C++ Course 11 : Using cmake and make

By Oleksiy Grechnyev

Using C++ compilers

Compile to executable (**.exe** on Windows) :

gcc:

g++ -o myprog main.cpp file1.cpp file2.cpp

clang:

clang -o myprog main.cpp file1.cpp file2.cpp

cl (Microsoft):

cl /o myprog main.cpp file1.cpp file2.cpp

Where: **myprog** = name of the executable

main.cpp file1.cpp file2.cpp = C++ source files

Compile ONE source file to the *object* file (**.o** or **.obj**) :

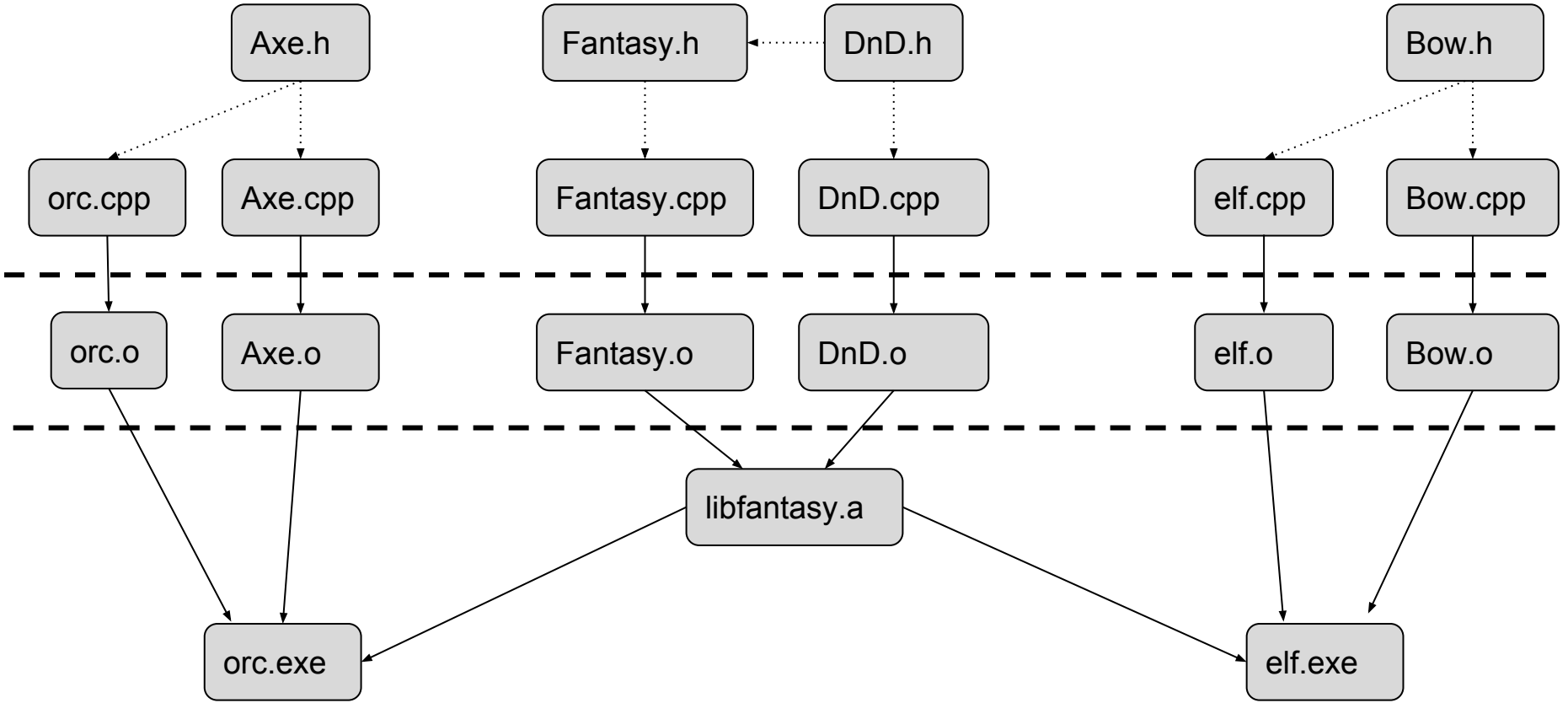
g++ -c main.cpp

...

