


Branch: master ▾

lessons / cmake-basics.mkd

Find file

Copy path

 **bast** initial layout

f31a3a0 on Sep 25, 2016

1 contributor

670 lines (476 sloc) | 14.4 KB

name: inverse layout: true class: center, middle, inverse

# CMake basics for busy scientists

## Radovan Bast

Licensed under [CC BY 4.0](#). Code examples: [OSI](#)-approved [MIT license](#).

layout: false

## Many projects use Makefiles

- Simple and many people are comfortable with it
- Relatively low-level
- Portability is a problem (impossible to port to Windows)
- Typically needs to be configured (configure scripts)
- Difficult to handle multi-language projects
- Difficult to manage projects that depend on many libraries
- Offers no functions to discover OS, processor, libraries, etc.
- Fortran 90 dependencies need to be explicitly expressed
- Difficult to do complex tasks and remain portable
- Typically we need to accommodate multiple languages, complex dependencies, math libraries, MPI, OpenMP, conditional builds and testing, external libraries, etc.
- Typically projects grow out of Makefiles and use GNU Autotools to generate Makefiles

## About CMake

- Cross-platform
- Open-source
- Actively developed
- Manages the build process in a compiler-independent manner
- Designed to be used in conjunction with the native build environment
- Written in C++ (does not matter but if you want to build CMake yourself, C++ is all you need)

- CMake is a generator, so it does not compile
- Not a replacement for make, rather replacement for Autotools

---

## Why CMake (1/2)

---

- Out-of-source compilation (possibility to compile several builds with the same source)
- Really cross-platform (Linux, Mac, Windows, AIX, iOS, Android)
- Excellent support for Fortran, C, C++, and Java, as well as mixed-language projects
- CMake understands Fortran 90 dependencies very well; no need to program a dependency scanner
- Excellent support for multi-component and multi-library projects
- Makes it possible and relatively easy to download, configure, build, install, and link external modules
- CMake defines portable variables about the system for us
- Cross-platform system- and library-discovery
- CTest uses a Makefile (possible to run tests with -jN)

---

## Why CMake (2/2)

---

- Generates user interface (command-line or text-UI or GUI)
- Can achieve complex build tasks with less typing and less headaches in a portable way
- We are in a good company: CMake is used by many prominent projects: MySQL, Boost, VTK, Blender, KDE, LyX, Mendeley, MikTeX, Compiz, Google Test, ParaView, Second Life, Avogadro, and many more ...
- Not bound to the generation of Makefiles
- Full-fledged testing and packaging framework with CTest and CPack
- We spend less time writing build system files and more time developing code
- Personal opinion: easier than Autotools

---

## How to install/upgrade CMake

---

- Download tarball from <http://www.cmake.org>
- Extract and set path

```
$ export PATH='/home/user/cmake/bin/':$PATH
```

- Enjoy
- Alternatively on Debian and related

```
$ sudo apt-get install cmake
```

- Alternatively on Fedora and related

```
$ sudo yum install cmake
```

---

## Hello-world example

---

- We have a `hello.F90` program and wish to compile it to `hello.x`
- We create a file called `CMakeLists.txt` which contains

```
cmake_minimum_required(VERSION 2.8 FATAL_ERROR)

project(HelloWorld)

enable_language(Fortran)

add_executable(hello.x hello.F90)
```

- We configure the project

```
$ mkdir build
$ cd build/
$ cmake ..

-- The Fortran compiler identification is GNU
-- Check for working Fortran compiler: /usr/bin/f95
-- Check for working Fortran compiler: /usr/bin/f95  -- works
-- Detecting Fortran compiler ABI info
-- Detecting Fortran compiler ABI info - done
-- Checking whether /usr/bin/f95 supports Fortran 90
-- Checking whether /usr/bin/f95 supports Fortran 90 -- yes
-- Configuring done
-- Generating done
-- Build files have been written to: /home/user/example/build
```

- We compile the code

```
$ make

Scanning dependencies of target hello.x
[100%] Building Fortran object CMakeFiles/hello.x.dir/hello.F90.o
Linking Fortran executable hello.x
[100%] Built target hello.x
```

