Demonstrate Usage of CMake & Make with Demo Project

What is CMake?
A cross-platform Buildsystem Generator which can create input for native build tools like make / ninja.

How to START?
Mini. Req.: only an empty '**CMakeLists.txt**'
Mini. Req. for none empty 'CMakeLists.txt':
* **cmake_minimum_required**(…)
* **project**(…)
* optional: a target of the type 'executable', 'library' or 'custom_target'
* init cmd: cmake -S<src-path> -B<build-path>

```
# A very simple CMake Project
cmake_minimum_required(VERSION 3.16)

project(Test) # VERSION 1.0.0

add_custom_target(Test # language CXX
    COMMAND
        ${CMAKE_COMMAND} -E echo
        "A simplest one custom-target CMake project."
)
```

```
yw@yw-HP-Compaq-Elite-8300-SFF in CMakeDemo on Demo_Branch [!]
$ cmake -H. -B build/
-- Configuring done
-- Generating done
-- Build files have been written to: /mnt/yw/SSD 250/00 projects/
l/programming training/Toolchain/CMake_Make/CMakeDemo/build

yw@yw-HP-Compaq-Elite-8300-SFF in CMakeDemo on Demo_Branch [!]
$ cmake --build ./build/ --target Test
Scanning dependencies of target Test
A simplest one custom-target CMake project.
Built target Test
```

The most important BASICs of CMake:

* CMake stages (without CTest & CPack):
  configure-step
  Generation-step => execute_process(...)
  Build-step => add_custom_command(OUTPUT …)
  add_custom_target(<target> …)
  add_custom_command(TARGET ...)

* Three types of target:
  add_executable(<name> …)
  add_library(<name> …)
  add_custom_library(<name> [ALL])

* Target specific helper func.:
  target_include_directories(<target> <visibility> <dir>)
  target_link_directories(<target> <visibility> <dir>)
  target_link_libraries(<target> <visibility> <lib>)
  target_compile_definitions(<target> <visibility> <items>)
  set_target_properties(<target...> PROPERTIES <prop value>...)
  get_target_property(<VAR> <target> <property>)

* Global/cross-targets helper func.:
  include_directories(<dir>…)
  link_directories(<dir>…)
  link_libraries(AFTER|BEFORE <dir>…)
  add_definitions(<definition>…)
  remove_definitions(-D<def>…)
  add_compile_definitions(<definition>…)
  set_property(<scope> PROPERTY <name> <value>…)
  get_property(<VAR> <scope> PROPERTY <name>)

* Structuring project:
  add_subdirectory(<subdir>)
  include(<filename>…)
  include_directories(<dir>…)

* Set variable:
  normal : set(<VAR> <value>) // get: ${VAR}
  cache entry: set(<VAR> <value> CACHE <type> <desc.>)
  env. variable: set(ENV{VAR} [value])
  <type>: STRING, BOOL, PATH etc.

* Manipulate list:
  list(<operation> <list> [<element> …])
  <operation>: LENGTH, GET, FIND, APPEND etc.

* View settings:
  MESSAGE(<type> "…")
  <type>: STRING, TRACE, ERROR etc.

* Flow control structures:
  <element>(...)
  #do something
  <endelement>()
  <element>: if, foreach, while, function, macro

```
cmake_minimum_required(VERSION 3.15.0)

project(CMAKEDEMO VERSION 1.0)

set(MAIN_TARGET CMakeDemo)

# include sources needs for this target
include(${CMAKE_CURRENT_SOURCE_DIR}/sources.cmake)

add_executable(${MAIN_TARGET} ${MAIN_SRC})

# Add local targets or subdirectory using the 'option'-function
# Advantage of 'option'-func is it can turn on/off from outside
# like a cached boolean variable
option(USE_EXTLIB01 "Use local library extlib01." ON)
option(USE_M2 "Use local library module02." ON)
option(USE_M3 "Use local library module03." ON)

# This is only a demostration about how to use 'find_library'
if(USE_EXTLIB01)
        find_library(EXT_LIBRARY
                NAMES extlib01.a
                PATHS ${PROJECT_SOURCE_DIR}/extlibs/src
        )
endif()

if(USE_M2)
        if(NOT EXISTS "${PROJECT_SOURCE_DIR}/module02")
                message(FATAL_ERROR "The module02 doesn't exist")
        else()
                add_subdirectory(module02)

                list(APPEND EXTRA_LIB_DIRS "module02")
                list(APPEND EXTRA_INCLUDE_DIRS "${PROJECT_SOURCE_DIR}/module02")
                list(APPEND EXTRA_LINKS module02)
        endif()
endif()

if(USE_M3)
        if(NOT EXISTS "${PROJECT_SOURCE_DIR}/module03")
                message(FATAL_ERROR "The module03 doesn't exist!")
        else()
                add_subdirectory(module03)

                list(APPEND EXTRA_LIB_DIRS "module03")
                list(APPEND EXTRA_INCLUDE_DIRS "${PROJECT_SOURCE_DIR}/module03")
                list(APPEND EXTRA_LINKS module03)
        endif()
endif()

target_include_directories(${MAIN_TARGET}
        PUBLIC ${EXTRA_INCLUDE_DIRS}
#       PUBLIC ${PROJECT_BINARY_DIR}
)

target_link_directories(${MAIN_TARGET}
        PRIVATE ${EXTRA_LIB_DIRS}
)

target_link_libraries(${MAIN_TARGET}
        ${EXTRA_LINKS}
    ${EXT_LIBRARY}
)
```

```
CMakeLists.txt
extlibs
  inc
    extlib01.h
  src
    extlib01.a
module01
  inc
    module01.h
  src
    module01.c
module02
  CMakeLists.txt
  inc
    module02.h
  module0201
    inc
      module0201.h
    src
      module0201.c
  sources.cmake
  src
    module02.c
module03
  CMakeLists.txt
  inc
    module03.h
  sources.cmake
  src
    module03.c
sources.cmake
src
  main.c
```