Link *object* files (and *static libraries*) to executable:

g++ -o myprog main.o file1.o file2.o

Build process



Low-level build systems for C/C++: make, nmake, ninja, ...

Unix/Linux and MinGW/Msys use **make** . Project file: **Makefile** :

To build a project:

make	Unix/Linux
mingw32-make	MinGW (Both 32 and 64 bit)

More options:

make clean	Clean project (delete executables and **.o**)
make hello	Build target **hello**
make all	Build all targets
make -f Makefile2 -j3	Build file **Makefile2** on 3 cores

Compile and install software on Unix/Linux (OpenCV, ffmpeg etc.) :

./configure **<options>**

make

make install

Makefile example (Makefile2)

```
CXX = g++
CXXFLAGS = -Wall -g
all: example
example : main.o a.o B.o
<tab>$(CXX) $(CXXFLAGS) -o $@ $^
main.o : main.cpp a.h B.h
<tab>$(CXX) $(CXXFLAGS) -c $<
a.o : a.cpp a.h
<tab>$(CXX) $(CXXFLAGS) -c $<
B.o : B.cpp B.h
<tab>$(CXX) $(CXXFLAGS) -c $<
clean:
<tab>-rm *.o example
<tab>-del *.o example.exe
```

Makefile targets

make targets:

target : dependencies

<tab>command

main.o : main.cpp a.h B.h

<tab>$(CXX) $(CXXFLAGS) -c $<

make variables:

CXX = g++

CXXFLAGS = -Wall -g

$(CXX) $(CXXFLAGS) -c $<

Special variables:

$@ : target name

$^ : All dependencies

$< : First dependency

Makefile with rules (Makefile)

```
CXX = g++
CXXFLAGS = -Wall -g
LIBS =
OBJS = main.o a.o B.o
DEPS = a.h B.h
.PHONY: all
all: example
example : $(OBJS)
<tab>$(CXX) $(CXXFLAGS) -o $@ $^ $(LIBS)
%.o : %.cpp $(DEPS)
<tab>$(CXX) $(CXXFLAGS) -o $@ -c $<
.PHONY: clean
clean:
<tab>-rm *.o example
<tab>-del *.o example.exe
```

Why make is not enough?

1. Different incompatible versions (**gmake, remake, mingw32-make, nmake ...**)
2. No configuration/library finding abilities. Extensions (Unix/Linux mostly):

pkgconfig Find a package (headers + libraries), used by **make**

GNU Build System : **autoconf, automake, libtool, gnulib**

autoconf Generate **./configure** script

automake Generate **Makefile.in** (used by **./configure**)

3. This is not very popular (or convenient) on Windows
4. Compiler dependent, works best for **gcc**
5. Outdated and can be difficult to use.

CMake (since 2000) : Alternative to GNU Build System for **C, C++, Fortran, Assembler**

1. Generates projects for **make, nmake, ninja, Code::Blocks, XCode, Visual Studio**
2. Cross platform, including Windows + Microsoft Compiler and Android NDK (**ninja**)
3. Package and library search
4. Powerful, with (relatively) easy-to-use syntax

Simplest cmake projects

My first CMake project (**CMakeLists.txt**)

```
add_executable(hello hello.cpp)
```

My second CMake project

```
# This is a comment
cmake_minimum_required(VERSION 3.1)

project(hello)

set(CMAKE_CXX_STANDARD 14)

set(SRCS
#    somefile.h somefile.cpp
     hello.cpp
)

add_executable(${PROJECT_NAME} ${SRCS})
```

How to build a CMake project ?

```
mkdir build
cd build
cmake ..
cmake --build .
```

Rebuild after you have edited some source files ...

```
cmake --build .
```

Using *generators* (Example: Windows, MinGW)

```
mkdir build
cd build
cmake -G "MinGW Makefiles" ..
cmake --build .
```

CMake does not call the C++ compiler directly.
Generators use low-level build systems (**make**, **nmake**, **ninja**, **...**)
and IDEs (Visual Studio, Code.Blocks, xcode)

How does it work :

Project directory (folder) : Директория (папка) проекта:
CMakeLists.txt
hello.cpp

We run:
mkdir build
cd build

Directory **build** appears:
build
CMakeLists.txt
hello.cpp

All build process takes place in the directory **build** !
Весь процесс сборки происходит в директории **build** !

