3.6.1. Regular variables

3.6.1.1. Regular vs cache

对于一个变量，搞懂其作用域以及生命周期就可以了

Unlike cache variables regular (normal) CMake variables have scope and don't outlive CMake runs.

If in the next example you run the CMake configure step twice, without removing the cache:

```
cmake_minimum_required(VERSION 2.8)
project(foo NONE)

message("Regular variable (before): ${abc}")
message("Cache variable (before): ${xyz}")

set(abc "123")
set(xyz "321" CACHE STRING "")

message("Regular variable (after): ${abc}")
message("Cache variable (after): ${xyz}")
```

```
[usage-of-variables]> rm -rf _builds

[usage-of-variables]> cmake -Hcache-vs-regular -B_builds
Regular variable (before):
Cache variable (before):
Regular variable (after): 123
Cache variable (after): 321
-- Configuring done
-- Generating done
-- Build files have been written to: /.../usage-of-variables/_builds

[usage-of-variables]> cmake -Hcache-vs-regular -B_builds
Regular variable (before):
Cache variable (before): 321
Regular variable (after): 123
Cache variable (after): 321
-- Configuring done
-- Generating done
-- Build files have been written to: /.../usage-of-variables/_builds
```

第二次运行的时候，cache变量是被cache起来的

You can see that the regular CMake variable `abc` is created from scratch each time

```
[usage-of-variables]> rm -rf _builds

[usage-of-variables]> cmake -Hcache-vs-regular -B_builds
Regular variable (before):
Cache variable (before):
Regular variable (after): 123
Cache variable (after): 321
-- Configuring done
-- Generating done
-- Build files have been written to: /.../usage-of-variables/_builds

[usage-of-variables]> cmake -Hcache-vs-regular -B_builds
Regular variable (before):
Cache variable (before): 321
Regular variable (after): 123
Cache variable (after): 321
-- Configuring done
-- Generating done
-- Build files have been written to: /.../usage-of-variables/_builds
```

And the cache variable `xyz` is created only once and reused on second run

```
[usage-of-variables]> rm -rf _builds

[usage-of-variables]> cmake -Hcache-vs-regular -B_builds
Regular variable (before):
Cache variable (before):
Regular variable (after): 123
Cache variable (after): 321
-- Configuring done
-- Generating done
-- Build files have been written to: /.../usage-of-variables/_builds

[usage-of-variables]> cmake -Hcache-vs-regular -B_builds
Regular variable (before):
Cache variable (before): 321
Regular variable (after): 123
Cache variable (after): 321
-- Configuring done
-- Generating done
-- Build files have been written to: /.../usage-of-variables/_builds
```

You can find cache variable `xyz` in CMakeCache.txt:

```
[usage-of-variables]> grep xyz _builds/CMakeCache.txt
xyz:STRING=321
```

Unlike regular `abc`:

```
[usage-of-variables]> grep abc _builds/CMakeCache.txt
[usage-of-variables]> echo $?
1
```

3.6.1.2. Scope of variable

Each variable is linked to the scope where it was defined. Commands add_subdirectory and function introduce their own scopes:

```
# Top level CMakeLists.txt

cmake_minimum_required(VERSION 2.8)
project(foo NONE)

set(abc "123")

message("Top level scope (before): ${abc}")

add_subdirectory(boo)

message("Top level scope (after): ${abc}")
```

```
# CMakeLists.txt from 'boo' directory

set(abc "456")

message("Directory 'boo' scope: ${abc}")
```

There are two variables `abc` defined. One in top level scope and another in scope of `boo` directory:

```
[usage-of-variables]> rm -rf _builds
[usage-of-variables]> cmake -Hdirectory-scope -B_builds
Top level scope (before): 123
Directory 'boo' scope: 456
Top level scope (after): 123
-- Configuring done
-- Generating done
-- Build files have been written to: /.../usage-of-variables/_builds
```

