Short Circuit

CMake

Professional CMake, A Practical Guide
by Craig Scott
12th Edition
Up until Part II

Summary by Emiel Bos

1 Intro

CMake is sort of a meta-build system, meaning it's not a build system itself, but it generates another build system's build files, such as Make, Qt Creator, Ninja, Android Studio, Apple's Xcode, and Microsoft Visual Studio. It has minimal dependencies, requiring only a C++ compiler on its own build system. A (CMake) project generally looks something like this:

- Base directory
  - Source directory (often under version control with Git)
    - Source files
    - `CMakeLists.txt` (defines what should be built and how in a platform independent way, such that CMake can generate platform specific build tool project files. Developers write this.)
  - Build directory (CMake calls this the Binary directory)
    - Build files
    - `CMakeCache.txt` (stores various data for reuse on subsequent runs. Usually not touched by developers, but developer options may be saved between runs.)

Having the source and build directory be the same is called an in-source build, and is stupid and inconvenient. You generally want an out-of-source build, though you could place the build directory inside the source directory.

CMake is actually a suite of tools that also include CTest and CPack. The CMake pipeline looks as follows:

1. Project file generation – done by CMake by running `cmake <source_dir> [-G generator]` inside the build directory. The optional `-G` option specifies the *generator*, which dictates the type of project file to be created, e.g. Visual Studio 17 solutions/projects or makefiles. If the `-G` option is omitted, CMake will choose a default generator type based on the host platform (can be overridden with the `CMAKE_GENERATOR` environment variable). This step is subdivided into two steps:
   (a) Configure; reads the `CMakeLists.txt` file and builds an internal representation of the project.
   (b) Generate; creates project files in the build directory
2. Build – done by build system, but can be invoked from CMake using `cmake --build <build_dir> [--config Debug|Release] --target <target_name>`, where the `--target` option specifies one or more (seperated by spaces) targets to build.
3. Test – done by CTest;
4. Package – done by CPack;

2 `CMakeLists.txt`

The `CMakeLists.txt` file is the heart of CMake and defines everything about the build from sources and targets through to testing, packaging and other custom tasks. CMake commands do not return values, their arguments are separated by whitespace, and their names are case insensitive (but convention is all-lowercase). A minimal `CMakeLists.txt` looks like this:

```
cmake_minimum_required(VERSION <major.minor>[.patch[.tweak][...<major.minor>[.patch[.tweak]]])
project(<ProjectName> VERSION [major[.minor[.patch[.tweak]]]] LANGUAGES
    [NONE|C|CXX|CUDA|ASM|Fortran])
add_executable(<TargetName> [WIN32] [MACOSX_BUNDLE] [EXCLUDE_FROM_ALL] <source1.cpp> [source2.cpp])
```

The `cmake_minimum_required()` specifies the minimum version (or a range of versions) of CMake this project needs and enforces policy settings to match CMake behavior to the specified version. If a range of versions is given, the CMake version must be at least the minimum and the behavior should be the lesser of the specified maximum and the running version.
The `project()` command checks the compilers for each specified language to ensure they are able to compile and link successfully, and populates all built-in variables (that we'll encounter one-by-one) and properties that control the build for the enabled languages. The version specified is the version of the project, and has nothing to do with CMake versions.

3 Executables

The `CMakeLists.txt` file defines targets, which are executables or libraries. The `add_executable()` command creates an executable (e.g. `TargetName.exe` on Windows) from the specified set of source files. It can be run multiple times for multiple different targets. When building on Windows, the optional `WIN32` option will build the executable as a Windows GUI application, which means it will be created with a `WinMain()` entry point instead of just `main()` and it will be linked with the `/SUBSYSTEM:WINDOWS` option. When building on an Apple platform (not only macOS, contrary to the name), the optional `MACOSX_BUNDLE` option will build an app bundle, with the exact effects depending on the platform (e.g. macOS, iOS, etc.). With `EXCLUDE_FROM_ALL`, it will not be included in the default `ALL`[1] target, which is built when no target is specified at build time, and will only be built if it is explicitly requested by the build command or if it is a dependency for another `ALL` build target.

It is best practice to not have the `ProjectName` be equal to any `TargetName` (even though they could).

4 Libraries

Besides executables, CMake can build libraries as well, analogously to `add_executable()`:

```
add_library(<TargetName> [STATIC|SHARED|MODULE] [EXCLUDE_FROM_ALL] <source1.cpp> [source2.cpp ...])
```

This creates a library from the specified set of source files. The types of library are:

- `STATIC`; specifies a static library or archive (Windows: `.lib` Unix: `.a` macOS: `.a`). Such libraries are compiled into an executable at compile-time.
- `SHARED`; specifies a shared library or archive (Windows: `.dll` Unix: `.so` macOS: `.dylib`). Such libraries are linked dynamically to an executable at compile-time and referenced/loaded by programs using it at run-time, and they exist separate from these executable.
- `MODULE`; specifies a plugin that is not linked into other targets, but may be loaded dynamically at runtime using `dlopen()`-like functionality.

If no type is given, the type is `STATIC` or `SHARED` based on whether the value of the `BUILD_SHARED_LIBS` variable is `ON`, which is more flexible when choosing between static or dynamic libraries as a project-wide strategy. It is actually recommended to omit the type until needing to specify individual libraries.

Don't start library target names with `lib...`, because on almost all platforms (except Windows), a leading `lib...` will be prefixed automatically when constructing the actual library name to make it conform to the platform's convention.

[1] This name may differ per build system, e.g. `ALL_BUILD` for Xcode.

5 Linking

To link a library to a target, i.e. an executable or other library, use:

```
target_link_libraries(<TargetName>
  <PRIVATE|PUBLIC|INTERFACE> <lib1> [lib2 ...]
  [<PRIVATE|PUBLIC|INTERFACE> <lib3> [lib4 ...]]
  ...
)
```

where the specified libraries can be any of the following types[2]:

- Other existing CMake target
- Full path to a library file
- Plain library name, in which case CMake will search for that library (e.g. `foo` becomes `-lfoo` or `foo.lib`, depending on the platform). Common for system libraries.
- Link flag, i.e. items starting with a hyphen other than `-l` or `-framework`, in which case CMake will treat these as flags to be added to the linker command. (Use this only for `PRIVATE` libraries, or else they would be carried through to other targets which isn't always safe.)

and where the types of dependency relationship[3] are:

- `PRIVATE`; the target uses the library only in its own internal representation, and anything else that in turn links to the target doesn't need to know about the linked library.
- `PUBLIC`; the target uses the library in its own internal representation as well as its interface, so anything that in turn links to the target will also have to link the library. An example would be one of the target's interface functions having a parameter of a type defined and implemented by the library.
- `INTERFACE`; the target uses the library only in it's interface.