Configuring cmake project:

We run (in the directory **build**):

cmake -G ”MinGW Makefiles” ..

Configure project using the generator **MinGW Makefiles**.

.. = path to the directory with the file **CMakeLists.txt**

Directory **build** :

CMakeFiles

cmake_install.cmake

CMakeCache.txt Project configuration (can be edited)

Makefile Project file for **make**

Directory **build/CMakeFiles** :

...

hello.dir Configuration/build directory for target **hello**

...

Building

We run (in the directory **build**):
cmake --build .
Build the project in the directory **.** (current directory) using e.g. **make**

New files in **build** :
hello (or **hello.exe** in Windows) : executable
New files in **build/CMakeFiles/hello.dir** :
hello.cpp.obj Object file
objects.a Object files packed as a static lib

Build commands:
make or **mingw32-make** or **nmake** or **ninja** Build project
cmake --build . --target hello -- -j2 Build target **hello** on 2 cores (**make** flag **-j2**)
cmake --build . --target clean
cmake --build . --target install

CMake language (example lang)

set() : Create/Assign CMake variable: Присваивание :

```
set(CMAKE_CXX_STANDARD 14)
```

Variable value: Значение переменной: **${CMAKE_CXX_STANDARD}**

message() : Print string or variable

```
message("Hello world !")
message("PROJECT_NAME = ${PROJECT_NAME}")
message("CMAKE_CXX_STANDARD = ${CMAKE_CXX_STANDARD}")
message(${CMAKE_CXX_STANDARD})
```

math() : Evaluate a mathematical expression, put result to **c** :

```
math(EXPR c "5*(10+13) + 7")
message("5*(10+13) + 7 = ${c}")
```

if(), while()

if() statement :

```
set(n 15)
if(n GREATER 10)
        message("${n} > 10")
else()
        message("${n} < 10")
endif()
```

while() statement :

```
set(n 1)
while(n LESS_EQUAL 10)
        message(${n})
        math(EXPR n "${n}+1")
endwhile()
```

CMake standard variables

```
message("CMAKE_BINARY_DIR = ${CMAKE_BINARY_DIR}")
message("CMAKE_SOURCE_DIR = ${CMAKE_SOURCE_DIR}")
message("CMAKE_BUILD_TYPE = ${CMAKE_BUILD_TYPE}")
message("CMAKE_CXX_FLAGS = ${CMAKE_CXX_FLAGS}")
message("CMAKE_CXX_FLAGS_DEBUG = ${CMAKE_CXX_FLAGS_DEBUG}")
message("CMAKE_CXX_FLAGS_RELEASE = ${CMAKE_CXX_FLAGS_RELEASE}")
message("CMAKE_EXECUTABLE_SUFFIX = ${CMAKE_EXECUTABLE_SUFFIX}")
message("CMAKE_SYSTEM = ${CMAKE_SYSTEM}")
message("CMAKE_SYSTEM_NAME = ${CMAKE_SYSTEM_NAME}")
message("CMAKE_SIZEOF_VOID_P = ${CMAKE_SIZEOF_VOID_P}")   # Size of void *
message("WIN32 = ${WIN32}")
message("APPLE = ${APPLE}")
message("UNIX = ${UNIX}")
message("MINGW = ${MINGW}")

if (WIN32)
    message("Windows !!!")
else()
    message("NOT Windows !!!")
endif()
```

CMake standard variables

```
Useful CMake variables :
CMAKE_BINARY_DIR = D:/alex/w/cpp-course/examples_11/lang/build
CMAKE_SOURCE_DIR = D:/alex/w/cpp-course/examples_11/lang
CMAKE_BUILD_TYPE =
CMAKE_CXX_FLAGS =
CMAKE_CXX_FLAGS_DEBUG = -g
CMAKE_CXX_FLAGS_RELEASE = -O3 -DNDEBUG
CMAKE_EXECUTABLE_SUFFIX = .exe
CMAKE_SYSTEM = Windows-10.0.15063
CMAKE_SYSTEM_NAME = Windows
CMAKE_SIZEOF_VOID_P = 8
WIN32 = 1
APPLE =
UNIX =
MINGW = 1
Windows !!!
```

Executables and libraries :

Build an executable : Собрать исполняемый файл :

```
add_executable(<target> <sources>)
add_executable(hello hello.cpp)                    # Build hello from hello.cpp
add_executable(${PROJECT_NAME} ${SRCS})            # The same with variables
```