3.6.1.3. New scope

When a new scope is created it will be initialized with the variables of the parent scope. Command unset can remove a variable from the current scope. If a variable is not found in the current scope it will be dereferenced to an empty string:

```
cmake_minimum_required(VERSION 2.8)
project(foo NONE)

function(foo)
  message("[foo]: Scope for function 'foo' copied from parent 'boo': { abc = '${abc}', xyz =
'${xyz}' }")
  unset(abc)
  message("[foo]: Command 'unset(abc)' will remove variable from current scope: { abc =
'${abc}', xyz = '${xyz}' }")
endfunction()

function(boo)
  message("[boo]: Scope for function 'boo' copied from parent: { abc = '${abc}', xyz = '${xyz}'
}")
  set(abc "789")
  message("[boo]: Command 'set(abc ...)' modify current scope, state: { abc = '${abc}', xyz =
'${xyz}' }")
  foo()
endfunction()

set(abc "123")
set(xyz "456")

message("Top level scope state: { abc = '${abc}', xyz = '${xyz}' }")

boo()
```

```
[usage-of-variables]> rm -rf _builds
[usage-of-variables]> cmake -Htake-from-parent-scope -B_builds
Top level scope state: { abc = '123', xyz = '456' }
[boo]: Scope for function 'boo' copied from parent: { abc = '123', xyz = '456' }
[boo]: Command 'set(abc ...)' modify current scope, state: { abc = '789', xyz = '456' }
[foo]: Scope for function 'foo' copied from parent 'boo': { abc = '789', xyz = '456' }
[foo]: Command 'unset(abc)' will remove variable from current scope: { abc = '', xyz = '456' }
-- Configuring done
-- Generating done
-- Build files have been written to: /.../usage-of-variables/_builds
```

3.6.1.4. Same scope

`include` and `macro` don't introduce a new scope, so commands like `set` and `unset` will affect the current scope:

```
# Top level CMakeLists.txt

cmake_minimum_required(VERSION 2.8)
project(foo NONE)

set(abc "123")

message("abc (before): ${abc}")
include("./modify-abc.cmake")
message("abc (after): ${abc}")

macro(modify_xyz)
  set(xyz "789")
endmacro()

set(xyz "336")

message("xyz (before): ${xyz}")
modify_xyz()
message("xyz (after): ${xyz}")
```

```
# modify-abc.cmake module

set(abc "456")
```

```
[usage-of-variables]> rm -rf _builds
[usage-of-variables]> cmake -Hsame-scope -B_builds
abc (before): 123
abc (after): 456
xyz (before): 336
xyz (after): 789
-- Configuring done
-- Generating done
-- Build files have been written to: /.../usage-of-variables/_builds
```

3.6.1.5. Parent scope

A variable can be set to the parent scope by specifying `PARENT_SCOPE`:

```
cmake_minimum_required(VERSION 2.8)
project(foo NONE)

set(abc "") # clear

function(scope_2)
  message("Scope 2 (before): '${abc}'")
  set(abc "786" PARENT_SCOPE)
  message("Scope 2 (after): '${abc}'")
endfunction()

function(scope_1)
  message("Scope 1 (before): '${abc}'")
  scope_2()
  message("Scope 1 (after): '${abc}'")
endfunction()

message("Top level (before): '${abc}'")
scope_1()
message("Top level (after): '${abc}'")
```

Variable **will only be set to parent** scope:

```
[usage-of-variables]> rm -rf _builds
[usage-of-variables]> cmake -Hparent-scope -B_builds
Top level (before): ''
Scope 1 (before): ''
Scope 2 (before): ''
Scope 2 (after): ''
Scope 1 (after): '786'
Top level (after): ''
-- Configuring done
-- Generating done
-- Build files have been written to: /.../usage-of-variables/_builds
```

Current scope will not be affected:

```
[usage-of-variables]> rm -rf _builds
[usage-of-variables]> cmake -Hparent-scope -B_builds
Top level (before): ''
Scope 1 (before): ''
Scope 2 (before): ''
Scope 2 (after): ''
Scope 1 (after): '786'
Top level (after): ''
-- Configuring done
-- Generating done
-- Build files have been written to: /.../usage-of-variables/_builds
```

As well as parent of the parent:

```
[usage-of-variables]> rm -rf _builds
[usage-of-variables]> cmake -Hparent-scope -B_builds
Top level (before): ''
Scope 1 (before): ''
Scope 2 (before): ''
Scope 2 (after): ''
Scope 1 (after): '786'
Top level (after): ''
-- Configuring done
-- Generating done
-- Build files have been written to: /.../usage-of-variables/_builds
```

3.6.1.6. From cache

如果一个变量在当前的作用域中没有找到，那么他将在cache中去查找；

Q：如果在cache中也没有找到怎么办？

If variable is not found in the current scope, it will be taken from the cache:

```
cmake_minimum_required(VERSION 2.8)
project(foo NONE)

set(a "789" CACHE STRING "")
set(a "123")

message("Regular variable from current scope: ${a}")

unset(a)

message("Cache variable if regular not found: ${a}")
```

```
[usage-of-variables]> rm -rf _builds
[usage-of-variables]> cmake -Hfrom-cache -B_builds
Regular variable from current scope: 123
Cache variable if regular not found: 789
-- Configuring done
-- Generating done
-- Build files have been written to: /.../usage-of-variables/_builds
```

