PROFESSIONAL CMAKE

A PRACTICAL GUIDE

CRAIG SCOTT

Professional CMake: A Practical Guide

14th Edition

ISBN 978-1-925904-22-2

https://crascit.com

Table of Contents

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 24.2, “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 18, *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 24.3, “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 18, *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 Section 18.2.4, "Interface Libraries").

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

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

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

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

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

In this example, the `Ui` library is linked to the `Collector` library as `PUBLIC`, so even though `MyApp` only directly links to `Collector`, `MyApp` will also be linked to `Ui` because of that `PUBLIC` relationship. The `Algo` and `Engine` libraries, on the other hand, are linked to `Collector` as `PRIVATE`, so `MyApp` will not be directly linked to them. Section 18.2, "Libraries" discusses additional behaviors for static libraries which may result in further linking to satisfy dependency relationships, including cyclic dependencies.

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

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

4.4. Linking Non-targets

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

Full path to a library file

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

Plain library name

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

Link flag

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

4.5. Old-style CMake

For historical reasons, any link item specified in `target_link_libraries()` may be preceded by one of the keywords `debug`, `optimized` or `general`. The effect of these keywords is to further refine when the item following it should be included based on whether the build is configured as a debug build (see Chapter 14, *Build Type*). If an item is preceded by the `debug` keyword, then it will only be added if the build is a debug build. If an item is preceded by the `optimized` keyword, it will only be added if the build is not a debug build. The `general` keyword specifies that the item should be added for all build configurations, which is the default behavior anyway if no keyword is used. The `debug`, `optimized` and `general` keywords should be avoided for new projects as there are clearer, more flexible and more robust ways to achieve the same thing with today's CMake features.

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:

```
# 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 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 14. Build Type

This chapter and the next cover two closely related topics. The build type (also known as the build configuration or build scheme in some IDE tools) is a high level control which selects different sets of compiler and linker behavior. Manipulation of the build type is the subject of this chapter, while the next chapter presents more specific details of controlling compiler and linker options. Together, these chapters cover material every CMake developer will typically use for all but the most trivial projects.

14.1. Build Type Basics

The build type has the potential to affect almost everything about the build in one way or another. While it primarily has a direct effect on the compiler and linker behavior, it also has an effect on the directory structure used for a project. This can in turn influence how a developer sets up their own local development environment, so the effects of the build type can be quite far-reaching.

Developers commonly think of builds as being one of two arrangements: debug or release. For a debug build, compiler flags are used to enable the recording of information that debuggers can use to associate machine instructions with the source code. Optimizations are frequently disabled in such builds so that the mapping from machine instruction to source code location is direct and easy to follow when stepping through program execution. A release build, on the other hand, generally has full optimizations enabled and no debug information generated.

These are examples of what CMake refers to as the *build type*. While projects are able to define whatever build types they want, the default build types provided by CMake are usually sufficient for most projects:

Debug

With no optimizations and full debug information, this is commonly used during development and debugging, as it typically gives the fastest build times and the best interactive debugging experience.

Release

This build type typically provides full optimizations for speed and no debug information, although some platforms may still generate debug symbols in certain circumstances. It is generally the build type used when building software for final production releases.

RelWithDebInfo

This is somewhat of a compromise of the previous two. It aims to give performance close to a `Release` build, but still allow some level of debugging. Most optimizations for speed are typically applied, but most debug functionality is also enabled. This build type is therefore most useful when the performance of a `Debug` build is not acceptable even for a debugging session. Note that the default settings for `RelWithDebInfo` will disable assertions.

MinSizeRel

This build type is typically only used for constrained resource environments such as embedded devices. The code is optimized for size rather than speed and no debug information is created.

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

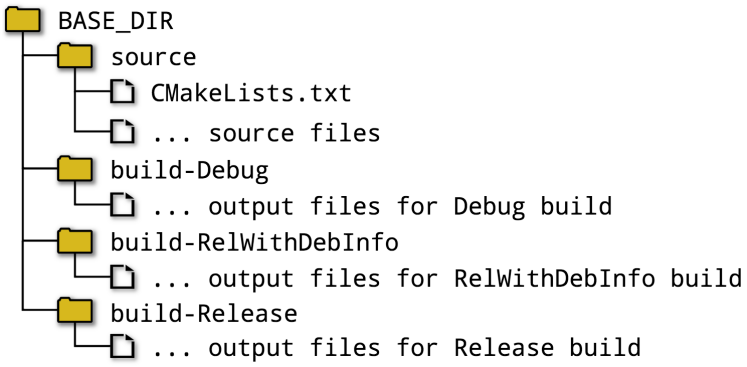
14.1.1. Single Configuration Generators

Back in Section 2.3, "Generating Project Files", the different types of project generators were introduced. Some, like Makefiles and Ninja, support only a single build type per build directory. For these generators, the build type is chosen by setting the `CMAKE_BUILD_TYPE` cache variable. For example, to configure and then build a project with Ninja, one might use commands like this:

```
cmake -G Ninja -DCMAKE_BUILD_TYPE:STRING=Debug ../source
cmake --build .
```

The `CMAKE_BUILD_TYPE` cache variable can also be changed in the CMake GUI application instead of from the command line, but the end effect is the same. With CMake 3.22 or later, if the `CMAKE_BUILD_TYPE` cache variable is not set, it will be initialized from the `CMAKE_BUILD_TYPE` environment variable (if defined).

Rather than switching between different build types in the same build directory, an alternative strategy is to set up separate build directories for each build type, all still using the same sources. Such a directory structure might look something like this:



If frequently switching between build types, this arrangement avoids having to constantly recompile the same sources just because compiler flags change. It also allows a single configuration generator to effectively act like a multi configuration generator. IDE environments like Qt Creator support switching between build directories just as easily as Xcode or Visual Studio allow switching between build schemes or configurations.

14.1.2. Multiple Configuration Generators

Some generators, notably Xcode and Visual Studio, support multiple configurations in a single build directory. From CMake 3.17, the Ninja Multi-Config generator is also available. These multi-config generators ignore the `CMAKE_BUILD_TYPE` cache variable and instead require the developer to choose the build type within the IDE or with a command line option at build time. Configuring and building such projects would typically look something like this:

```
cmake -G Xcode ../source
cmake --build . --config Debug
```

When building within the Xcode IDE, the build type is controlled by the build scheme, while within the Visual Studio IDE, the current solution configuration controls the build type. Both environments keep separate directories for the different build types, so switching between builds doesn't cause constant rebuilds. In effect, the same thing is being done as the multiple build directory arrangement described above for single configuration generators, it's just that the IDE is handling the directory structure on the developer's behalf.

For command-line builds, the Ninja Multi-Config generator has a little more flexibility compared to the other multi-config generators. The `CMAKE_DEFAULT_BUILD_TYPE` cache variable can be used to change the default configuration to use when no configuration is specified on the build command line. The Xcode and Visual Studio generators have their own fixed logic for determining the default configuration in this scenario. The Ninja Multi-Config generator also supports advanced features that allow custom commands to execute as one configuration, but other targets to be built with one or more other configurations. Most projects would not typically need or benefit from these more advanced features, but the CMake documentation for the Ninja Multi-Config generator provides the essential details, with examples.

14.2. Common Errors

Note how for single configuration generators, the build type is specified at *configure* time, whereas for multi configuration generators, the build type is specified at *build* time. This distinction is critical, as it means the build type is not always known when CMake is processing a project's `CMakeLists.txt` file. Consider the following piece of CMake code, which unfortunately is rather common, but demonstrates an incorrect pattern:

```
# WARNING: Do not do this!
if(CMAKE_BUILD_TYPE STREQUAL "Debug")
    # Do something only for debug builds
endif()
```

The above would work fine for Makefile-based generators and Ninja, but not for Xcode, Visual Studio or Ninja Multi-Config. In practice, just about any logic based on `CMAKE_BUILD_TYPE` within a project is questionable unless it is protected by a check to confirm a single configuration generator is being used. For multi configuration generators, this variable is likely to be empty, but even if it isn't, its value should be considered unreliable because the build will ignore it. Rather than referring to `CMAKE_BUILD_TYPE` in the `CMakeLists.txt` file, projects should instead use other more robust alternative techniques, such as generator expressions based on `$<CONFIG:⋯>`.

When scripting builds, a common deficiency is to assume a particular generator is used or to not properly account for differences between single and multi configuration generators. Developers should ideally be able to change the generator in one place and the rest of the script should still function correctly. Conveniently, single configuration generators will ignore any build-time specification and multi configuration generators will ignore the `CMAKE_BUILD_TYPE` variable, so by specifying both, a script can account for both cases. For example:

```
mkdir build
cd build
cmake -G Ninja -DCMAKE_BUILD_TYPE=Release ../source
cmake --build . --config Release
```

With the above example, a developer could simply change the generator name given to the `-G` parameter and the rest of the script would work unchanged.

Not explicitly setting the `CMAKE_BUILD_TYPE` for single configuration generators is also common, but usually not what the developer intended. A behavior unique to single configuration generators is that if `CMAKE_BUILD_TYPE` is not set, the build type will usually be empty. This can lead to the misunderstanding that an empty build type is equivalent to Debug, but this is not so. An empty build type is its own unique, nameless build type. In such cases, no configuration-specific compiler or linker flags are used, which often results in invoking the compiler and linker with minimal flags. The behavior is then determined by the compiler's and linker's own defaults. While this may often be similar to the Debug build type's behavior, it is by no means guaranteed.

Using the Visual Studio compilers with a single configuration generator is somewhat of a special case. For that toolchain, there are different runtime libraries for debug and non-debug builds. An empty build type would make it unclear which runtime should be used. To avoid this ambiguity, the build type will default to Debug for this combination.

14.3. Custom Build Types

Sometimes a project may want to limit the set of build types to a subset of the defaults, or it may want to add other custom build types with a special set of compiler and linker flags. A good example of the latter is adding a build type for profiling or code coverage, both of which require specific compiler and linker settings.

There are two main places where a developer may see the set of build types. When using IDE environments for multi configuration generators like Xcode and Visual Studio, the IDE provides a drop-down list or similar from which the developer selects the configuration they wish to build. For single configuration generators like Makefiles or Ninja, the build type is entered directly for the `CMAKE_BUILD_TYPE` cache variable, but the CMake GUI application can be made to present a combo box of valid choices instead of a simple text edit field. The mechanisms behind these two cases are different, so they must be handled separately.

The set of build types known to multi configuration generators is controlled by the `CMAKE_CONFIGURATION_TYPES` cache variable, or more accurately, by the value of this variable at the end of processing the top level `CMakeLists.txt` file. The first encountered `project()` command populates the cache variable if it has not already been defined. With CMake 3.22 or later, a `CMAKE_CONFIGURATION_TYPES` environment variable can provide the default value. If that environment variable isn't set or an earlier CMake version is used, the default value will be (possibly a subset of) the four standard configurations mentioned in Section 14.1, "Build Type Basics" (`Debug`, `Release`, `RelWithDebInfo` and `MinSizeRel`).

Projects may modify the `CMAKE_CONFIGURATION_TYPES` variable after the first `project()` command, but only in the top level `CMakeLists.txt` file. Some CMake generators rely on this variable having a

consistent value throughout the whole project. Custom build types can be defined by adding them to CMAKE_CONFIGURATION_TYPES and unwanted build types can be removed from that list. Note that only the non-cache variable should be modified, as changing the cache variable may discard changes made by the developer.

Care needs to be taken to avoid setting CMAKE_CONFIGURATION_TYPES if it is not already defined. Prior to CMake 3.9, a very common approach for determining whether a multi configuration generator was being used was to check if CMAKE_CONFIGURATION_TYPES was non-empty. Even parts of CMake itself did this prior to 3.11. While this method is usually accurate, it is not unusual to see projects unilaterally set CMAKE_CONFIGURATION_TYPES even if using a single configuration generator. This can lead to wrong decisions being made regarding the type of generator in use. To address this, CMake 3.9 added a new GENERATOR_IS_MULTI_CONFIG global property which is set to true when a multi configuration generator is being used, providing a definitive way to obtain that information instead of relying on inferring it from CMAKE_CONFIGURATION_TYPES. Even so, checking CMAKE_CONFIGURATION_TYPES is still such a prevalent pattern that projects should continue to only modify it if it exists and never create it themselves. It should also be noted that prior to CMake 3.11, adding custom build types to CMAKE_CONFIGURATION_TYPES was not safe. Certain parts of CMake only accounted for the default build types, but even so, projects may still be able to usefully define custom build types with earlier CMake versions, depending on how they are going to be used. That said, for better robustness, it is recommended that at least CMake 3.11 be used if custom build types are going to be defined.

Another aspect of this issue is that developers may add their own types to the CMAKE_CONFIGURATION_TYPES cache variable and/or remove those they are not interested in. Projects should therefore not make any assumptions about what configuration types are or are not defined.

Taking the above points into account, the following pattern shows the preferred way for projects to add their own custom build types for multi configuration generators:

```
cmake_minimum_required(3.11)
project(Foo)

# Only make changes if we are the top level project
if(CMAKE_SOURCE_DIR STREQUAL CMAKE_CURRENT_SOURCE_DIR)
    get_property(isMultiConfig GLOBAL
        PROPERTY GENERATOR_IS_MULTI_CONFIG
    )
    if(isMultiConfig)
        if(NOT "Profile" IN_LIST CMAKE_CONFIGURATION_TYPES)
            list(APPEND CMAKE_CONFIGURATION_TYPES Profile)
        endif()
    endif()
    # Set Profile-specific flag variables as needed...
endif()
```

For single configuration generators, there is only one build type. This 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”, 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(allowedBuildTypes Debug Release Profile)

# WARNING: This logic is not sufficient
if(NOT CMAKE_BUILD_TYPE IN_LIST allowedBuildTypes)
    message(FATAL_ERROR "${CMAKE_BUILD_TYPE} is not a known build type")
endif()
```

The default value for `CMAKE_BUILD_TYPE` is an empty string, so the above would cause a fatal error for both single and multi configuration generators unless the developer explicitly set it. This is undesirable, especially for multi configuration generators which don't even use the `CMAKE_BUILD_TYPE` variable's value. This can be handled by having the project provide a default value if `CMAKE_BUILD_TYPE` hasn't been set. Furthermore, the techniques for multi and single configuration generators can and should be combined to give robust behavior across all generator types. The end result would look something like this:

```
cmake_minimum_required(3.11)
project(Foo)

if(CMAKE_SOURCE_DIR STREQUAL CMAKE_CURRENT_SOURCE_DIR)
    get_property(isMultiConfig GLOBAL PROPERTY GENERATOR_IS_MULTI_CONFIG)
    if(isMultiConfig)
        if(NOT "Profile" IN_LIST CMAKE_CONFIGURATION_TYPES)
            list(APPEND CMAKE_CONFIGURATION_TYPES Profile)
        endif()
    else()
        set(allowedBuildTypes Debug Release Profile)
        set_property(CACHE CMAKE_BUILD_TYPE PROPERTY
            STRINGS "${allowedBuildTypes}"
        )
        if(NOT CMAKE_BUILD_TYPE)
            set(CMAKE_BUILD_TYPE Debug CACHE STRING "" FORCE)
        elseif(NOT CMAKE_BUILD_TYPE IN_LIST allowedBuildTypes)
            message(FATAL_ERROR "Unknown build type: ${CMAKE_BUILD_TYPE}")
        endif()
    endif()

    # Set relevant Profile-specific flag variables as needed...
endif()
```

The above techniques enable *selecting* a custom build type, but they don't *define* anything about that build type. Selecting a build type specifies which configuration-specific variables to use. It also affects any generator expressions whose logic depends on the current configuration (`$<CONFIG>` and `$<CONFIG:⋯>`). These variables and generator expressions are discussed in the next chapter. For now, the following two families of variables are of primary interest.

- `CMAKE_<LANG>_FLAGS_<CONFIG>`
- `CMAKE_<TARGETTYPE>_LINKER_FLAGS_<CONFIG>`

The flags specified in these variables are added to the default set provided by the same-named variables without the `_<CONFIG>` suffix. A custom `Profile` build type might be defined like so:

```
set(CMAKE_C_FLAGS_PROFILE             "-p -g -O2" CACHE STRING "")
set(CMAKE_CXX_FLAGS_PROFILE           "-p -g -O2" CACHE STRING "")
set(CMAKE_EXE_LINKER_FLAGS_PROFILE    "-p -g -O2" CACHE STRING "")
set(CMAKE_SHARED_LINKER_FLAGS_PROFILE "-p -g -O2" CACHE STRING "")
set(CMAKE_STATIC_LINKER_FLAGS_PROFILE ""          CACHE STRING "")
set(CMAKE_MODULE_LINKER_FLAGS_PROFILE "-p -g -O2" CACHE STRING "")
```