Specify libraries needed by target : Библиотеки, необходимые для цели :

```
target_link_libraries(<target> <libraries>)
target_link_libraries(${PROJECT_NAME} a b)
```

Build a library : Собрать библиотеку :

```
add_library(<target> [STATIC|SHARED] <libraries>)
add_library(b ${SRCS_B})                           # Default (static?) library
add_library(b STATIC ${SRCS_B})                    # Static library  .a/.lib
add_library(b SHARED ${SRCS_B})                    # Shared (dynamic) library   .so/.dll
```

Include a subdirectory (with its own **CMakeLists.txt**) :

```
add_subdirectory(liba)
```

Example lib : hello.exe + 2 libraries : Main CMakeLists.txt

```
cmake_minimum_required(VERSION 3.1)
project(hello)
set(CMAKE_CXX_STANDARD 14)

# Build library a from the separate directory liba
add_subdirectory(liba)
include_directories(liba)           # Search for header files (a.h) in liba

# Build SHARED library b : in this directory
set(SRCS_B B.cpp)
add_library(b SHARED ${SRCS_B})

# Build hello
set(SRCS_HELLO main.cpp)
add_executable(${PROJECT_NAME} ${SRCS_HELLO})
target_link_libraries(${PROJECT_NAME} a b)
```

Here **hello** and **b** are built in the same directory. Собираются в одной директории.
It's better to put every target to a separate directory.

Example lib : CMakeLists.txt in subdirectory liba

```
cmake_minimum_required(VERSION 3.1)
project(a)
set(CMAKE_CXX_STANDARD 14)

set(SRCS
    a.cpp
)
add_library(${PROJECT_NAME} STATIC ${SRCS})    #  Static library a
```

Files after build in the directory **build** :

hello.exe	Executable
libb.dll	Shared library **b**
libb.dll.a	Import lib for **libb.dll** (Used for **.dll**, not **.so** !)
CMakeFiles/b.dir/	
CMakeFiles/hello.dir/	
liba/liba.a	Static library **a**
liba/CMakeFiles/a.dir/	

Configuring a CMake project

Build types (**CMAKE_BUILD_TYPE**) : **Release, Debug, MinSizeRel, RelWithDebInfo**

```
cmake -DCMAKE_BUILD_TYPE=Debug ..
```

Set default build type in **CMakeLists.txt** (Not recommended !) :

```
if(NOT CMAKE_BUILD_TYPE )
  set( CMAKE_BUILD_TYPE "Release" )
endif()
```

Configuring parameters:

```
cmake -DWITH_MAGIC=YES ..
```

option() : declare boolean parameters (with description+default !) in **CMakeLists.txt**:

```
option(WITH_DRAGONS "Any dragons in our story ?" OFF)
option(WITH_ELVES "Any elves in our story ?" OFF)
option(WITH_ORCS "Any orcs in our story ?" ON)
```

Configuring options:

```
cmake -DWITH_ELVES=ON ..
```

Multiple parameters?

Do we really have to write ?

cmake -G ”MinGW Makefiles” -DCMAKE_BUILD_TYPE=Debug -DWITH_MAGIC=YES -DWITH_ELVES=ON -DWITH_DRAGONS=ON -DWITH_ORCS=OFF ..

No, we can work in steps :

```
cmake -G ”MinGW Makefiles” ..
cmake -DCMAKE_BUILD_TYPE=Debug ..
cmake -DWITH_MAGIC=YES ..
cmake -DWITH_ELVES=ON ..
cmake -DWITH_DRAGONS=ON ..
cmake -DWITH_ORCS=OFF ..
```

CMake stores variables in **CMakeCache.txt**

Delete this file to reset cache !

CMakeCache.txt

```
//Any dragons in our story ?
WITH_DRAGONS:BOOL=ON

//Any elves in our story ?
WITH_ELVES:BOOL=ON

//No help, variable specified on the command line.
WITH_MAGIC:UNINITIALIZED=YES

//Any orcs in our story ?
WITH_ORCS:BOOL=OFF
```

To view CMake cache :

cmake -L ..