3.6.1.7. Cache unset regular

Note that the order of commands is important because `set(... CACHE ...)` will remove the regular variable with the same name from current scope:

和加载的顺序相关联

```
cmake_minimum_required(VERSION 2.8)
project(foo NONE)

set(a "123")
set(a "789" CACHE STRING "")

message("Regular variable unset, take from cache: ${a}")
```

从cache中对a变量的加载，将会remove之前设置的值

```
[usage-of-variables]> rm -rf _builds
[usage-of-variables]> cmake -Hcache-remove-regular -B_builds
Regular variable unset, take from cache: 789
-- Configuring done
-- Generating done
-- Build files have been written to: /.../usage-of-variables/_builds
```

3.6.1.8. Confusing

This may lead to a quite confusing behavior:

cmake中定义函数的方式

很显然，这个是不带参数的函数定义

一定函数的定义将会产生一个新的scope

Q：如何定义带参数的函数？

```
cmake_minimum_required(VERSION 2.8)
project(foo NONE)

function(set_abc_globally)
  message("Function scope before cache modify = ${abc}")
  set(abc "789" CACHE STRING "")
  message("Function scope after cache modify = ${abc}")
endfunction()

set(abc "123")

set_abc_globally()

message("Parent scope is not affected, take variable from current scope, not cache = ${abc}")
```

难道cmake中 child scope 无法 修改 father scope中的变量？

set(abc "789" CACHE STRING "") 意思应该是：如果cache变量a的值为""，那么就设置cache变量a的值为789；显然地，当cache变量a的值不是""的时候，自然地，不会设置变量a的值为789了

In this example `set(... CACHE ...)` will remove `abc` only from scope of function and not from top level scope:

```
[usage-of-variables]> rm -rf _builds
[usage-of-variables]> cmake -Hcache-confuse -B_builds
Function scope before cache modify = 123
Function scope after cache modify = 789
Parent scope is not affected, take variable from current scope, not cache = 123
-- Configuring done
-- Generating done
-- build files have been written to: /.../usage-of-variables/_builds
```

This will be even more confusing if you run this example one more time without removing cache:

```
[usage-of-variables]> cmake -Hcache-confuse -B_builds
Function scope before cache modify = 123
Function scope after cache modify = 123
Parent scope is not affected, take variable from current scope, not cache = 123
-- Configuring done
-- Generating done
-- Build files have been written to: /.../usage-of-variables/_builds
```

Since variable `abc` already stored in cache command `set(... CACHE ...)` has no effect and **will not remove** regular `abc` from scope of function.

3.6.1.9. Names

Variable names are case-sensitive:

```
cmake_minimum_required(VERSION 2.8)
project(foo NONE)

set(a "123")
set(b "567")
set(aBc "333")

set(A "321")
set(B "765")
set(ABc "777")

message("a: ${a}")
message("b: ${b}")
message("aBc: ${aBc}")

message("A: ${A}")
message("B: ${B}")
message("ABc: ${ABc}")
```

```
[usage-of-variables]> rm -rf _builds
[usage-of-variables]> cmake -Hcase-sensitive -B_builds
a: 123
b: 567
aBc: 333
A: 321
B: 765
ABc: 777
-- Configuring done
-- Generating done
-- Build files have been written to: /.../usage-of-variables/_builds
```

Name of variable may consist of **any** characters:

cmake中变量的名字区分大小写以及，可以有任意字符来组成

```
cmake_minimum_required(VERSION 2.8)
project(foo NONE)

set("abc" "123")
set("ab c" "456")
set("ab?c" "789")
set("/usr/bin/bash" "987")
set("C:\\Program Files\\" "654")
set(" " "321")

function(print_name varname)
  message("Variable name: '${varname}', value: '${${varname}}'")
endfunction()

print_name("abc")
print_name("ab c")
print_name("ab?c")
print_name("/usr/bin/bash")
print_name("C:\\Program Files\\")
print_name(" ")
```

