To modify or create a variable, use the `set(ENV{<variable>} <value>)` command, like so:

```
set(ENV{CXX} "clang++")
```

To clear an environment variable, use `unset(ENV{<variable>})`, like so:

```
unset(ENV{VERBOSE})
```

Be aware that there are a few environment variables that affect different aspects of CMake behavior. The `CXX` variable is one of them – it specifies what executable will be used for compiling C++ files. We'll cover other environmental variables, as they will become relevant for this book. A full list is available in the documentation:

`https://cmake.org/cmake/help/latest/manual/cmake-env-variables.7.html`

If you use `ENV` variables as arguments to your commands, the values will be interpolated during the generation of the buildsystem. This means that they will get baked into the build tree, and changing the environment for the build stage won't have any effect.

For example, take the following project file:

chapter02/03-environment/CMakeLists.txt

```
cmake_minimum_required(VERSION 3.20.0)
project(Environment)

message("generated with " $ENV{myenv})
add_custom_target(EchoEnv ALL COMMAND echo "myenv in build
  is" $ENV{myenv})
```

The preceding example has two steps: it will print the `myenv` environment variable during the configuration, and it will add a build stage through `add_custom_target()`, which echoes the same variable as part of the build process. We can test what happens with a bash script that uses one value for the configuration stage and another for the build stage:

chapter02/03-environment/build.sh

```
#!/bin/bash
export myenv=first
echo myenv is now $myenv
```

```
cmake -B build .
cd build
export myenv=second
echo myenv is now $myenv
cmake --build .
```

Running the preceding code clearly shows that the value set during the configuration is persisted to the generated buildsystem:

```
$ ./build.sh | grep -v "\-\-"
myenv is now first
generated with first
myenv is now second
Scanning dependencies of target EchoEnv
myenv in build is first
Built target EchoEnv
```

Using the cache variables

We first mentioned cache variables when discussing command-line options for `cmake` in *Chapter 1*, *First Steps with CMake*. Essentially, they're persistent variables stored in a `CMakeCache.txt` file in your build tree. They contain information gathered during the project configuration stage, both from the system (path to compilers, linkers, tools, and others) and from the user through the **GUI**. Cache variables are not available in *scripts* (since there's no `CMakeCache.txt` file) – they only exist in *projects*.

Cache variables can be referenced with the `$CACHE{<name>}` syntax.

To set a cache variable, use `set()` with the following syntax:

```
set(<variable> <value> CACHE <type> <docstring> [FORCE])
```

As you can see, there are some new required arguments (in comparison to the `set()` command for normal variables), and it also introduces first keywords: `CACHE` and `FORCE`.

Specifying `CACHE` as a `set()` argument means that we intend to change what was provided during the configuration stage, and it imposes a requirement to provide the variable `<type>` and `<docstring>` values. This is because these variables are configurable by the user and the GUI needs to know how to display it. The following types are accepted:

- `BOOL`: A Boolean on/off value. The GUI will show a checkbox.
- `FILEPATH`: A path to a file on a disk. The GUI will open a file dialog.
- `PATH`: A path to a directory on a disk. The GUI will open a directory dialog.
- `STRING`: A line of text. The GUI offers a text field to be filled. It can be replaced by a drop-down control by calling `set_property(CACHE <variable> STRINGS <values>)`.
- `INTERNAL`: A line of text. The GUI skips internal entries. The internal entries may be used to store variables persistently across runs. Use of this type implicitly adds the `FORCE` keyword.

The `<doctring>` value is simply a label that will be displayed by the GUI next to the field to provide more detail about this setting to the user. It is required even for an `INTERNAL` type.

Setting cache variables follows the same rules as environmental variables to some extent – values are overwritten only for the current execution of CMake. Take a look at this example:

```
set(FOO "BAR" CACHE STRING "interesting value")
```

The above call has no permanent effect if the variable exists in the cache. However, if the value doesn't exist in cache or an optional `FORCE` argument is specified, the value will be persisted:

```
set(FOO "BAR" CACHE STRING "interesting value" FORCE)
```

Setting the cache variables has some unobvious implications. That is, any normal variable with the same name will be removed. We'll find out why in the next section.

As a reminder, cache variables can be managed from the command line as well (check the appropriate section in *Chapter 1*, *First Steps with CMake*).

How to correctly use the variable scope in CMake

Variable scope is probably the hardest part of the whole concept of the CMake Language. This is maybe because we're so accustomed to how things are done in more advanced languages that support namespaces and scope operators. CMake doesn't have those mechanisms, so it deals with this issue in its own, somewhat unusual way.

Just to clarify, variable scopes as a general concept are meant to separate different layers of abstraction so that when a user-defined function is called, variables set in that function are local to it. These *local* variables aren't affecting the *global* scope, even if the names of the local variables are exactly the same as the global ones. If explicitly needed, functions should have read/write access to global variables as well. This separation of variables (or scopes) has to work on many levels – when one function calls another, the same separation rules apply.

CMake has two scopes:

- **Function scope**: For when custom functions defined with `function()` are executed
- **Directory scope**: For when a `CMakeLists.txt` listfile in a nested directory is executed from the `add_subdirectory()` command

We'll cover the preceding commands later in this book, but first, we need to know how the concept of variable scope is implemented. When a nested scope is created, CMake simply fills it with copies of all the variables from the current scope. Subsequent commands will affect these copies. But as soon as the execution of the nested scope is completed, all copies are deleted and the original, parent scope is restored.

Let's consider the following scenario:

1. The parent scope sets the `VAR` variable to `ONE`.
2. The nested scope starts and `VAR` is printed to console.
3. The `VAR` variable is set to `TWO`, and `VAR` is printed to the console.
4. The nested scope ends, and `VAR` is printed to the console.

The console's output will look like this: `ONE`, `TWO`, `ONE`. This is because the copied `VAR` variable is discarded after the nested scope ends.

How the concept of scope works in CMake has interesting implications that aren't that common in other languages. If you unset (`unset()`) a variable created in the parent scope while executing in a nested scope, it will disappear, but only in the nested scope. When the nested scope is completed, the variable is restored to its previous value.

This brings us to the behavior of variable referencing and the `${}` syntax. Whenever we try to access the normal variable, CMake will search for the variables from the current scope, and if the variable with such a name is defined, it will return its value. So far, so good. However, when CMake can't find a variable with that name (for example, if it didn't exist or was unset (`unset()`)), it will search through the cache variables and return a value from there if a match is found.

That's a possible gotcha if we have a nested scope calling `unset()`. Depending on where we reference that variable – in the inner or the outer scope – we'll be accessing the cache or the original value.

But what can we do if we really need to change the variable in the calling (parent) scope? CMake has a `PARENT_SCOPE` flag you can add at the end of the `set()` and `unset()` commands:

```
set(MyVariable "New Value" PARENT_SCOPE)
unset(MyVariable PARENT_SCOPE)
```

That workaround is a bit limited, as it doesn't allow accessing variables more than one level up. Another thing worth noting is the fact that using `PARENT_SCOPE` doesn't change variables in the current scope.

Let's see how variable scope works in practice and consider the following example:

---

chapter02/04-scope/CMakeLists.txt

```
function(Inner)
  message("  > Inner: ${V}")
  set(V 3)
  message("  < Inner: ${V}")
endfunction()

function(Outer)
  message(" > Outer: ${V}")
  set(V 2)
  Inner()
  message(" < Outer: ${V}")
endfunction()

set(V 1)
message("> Global: ${V}")
```