Even though it's not needed (the default is `PUBLIC`), it is recommended to always specify the type of dependency relationship.

6 Variables

Even though variables may be interpreted as a different type in some contexts, they're all strings. Variables are set with the `set()` command and used with

```
set(varName stringVal) # varName = "stringVal"
set(varName a b c) # varName = "a;b;c", multiple values are joined together with a ";", which CMake
    considers a list
set(varName a;b;c) # varName = "a;b;c", same effect
set(varName "a b c") # varName = "a b c", i.e. you need quotes for spaces
set(varName a b;c) # varName = "a;b;c"
set(varName a "b c") # varName = "a;b c"
set(varName var) # varName = "var"
set(${varName}Name ${varName}) # varName = "var"

# Strings can be multi-line, with the newlines embedded into it
# Also, instead of quotes, double brackets can be used so that quotes don't have to be escaped
# Between the brackets, any number of ='s may be placed to avoid misinterpretation, as long as the
    open and close have the same number of ='s. Useful for code.
set(varName [=[
#!/bin/bash
[[ -n "${USER}" ]] && echo "Have USER"
]=])

unset(varName) # Unset variable varName
set(varName) # Calling set() with no value does exactly the same
```

[2]For historical reasons, the library specifications may be preceded by one of the keywords `debug`, `optimized` or `general`, which dictates the build type for which that library is included, but don't use these anymore (there are better ways).

[3]Also, `LINK_PRIVATE`, `LINK_PUBLIC` and `LINK_INTERFACE_LIBRARIES` are precursors to the above relationship types, but don't use these as well, because how they affect target properties is dependant on the policy settings, and this can quickly lead to confusion.

```
set(ENV{varName} "value") # Set environment variable, but only affects the currently running CMake
    instance and won't be visible at build time (so not very useful). Use with $ENV{varName}
```

Variables can be printed as follows:

```
message("The value of varName = ${varName}")
```

There are also a number of string operations that use the `string()` command. The first argument specifies the operation, and subsequent arguments generally include an input string and – since CMake commands cannot return a value – an output string for the result. Multiple input strings are concatenated before substitution.

```
string(FIND <inputString> <subString> <outputVar> [REVERSE]) # Searches inputString for subString
    and stores the (character) index if found, or -1
string(REPLACE <matchString> <replacementString> <outVar> <inputString1> [inputString2] ...) #
    Replaces every occurrence of matchString in the input string(s) with replacementString
string(SUBSTRING <inputString> <index> <length> <outVar>) # Extracts length characters from
    inputString starting at index
string(LENGTH <inputString> <outVar>) # Returns the length (in bytes, which is often fine)
string(TOLOWER <inputString> <outVar>) # Converts inputString to all-lowercase
string(TOUPPER <inputString> <outVar>) # Converts inputString to all-uppercase
string(STRIP <inputString> <outVar>) # Strips whitespace from the start and end of a string
string(REGEX MATCH <regex> <outVar> <inputString1> ...) # Finds the first match
string(REGEX MATCHALL <regex> <outVar> <inputString1> ...) # Finds all matches and stores them as
    list
string(REGEX REPLACE <regex> <replacementString> <outVar> <inputString1> ...) # Returns the entire
    input with each match replaced by replacementString
```

Likewise, there are a number of operations on lists:

```
list(LENGTH <inputList> <outVar>) # Return the number of items
list(GET <inputList> <index> [index...] <outVar>) # Get the item(s) at the given indices
list(FIND <inputList> <value> <outVar>) # Returns the index of the item in inputList if found, or -1
list(INSERT <listVar> <index> <item> [item...]) # Inserts the item(s) at the index in-place
list(APPEND <listVar> <item> [item...]) # Appends the item(s) in-place
list(PREPEND <listVar> <item> [item...]) # Prepends the item(s) in-place
list(REMOVE_ITEM <listVar> <value> [value...]) # Removes all instances of one or more items in-place
list(REMOVE_AT <listVar> <index> [index...]) # Removes items at the specified indices in-place
list(REMOVE_DUPLICATES <listVar>) # Removes duplicate items in-place
list(POP_FRONT <listVar> [outVar1 [outVar2...]]) # Pops the first max(1, #outVars) number of items
    from the front
list(POP_BACK <listVar> [outVar1 [outVar2...]]) # Pops the first max(1, #outVars) number of items
    from the back
list(REVERSE <listVar>) # Who knows?
list(SORT <listVar> [COMPARE STRING|FILE_BASENAME|NATURAL] [CASE SENSITIVE|INSENSITIVE] [ORDER
    ASCENDING|DESCENDING])
```

One peculiarity about lists: openings square brackets (`[`) have to be closed (`]`) within the same list item, or else everything between is part of one item.

Math expressions can be evaluated as follows:

```
math(EXPR <outVar> <mathExpr> [OUTPUT_FORMAT DECIMAL|HEXADECIMAL]) # mathExpr is a string that may
    contain +-*/%|&^~<<>>, parentheses, and variables
```

CMake defines and uses a slew of built-in variables, that affect all sorts of program behaviour.

6.1 Cache variables

Cache variables are stored in `CMakeCache.txt` and persist between runs. These allow developers of a project to change their build – e.g. they may affect paths to external packages, flags for compilers and linkers, which parts of the build are active, etc. – without having to edit `CMakeLists.txt`. There are three ways to set them:

- In `CMakeLists.txt` with `set()` or `option()`:

  ```
  set(varName value CACHE <type> "docstring (for GUI tooltips)" [FORCE]) # Set cache
      variable. Only overwrites a cache variable if FORCEd, unlike normal variables
  option(varName "docstring" [initialValue]) # Sets a boolean cache variable; more or less
      the same as set(varName initialValue CACHE BOOL "docstring"). Omitting initialValue
  ```

```
sets varName to OFF
```

The type given for cache variables is mostly used for GUI, with some exceptions. It can be any of:

- `BOOL`; GUI tools use a checkbox or similar to represent the variable, and the string should be any of `ON`/`OFF`, `TRUE`/`FALSE`, `1`/`0`, etc. (and this should also be the `initialValue` when using `option()`)
- `FILEPATH`; GUI tools present a file dialog to the user
- `PATH`; GUI tools present a dialog that selects a directory rather than a file
- `STRING`; GUI tools use a single-line text edit widget
- `INTERNAL`; not intended to be made available to the user, but used to persistently record internal information by the project. Can only be changed in `CMakeLists.txt`

Cache variables are accessed in the usual `option()` does nothing if a normal variable with the same name already exists with the same name, and `set()` does nothing if both a normal and cache variable with the same name already exist (except if `FORCE`d or `INTERNAL`d). In short, just don't have a normal and a cache variable share the same name.