Passing variables to C++

Using **add_definition() :**

```
if(DEFINED WITH_DRAGONS)
      add_definitions(-DWITH_DRAGONS=${WITH_DRAGONS})
endif()
```

Using **configure_file() :**

```
configure_file(config.h.in config.h)
include_directories("${PROJECT_BINARY_DIR}")    # To find config.h
```

File **config.h.in** :

```
#cmakedefine  WITH_MAGIC   @WITH_MAGIC@       // Defined only if NOT OFF/0/NO
#define WITH_ELVES   @WITH_ELVES@            // Defined always
#cmakedefine01  WITH_ORCS
```

File **config.h** :

```
#define WITH_MAGIC   YES             // Defined only if NOT OFF/0/NO
#define WITH_ELVES   ON                  // Defined always
#define WITH_ORCS 0
```

The complete example (vars) CMakeLists.txt

```
cmake_minimum_required(VERSION 3.1)
project(hello)
set(CMAKE_CXX_STANDARD 14)
# Options
option(WITH_DRAGONS "Any dragons in our story ?" OFF)
option(WITH_ELVES "Any elves in our story ?" OFF)
option(WITH_ORCS "Any orcs in our story ?" ON)
message("WITH_MAGIC = ${WITH_MAGIC}")    # And all others
..
# Pass definition to C++ using add_definition()
if(DEFINED WITH_DRAGONS)
 add_definitions(-DWITH_DRAGONS=${WITH_DRAGONS})
endif()
# Pass definitions to C++ using configure_file()
configure_file(config.h.in config.h)
include_directories("${PROJECT_BINARY_DIR}")     # To find config.h
# Build executable hello
set(SRCS  hello.cpp)
add_executable(${PROJECT_NAME} ${SRCS})
```

Useful external libraries and APIs (C++/C) :

1. GUI : **Qt, gtkmm**
2. Video/Audio processing : **ffmpeg (C), gstreamer (C), openmax (C)**
3. Image processing : **CImg, OpenCV**
4. Cross-platform TCP/UDP : **Boost.Asio**
5. Unit tests : **CppUnit, CppTest, Google Test, Boost**
6. 3D graphics : **OpenGL (C) + glew/glad/epoxy, glfw, glm**

Finding external libraries in CMake : 1. The CMake way

```
cmake_minimum_required (VERSION 3.0)
project( grabcam )
set (CMAKE_CXX_STANDARD 14)

find_package( OpenCV REQUIRED )
include_directories( ${OpenCV_INCLUDE_DIRS} )
add_executable( grabcam grabcam.cpp )
target_link_libraries( grabcam ${OpenCV_LIBS} )
```

Finds OpenCV package installed at the standard locations (Linux, MinGW).
Находит пакет OpenCV установленный в стандартном месте (Linux, MinGW).

Most of the time you need to specify directory:
Обычно надо указать директорию:
cmake -DOpenCV_DIR=d:/opencv ..
The directory must contain : **OpenCVConfig.cmake**
Директория должна содержать : **OpenCVConfig.cmake**

Finding external libraries in CMake : 2. find_library()

Standard libraries of Linux/MinGW are found simply by name (without *lib-* prefix):

```
target_link_libraries(dijkdemo gdi32 png)        # Needed by CImg
```

Alternative names can be checked by **find_library()**:

```
find_library(GLFW_LIB NAMES glfw glfw3)
find_library(GLEW_LIB NAMES glew GLEW glew32)
find_package(OpenGL REQUIRED)
target_link_libraries(triangle ${GLEW_LIB} ${GLFW_LIB} ${OPENGL_gl_LIBRARY})
```

C++ threads (links thread library on Unix/Linux):

```
find_package(Threads)
target_link_libraries(${PROJECT_NAME} ${CMAKE_THREAD_LIBS_INIT})
```

Finding external libraries in CMake : 3. pkg-config

```
cmake_minimum_required(VERSION 3.1)
project(hello)
set(CMAKE_CXX_STANDARD 14)

# gtkmm libraries
find_package(PkgConfig)        # Find pkg-config
pkg_check_modules(GTKMM gtkmm-3.0)
link_directories(${GTKMM_LIBRARY_DIRS})
include_directories(${GTKMM_INCLUDE_DIRS})

add_executable(${PROJECT_NAME}
   HelloWorld.h
   main.cpp
   )
target_link_libraries(${PROJECT_NAME} ${GTKMM_LIBRARIES}   )
```

Thank you for your attention !

title

text