABLES, 336
CPACK_WIX_PRODUCT_GUID, 352
CPACK_WIX_UPGRADE_GUID, 352
CPackComponent, 339
CROSSCOMPILING_EMULATOR, 278
CTEST_OUTPUT_ON_FAILURE, 265
CTEST_PARALLEL_LEVEL, 273
CUDA_STANDARD, 124
CXX_EXTENSIONS, 125
CXX_STANDARD, 124
CXX_STANDARD_REQUIRED, 124
CXX_VISIBILITY_PRESET, 194
C_EXTENSIONS, 125
C_STANDARD, 124
C_STANDARD_REQUIRED, 124
C_VISIBILITY_PRESET, 194
CheckCCompilerFlag, 91
CheckCSourceCompiles, 89
CheckCSourceRuns, 90
CheckCXXCompilerFlag, 91
CheckCXXSourceCompiles, 89
CheckCXXSourceRuns, 90
CheckCXXSymbolExists, 92
CheckFortranCompilerFlag, 91
CheckFortranFunctionExists, 92
CheckFortranSourceCompiles, 89
CheckFunctionExists, 92
CheckIncludeFile, 92
CheckIncludeFileCXX, 92
CheckIncludeFiles, 92
CheckLanguage, 92
CheckLibraryExists, 92
CheckPrototypeDefinition, 92

CheckStructHasMember, 92
CheckSymbolExists, 92
CheckTypeSize, 92
CheckVariableExists, 92
Commenting, 13
ccmake, 31
check_c_source_runs(), 90
check_cxx_source_runs(), 90
check_cxx_symbol_exists(), 92
check_symbol_exists(), 92
cmake, 8, 27
cmake -E command mode, 151, 169
cmake-gui, 28
cmake_minimum_required(), 10, 96
cmake_parse_arguments(), 63
cmake_policy(), 96
cmake_pop_check_state(), 93
cmake_print_properties(), 87
cmake_print_variables(), 88
cmake_push_check_state(), 93
cmake_reset_check_state(), 93
configure_file(), 159, 177
configure_package_config_file(), 328
continue(), 48
cpack command, 334
cpack_add_component(), 339
cpack_add_component_group(), 340
cpack_add_install_type(), 340
cpack_configure_downloads(), 350
cpack_ifw_add_repository(), 351
cpack_ifw_configure_component(), 348
cpack_ifw_configure_component_group(), 348
ctest command, 265

D

DEBUG_CONFIGURATIONS, 109, 118
DEPENDS, 275
DESTDIR, 301
DISABLED, 268
DeployQt4, 322
define_property(), 72

E

ENVIRONMENT, 265, 406
EP_STEP_TARGETS, 374
ExternalData, 387
ExternalProject, 366, 390
ExternalProject_Add(), 367
ExternalProject_Add_Step(), 375
ExternalProject_Add_StepDependencies(), 374
ExternalProject_Add_StepTargets(), 376
enable_language(), 205
enable_testing(), 263, 284
execute_process(), 149
export(), 216, 256, 317

F

FAIL_REGULAR_EXPRESSION, 266
FIND_LIBRARY_USE_LIB32_PATHS, 248
FIND_LIBRARY_USE_LIB64_PATHS, 248
FIND_LIBRARY_USE_LIBX32_PATHS, 248
FIXTURES_CLEANUP, 276
FIXTURES_REQUIRED, 276
FIXTURES_SETUP, 276
FOLDER, 397
FRAMEWORK, 226
FRAMEWORK_VERSION, 226
FetchContent, 380, 392
FetchContent_Declare(), 381
FetchContent_GetProperties(), 381
FetchContent_Populate(), 382
FindPkgConfig, 257
file(), 157, 161, 165, 166, 167
find_dependency(), 324
find_file(), 240
find_library(), 247
find_package(), 87, 249
find_path(), 246
find_program(), 246
foreach(), 46
function(), 59

G

GENERATOR_IS_MULTI_CONFIG, 106
GNUInstallDirs, 299, 405
GNUtoMS, 186
GenerateExportHeader, 195
Generators
  Package, 344
  Project, 7

GetPrerequisites, 322
generate_export_header(), 195
get_cmake_property(), 73, 342
get_directory_property(), 74
get_filename_component(), 156
get_property(), 71
get_source_file_property(), 75
get_target_property(), 75
get_test_property(), 77
gtest_add_test(), 290
gtest_discover_tests(), 292

H

HEADER_FILE_ONLY, 399

I

IMPORTED_GLOBAL, 139, 395
IMPORTED_IMPLIB, 134
IMPORTED_LOCATION, 132, 134
IMPORTED_LOCATION_<CONFIG>, 132, 134
IMPORTED_OBJECTS, 134
INCLUDE, 242
INCLUDE_DIRECTORIES
  Directory property, 116
  Target property, 111
INSTALL_INTERFACE, 307
INSTALL_NAME_DIR, 312
INSTALL_RPATH, 309
INSTALL_RPATH_USE_LINK_PATH, 309
INTERFACE_COMPILE_DEFINITIONS, 111
INTERFACE_COMPILE_FEATURES, 126
INTERFACE_COMPILE_OPTIONS, 111
INTERFACE_INCLUDE_DIRECTORIES, 111, 307
INTERFACE_LINK_LIBRARIES, 113
IOS_INSTALL_COMBINED, 237, 314
InstallRequiredSystemLibraries, 321
if(), 39
include(), 54, 86, 122
include_directories(), 116
include_guard(), 57
install(), 302