- In the command line:

```
cmake -D varName[:type]=value1 [var2Name[:type2]=value2] ...
```

which will always overwrite/replace any previous value of `varName` (so it's the same as using `set()` with `CACHE` and `FORCE`). It will be set with an empty docstring. The `type` can be omitted, but this is not recommended.[4] Some examples:

```
cmake -D varName:BOOL=ON
cmake -D "varName:STRING=This contains spaces"
cmake -D varName:FILEPATH=subdir/helpers.txt
```

Variables can be removed from the cache with:

```
cmake -U varName [-U varName] ...
```

This last option supports `*` and `?` wildcards.

- In one of the two (equivalent) GUIs that CMake provides: `cmake-gui` (supported on all major desktop platforms) and `ccmake` (for all platforms except Windows; it's a curses-based interface which can be used in text-only environments such as over a SSH connection). They allow build and source directories to be defined, cache variables to be added (which is the same as `set()`), removed, viewed (with hoverover docstring tooltip) and edited (according to their type), and to configure and generate. When the configure stage is run for the first time, the a dialog shows where the CMake generator and toolchain can be specified. Each time the configure step is initiated, the cache variables shown are updated, with added or changed values highlighted in red. It is good practice to re-run the configure stage until there are no changes.

Variables can be marked as advanced to not have them show up by default:

```
mark_as_advanced([CLEAR|FORCE] varName1 [varName2...]) # Add (FORCE) or remove (CLEAR) the
    Advanced-property of the given variables. Without either keyword, the variables will
    only be marked if they don't already have a mark state set.
```

The GUI also allows grouping variables by the first part of their names up to the first underscore.

7 Flow Control

If-then-else:

```
if(<expression1>)
   # commands ...
```

[4] It is given a special type that is similar to `INTERNAL` but which CMake interprets to mean undefined. Also, if the project's `CMakeLists.txt` file tries to overwrite this cache variable and with type `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 very robust, since `cmake` could be invoked from any directory, not just the build directory.

```
elseif(<expression2>)
   # commands ...
else()
   # commands ...
endif()
```

The expression(s) can be a

- Constant (quoted or unquoted)
  - ON, YES, TRUE, Y or a non-zero value evaluate to true
  - OFF, NO, FALSE, N, IGNORE, NOTFOUND, an empty string, a string that ends in -NOTFOUND or a zero value evaluate to false
- Variables of the form
- Variable names (unquoted)
  - If it's value doesn't match any of the false constants it evaluates to true
  - If it's value matches any of the false constants or is undefined (evaluated as the empty string) it is false
- Strings (quoted)
  - If it's value doesn't match any of the false constants it evaluates to true
  - Else it is false
- Combinations of expressions using
  - Logical operators: AND, OR and NOT
  - Comparison operators for:
    * Numbers[5]: LESS, GREATER, EQUAL, LESS_EQUAL and GREATER_EQUAL
    * Strings[6]: STRLESS, STRGREATER, STREQUAL, STRLESS_EQUAL and STRGREATER_EQUAL
    * Version numbers[7]: VERSION_LESS, VERSION_GREATER, VERSION_EQUAL, VERSION_LESS_EQUAL and VERSION_GREATER_EQU
- Other tests
  - File system tests: EXISTS, IS_DIRECTORY, IS_SYMLINK, IS_ABSOLUTE and IS_NEWER_THAN[8]
  - Existence tests:
    * DEFINED; checks if the specified (cache) variable exists
    * COMMAND; checks if the specified command, function or macro exists. Useful for checking whether something is defined before trying to use it
    * POLICY; checks if the specified policy is knows to CMake
    * TARGET; checks if the specified target has been defined by add_executable(), add_library() or add_custom_target()
    * TEST; checks if the specified test has been defined by add_test()
    * IN_LIST; checks if the specified value is in the specified list

Looping takes a few different forms:

```
foreach(<loopVar> <arg1> [arg2 ...])
   # Each loop, loopVar is assigned the next argument
endforeach()

foreach(<loopVar> IN [LISTS listVar1 ...] [ITEMS item1 ...])
   # More general. loopVar iterates first through all lists, then through all items
endforeach()
```

[5]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, so be careful.

[6]CMake also supports testing a string against a regular expression.

[7]Same robustness caveats as numeric comparisons.

[8]Compares two files and also returns true if both files have the same timestamp.

```
foreach(<loopVar> IN [LISTS listVar1 ...] [ITEMS item1 ...])
   # More general. loopVar iterates first through all lists, then through all items
endforeach()

foreach(<loopVar1> [loopVar2...] IN ZIP_LISTS <listVar1> [listVar2...])
   # If only one loopVar is given, then there is a loopVar_N variable available for each listVarN
   # If there is a loopVar for each listVar, then those are mapped one-to-one instead of creating
       loopVar_N variables
   # When iteration moves past the end of a shorter list, the associated variable is undefined,
       i.e. the empty string
endforeach()

foreach(<loopVar> RANGE <start> <stop> [step])
   # loopVar loops through the values from start to stop (inclusive) with the given step
endforeach()

foreach(<loopVar> RANGE value)
   # Equal to foreach(loopVar RANGE 0 value) (so it executed value+1 times)
endforeach()

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

CMake supports `break()`ing from the innermost loop and `continue()`ing to the next iteration.

8 Subdirectories

A very basic `CMakeLists.txt` file will only work with the top-level directory of the source tree. There are two ways to have CMake access subdirectories.

- `add_subdirectory()`:

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

  brings `sourceDir` into the build, which must have it's own `CMakeLists.txt`, but that file doesn't need it's own `project()`.[9] This means that CMake creates a corresponding directory in the project's build tree and runs `sourceDir`'s `CMakeLists.txt` within a child scope. That child scope has its own policies and receives a copy of all of the variables defined in the calling scope at that point, meaning any added variables or changes to variables in the child are performed in isolation on the child's version of the set of variables, leaving the caller's variables unchanged. You can circumvent this with the `PARENT_SCOPE` keyword in the `set()` command, which sets the specified variable in the parent scope, and only that variable. However, this is best avoided for clarity.
  `sourceDir` can be (the absolute or relative path of) any directory; it doesn't have to be within the source tree (though it usually is). If `buildDir` is omitted, CMake creates a directory in the build tree with the same name as `sourceDir`. `buildDir` can also be an absolute path or relative path, with the latter being relative to the current build directory (so not the current source directory). If the given `sourceDir` is outside the source tree, `buildDir` does need to be specified. `EXCLUDE_FROM_ALL` controls whether targets defined in the subdirectory being added should be included in the project's `ALL` target by default.[10]
  CMake stores the absolute paths of the top-most directory of the source and build tree in the `CMAKE_SOURCE_DIR` and `CMAKE_BINARY_DIR` built-in variables, respectively, and the absolute paths of the source and build directory of the currently processed `CMakeLists.txt` file in the `CMAKE_CURRENT_SOURCE_DIR` and `CMAKE_CURRENT_BINARY_DIR` built-in variables.

- `include()`:

  ```
  include(<fileName|moduleName> [OPTIONAL] [RESULT_VARIABLE <varName>] [NO_POLICY_SCOPE])
  ```

  `include()` does not introduce a new variable scope, and neither a new policy scope if `NO_POLICY_SCOPE` is passed.

[9] Often, such additional `project()` commands create additional – mostly superfluous – files. For example, when using a Visual Studio project generator, each `project()` command creates an associated solution file. The top-level solution file will contain all targets in the project, while any solution file in subdirectories will only contain the targets in that scope and below. This may be useful in some cases, however. The Xcode generator behaves in a similar way, but they do not include the logic for building targets from outside of that directory scope or below.

[10] This doesn't always act as expected and can even result in broken builds.

If OPTIONAL is given, no error is raised if the file or module does not exist. If a RESULT_VARIABLE is given, it will store the full filename upon success or NOTFOUND upon failure. This command can also be used to load modules.
The CMAKE_CURRENT_SOURCE_DIR and CMAKE_CURRENT_BINARY_DIR variables do not change, but CMake provides CMAKE_CURRENT_LIST_DIR (which is essentially CMAKE_CURRENT_SOURCE_DIR but which is updated when using include()), CMAKE_CURRENT_LIST_FILE (which is the name of the current file), and CMAKE_CURRENT_LIST_LINE (which gives the line number and may be useful in some debudding scenarios).[11]
A benefit of include() is that content can be included twice, so that different subdirectories of a large, complex project can independently include some file with CMake code in a common area of the project. If such a file should only be processed once, include guards can be used:

```
if(DEFINED include_guard)
   return()
endif()

set(include_guard 1) # The next time this file is executed, it will return()
```

or more shortly

```
include_guard([GLOBAL|DIRECTORY]) # Analogous to C/C++'s #pragma once. By default, variable
    scope is assumed and the effect is equivalent to the above, GLOBAL checks processing in
    the entire project, and DIRECTORY checks processing in the current directory scope and
    below
```

If not called from inside a function, return() ends processing of the current file and exits its scope, regardless of whether it was brought in via include() or add_subdirectory().

If a project is incorporated into another (parent) project, e.g. as a Git submodule, and that parent project add_subdirectory()s the project, then variables CMAKE_SOURCE|BINARY_DIR in the project's CMakeLists.txt no longer point to that project's source tree, as it was intended, but to the parent project's source tree. To overcome this, the project() command sets some variables that provides relative paths in a more robust way:

- PROJECT_SOURCE|BINARY_DIR; the source/binary directory of the most recent call to project() in the current scope or any parent scope
- <projectName>_SOURCE|BINARY_DIR; the source/binary directory of the most recent call to project(projectName) in the current scope or any parent scope, i.e. it is tied to a specific project name/project() call

A project can check whether it is the top project with CMAKE_CURRENT_SOURCE_DIR STREQUAL CMAKE_SOURCE_DIR. The variable PROJECT_IS_TOP_LEVEL expresses this more clearly, which will be true if the most recent call to project() in the current directory scope or above was in the top level CMakeLists.txt. <projectName>_IS_TOP_LEVEL is defined for every call to project() as a cache variable, so it can be read from any directory, which may be useful when there are intervening calls to project() between the current scope and the scope of interest.

9 Functions

Functions have their own scope and their arguments become variables in the function body, while macros effectively paste their body into the call and their arguments are string replacements, so they're more akin to C/C++ #define macros:

```
function(<func_name> [arg1 [arg2 [...]]])
   # Function body. Each argument is a variable
endfunction()

func_name() # Call the function

macro(<mac_name> [arg1 [arg2 [...]]])
   # Macro body. Each argument is a string substitutions (but ${arg1} still works)
endmacro()

mac_name() # Call the macro
```

Similarly to add_subdirectory(), functions have their own scope and variables defined or modified inside a function have no effect on variables of the same name outside of the function. Values can be "returned" by set()ing with the PARENT_SCOPE keyword. About the only reason one would use a macro instead of a function is if many variables need

[11] These variables are also updated with add_subdirectory().

to be set in the calling scope, so that `set()` doesn't have to be called with `PARENT_SCOPE`. Also, any `return()` statement from a macro will actually be returning from the scope of whatever called the macro, not from the macro itself.

A function call can give more arguments than there are parameters in the function definition; these extraneous arguments are unnamed arguments. Inside function and macro bodies, the following variables/variable-likes are available: `ARGC`, the number of (named or unnamed) arguments passed; `ARGV`, a list containing the (named or unnamed) arguments passed; and `ARGN`, a list containing only the unnamed arguments. Each individual (named or unnamed) argument can be referenced with variable(-like) `ARGV<x>`, with `x` the number of the argument. Be careful with macros and their string substitution though, because if they use `ARGN` in a place where a variable name is expected (such as `foreach()` with the `LISTS` keyword), 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.

In order to support multiple, keyworded argument sets (like `target_link_libraries()`[12]), CMake provides the built-in `cmake_parse_arguments()` command:

```
cmake_parse_arguments(
   <prefix>
   [valuelessKeywords] [singleValueKeywords] [multiValueKeywords]
   argsToParse... # Typically given as ${ARGN} without quotes
)
```

When `cmake_parse_arguments()` returns, variables are defined of the form `<prefix>_<keyword>`. Each of the `...Keywords` is a quoted list of keywords, conventionally all-uppercase:

- `valuelessKeywords` define standalone keyword arguments which act like boolean switches. If the keyword is present, its argument variable is true, else it's false.
- `singleValueKeywords` define keywords that each require exactly one additional argument after the keyword when they are used. If the keyword with argument is present, its argument variable has the given value.
- `multiValueKeywords` define keywords that each require zero or more additional arguments after the keyword. If the keyword with at least one argument is present, its argument variable has as value a list with the given values.

An example:

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

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

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

   foreach(arg IN LISTS singleValues multiValues)
      message(" ${arg} = ${${prefix}_${arg}}")
   endforeach()
endfunction()

func(ENABLE_DEBUG
   TARGET dummy
   IMAGES here.png there.png gone.png
)
```

[12] Unfortunately, the about-to-be-discussed method does not support keywords being used more than once; only the last occurence of a keyword and its succeeding arguments are taken into account.

which will print

```
ENABLE_NET disabled
ENABLE_DEBUG enabled
TARGET = dummy
SOURCES =
IMAGES = here.png;there.png;gone.png
```

Arguments that are given before a keyword in the function call are leftover arguments that can be retrieved as a list from the variable `<prefix>_UNPARSED_ARGUMENTS`. The list variable `<prefix>_KEYWORDS_MISSING_VALUES` will be populated with a list containing all single- or multi-value keywords that were present but which did not have any value following them.