这里的变量定义是引用起来的；
而之前的定义则是 set
(abc,123)

```
[usage-of-variables]> rm -rf _builds
[usage-of-variables]> cmake -Hany-names -B_builds
Variable name: 'abc', value: '123'
Variable name: 'ab c', value: '456'
Variable name: 'ab?c', value: '789'
Variable name: '/usr/bin/bash', value: '987'
Variable name: 'C:\Program Files\', value: '654'
Variable name: ' ', value: '321'
-- Configuring done
-- Generating done
-- Build files have been written to: /.../usage-of-variables/_builds
```

3.6.1.10. Quotes

In the previous example, the quote character `"` was used to create a name containing a space - this is called *quoted argument*. Note that the argument must start and end with a quote character, otherwise it becomes an *unquoted argument*. In this case, the quote character will be treated as part of the string:

```
cmake_minimum_required(VERSION 2.8)
project(foo NONE)

set(a "Quoted argument")
set(b x-"Unquoted argument")
set(c x"a;b;c")

message("a = '${a}'")
message("b = '${b}'")

message("c =")
foreach(x ${c})
  message("  '${x}'")
endforeach()
```

c变量中的分号有什么作用
c好像是一个list变量

```
[usage-of-variables]> rm -rf _builds
[usage-of-variables]> cmake -Hquotes -B_builds
a = 'Quoted argument'
b = 'x-"Unquoted argument"'
c =
  'x"a'
  'b'
  'c"'
-- Configuring done
-- Generating done
-- Build files have been written to: /.../usage-of-variables/_builds
```

As you can see the variable `b` contains quotes now and for list `c` quotes are part of the elements: `x"a` , `c"` .

CMake documentation

- Quoted argument
- Unquoted argument

3.6.1.11. Dereferencing

Dereferenced variable can be used in creation of new variable:

```
cmake_minimum_required(VERSION 2.8)
project(foo NONE)

set(a "xyz")

set(b "${a}_321")
set(${a}_1 "456")
set(variable_${a} "${a} + ${b} + 155")

message("b: '${b}'")
message("xyz_1: '${xyz_1}'")
message("variable_xyz: '${variable_xyz}'")
```

```
[usage-of-variables]> rm -rf _builds
[usage-of-variables]> cmake -Hdereference -B_builds
b: 'xyz_321'
xyz_1: '456'
variable_xyz: 'xyz + xyz_321 + 155'
-- Configuring done
-- Generating done
-- Build files have been written to: /.../usage-of-variables/_builds
```

Or new variable name:

```
cmake_minimum_required(VERSION 2.8)
project(foo NONE)

set(a "xyz")

set(b "${a}_321")
set(${a}_1 "456")
set(variable_${a} "${a} + ${b} + 155")

message("b: '${b}'")
message("xyz_1: '${xyz_1}'")
message("variable_xyz: '${variable_xyz}'")
```

```
[usage-of-variables]> rm -rf _builds
[usage-of-variables]> cmake -Hdereference -B_builds
b: 'xyz_321'
xyz_1: '456'
variable_xyz: 'xyz + xyz_321 + 155'
-- Configuring done
-- Generating done
-- Build files have been written to: /.../usage-of-variables/_builds
```

Or even both:

```
cmake_minimum_required(VERSION 2.8)
project(foo NONE)

set(a "xyz")

set(b "${a}_321")
set(${a}_1 "456")
set(variable_${a} "${a} + ${b} + 155")

message("b: '${b}'")
message("xyz_1: '${xyz_1}'")
message("variable_xyz: '${variable_xyz}'")
```

```
[usage-of-variables]> rm -rf _builds
[usage-of-variables]> cmake -Hdereference -B_builds
b: 'xyz_321'
xyz_1: '456'
variable_xyz: 'xyz + xyz_321 + 155'
-- Configuring done
-- Generating done
-- Build files have been written to: /.../usage-of-variables/_builds
```

3.6.1.12. Nested dereferencing

Dereferencing of variable by `${...}` will happen as many times as needed:

```
cmake_minimum_required(VERSION 2.8)
project(foo)

foreach(lang C CXX)
  message("Compiler for language ${lang}: ${CMAKE_${lang}_COMPILER}")
  foreach(build_type DEBUG RELEASE RELWITHDEBINFO MINSIZEREL)
    message("Flags for language ${lang} + build type ${build_type}:
${CMAKE_${lang}_FLAGS_${build_type}}")
  endforeach()
endforeach()
```

