Using `CMake` tool

D. S. Hwang
IDA Lab.
Dankook University

June 9, 2020

1 `Make` vs. `CMake`

★ The `make` utility and `Makefile`s provide a build system that can be used to manage the compilation and recompilation of programs that are written in any programming language.

★ `CMake` is a cross-platform `Makefile`

- UNIX/Linux ⇒ Makefiles
- Windows ⇒ Visual Studio Projects/Workspaces
- Apple ⇒ Xcode

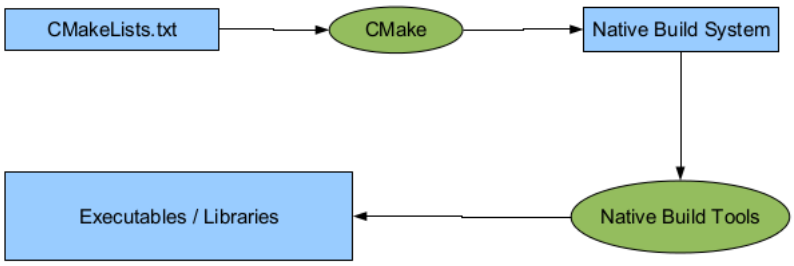
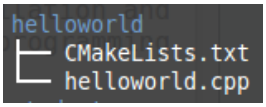
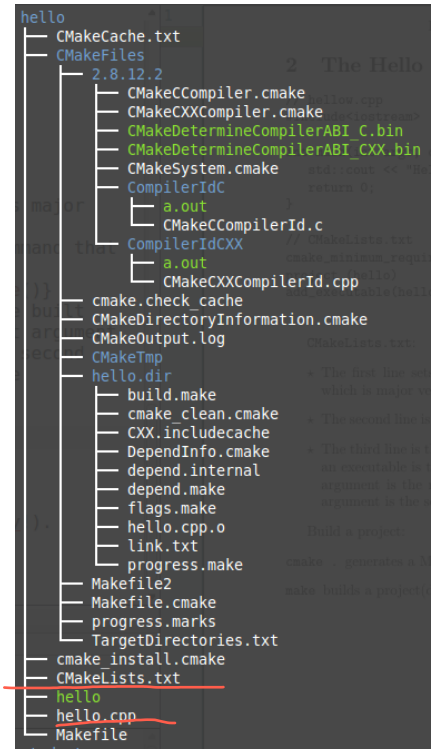


Figure 1: `CMake` builds a binary



(a) Project `helloworld`



(b) `CMake` generation of project `Hello`

Figure 2: Changes in the directory

2 The Hello World Example

```
// hellow.cpp
#include<iostream>

int main(int argc, char *argv[]){
   std::cout << "Hello World!" << std::endl;
   return 0;
}

// CMakeLists.txt
cmake_minimum_required(VERSION 2.8.9)
project (hello)
add_executable(hello helloworld.cpp)
```

`CMakeLists.txt`:

★ The first line sets the minimum version of `CMake` for `hello` project, which is major version 2, minor version 8, and patch version 9.

★ The second line is the `project()` command that sets the project name.

★ The third line is the `add_executable()` command, which requests that an executable is to be built using the `hello.cpp` source file. The first argument is the name of the executable to be built, and the second argument is the source file from which to build the executable.

Using `CMake`:

`mkdir build; cd build` creates a `build` directory and changes the working directory.

`cmake ..` configures the package for your system and generates a makefile.

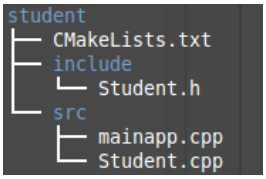
`make` builds the package(don't modifiy!).

`make install` installs the package.

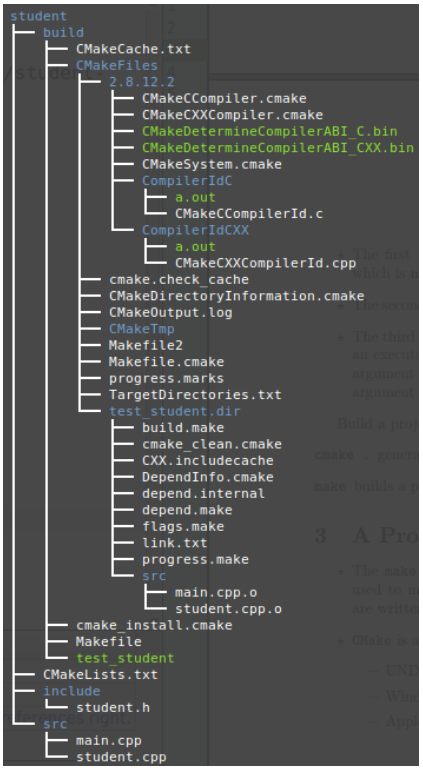
`make install` merges the last 2 steps into one.

