

Building Projects

If you use vim, you can bind `CTRL-B` to build a cmake project

```
nnoremap <C-b> :!cmake -S . -B ./build/ && cmake --build ./build
```

Building from Bash

Within bash, you can easily build a single C++ source file into an executable with `g++ -g -Wall source.cpp`, this will output an `a.out` file which is the default name for the resulting executable. If you want to name this file, pass the `-o` argument as follows `g++ -g -Wall source.cpp -o executable`

Now, we can run the executable with `./executable` or `./a.out`, so long as you are within the directory where it exists and the build completed normally.

Note that you can also build with `clang++`, and a C++ standard can be specified with `clang++ -std=c++11 source.cpp`

Make

A simple makefile can be seen below, which can compile a project with a single `make` command. Create your own with details specific to your project, and name it `Makefile`

```
# Makefile
#-----

# Set Local Variables
CXX = g++
CXXFLAGS = -g -Wall

#-----

# Build executable
exe: driver.cpp lib.o
    ${CXX} ${CXXFLAGS} driver.cpp lib.o -o exe

#-----

# Compile sources
lib.o: lib.cpp lib.h
    ${CXX} ${CXXFLAGS} -c lib.cpp -o lib.o
```

```
#-----  
# Clean last build  
clean:  
rm -f *.o exe
```

Note that `CXX` and `CXXFLAGS` are just local variables that define our compiler, `g++`, and the flags we'd like to set for it to use. Later, within this makefile, we can use them with the `${VARIABLE}` syntax. This lets us define things like filepaths, flags, and other lines we reuse frequently within our makefile.

So, the `${CXX} ${CXXFLAGS} -c lib.cpp -o lib.o` line is equivalent to `g++ -g -Wall -c lib.cpp -o lib.o` - In this case, we add the `-c` argument to tell `g++` to only compile and output object files without linking and building an executable.

`make clean` will remove all previous build files, `make` will recompile our project given the sources have been updated. Make will not recompile sources that are not modified or dependent on modified files.

For a more portable makefile, feel free to use the template below, and just replace the variables with whatever is relevant to your project.

```
CC = cc  
  
FLAGS = -g -Wall  
  
SRC = source.c mylib-source.c  
  
LINK_TARGET = ls3  
  
REBUILDABLES = $(LINK_TARGET)  
  
# Build the example  
#####  
#####  
  
all: $(LINK_TARGET)  
  
$(LINK_TARGET): $(SRC)  
${CC} $(FLAGS) $^ -o $@  
  
# Clean previous builds
```

```
#####  
#####
```

clean:

```
rm -f $(REBUILDABLES)
```

CMake

Our example project will have the following file structure -

```
some/dir/project/  
├─ src  
│   ├── CMakeLists.txt  
│   ├── header.h  
│   ├── lib-test.cpp  
│   ├── one.cpp  
│   └─ two.cpp  
└─ CMakeLists.txt
```

This is a simple format and useful for learning CMake, once you have this working you can reorganize it as needed or follow a more in-depth tutorial elsewhere, this is only an example of a simple 'hello world program', so we won't need to create any extra subdirectories.

Within the root directory we can setup our project by defining the relevant subdirectories and cmake options, see the `CMakeLists.txt` file below

```
#####  
#####  
  
## A basic example of building an executable with CMake and linking libraries  
## Legal : All content (c) 2020 Shaun Reed, all rights reserved.  
## Author : Shaun Reed  
  
#####  
#####  
  
# project/CMakeLists.txt  
  
  
# Project setup  
cmake_minimum_required(VERSION 2.8)  
  
# Here, we name our project  
project(hello-world CXX)
```

```
include_directories(${CMAKE_CURRENT_SOURCE_DIR} ${CMAKE_CURRENT_BINARY_DIR})

# Testing
# An example of how an option could create multiple build paths
option(CMAKE_FIRST_TEST "Should we build our test application?" ON)
if(CMAKE_FIRST_TEST)
    ☐# Since this is set to ON, we add the subdirectory for our source code
    ☐# Be sure this reflects the folder name containing the next CMakeLists to further direct cmake
    add_subdirectory(src)
endif()
```

Now, from within the `project/src/` directory, we create the following CMakeLists.txt to build our libraries and executable -

```
#####
#####
## About : Building an executable with CMake and linking custom libraries
## Legal : All content (c) 2020 Shaun Reed, all rights reserved.
## Author : Shaun Reed
#####
#####
# project/src/CMakeLists.txt

# Creating executables
set(FIRST_EXECUTABLE_CMAKE_SOURCES one.cpp)
add_executable(one ${FIRST_EXECUTABLE_CMAKE_SOURCES})

set(SECOND_EXECUTABLE_CMAKE_SOURCES two.cpp)
add_executable(two ${SECOND_EXECUTABLE_CMAKE_SOURCES})

#Creating libraries
set(FIRST_STATIC_LIBRARY lib-test.cpp)
add_library(TestLibrary STATIC ${FIRST_STATIC_LIBRARY})

#Linking libraries to executables
target_link_libraries(one TestLibrary)
target_link_libraries(two TestLibrary)
```

Now, assuming all the source files configured with CMake above are present and valid, we can run the following commands to build and compile our project.

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

That's it! Running the commands above ensure that our build files output won't clutter up our project. First, we create a new directory to build into, then we move inside it and run `cmake ..` - this tells cmake to run on the previous directory and as a result outputs the build files into our current directory. Then, we can `cmake --build .` to build the files cmake created, which outputs and executables defined in our project.

Revision #11

Created 28 February 2020 23:04:17 by Shaun Reed

Updated 1 March 2025 13:53:33 by Shaun Reed