Compare Demo with same Resources
* The major exe target contains source under ./src and and ./module01
* module02 & -03 shall be compiled into own libraries and linked to the major target
* A static extra-lib 'extlib01' needs to be linked into the final exe target
* Based on this structure, there are three CMakeLists.txt and three Makefile on each level, pls. refer to correspond screenshots
* Mikefile of module use same structure therefore it's not be shown completely

```
extlibs
  inc
    extlib01.h
  src
    extlib01.a
inc
Makefile
module01
  inc
    module01.h
  src
    module01.c
module02
  inc
    module02.h
  module0201
    inc
      module0201.h
    src
      module0201.c
  module02.mk
  src
    module02.c
module03
  inc
    module03.h
  module0301
    inc
    src
  module03.mk
  src
    module03.c
src
  main.c
```

```
# This file is adapted and expanded based on the https://makefiletutorial.com/#makefile-cookbook
# Many thanks to the original author!

# Main target
TARGET_EXEC := demo

# directories
BUILD_DIR := ./build
SRC_DIRS := ./
LIB_DIRS := ./extlibs
NAVI_SRCS_STR := \( -name '*.cpp' -or -name '*.c' -or -name '*.s' \)
SKIP_DIRS_STR := \( -name .git -o -name .vscode -o -name module02 -o -name module03 \)
NAVI_LIBS_STR := \( -name '*.a' \)

# Unix/linux standard shell commands used in these scripts;
# replace with your own ones if you're working under another envrionment
CMD_FIND = find
CMD_MKDIR := mkdir
CMD_RM := rm

# Dummy first default target
.PHONY : nothing
nothing:
        $(info "Nothing to do with the default/first target; please use an valid target like 'all'.")

# Find all the C and C++ files we want to compile
# Note the single quotes around the * expressions. Make will incorrectly expand these otherwise.
SRCS := $(shell $(CMD_FIND) $(SRC_DIRS) -type d $(SKIP_DIRS_STR) -prune -o $(NAVI_SRCS_STR) -print)

# add all building libraires into $(LIBS)
LIBS := $(shell $(CMD_FIND) $(LIB_DIRS) $(NAVI_LIBS_STR) )

# String substitution for every C/C++ file.
# As an example, hello.cpp turns into ./build/hello.cpp.o
OBJS := $(SRCS:%=$(BUILD_DIR)/%.o)

# String substitution (suffix version without %).
# As an example, ./build/hello.cpp.o turns into ./build/hello.cpp.d
DEPS := $(OBJS:.o=.d)

# Every folder in ./src will need to be passed to GCC so that it can find header files
# INC_DIRS := $(shell $(CMD_FIND) $(SRC_DIRS) -type d -not -path "$(SKIP_DIRS)")
INC_DIRS := $(shell $(CMD_FIND) $(SRC_DIRS) -type d $(SKIP_DIRS_STR) -prune -o -print)

# Add a prefix to INC_DIRS. So moduleA would become -ImoduleA. GCC understands this -I flag
INC_FLAGS := $(addprefix -I,$(INC_DIRS))

# The -MMD and -MP flags together generate Makefiles for us!
# These files will have .d instead of .o as the output.
CPPFLAGS := $(INC_FLAGS) -MMD -MP

# include makefiles for libraires building
-include ./module02/module02.mk
-include ./module03/module03.mk

# -include ./extlibs/extlib01.mk

# The final build step.
.PHONY : all
all: $(BUILD_DIR)/$(TARGET_EXEC)

$(BUILD_DIR)/$(TARGET_EXEC): $(OBJS) $(LIBS)
        $(CC) $(OBJS) $(LIBS) -o $@ $(LDFLAGS)

# Build step for C source
$(BUILD_DIR)/%.c.o: %.c
        $(CMD_MKDIR) -p $(dir $@)
        $(CC) $(CPPFLAGS) $(CFLAGS) -c $< -o $@

# Build step for C++ source
$(BUILD_DIR)/%.cpp.o: %.cpp
        $(CMD_MKDIR) -p $(dir $@)
        $(CXX) $(CPPFLAGS) $(CXXFLAGS) -c $< -o $@

# Build step for assembly source
$(BUILD_DIR)/%.s.o: %.s
        $(CMD_MKDIR) -p $(dir $@)
        $(CC) $(CPPFLAGS) $(CFLAGS) -c $< -o $@

.PHONY: clean
clean:
        if [ -d $(BUILD_DIR) ]; then $(CMD_RM) -r $(BUILD_DIR); fi

# Include the .d makefiles. The - at the front suppresses the errors of missing
# Makefiles. Initially, all the .d files will be missing, and we don't want those
# errors to show up.
-include $(DEPS)
```

```
# include sources needs for this target
include(${CMAKE_CURRENT_SOURCE_DIR}/sources.cmake)

# Available lib-type:
# STATIC
# SHARED
# OBJECT
# MODULE
# Default type is STATIC, if BUILD_SHARED_LIBS is false
add_library( module02
    STATIC
    ${MODULE02_SRC}
)
```

```
get(MODULE02_SRC
    # header files
    ${CMAKE_CURRENT_SOURCE_DIR}/inc/module02.h
    ${CMAKE_CURRENT_SOURCE_DIR}/module0201/inc/module0201.h

    # source files
    ${CMAKE_CURRENT_SOURCE_DIR}/src/module02.c
    ${CMAKE_CURRENT_SOURCE_DIR}/module0201/src/module0201.c
)
```

```
# Different as it's for library 'module02' we declare Lib 'module03'
# with NO sources. This is becuase we are going to use target_sources
# to populate the soruces of this target.
# Therefore, we're forced to 'include' the sources.cmake before the
# declaration of the add_library but be able to do it later

add_library(module03 "")

# include sources needs for this target
include(${CMAKE_CURRENT_SOURCE_DIR}/sources.cmake)
```

```
# Use target_sources to add target's sources instead of defining
# an variable and then appendding all sources to it, as we're done
# for library 'module02'; further more, we can use different visibility
# PRIVATE or PUBLIC for different parts of the sources

target_sources(module03
    PRIVATE
        # source files
        ${CMAKE_CURRENT_SOURCE_DIR}/src/module03.c

    PUBLIC
        # header files
        ${CMAKE_CURRENT_SOURCE_DIR}/inc/module03.h
)
```

```
# This module as an own logic unit will
M2_TARGET := module02.a
M2_BUILD_DIR := ./build/module02/build
M2_SRC_DIRS := ./module02
M2_SKIP_DIRS := ./.git
```

```
# The final build step for this logic unit
.PHONY: module02
module02: $(M2_BUILD_DIR)/$(M2_TARGET)

$(M2_BUILD_DIR)/$(M2_TARGET): $(M2_OBJS)
        $(AR) $(ARFLAGS) $@ $^
```

