3.5. Project declaration

Next must-have command is project. Command `project(foo)` will set languages to C and C++ (default), declare some `foo_*` variables and run basic build tool checks.

CMake documentation

- project

3.5.1. Tools discovering

就是说cmake在调用project的时候，去check c以及 c++ 编译器

By default on calling `project` command CMake will try to detect compilers for default languages: C and C++. Let's add some variables and check where they are defined:

```
cmake_minimum_required(VERSION 2.8)

message("Before 'project':")
message("  C: '${CMAKE_C_COMPILER}'")
message("  C++: '${CMAKE_CXX_COMPILER}'")

project(Foo)

message("After 'project':")
message("  C: '${CMAKE_C_COMPILER}'")
message("  C++: '${CMAKE_CXX_COMPILER}'")
```

Examples on GitHub

- Repository
- Latest ZIP

Run test on `Linux`:

```
[project-examples]> rm -rf _builds
[project-examples]> cmake -Hset-compiler -B_builds
Before 'project':
  C: ''
  C++: ''
-- 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
After 'project':
  C: '/usr/bin/cc'
  C++: '/usr/bin/c++'
-- Configuring done
-- Generating done
-- Build files have been written to: /.../project-examples/_builds
```

CMake will run tests for other tools as well, so try to avoid checking of anything before `project`, place all checks **after project declared**.

就是说，如果你要check一些配置项的时候，一定要在PROJECT之后来写这些要进行check的代码

Stackoverflow

- Why CMAKE_COMPILER_IS_GNUCXX and CMAKE_CXX_COMPILER_ID are empty?
- Why CMAKE_SYSTEM_NAME is blank?
- Why MSVC is empty?

Also `project` is a place where toolchain file will be read.

所谓的一些工具链文件，应该就是cmake向cpp传递消息的中间文件，一般是xxxx.cmake文件

```
# CMakeLists.txt

cmake_minimum_required(VERSION 2.8)

message("Before 'project'")

project(Foo)

message("After 'project'")
```

```
# toolchain.cmake

message("Processing toolchain")
```

```
[project-examples]> rm -rf _builds
[project-examples]> cmake -Htoolchain -B_builds -DCMAKE_TOOLCHAIN_FILE=toolchain.cmake
Before 'project'
Processing toolchain
Processing toolchain
-- 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
Processing toolchain
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
Processing toolchain
-- Detecting C compiler ABI info - done
-- Detecting C compile features
Processing toolchain
Processing toolchain
Processing toolchain
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
Processing toolchain
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
Processing toolchain
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
Processing toolchain
Processing toolchain
Processing toolchain
-- Detecting CXX compile features - done
After 'project'
-- Configuring done
-- Generating done
-- Build files have been written to: /.../project-examples/_builds
```

Note

You may notice that toolchain read several times

Stackoverflow

- In which Order are Files parsed (Cache, Toolchain, ...)?

3.5.2. Languages

可以在 PROJECT 之后指定 语言

If you don't have or don't need support for one of the default languages you can set language explicitly after name of the project. This is how to setup C-only project:

```
cmake_minimum_required(VERSION 2.8)

message("Before 'project':")
message("  C: '${CMAKE_C_COMPILER}'")
message("  C++: '${CMAKE_CXX_COMPILER}'")

project(Foo C)

message("After 'project':")
message("  C: '${CMAKE_C_COMPILER}'")
message("  C++: '${CMAKE_CXX_COMPILER}'")
```

There is no checks for C++ compiler and variable with path to C++ compiler is empty now:

```
[project-examples]> rm -rf _builds
[project-examples]> cmake -Hc-compiler -B_builds
Before 'project':
  C: ''
  C++: ''
-- The C 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
After 'project':
  C: '/usr/bin/cc'
  C++: ''
-- Configuring done
-- Generating done
-- Build files have been written to: /.../project-examples/_builds
```

Of course you will not be able to build C++ targets anymore. Since CMake thinks that `*.cpp` extension is for C++ sources (by default) there will be error reported if C++ is not listed (discovering of C++ tools will not be triggered):

```
cmake_minimum_required(VERSION 2.8)
project(Foo C)

add_library(foo foo.cpp)
```

```
[project-examples]> rm -rf _builds
[project-examples]> cmake -Hcpp-not-found -B_builds
-- The C 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
-- Configuring done
CMake Error: Cannot determine link language for target "foo".
CMake Error: CMake can not determine linker language for target: foo
-- Generating done
-- Build files have been written to: /.../project-examples/_builds
```

We can save some time by using special language `NONE` when we don't need any tools at all:

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

No checks for C or C++ compiler as you can see:

```
[project-examples]> rm -rf _builds
[project-examples]> cmake -Hno-language -B_builds
-- Configuring done
-- Generating done
-- Build files have been written to: /.../project-examples/_builds
```

Note

Such form will be used widely in examples in cases when we don't need to build targets.