可以进行内嵌

```
[usage-of-variables]> rm -rf _builds
[usage-of-variables]> cmake -Hnested-dereference -B_builds
-- The C compiler identification is GNU 4.8.4
-- The CXX compiler identification is GNU 4.8.4
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
Compiler for language C: /usr/bin/cc
Flags for language C + build type DEBUG: -g
Flags for language C + build type RELEASE: -O3 -DNDEBUG
Flags for language C + build type RELWITHDEBINFO: -O2 -g -DNDEBUG
Flags for language C + build type MINSIZEREL: -Os -DNDEBUG
Compiler for language CXX: /usr/bin/c++
Flags for language CXX + build type DEBUG: -g
Flags for language CXX + build type RELEASE: -O3 -DNDEBUG
Flags for language CXX + build type RELWITHDEBINFO: -O2 -g -DNDEBUG
Flags for language CXX + build type MINSIZEREL: -Os -DNDEBUG
-- Configuring done
-- Generating done
-- Build files have been written to: /.../usage-of-variables/_builds
```

3.6.1.13. Types of variable

Variables always have type string but some commands can interpret them differently. For example the command `if` can treat strings as boolean, path, target name, etc.:

```
cmake_minimum_required(VERSION 2.8)
project(foo)

set(condition_a "TRUE")
set(condition_b "NO")

set(path_to_this "${CMAKE_CURRENT_SOURCE_DIR}/CMakeLists.txt")

set(target_name foo)
add_library("${target_name}" foo.cpp)

if(condition_a)
  message("condition_a")
else()
  message("NOT condition_a")
endif()

if(condition_b)
  message("condition_b")
else()
  message("NOT condition_b")
endif()

if(EXISTS "${path_to_this}")
  message("File exists: ${path_to_this}")
else()
  message("File not exist: ${path_to_this}")
endif()

if(TARGET "${target_name}")
  message("Target exists: ${target_name}")
else()
  message("Target not exist: ${target_name}")
endif()
```

```
[usage-of-variables]> rm -rf _builds
[usage-of-variables]> cmake -Htypes-of-variable -B_builds
-- The C compiler identification is GNU 4.8.4
-- The CXX compiler identification is GNU 4.8.4
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
condition_a
NOT condition_b
File exists: /.../usage-of-variables/types-of-variable/CMakeLists.txt
Target exists: foo
-- Configuring done
-- Generating done
-- Build files have been written to: /.../usage-of-variables/_builds
```

CMake documentation

- if

3.6.1.14. Create list

Some commands can treat a variable as list. In this case the string value is split into elements separated by `;`. The command `set` can create such lists:

```
cmake_minimum_required(VERSION 2.8)
project(foo NONE)

set(l0 a b c)
set(l1 a;b;c)
set(l2 "a b" "c")
set(l3 "a;b;c")
set(l4 a "b;c")

message("l0 = 'a' + 'b' + 'c' = '${l0}'")
message("l1 = 'a;b;c' = '${l1}'")
message("l2 = 'a b' + 'c' = '${l2}'")
message("l3 = \"'a;b;c'\" = '${l3}'")
message("l4 = 'a' + 'b;c' = '${l4}'")

message("print by message: " ${l3})
message("print by message: " "a" "b" "c")
```

`set` creates **string** from elements and puts the `;` between them:

```
[usage-of-variables]> rm -rf _builds
[usage-of-variables]> cmake -Hlist -B_builds
l0 = 'a' + 'b' + 'c' = 'a;b;c'
l1 = 'a;b;c' = 'a;b;c'
l2 = 'a b' + 'c' = 'a b;c'
l3 = "'a;b;c'" = 'a;b;c'
l4 = 'a' + 'b;c' = 'a;b;c'
print by message: abc
print by message: abc
-- Configuring done
-- Generating done
-- Build files have been written to: /.../usage-of-variables/_builds
```

In case you want to add an element with space you can protect the element with `"`:

```
[usage-of-variables]> rm -rf _builds
[usage-of-variables]> cmake -Hlist -B_builds
l0 = 'a' + 'b' + 'c' = 'a;b;c'
l1 = 'a;b;c' = 'a;b;c'
l2 = 'a b' + 'c' = 'a b;c'
l3 = "'a;b;c'" = 'a;b;c'
l4 = 'a' + 'b;c' = 'a;b;c'
print by message: abc
print by message: abc
-- Configuring done
-- Generating done
-- Build files have been written to: /.../usage-of-variables/_builds
```

As seen with `l4` variable protecting `;` with `"` doesn't have any effect:

```
[usage-of-variables]> rm -rf _builds
[usage-of-variables]> cmake -Hlist -B_builds
l0 = 'a' + 'b' + 'c' = 'a;b;c'
l1 = 'a;b;c' = 'a;b;c'
l2 = 'a b' + 'c' = 'a b;c'
l3 = "'a;b;c'" = 'a;b;c'
l4 = 'a' + 'b;c' = 'a;b;c'
print by message: abc
print by message: abc
-- Configuring done
-- Generating done
-- Build files have been written to: /.../usage-of-variables/_builds
```

We are concatenating **string** `a` with **string** `b;c` and putting `;` between them. Final result is the **string** `a;b;c`. When a command interprets this string as list, such list has 3 elements. Hence **it's not a list** with two elements `a` and `b;c`.