L

LABELS, 271
LIBRARY_OUTPUT_DIRECTORY, 404
LIBRARY_OUTPUT_NAME, 405
LINK_FLAGS, 113
LINK_INTERFACE_LIBRARIES, 113
LINK_INTERFACE_MULTIPLICITY, 186
LINK_LIBRARIES, 113
LOCATION, 110, 406
link_libraries(), 118
list(), 35

M

MACOSX_BUNDLE, 221
MACOSX_BUNDLE_BUNDLE_NAME, 222
MACOSX_BUNDLE_BUNDLE_VERSION, 222
MACOSX_BUNDLE_COPYRIGHT, 222
MACOSX_BUNDLE_GUI_IDENTIFIER, 222
MACOSX_BUNDLE_ICON_FILE, 222
MACOSX_BUNDLE_INFO_PLIST, 222
MACOSX_BUNDLE_INFO_STRING, 222
MACOSX_BUNDLE_LONG_VERSION_STRING, 222
MACOSX_BUNDLE_SHORT_VERSION_STRING, 222
MACOSX_FRAMEWORK_BUNDLE_VERSION, 227
MACOSX_FRAMEWORK_ICON_FILE, 227
MACOSX_FRAMEWORK_IDENTIFIER, 227
MACOSX_FRAMEWORK_INFO_PLIST, 227
MACOSX_FRAMEWORK_SHORT_VERSION_STRING, 227
MACOSX_PACKAGE_LOCATION, 225, 399
MACOSX_RPATH, 312
MEMORYCHECK_COMMAND, 286
MEMORYCHECK_TYPE, 286
MODULE, 229
macro(), 59
mark_as_advanced(), 29
math(), 37
message(), 31

N

NO_SYSTEM_FROM_IMPORTED, 115

O

OSX_DEPLOYMENT_TARGET, 230
OUTPUT_NAME, 387, 405
option(), 25, 46

P

PARENT_SCOPE, 22, 53, 65
PASS_REGULAR_EXPRESSION, 266
PATH, 242
PDB_NAME, 407
PDB_NAME_<CONFIG>, 407
PDB_OUTPUT_DIRECTORY, 407
PDB_OUTPUT_DIRECTORY_<CONFIG>, 407
PKG_CONFIG_USE_CMAKE_PREFIX_PATH, 258
POSITION_INDEPENDENT_CODE, 202
PREDEFINED_TARGETS_FOLDER, 398
PRIVATE_HEADER, 227, 303, 314
PROCESSORS, 273
PROJECT_LABEL, 397
PROJECT_VERSION, 175, 328
PROJECT_VERSION_MAJOR, 175
PROJECT_VERSION_MINOR, 175
PROJECT_VERSION_PATCH, 175
PROJECT_VERSION_TWEAK, 175
PUBLIC_HEADER, 227, 303, 314
pkg-config, 257
pkg_check_modules(), 257
pkg_search_module(), 257
project(), 12, 175, 205

R

RESOURCE, 223, 303
RESOURCE_LOCK, 274
RPATH, 308, 406
RUNTIME_OUTPUT_DIRECTORY, 404
RUNTIME_OUTPUT_NAME, 405
RUN_SERIAL, 274
remove_definitions(), 117
return(), 56

S

SKIP_BUILD_RPATH, 309
SKIP_RETURN_CODE, 267
SOVERSION, 187
STATIC_LIBRARY_FLAGS, 113
Superbuilds, 390
Syntax
  Generator expressions, 80
  Lua-style brackets, 23, 165
set(), 22, 24
set_directory_properties(), 74
set_property(), 70
set_source_files_properties(), 75
set_target_properties(), 75
set_tests_properties(), 77
source_group(), 398
string(), 33, 161

T

TARGET_BUNDLE_CONTENT_DIR, 313
TARGET_BUNDLE_DIR, 313
TARGET_FILE, 318
TEST_INCLUDE_FILES, 293
TIMEOUT, 268
TIMEOUT_AFTER_MATCH, 269
TestBigEndian, 89
Toolchain files, 206
target_compile_definitions(), 115
target_compile_features(), 126
target_compile_options(), 115
target_include_directories(), 114
target_link_libraries(), 17, 109, 114
target_sources(), 402
test_big_endian(), 89
try_compile(), 90, 211
try_run(), 91, 211

U

USE_FOLDERS, 397
unset(), 24

V

VALGRIND_COMMAND_OPTIONS, 286
VERSION
  Target property, 187
  project() command, 12, 175
VISIBILITY_INLINES_HIDDEN, 194
VS_USER_PROPS, 406
Variables
  Cache, 24
  Environment, 24
  Regular, 22
Versioning
  Comparisons, 42
  Projects, 175

Shared libraries, 186
variable_watch(), 32

W

WILL_FAIL, 268
WINDOWS_EXPORT_ALL_SYMBOLS, 194
WriteCompilerDetectionHeader, 128
while(), 47
write_basic_package_version_file(), 328
write_compiler_detection_header(), 128

X

XCODE_ATTRIBUTE_CODE_SIGN_ENTITLEMENTS, 233
XCODE_ATTRIBUTE_CODE_SIGN_IDENTITY, 232
XCODE_ATTRIBUTE_DEVELOPMENT_TEAM, 231
XCODE_ATTRIBUTE_IPHONEOS_DEPLOYMENT_TARGET, 230
XCODE_ATTRIBUTE_XXX, 220
xcodebuild, 233

Z

ZERO_CHECK, 236