- We are done
- In the following we will learn what happened here behind the scenes

---

## Building a Fortran project

---

- We have a simple Fortran project consisting of 3 files ( `main.F90` , `foo.F90` , and `bar.F90` ):

```
program main

    use foo, only: print_foo
    use bar, only: print_bar

    implicit none

    call print_foo()
    call print_bar()
```

```
end program
```

---

## Building a Fortran project

---

- We can build them with the following simple `CMakeLists.txt` :

```
cmake_minimum_required(VERSION 2.8 FATAL_ERROR)

project(MyProject)

enable_language(Fortran)

# this is where we will place the Fortran module files
set(CMAKE_Fortran_MODULE_DIRECTORY ${PROJECT_BINARY_DIR}/modules)

# the executable is built from 3 source files
add_executable(
    myproject.x
    main.F90
    foo.F90
    bar.F90
)
```

- We can list the source files in any order
- We do not have to worry about module dependencies

---

## Building a Fortran project

---

- We build the project

```
$ mkdir build
$ cd build/
$ cmake ..
$ make
```

- There is nothing special about `build/` - we can do this instead:

```
$ cd /tmp/debug
$ cmake /path/to/source
$ make
```

- CMake looks for `CMakeLists.txt` and processes this file
- CMake puts everything into `${PROJECT_BINARY_DIR}` , does not pollute `${PROJECT_SOURCE_DIR}`
- We can build different binaries with the same source!

---

## Multi-language projects

---

- It is easy to support other languages (C, CXX):

```
project(language-mix)

enable_language(Fortran CXX)

# tell CMake where to find header files
include_directories(${PROJECT_SOURCE_DIR}/include)

add_executable(
    mix.x
    mix/main.F90
    mix/sub.cpp
    mix/baz.c
)

set_property(TARGET mix.x PROPERTY LINKER_LANGUAGE Fortran)
```

- That was easy
- But how can we select the compiler?

---

## How to set the compiler

- We set the compilers like this:

```
$ FC=gfortran CC=gcc CXX=g++ cmake ..
$ make
```

- Or via export:

```
$ export FC=gfortran
$ export CC=gcc
$ export CXX=g++
$ cmake ..
$ make
```

---

## Controlling compiler flags

- We can define compiler flags for different compilers and build types

```
if(CMAKE_Fortran_COMPILER_ID MATCHES Intel)
    set(CMAKE_Fortran_FLAGS "${CMAKE_Fortran_FLAGS} -Wall"
        set(CMAKE_Fortran_FLAGS_DEBUG "-g -traceback")
        set(CMAKE_Fortran_FLAGS_RELEASE "-O3 -ip -xHOST")
endif()

if(CMAKE_Fortran_COMPILER_ID MATCHES GNU)
    set(CMAKE_Fortran_FLAGS "${CMAKE_Fortran_FLAGS} -Wall")
    set(CMAKE_Fortran_FLAGS_DEBUG "-O0 -g3")
    set(CMAKE_Fortran_FLAGS_RELEASE "-Ofast -march=native")
endif()

...
```

- Similarly you can set `CMAKE_C_FLAGS` and `CMAKE_CXX_FLAGS`

- This is how we set CPP definitions:

```
add_definitions(-DVAR_SOMETHING -DENABLE_DEBUG -DTHIS_DIMENSION=137)
```

---

## Controlling the build type

- We can select the build type on the command line:

```
$ cd build
$ cmake -DCMAKE_BUILD_TYPE=Debug ..
$ make
```

- It is often useful to set

```
# we default to Release build type
if(NOT CMAKE_BUILD_TYPE)
    set(CMAKE_BUILD_TYPE "Release")
endif()
```

---

## See actual compiler flags and link line

- The default compilation output is nice and compact
- But sometimes we want to see the current compiler flags and the gory compiler output

```
$ make VERBOSE=1
```

- The link line is saved in `CMakeFiles/<target>.dir/link.txt`

---

## Presenting options to the user

```
# default is OFF
option(ENABLE_MPI "Enable MPI parallelization" OFF)

if(ENABLE_MPI)
    message("MPI is enabled")
else()
    message("MPI is disabled")
endif()
```