3 A Project with Directories

```
// student.h
#include<string>

class Student{
private:
std::string name;
public:
Student(std::string);
virtual void display();
};

// student.cpp
#include <iostream>
#include "student.h"

using namespace std;

Student::Student(string name):name(name){}
```



(a) Project `directory_test`



(b) `CMake` generation of project `directory_test`

Figure 3: Changes in the directory

```
void Student::display(){
    cout << "A student with name " << this->name << endl;
}

// main.cpp
#include <iostream>
#include "student.h"

using namespace std;

Student::Student(string name):name(name){}

void Student::display(){
    cout << "A student with name " << this->name << endl;
```

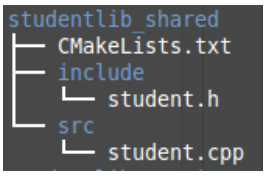
```
}

// CMakeLists.txt
cmake_minimum_required(VERSION 2.8.9)
project(directory_test)

#Bring the headers, such as Student.h into the project
include_directories(include)

#Can manually add the sources using the set command as follows:
#set(SOURCES src/main.cpp src/student.cpp)

#However, the file(GLOB...) allows for wildcard additions:
file(GLOB SOURCES "src/*.cpp")

add_executable(test_student ${SOURCES})
```

`CMakeLists.txt`:

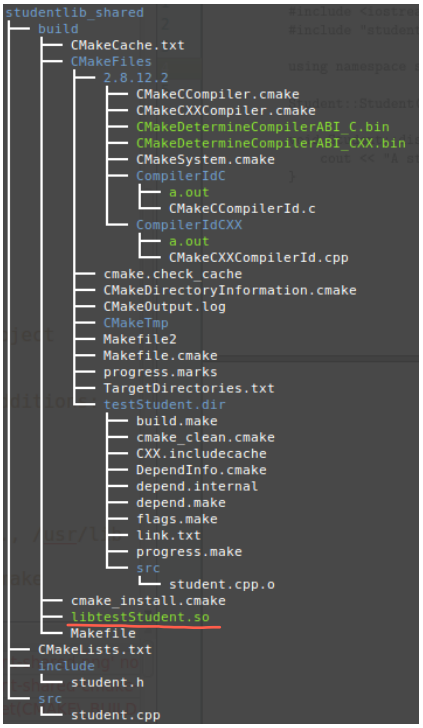
- ★ The `include_directories()` function is used to bring the header files into the build environment.
- ★ The `set(SOURCES ... )` function can be used to set a variable `SOURCES` that contains the name values of all of the source files `.cpp` in the project.
- ★ The `file()` command is used to add the source files to the project.
- ★ `GLOB` (or `GLOB_RECURSE`) is used to create a list of all of the files that meet the globbing expression (i.e., "src/*.cpp") and add them to a variable `SOURCES`.

4 Building a Shared Library (`.so`)

- ★ A shared library is built using the project `student`.
- ★ The project is almost the same, except that the `main.cpp` file is removed, as it is not relevant to a library build.
- ★ The shared library only contains a single `Student` class, however, that is sufficient to demonstrate the principles of building a library using `CMake`.



(a) Project `directory_test`



(b) `CMake` generation of project `directory_test`

Figure 4: Changes in the directory

```
// CMakeLists.txt
cmake_minimum_required(VERSION 2.8.9)
project(directory_test)
set(CMAKE_BUILD_TYPE Release)

# Bring the headers, such as Student.h into the project
include_directories(include)

# However, the file(GLOB...) allows for wildcard additions:
file(GLOB SOURCES "src/*.cpp")

# Generate the shared library from the sources
add_library(testStudent SHARED ${SOURCES})

# Set the location for library installation -- i.e., /usr/lib in this case
# not really necessary in this example. Use "sudo make install" to apply
```

```
install(TARGETS testStudent DESTINATION /usr/lib)
```

`CMakeLists.txt`:

- ★ The `set(CMAKE_BUILD_TYPE Release)` function is used to set the build type to be a release build. Instead of the `add_executable()` function that is used in previous examples, this example uses the `add_library()` function.
- ★ The library is built as a shared library using the `SHARED` flag (other options are: `STATIC` or `MODULE`) , and the `testStudent` name is used as the name of the shared library.
- ★ The last line uses the `install()` function to define an installation location for the library (in this case it is `/usr/lib`).
- ★ Deployment is invoked using a call to `sudo make install` in this case.
- ★ The shared library is `libtestStudent.so`.