Arguments in a function call can be separated by (one or more consecutive) spaces or semicolons. However, spaces only act as argument separators before any variable evaluation is performed, so a space in a variable's string is (part of) an argument.
In the example, the quoting around the evaluation of `noValues`, `singleValues` and `multiValues` is necessary to prevent the embedded semicolons from acting as argument separators, such that the keywords are packed together as a single argument, which is what `cmake_parse_arguments()` needs. In contrast, the

```
func(a "" c) # ${ARGV} = a;;c, so only a and c are passed to cmake_parse_arguments()
func("a;b;c" "1;2;3") # ${ARGV} = a;b;c;1;2;3, so the two-level argument structure gets flattened
```

To circumvent this, CMake introduced another variant of `cmake_parse_arguments()`, which avoid evaluating

```
cmake_parse_arguments(
   PARSE_ARGV <startIndex>
   <prefix>
   [valuelessKeywords] [singleValueKeywords] [multiValueKeywords]
)
```

However, because it reads An advantage of this form is that if any leftover arguments (that don't correspond to any keyword) are themselves a list, their embedded semicolons will be escaped in the `<prefix>_UNPARSED_ARGUMENTS` list variable. Another use case of this form is inside a wrapper function to avoid list flattening when forwarding arguments. In most cases, dropping empty arguments or flattening lists has no real impact, and either form of `cmake_parse_arguments()` can safely be called.

When `function()` or `macro()` is called to define a new command and a command already exists with that name, CMake makes the old command available using the same name except with an underscore prepended. This only works once – i.e. CMake doesn't infinitely prepend underscores – and you can't write wrapper functions exploiting this mechanism without the danger of infinite recursion. This is undocumented behaviour and should generally not be used.

During execution of a function, the following built-in variables are available: `CMAKE_CURRENT_FUNCTION` holds the name of the function currently being executed, `CMAKE_CURRENT_FUNCTION_LIST_FILE` contains the full path to the file that defined the function currently being executed, `CMAKE_CURRENT_FUNCTION_LIST_DIR` holds the absolute directory containing that file, and `CMAKE_CURRENT_FUNCTION_LIST_LINE` holds the line number at which the currently executing function was defined within that file.

Besides functions and macros, a third way of invoking CMake code is using the `cmake_language()` command:

```
cmake_language(CALL <command> [args...]) # Calls the specified command (which can be a variable)
    with the specified arguments. Only useful to parameterize a command without having to hard-code
    all available choices
cmake_language(EVAL CODE <code...>) # Executes any valid CMake script. Handy for e.g. call stack
    tracing
cmake_language(DEFER # Queues a command until the end of current directory scope
   [DIRECTORY dir] # Allows specifying a parent directory scope instead, e.g. ${CMAKE_SOURCE_DIR}
   [ID id | ID_VAR outVar] # Allows grouping of commands by ID
   CALL <command> [args...] # The command variable is evaluated immediately, while argument
       evaluation is deferred
)
```

10 Properties

There are many entities in CMake that can have properties assigned to it. A property is typically well defined and documented by CMake and always applies to a specific entity.

Properties can be set as follows:

```
set_property(<entityType> [entity|entities...] [APPEND|APPEND_STRING] PROPERTY <propertyName>
    <values...>)
```

By default, the new value(s) overwrite the old; `APPEND` appends the new value(s) as a list, and `APPEND_STRING` concatenates the new value(s) as a list. `propertyName` usually corresponds to one of the many predefined CMake propoerties, but can also be a custom one.[13]
Entities are grouped by type:

- `GLOBAL` properties relate to the overall build as a whole.
  Besides the generic `get_property()`, CMake provides `get_cmake_property()` for querying global properties, and some "exclusive" pseudo-properties can be queried: `VARIABLES`, a list of all regular variables; `CACHE_VARIABLES`, a list of all cache variables; `COMMANDS`, a list of all defined commands, functions and macros; `MACROS`, a list of just the defined macros; `COMPONENTS`, a list of all components defined by `install()` commands.

- `DIRECTORY` properties sit somewhere between global properties which apply everywhere and target properties which only affect individual targets. You can optionally supply a `[dirName]` as `entityName`, but by default the current directory is taken.
  These properties can also be get and set in a more concise manner:

```
   set_directory_properties(PROPERTIES <prop1> <val1> [prop2 val2] ...)
   get_directory_property(<resultVar> [DIRECTORY dirName] <property>) # Get the value of a
property from dirName (current directory by default)
   get_directory_property(<resultVar> [DIRECTORY dirName] DEFINITION <varName>) # Get the
       value of a
variable from dirName (current directory by default, but that's not very useful)
```

- `TARGET` properties have a strong and direct influence on how targets are built from source files into binaries.
  These properties can also be get and set in a more concise manner:

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

  To go even more specific, `TARGET` properties each have their own `target_...()` setters, which are strongly recommended because they also set up dependency relationships between targets so that CMake can propagate some properties automatically.

- `SOURCE` properties enable fine-grained manipulation of compiler flags and additional information on how to treat a file on a file-by-file basis rather than for all of a target's sources. They're best avoided, however, because they may have undesirable impacts on the build behavior of a project and may rebuild more than should be necessary when compile options for only a few source files change.[14]
  These properties can also be get and set with more specific commands:

```
   set_source_files_properties(
      <sources...>
      [DIRECTORY dirs...] # Can be used to specify one or more directories in which the source
          properties should be set, with any targets in it aware of those properties
      [TARGET_DIRECTORY targets...] # Treats the specified target(s)'s directory as though
          specified with the above DIRECTORY keyword
      PROPERTIES <propertyName1> <value1> [propertyName2 value2] ...
   )
   get_source_file_property(
      <resultVar> <sourceFile>
      [DIRECTORY dir | TARGET_DIRECTORY target]
      propertyName
   )
```

[13] It's generally a good idea to use a project-specific prefix on the property name to avoid potential name clashes with properties defined by CMake or other third party packages

[14] The Xcode generator also has limitations which prevent it from supporting configuration-specific source file properties.