The above assumes a GCC-compatible compiler to keep the example simple and turns on profiling as well as enabling debugging symbols and most optimizations. An alternative is to base the compiler and linker flags on one of the other build types and add the extra flags needed. This can be done as long as it comes after the `project()` command, since that command populates the default compiler and linker flag variables. For profiling, the `RelWithDebInfo` default build type is a good one to choose as the base configuration since it enables both debugging and most optimizations:

```
set(CMAKE_C_FLAGS_PROFILE
    "${CMAKE_C_FLAGS_RELWITHDEBINFO} -p"
    CACHE STRING ""
)
set(CMAKE_CXX_FLAGS_PROFILE
    "${CMAKE_CXX_FLAGS_RELWITHDEBINFO} -p"
    CACHE STRING ""
)
set(CMAKE_EXE_LINKER_FLAGS_PROFILE
    "${CMAKE_EXE_LINKER_FLAGS_RELWITHDEBINFO} -p"
    CACHE STRING ""
)
set(CMAKE_SHARED_LINKER_FLAGS_PROFILE
    "${CMAKE_SHARED_LINKER_FLAGS_RELWITHDEBINFO} -p"
    CACHE STRING ""
)
set(CMAKE_STATIC_LINKER_FLAGS_PROFILE
    "${CMAKE_STATIC_LINKER_FLAGS_RELWITHDEBINFO}"
    CACHE STRING ""
)
set(CMAKE_MODULE_LINKER_FLAGS_PROFILE
    "${CMAKE_MODULE_LINKER_FLAGS_RELWITHDEBINFO} -p"
    CACHE STRING ""
)
```

Each custom configuration should have the associated compiler and linker flag variables defined. For some multi configuration generator types, CMake will check that the required variables exist and will fail with an error if they are not set.

Another variable which may sometimes be defined for a custom build type is `CMAKE_<CONFIG>_POSTFIX`. It is used to initialize the `<CONFIG>_POSTFIX` property of each library target, with its value being appended to the file name of such targets when built for the specified configuration. This allows libraries from multiple build types to be put in the same directory without overwriting each other. `CMAKE_DEBUG_POSTFIX` is often set to values like `d` or `_debug`, especially for Visual Studio builds where different runtime DLLs must be used for Debug and non-Debug builds, so packages may need to include libraries for both build types. In the case of the custom `Profile` build type defined above, an example might be:

```
set(CMAKE_PROFILE_POSTFIX _profile)
```

If creating packages that contain multiple build types, setting `CMAKE_<CONFIG>_POSTFIX` for each build type is highly recommended. By convention, the postfix for Release builds is typically empty. Note though that the `<CONFIG>_POSTFIX` target property is ignored on Apple platforms.

For historical reasons, the items passed to the `target_link_libraries()` command can be prefixed with the `debug` or `optimized` keywords to indicate that the named item should only be linked in for debug or non-debug builds respectively. A build type is considered to be a `debug` build if it is listed in the `DEBUG_CONFIGURATIONS` global property, otherwise it is considered to be `optimized`. For custom build types, they should have their name added to this global property if they should be treated as a `debug` build in this scenario. As an example, if a project defines its own custom build type called `StrictChecker` and that build type should be considered a non-optimized debug build type, it can (and should) make this clear like so:

```
set_property(GLOBAL APPEND PROPERTY
    DEBUG_CONFIGURATIONS StrictChecker
)
```

New projects should normally prefer to use generator expressions instead of the `debug` and `optimized` keywords with the `target_link_libraries()` command. The next chapter discusses this area in more detail.

14.4. Recommended Practices

Developers should not assume a particular CMake generator is being used to build their project. Another developer on the same project may prefer to use a different generator because it integrates better with their IDE tool, or a future version of CMake may add support for a new generator type which might bring other benefits. Certain build tools may contain bugs which a project may later be affected by, so it can be useful to have alternative generators to fall back on until such bugs are fixed. Expanding a project's set of supported platforms can also be hindered if a particular CMake generator has been assumed.

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. Only make such changes in the top level `CMakeLists.txt` file.

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 19, *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.