- We can select options on the command line:

```
$ cd build
$ cmake -DENABLE_MPI=ON ..
$ make
```

- These options become automatically available in GUI/ccmake
-

## Good way to set defaults

---

```
set(MAX_BUFFER "1024" CACHE STRING "Max buffer size")

# this is passed on to the code we compile
add_definitions(-DMAX_BUFFER=${MAX_BUFFER})

message(STATUS "Set max buffer to ${MAX_BUFFER}")
```

- We can override them from the command line

```
$ cmake ..
-- Set max buffer to 1024

$ cmake -DMAX_BUFFER=2048 ..
-- Set max buffer to 2048
```

---

## Messages to the user (or to you)

- Useful for debugging CMake files

```
# display message
message("We are right here.")

# display STATUS message with a -- in the command line
message(STATUS "Still everything under control ...")

# display message and halt configuration
message(FATAL_ERROR "Something unexpected happened!")
```

---

## Variables, lists and loops

```
set(CURRENT_WEATHER "sunny day")

message("We'll meet again some ${CURRENT_WEATHER} ...")

set(FRUITS Apple Banana Orange Kiwi Mango)

foreach(fruit ${FRUITS})
    message("${fruit} is a tasty fruit")
endforeach()
```

---

## Functions and macros

```
function(my_function foo bar)
    message("called my_function with the arguments: '${foo}' and '${bar}'")
    set(MY_VARIABLE "local scope")
endfunction()
```

```
macro(my_macro foo bar)
    message("called my_macro with the arguments: '${foo}' and '${bar}'")
    set(MY_VARIABLE "${bar}")
endmacro()

my_function(this that)
message("MY_VARIABLE set to: '${MY_VARIABLE}'")

my_macro(this that)
message("MY_VARIABLE set to: '${MY_VARIABLE}'")
```

- Function variables have scope

```
called my_function with the arguments: 'this' and 'that'
MY_VARIABLE set to: ''
called my_macro with the arguments: 'this' and 'that'
MY_VARIABLE set to: 'that'
```

---

## Configuring files

- We can configure files

```
configure_file(
    ${PROJECT_SOURCE_DIR}/infile
    ${PROJECT_BINARY_DIR}/outfile
    @ONLY
)
```

- CMake takes `${PROJECT_SOURCE_DIR}/infile` :

```
My system is @CMAKE_SYSTEM_NAME@ and my
processor is @CMAKE_HOST_SYSTEM_PROCESSOR@.
```

- And generates `${PROJECT_BINARY_DIR}/outfile` :

```
My system is Linux and my
processor is x86_64.
```

---

## Organizing larger projects: includes

- CMake looks for `CMakeLists.txt`
- For larger projects it is not practical to put everything into one huge `CMakeLists.txt`
- We can organize the CMake code like this:

```
CMakeList.txt
cmake/ConfigureCompilerFlags.cmake
cmake/ConfigureMPI.cmake
cmake/ConfigureMath.cmake
```

- Then we can include these files into `CMakeLists.txt` (or other included files):



```
# these are paths that cmake will search for module files
```

```
set(CMAKE_MODULE_PATH  
    ${CMAKE_MODULE_PATH}  
    ${PROJECT_SOURCE_DIR}/cmake)
```

```
include(ConfigureCompilerFlags)
```

```
include(ConfigureMPI)
```

```
include(ConfigureMath)
```

---

## Organizing larger projects: split CMakeLists.txt

---

- Let us assume we have a source tree structure like this:

```
CMakeLists.txt
```

```
main.F90
```

```
fun1/foo.F90
```

```
fun1/bar.F90
```

```
fun2/baz.F90
```

```
fun2/oof.F90
```

- Assume module fun2 depends on fun1 (module baz uses module foo)

```
module baz  
    use foo, only: print_foo  
    implicit none  
    public print_baz  
    private  
contains  
    subroutine print_baz()  
        print *, 'baz'  
        call print_foo()  
    end subroutine  
end module
```