  The `DIRECTORY` and `TARGET_DIRECTORY` keywords are also available in the generic `set|get_property()` commands with `SOURCE` as `entityType`.
  Because a source file can be compiled into multiple targets, in each of the directory scopes where the source properties are set, the properties should make sense for all targets using those files.

- `CACHE` properties are aimed more at how the cache variables are handled in the CMake GUI(s) rather than affecting the build in any tangible way. The most significant cache variable properties have been discussed before in Section 6.1: `TYPE`, `ADVANCED`, `HELPSTRING`, and `STRINGS` (containing the values for a potential combo box) in case a cache variable's type is `STRING`.
- `INSTALL` properties are specific to the type of packaging being used and are typically not needed by most projects.
- `TEST` properties can also be get and set with more specific commands:

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

The specific `DIRECTORY`, `TARGET`, and `TEST` property setters `set_directory|target|tests_properties()` are more concise versions of the general `set_property()` command that lack the flexibility to `APPEND` or `APPEND_STRING` or to choose a different

Properties can be get as follows:

```
get_property(<resultVar> <entityType> [entityName] PROPERTY <propertyName>
    [DEFINED|SET|BRIEF_DOCS|FULL_DOCS])
```

which will store the property's value in `resultVar` if none of the keywords are given. If `SET` is given, a boolean is stored indicating whether the property has been set to some value; if `DEFINED` is given, a boolean is stored indicating whether the property has been defined, which is not the same as and uncorrelated with it being set; and `BRIEF|FULL_DOCS` retrieve the brief/full documentation string (or `NOTFOUND` if none is set).
Defining a property (doesn't set a value and) is done with:

```
define_property(<entityType> # Property definition happens for an entire type, not a specific entity
   PROPERTY <propertyName>
   [INHERITED] #
   [BRIEF_DOCS <briefDoc> [moreBriefDocs...]] [FULL_DOCS <fullDoc> [moreFullDocs...]] # Likely to
       be deprecated, because not useful
   [INITIALIZE_FROM_VARIABLE <variableName>] # Specify a variable to be used to initialize the
       property (Only for TARGETs)
)
```

If `INHERITED` is given, if a property is not set in the named scope it will fallback to the parent scope(s) recursively up the scope hierarchy until the property is found or the top level is reached. The hierarchy is as follows: `TARGET`/`SOURCE`/`TEST` fallback to the `DIRECTORY` scope hierarchy, which falls back to `GLOBAL`. `CACHE` already chains to the parent variable scope by design. No inheriting occurs when using `set_property()` with `APPEND` or `APPEND_STRING`.

11 Generator Expressions

Generator expressions are, unline the rest of the `CMakeLists.txt` file, evaluated at the generator stage rather than the configure stage. They're useful, for example, to get/indicate the location of a directory that is dependent on the build configuration, e.g. Debug or Release, which developers normally choose when building, long after CMake is finished. They cannot be used everywhere; if a particular command or property supports generator expressions, it is mentioned in the CMake documentation. This set of places expands with most new versions of CMake.

Booleans in generator expressions have to be `0` or `1`. An expression has the form

```
$<0:...> # Evaluates to the empty string
$<1:...> # Evaluates to the expression/string on the ...
$<BOOL:...> # Evaluates to whatever the boolean (variable) on the ... (e.g. TRUE/FALSE, YES/NO,
    etc.) converts to
$<AND:expr[,expr...]> # Evaluates to the AND of the given expressions
$<OR:expr[,expr...]> # Evaluates to the OR of the given expressions
$<NOT:expr> # Evaluates to the inverse of the given expression
$<IF:expr,val1,val0> # Concise version of $<expr:val1>$<$<NOT:expr>:val0>
```

```
$<STREQUAL:string1,string2>
$<EQUAL:number1,number2>
$<VERSION_EQUAL:version1,version2>
$<VERSION_GREATER:version1,version2>
$<VERSION_LESS:version1,version2>
$<CONFIG:arg> # Evaluates to 1 if arg corresponds to the build type being built and 0 for all other
    build types
```

Any target property can be obtained using one of

```
$<TARGET_PROPERTY:target,property> # Provides the value of the named property from the specified
    target
$<TARGET_PROPERTY:property> # Provides the property from the target on which the generator
    expression is being used
```

But there are more direct expressions take care of extracting out parts of some properties or computing values based on raw properties, useful when defining custom build rules for copying files around in post build steps : `TARGET_FILE` yields the absolute path and file name of the target's binary, `TARGET_FILE_NAME` yields only the file name, and `TARGET_FILE_DIR` yields only the path.

Some utility expressions:

```
$<COMMA> # A comma that doesn't interfere with the generator expression syntax itself
$<SEMICOLON>
$<LOWER_CASE:...> # Converts content in ... to lower case, e.g. useful before string comparison
$<UPPER_CASE:...>
$<JOIN:list,...> # Replaces the semicolon in the list with the content in ..., should never be used
    without quoting the expression
$<GENEX_EVAL:...> # Force an evaluation in case the evaluation of a generator expression results in
    content that itself contains generator expressions
```

There are also some generator expressions that provide information, though they should be used with care, since there are likely more robust ways of obtaining the information:

```
$<CONFIG> # Evaluates to the build type
$<PLATFORM_ID> # Evaluates to the platform for which the target is being built, though prefer the
    CMAKE_SYSTEM_NAME variable
$<C|CXX_COMPILER_VERSION>
```

12 Modules

Modules are pre-built files with CMake code that can be included in two ways:

- `include(<fileName|moduleName> [OPTIONAL] [RESULT_VARIABLE <varName>] [NO_POLICY_SCOPE])`

  Already discussed in Section 8 for including subdirectories. When given a `moduleName`, the `include()` command will look for `moduleName.cmake`, first in the list of directories in the `CMAKE_MODULE_PATH` variable, then search in its own internal module directory.[15] A useful pattern is to have your own `cmake` folder with `.cmake` files, and to append that directory to the `CMAKE_MODULE_PATH` variable in the beginning of the top level `CMakeLists.txt` file.