5 Building a Static Library (`.a`)

- ★ A statically-linked library is created at compile time to contain all of the code code relating the library- essentially it makes copies of any dependency code, including that in other libraries.
- ★ This results in a library that is typically larger in size than the equivalent shared library, but because all of the dependencies are determined at compile time, there are fewer run-time loading costs and the library may be more platform independent.
- ★ Unless you are certain that you require a static library, you should use a shared library.

```
// CMakeLists.txt
cmake_minimum_required(VERSION 2.8.9)
project(directory_test)
set(CMAKE_BUILD_TYPE Release)

# Bring the headers, such as student.h into the project
include_directories(include)
```

```
# However, the file(GLOB...) allows for wildcard additions:
file(GLOB SOURCES "src/*.cpp")

# Generate the static library from the sources
add_library(testStudent STATIC ${SOURCES})

# Set the location for library installation -- i.e., /usr/lib in this case
# not really necessary in this example. Use "sudo make install" to apply
install(TARGETS testStudent DESTINATION /usr/lib)
```

`CMakeLists.txt`:

★ determine the constituents of a static library using the GNU `ar` (archive)

```
[/build % ar -t libtestStudent.a
 student.cpp.o
```

★ use the GNU `nm` command to list the symbols in object files and binaries.

★ In this case, the command lists the symbols in the student library and their types

`T` is code.

`U` is undefined.

`R` is read-only data.

```
[build % nm -C libtestStudent.a

student.cpp.o:
                 U __cxa_atexit
                 U __dso_handle
0000000000000000 t _GLOBAL__sub_I__ZN7StudentC2ESs
0000000000000000 T Student::display()
0000000000000090 T Student::Student(std::string)
0000000000000090 T Student::Student(std::string)
                 U std::ctype<char>::_M_widen_init() const
                 U std::ostream::put(char)
                 U std::ostream::flush()
                 U std::basic_string<char, std::char_traits<char>, std::allocator<char> >::basic_string(std::string const&)
                 U std::ios_base::Init::Init()
                 U std::ios_base::Init::~Init()
                 U std::basic_ostream<char, std::char_traits<char> >& std::__ostream_insert<char, std::char_traits<char> >(std::basic_ostream<char
                 U std::__throw_bad_cast()
                 U std::cout
0000000000000000 b std::__ioinit
0000000000000000 V typeinfo for Student
0000000000000000 V typeinfo name for Student
0000000000000000 V vtable for Student
                 U vtable for __cxxabiv1::__class_type_info
```

6 Using a Shared or Static Library

- ★ `CMake` can be used to generate the `Makefiles` in your project in order to simplify this process.
- ★ `CMakeLists.txt` file can be used to build a program that links to a library:- either shared or static.
- ★ For this example the shared library that is generated before is used and a short C++ program is written that utilizes the functionality of that library.

```
// CMakeLists.txt

cmake_minimum_required(VERSION 2.8.9)
project (TestLibrary)

# For the shared library:
set ( PROJECT_LINK_LIBS libtestStudent.so )
link_directories( ~/exploringBB/extras/cmake/studentlib_shared/build )

# For the static library:
#set ( PROJECT_LINK_LIBS libtestStudent.a )
#link_directories( ~/exploringBB/extras/cmake/studentlib_static/build )

include_directories(~/exploringBB/extras/cmake/studentlib_shared/include)

add_executable(libtest libtest.cpp)
target_link_libraries(libtest ${PROJECT_LINK_LIBS} )
://derekmolloy.ie/hello-world-introductions-to-cmake/

// libtest.cpp
#include"student.h"

int main(int argc, char *argv[]){
   Student s("Joe");
   s.display();
   return 0;
}
```

7 Summary

- ★ review a short and practical introduction to `CMake`
- ★ demonstrate how it can be used to build: a simple project, a separately compiled project, and a shared library.
- ★ These are the operations that you are likely to perform and the examples above can act as templates.
- ★ check out the up-to-date documentation on `CMake` (www.cmake.org)