---

## Organizing larger projects: split CMakeLists.txt

---

- The convenient and compact solution is to do this

```
cmake_minimum_required(VERSION 2.8 FATAL_ERROR)
```

```
project(MyProject)
```

```
enable_language(Fortran)
```

```
# this is where we will place the Fortran module files
```

```
set(CMAKE_Fortran_MODULE_DIRECTORY ${PROJECT_BINARY_DIR}/modules)
```

```
add_executable(  
    myproject.x
```

```
    main.F90
```

```
    fun1/foo.F90
```

```
    fun1/bar.F90
```

```
    fun2/baz.F90
```

```
    fun2/oof.F90
```

)

- We do not have to worry (we can compile with `make -j12`)

---

## Organizing larger projects: split CMakeLists.txt

---

- Alternative is to use separate `CMakeLists.txt` files

```
cmake_minimum_required(VERSION 2.8 FATAL_ERROR)

project(MyProject)

enable_language(Fortran)

# this is where we will place the Fortran module files
set(CMAKE_Fortran_MODULE_DIRECTORY ${PROJECT_BINARY_DIR}/modules)

add_subdirectory(fun1) # here we use fun1/CMakeLists.txt
add_subdirectory(fun2) # here we use fun2/CMakeLists.txt

# fun1 needs to be compiled before fun2
add_dependencies(fun2 fun1)

add_executable(myproject.x main.F90)

target_link_libraries(myproject.x fun1 fun2)
```

---

## Organizing larger projects: split CMakeLists.txt

---

- Where `fun1/CMakeLists.txt` contains

```
add_library(
    fun1
    foo.F90
    bar.F90
)
```

- And where `fun2/CMakeLists.txt` contains

```
add_library(
    fun2
    baz.F90
    oof.F90
)
```

---

## Organizing larger projects: split CMakeLists.txt

---

- The advantage is that we separate concerns

```
Scanning dependencies of target fun1
[ 20%] Building Fortran object fun1/CMakeFiles/fun1.dir/bar.F90.o
```

```
[ 40%] Building Fortran object fun1/CMakeFiles/fun1.dir/foo.F90.o
Linking Fortran static library libfun1.a
[ 40%] Built target fun1
Scanning dependencies of target fun2
[ 60%] Building Fortran object fun2/CMakeFiles/fun2.dir/baz.F90.o
[ 80%] Building Fortran object fun2/CMakeFiles/fun2.dir/oof.F90.o
Linking Fortran static library libfun2.a
[ 80%] Built target fun2
Scanning dependencies of target myproject.x
[100%] Building Fortran object CMakeFiles/myproject.x.dir/main.F90.o
Linking Fortran executable myproject.x
[100%] Built target myproject.x
```

- Useful for maintenance of larger projects
- Simplifies and speeds up (re)compilation because we limit possible dependencies

## Archaeology: Recover previous configuration settings

- Configuration settings are saved in `${PROJECT_BINARY_DIR}/CMakeCache.txt`

## CMake generators

- The list of CMake generators is long
- Unix Makefiles is only one of them:

visual studio 6	Xcode
visual studio 7	CodeBlocks - MinGW Makefiles
visual studio 10	CodeBlocks - NMake Makefiles
visual studio 11	CodeBlocks - Ninja
visual studio 12	CodeBlocks - Unix Makefiles
visual studio 7 .net 2003	Eclipse CDT4 - MinGW Makefiles
visual studio 8 2005	Eclipse CDT4 - NMake Makefiles
visual studio 9 2008	Eclipse CDT4 - Ninja
borland makefiles	Eclipse CDT4 - Unix Makefiles
nmake makefiles	KDevelop3
nmake makefiles jom	KDevelop3 - Unix Makefiles
watcom wmake	Sublime Text 2 - MinGW Makefiles
msys makefiles	Sublime Text 2 - NMake Makefiles
mingw makefiles	Sublime Text 2 - Ninja
unix makefiles	Sublime Text 2 - Unix Makefiles
Ninja	

- CMake makes it possible to use the native build environment