* Content under <build-path>:
  CmakeListsCache.txt # every config & parsing result
  Cmakefiles/ # everything that native tool need
  <NativeToolInput> # i.e. 'Makefile', 'build.ninja' etc.
  cmake_install.cmake # useful for install program/module

* CML usages:
  cmake -G <generator> => def. native generator i.e. make
  cmake -E <cmd> => perform cml tool i.e. echo or copy
  cmake -P <script> => perform a script

Where to get HELP?
* cmake --version
* cmake --help => main help page
* cmake --help <cmd> => page for the <cmd>
* cmake --hlep-comman-list
* cmake --system-information [file]
* cmake -E => list available CML tools
* cmake –build <build-path> --target help => list available targets of cur. scope / CMakeLists.txt

```
yw@yw-HP-Compaq-Elite-8300-SFF in CMakeDemo on Demo_Branch [!]
$ cmake --version
cmake version 3.16.3

CMake suite maintained and supported by Kitware (kitware.com/cma

yw@yw-HP-Compaq-Elite-8300-SFF in CMakeDemo on Demo_Branch [!]
$ cmake --help
Usage
```

```
yw@yw-HP-Compaq-Elite-8300-SFF in CMakeDemo on Demo_Branch [!]
$ cmake --system-information
Avoid ctest truncation of output: CTEST_FULL_OUTPUT
=============================================================
=== MAIN VARIABLES
=============================================================
CMAKE_STATIC_LIBRARY_PREFIX == "lib"
CMAKE_STATIC_LIBRARY_SUFFIX == ".a"
```

BASICs of Make:
* Major config file: 'Makefile'
* Run 'make' to call the default/first target
* Options: -e, -B, -s, -j=N
* get help: 'make --help'
* Target definition sytax:
  <name>: [rerequisites]
  cmd cmd cmd ...
* call func.: $(<func> <arg. List>)
* .PHONY: # avoid same name between target and file

* set variable: || to get: $(<VAR>)
  <VAR> := <value> # one time assignment
  <VAR> = <value> # renewing assignment
  <VAR> += <value> # append assignment
  <VAR> ?= <value> # safe assignment
  <VAR> += <value> # append assignment

* magic variables
out.o: src.c src.h
  $@ # "out.o" (target)
  $< # "src.c" (first prerequisite)
  $^ # "src.c src.h" (all prerequisites)

%.o: %.c
  $* # the 'stem' with which an implicit rule matches ("foo" in "foo.c")