- `find_package(<PackageName> [REQUIRED])`

  This command uses at least one of two methods of searching:

  – `CONFIG` mode is the more reliable, since it looks for a CMake script with CMake targets, variables and commands provided by the package itself. It is usually in the `lib/cmake/<PackageName>/` directory, with the filename being either `<PackageName>Config.cmake` or `<lowercasePackageName>-config.cmake`. A separate optional file named `<PackageName>ConfigVersion.cmake` or `<lowercasePackageName>-config-version.cmake` may also exist in the same directory to determine whether the version of the package satisfies any specified version constraint included.

    If the package isn't in a directory where CMake automatically looks (e.g. `Program Files` on Windows), then

[15] The order of search is flipped when `include`ing inside a file inside CMake internal module directory.

you should either add the base path of the package to the `CMAKE_PREFIX_PATH` list variable, or set this path in the <PackageName>_DIR variable.

- `MODULE` mode is for packages that aren't "CMake-aware" and that don't provide config files. This looks for a File module with the name `FindPackageName.cmake` in the locations specified in the `CMAKE_MODULE_PATH` variable. Such files are not provided by the package itself, but typically maintained independently. CMake also takes the burden of maintaining some of them.[16] For that reason, they can be out-of-date, and are not as reliable. They're mostly heuristic implementations that knows what the package normally provides and how to present that package to the project, including things like imported targets, variables defining locations of relevant files, libraries or programs, information about optional components, version details and so on.

If `REQUIRED`, the package is necessary for the build and will yield an error if not found.

CMake offers many modules. We will discuss a family of modules for checking support for code fragments.

```
include(CheckSourceCompiles)
check_source_compiles(<C|CXX|CUDA|etc> <code> <resultVar> [FAIL_REGEX regexes...] [SRC_EXT
    <extension>])
```

This compiles and links the code fragment in the `code` string variable into an executable and returns true in `resultVar` if succesful and some error string if unsuccesful.[17] A list of regular expressions can be supplied with `regexes`; if the test compilation and linking output matches any of the specified regexes, the check will fail even if the code compiles and links successfully. A file extension can be specified with the optional `extension` parameter, which is only useful in the case of `FORTRAN`, because the file extension affects how compilers treat source files.
A number of variables can be set to influence how the code is compiled: `CMAKE_REQUIRED_FLAGS` is a single string with multiple flags to pass to the compiler command line separated by spaces, `CMAKE_REQUIRED_DEFINITIONS` is a list of compiler definitions, `CMAKE_REQUIRED_INCLUDES` is a list of directories to search for headers, `CMAKE_REQUIRED_LIBRARIES` is a list of libraries to add to the linking stage, `CMAKE_REQUIRED_LINK_OPTIONS` is a list of options to be passed to the linker if building an executable or to the archiver if building a static library, and `CMAKE_REQUIRED_QUIET` indicates whether to print status messages. The state of the set of these `CMAKE_REQUIRED_...` variables can be saved and restored by pushing and popping them to and from a virtual stack, handy when 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:

```
include(CMakePushCheckState)
cmake_push_check_state([RESET]) # Starts a new virtual variable scope for just the
    CMAKE_REQUIRED_... variables
cmake_pop_check_state() # Discards the current values of the CMAKE_REQUIRED_... variables and
    restores them to the previous stack level's values
cmake_reset_check_state() # Clears all the CMAKE_REQUIRED_... variables for convenience (RESET
    keyword of cmake_push_check_state() does the same)
```

```
include(CheckSourceRuns)
check_source_runs((<C|CXX|CUDA|etc> <code> <resultVar> [SRC_EXT extension])
```

This additionally checks whether the given code runs, with the exit code of the executable indicating success (an unsuccessful built is also a failure).

```
include(CheckCompilerFlag)
check_compiler_flag(<C|CXX|CUDA|etc> <flag> <resultVar>)
```

A wrapper command that update the `CMAKE_REQUIRED_DEFINITIONS` variable internally to include `flag` in a call to `check_source_compiles()` with a trivial test file and an internal set of failure regexes that test for a diagnostic message being issued or not. The result of the call will be a true value if no matching diagnostic (so compiler warnings also yield failures). This command also assumes that any flags already present in the relevant CMAKE_<LANG>_FLAGS variables do not themselves generate any compiler warnings.

```
include(CheckLinkerFlag)
check_linker_flag(<C|CXX|CUDA|etc> <flag> <resultVar>)
```

Analogous to `check_compiler_flag()`, except it takes over handling of the `CMAKE_REQUIRED_LINK_OPTIONS` variable.

[16] I think they stopped adding new File modules to CMake.

[17] This result is cached, and subsequent CMake runs will use the cached result rather than perform the test again, even if the code being tested is changed. To force re-evaluation, the variable has to be manually removed from the cache.

```
include(CheckSymbolExists) # For C
check_symbol_exists(<symbol> <headers> <resultVar>)
include(CheckCXXSymbolExists) # For C++
check_cxx_symbol_exists(<symbol> <headers> <resultVar>)
```

Both these modules and commands build a test C/C++ executable and 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. A corresponding #include will be added to the test source code for each header in the headers list. (The symbol being checked will be defined by one of these headers in most cases. If it's about a function or variable provided by a library, that library must be linked using the CMAKE_REQUIRED_LIBRARIES variable.)

13 Policies

The cmake_minimum_required() states that the project expects CMake to behave like the specified version. However, sometimes more fine-grained control (i.e. multiple behaviours) is required. A behaviour change is called a policy. Policies can be set at two different granularity, but they're all set with the cmake_policy() command:

- cmake_policy(VERSION <major.minor>[.patch[.tweak]][...<major.minor>[.patch[.tweak]]) simply changes all policies to the specified version. cmake_minimum_required() implicitly calls this function. The two are largely interchangeable (except that cmake_minimum_required() is obligatory). As you can see, you can also specify a range, in which case the version must be at least the minimum and the behavior should be the lesser of the specified maximum and the running version. This command resets the state of all individually-set policies; i.e. all policies of the current version or earlier are set to NEW.

- cmake_policy(SET CMP<xxxx> NEW|OLD) specifies whether to use the old or new behaviour of the specified policy (i.e. behavior change), where the policy is specified with a four-digit identifier. Here is a list of all policies and their ID. You can check whether your version of CMake knows about a certain policy with a specialized form if if(), namely if(POLICY CMP0055) (and then set the policy inside the body).
  You can get the current state of a policy with cmake_policy(GET CMP<xxxx> <outVar>), with the result being OLD or NEW (or empty if policy is unknown).
  CMake provides a policy stack which can be used to push (cmake_policy(PUSH) and pop (cmake_policy(POP) the current state of all policies. The two commands isolate any cmake_policy() changes to the demarcated portion (but be careful with return() statements). Some commands, like add_subdirectory(), include() and find_package(), implicitly push a new policy state onto the stack and pop it again at a well defined point later.