Note

For CMake 3.0+ sub-option `LANGUAGES` added, since it will be:

```
cmake_minimum_required(VERSION 3.0)
project(foo LANGUAGES NONE)
```

Stackoverflow

- Detect project language in cmake

3.5.3. Variables

Command `project` declare `*_{SOURCE,BINARY}_DIR` variables. Since version `3.0` you can add `VERSION` which additionally declare `*_VERSION_{MAJOR,MINOR,PATCH,TWEAK}` variables:

```
cmake_minimum_required(VERSION 3.0)

message("Before project:")
message("  Source: ${PROJECT_SOURCE_DIR}")
message("  Binary: ${PROJECT_BINARY_DIR}")
message("  Version: ${PROJECT_VERSION}")
message("  Version (alt):
${PROJECT_VERSION_MAJOR}.${PROJECT_VERSION_MINOR}.${PROJECT_VERSION_PATCH}")

project(Foo VERSION 1.2.7)

message("After project:")
message("  Source: ${PROJECT_SOURCE_DIR}")
message("  Binary: ${PROJECT_BINARY_DIR}")
message("  Version: ${PROJECT_VERSION}")
message("  Version (alt):
${PROJECT_VERSION_MAJOR}.${PROJECT_VERSION_MINOR}.${PROJECT_VERSION_PATCH}")
```

```
[project-examples]> rm -rf _builds
[project-examples]> cmake -Hvariables -B_builds
Before project:
  Source:
  Binary:
  Version:
  Version (alt): ..
-- 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
After project:
  Source: /.../project-examples/variables
  Binary: /.../project-examples/_builds
  Version: 1.2.7
  Version (alt): 1.2.7
-- Configuring done
-- Generating done
-- Build files have been written to: /.../project-examples/_builds
```

You can use alternative `foo_{SOURCE,BINARY}_DIRS` / `foo_VERSION_{MINOR,MAJOR,PATCH}` synonyms.

This is useful when you have hierarchy of projects:

可以有多个 PROJECT

```
# Top level CMakeLists.txt

cmake_minimum_required(VERSION 2.8)
project(foo)

message("From top level:")
message("  Source (general): ${PROJECT_SOURCE_DIR}")
message("  Source (foo): ${foo_SOURCE_DIR}")

add_subdirectory(boo)
```

cmake通过这个命令，来分析不同文件夹中的 CMakeLists.txt

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

cmake_minimum_required(VERSION 2.8)
project(boo)

message("From subdirectory 'boo':")
message("  Source (general): ${PROJECT_SOURCE_DIR}")
message("  Source (foo): ${foo_SOURCE_DIR}")
message("  Source (boo): ${boo_SOURCE_DIR}")
```

子 CMakeLists.txt中也可以访问到 foo_source_dir

```
[project-examples]> rm -rf _builds
[project-examples]> cmake -Hhierarchy -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
From top level:
  Source (general): /.../project-examples/hierarchy
  Source (foo): /.../project-examples/hierarchy
From subdirectory 'boo':
  Source (general): /.../project-examples/hierarchy/boo
  Source (foo): /.../project-examples/hierarchy
  Source (boo): /.../project-examples/hierarchy/boo
-- Configuring done
-- Generating done
-- Build files have been written to: /.../project-examples/_builds
```

As you can see we are still able to use `foo_*` variables even if new command `project(boo)` called.

3.5.4. When not declared

如果你不明确地去调用PROJECT命令，那么cmake也会隐含地去调用这个命令，而且这个命令的调用实在cmake_minimum_required命令调用之前发生的，这样就会带来之前讨论的一系列问题

CMake will implicitly declare `project` in case there is no such command in top-level CMakeLists.txt. This will be equal to calling `project` before any other commands. It means that `project` will be called **before** `cmake_minimum_required` so can lead to problems described in previous section:

```
# Top level CMakeLists.txt

message("Before 'cmake_minimum_required'")
cmake_minimum_required(VERSION 2.8)

add_subdirectory(boo)
```

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

cmake_minimum_required(VERSION 2.8)
project(boo)
```

```
[project-examples]> rm -rf _builds
[project-examples]> cmake -Hnot-declared -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
Before 'cmake_minimum_required'
-- Configuring done
-- Generating done
-- Build files have been written to: /.../project-examples/_builds
```

3.5.5. Summary

各个CMakeLists.txt之间通过
add_subdirectory来串联起来

- You must have `project` command in your top-level `CMakeLists.txt`
- Use `project` to declare non divisible monolithic hierarchy of targets
- Try to minimize the number of instructions before `project` and verify that variables are declared in such block of CMake code

Stackoverflow

- What is the project?