The command `message` interprets `l3` as list with 3 elements, so in the end 4 arguments (value of type string) passed as input: `print by message:_`, `a`, `b`, `c`. Command `message` will concatenate them without any separator, hence string `print by message: abc` will be printed:

```
[usage-of-variables]> rm -rf _builds
[usage-of-variables]> cmake -Hlist -B_builds
l0 = 'a' + 'b' + 'c' = 'a;b;c'
l1 = 'a;b;c' = 'a;b;c'
l2 = 'a b' + 'c' = 'a b;c'
l3 = "'a;b;c'" = 'a;b;c'
l4 = 'a' + 'b;c' = 'a;b;c'
print by message: abc
print by message: abc
-- Configuring done
-- Generating done
-- Build files have been written to: /.../usage-of-variables/_builds
```

CMake documentation

- set

3.6.1.15. Operations with list

The `list` command can be used to calculate length of list, get element by index, remove elements by index, etc.:

```
cmake_minimum_required(VERSION 2.8)
project(foo NONE)

set(l0 "a;b;c")
set(l1 "a" "b;c")
set(l2 "a" "b c")

list(LENGTH l0 l0_len)
list(LENGTH l1 l1_len)
list(LENGTH l2 l2_len)

message("length of '${l0}' (l0) = ${l0_len}")
message("length of '${l1}' (l1) = ${l1_len}")
message("length of '${l2}' (l2) = ${l2_len}")

list(GET l1 2 l1_2)
message("l1[2] = ${l1_2}")

message("Removing first item from l1 list: '${l1}'")
list(REMOVE_AT l1 0)
message("l1 = '${l1}'")
```

```
[usage-of-variables]> rm -rf _builds
[usage-of-variables]> cmake -Hlist-operations -B_builds
length of 'a;b;c' (l0) = 3
length of 'a;b;c' (l1) = 3
length of 'a;b c' (l2) = 2
l1[2] = c
Removing first item from l1 list: 'a;b;c'
l1 = 'b;c'
-- Configuring done
-- Generating done
-- Build files have been written to: /.../usage-of-variables/_builds
```

CMake documentation

- list

3.6.1.16. Recommendation

Use **short laconic lower-case** names (`a` , `i` , `mylist` , `objects` , etc.) for local variables that used **only by the current scope**. Use **long detailed upper-case** names (`FOO_FEATURE` , `BOO_ENABLE_SOMETHING` , etc.) for variables that used by **several scopes**.

For example it make no sense to use long names in function since function has it's own scope:

```
function(foo_something)
  set(FOO_SOMETHING_A 1)
  # ...
endfunction()
```

Using just `a` will be fine:

```
function(foo_something)
  set(a 1)
  # ...
endfunction()
```

Same with scope of CMakeLists.txt:

```
# Foo/CMakeLists.txt

message("Files:")
foreach(FOO_FILES_ITERATOR ${files})
  message("  ${FOO_FILES_ITERATOR}")
endforeach()
```

Prefer instead:

```
# Foo/CMakeLists.txt

message("Files:")
foreach(x ${files})
  message("  ${x}")
endforeach()
```

See also

- Cache names

Compare it with C++ code:

```
// pretty bad idea
#define a

// good one
#define MYPROJECT_ENABLE_A
```

```
// does it make sense?
for (int array_iterator = 0; array_iterator < array.size(); ++array_iterator) {
  // use 'array_iterator'
}

// good one
for (int i = 0; i < array.size(); ++i) {
  // use 'i'
}
```

3.6.1.17. Summary

- All variables have a **string** type

- List is nothing but **string**, elements of list separated by `;`
- The way how variables are interpreted **depends on the command**
- Do not give same names for **cache** and **regular** variables
- `add_subdirectory` and `function` create **new scope**
- `include` and `macro` work in the **current scope**