It is recommended to work with policies at the CMake version level rather than manipulating specific policies.

14 Debugging

The general form of the message() command is

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

where multiple messages will be joined into a single string with no separators, and where mode specifies the type of message. In order of importance:

- Printed to stderr (implies a problem or something worth investigating):
  1. FATAL_ERROR; denotes a hard error. Processing will stop immediately after and the log will record the location of the message() command.
  2. SEND_ERROR; denotes a hard error. Processing will finish the configure stage, but stops before generation stage. (Prefer FATAL_ERROR.)
  3. WARNING; denotes a warning. Processing will continue and the log will record the location of the message() command.
  4. AUTHOR_WARNING; denotes a warning, but only shown if developer warnings are enabled (which is disabled with the -Wno-dev option). Usually only by CMake itself.
  5. DEPRECATION; denotes a depracation. Treated as an error is CMAKE_ERROR_DEPRECATED, and as a warning if CMAKE_WARN_DEPRECATED is true.

6. NOTICE; the default log level (which actually kind of sucks for regular printing; STATUS is preferred for that).

- Printed to stdout:
  7. STATUS; denotes concise status information.
  8. VERBOSE; denotes more detailed information.
  9. DEBUG; denotes a debug message not intended for project users, but rather for developers working on the project itself.
  10. TRACE; denotes very low level details, used almost exclusively for temporary messages during project development.

Instead of mode, you can also specify a checkState message for checking stuff: CHECK_START, CHECK_PASS or CHECK_FAIL. These messages have STATUS mode, and upon completion (CHECK_PASS or CHECK_FAIL), the CHECK_START message is repeated, which is useful for nested checks. The --log-level command line option allows specifying a minimal logging level. The CMAKE_MESSAGE_INDENT variable allows storing a string (prefarably only whitespace) that is prepended to each line of message() output, which can be used to structure it. If the --log-context command line option is given and the CMAKE_MESSAGE_CONTEXT (list) variable is not empty, then `[<context1>.<context2>....]` is prepended to each line of message() output. It is recommended to only append to the CMAKE_MESSAGE_INDENT and CMAKE_MESSAGE_CONTEXT variables with list(APPEND), but never to set them.

The CMakePrintHelpers provides two macros for quickly logging properties and variables:

```
include(CMakePrintHelpers)

cmake_print_properties(
   [TARGETS <target1> [target2...]]
   [SOURCES <source1> [source2...]]
   [DIRECTORIES <dir1> [dir2...]]
   [TESTS <test1> [test2...]]
   [CACHE_ENTRIES <var1> [var2...]]
   PROPERTIES <property1> [property2...]
)

cmake_print_variables(<var1> [var2...])
```

You can also log all read and write attempts of a variable: `variable_watch(<varName> [command])`, where you can optionally limit the logging to one CMake function or macro.

If you want to debug generator expressions, you have to write them to a file, because those are only evaluated at the generator stage:

```
file(GENERATE OUTPUT <file>.txt CONTENT "${<genex>}\n")
```

Lastly, CMake supports profiling performance of the configure stage, which, when the command line options `--profiling-output=<fil` and --profiling-format=google-trace are set, outputs a file that can be loaded into a Chrome web browser or some IDEs (e.g. Qt Creator).

15 vcpkg

vcpkg is a cross-platform open source package manager by Microsoft. The command-line utility is currently available on Windows, macOS and Linux. It's arguably the most popular package manager for C/C++ projects, with the biggest number of supported packages.

To use vcpkg, clone the repository, preferably as a Git submodule to an existing project:

```
git clone https://github.com/Microsoft/vcpkg.git
```

Then run the bootstrap script to build the vcpkg binary:

```
./vcpkg/bootstrap-vcpkg.sh # Linux
.\vcpkg\bootstrap-vcpkg.bat # Windows
```

This binary can now be used to install packages (which vcpkg also calls ports) in two ways:

- Classic mode: `vcpkg install <packageName>[feature1,feature2:triplet]`
  This installs packages individually. The `triplet` is a shorthand to specify the target environment (CPU, OS, compiler, runtime, etc.). It is important to specify the triplet, even though it's optional, because else it will pick some system-specific default. vcpkg defines many triplets (run `vcpkg help triplet` for a list), but you'll likely need `x64-windows` or `x64-linux`.
  Conceptually, this method associates installed packages with the vcpkg installation

- Manifest mode, which instead associated packages with individual projects. The set of installed packages is controlled by the project's manifest file, called `vcpkg.json`, and packages are installed somewhere in the project's directory. `vcpkg install` then takes no arguments, and instead (re)installs all packages in the manifest file. Here is a complete reference. An example:

```
{
  "name": "app-name", # Can only be lowercase letters, digits, and hyphens
  "version": "1.0", # There's also "version-date" and "version-string" for non-orderable
      versions
  "dependencies": [ # Lists all the dependencies. You can specify a dependency in two ways:
    "boost-system", # With a single string containing the name...
    { # ...or as an object
      "name": "ffmpeg",
      "default-features": false, # You don't want the author-prescribed set of features
      "features": [ "mp3lame" ] # Instead, you only want the MP3 encoding feature
      "version>=": 5.1.2 # Should be at least version 5.1.2
    }
  ]
}
```

  Features are components of a package. Packages and their components can be searched using `vcpkg search <package_name>`. You can also browse packages online, but that doesn't show features for some reason.

  A project can also specify its own set of (optional, modular) features (which can in turn be used by other projects to specify which features they need from our project) in the `"features"` list, together with what the author deems are the set of features most users will use in the `"default-features"` list:

```
{
  ...
  "default-features": [ "cbor", "json" ],
  "features": {
    "cbor": {
      "description": "The CBOR backend", # Required, contrary to the optional package description
      "dependencies": [
        {
          "name": "libdb",
          "default-features": false,
          "features": [ "json" ] # Your project's features can also have dependencies, e.g. your
              own features
        }
      ]
    },
    "json": {
      "description": "The JSON backend",
      "dependencies": [
        "jsoncons"
      ]
    }
  }
}
```

Both methods install the package(s) (by default in `vcpkg/installed/`) and prints the relevant CMake code to include in your `CMakeLists.txt` to the console log, e.g.:

```
The package <package>:x64-windows provides CMake targets:

    find_package(<PackageName> CONFIG REQUIRED)
    target_link_libraries(main PRIVATE <PackageName::SomeThing)
```

where the `PackageName::SomeThing` syntax is that of an imported target.