Introduction to Linux build systems

Contents

1 The make utility and the make files **1**
- 1.1 The build process . . . 2
- 1.2 makefile simple script . . . 2
- 1.3 recipes, targets and dependencies . . . 2
- 1.4 make implicit rules . . . 3
- 1.5 make variables . . . 3
  - 1.5.1 make built-in variables . . . 3
- 1.6 make built-in functions . . . 3
- 1.7 automatic variables . . . 3
- 1.8 Passing arguments to a makefile (Conditional Processing) . . . 4
- 1.9 Static pattern rules . . . 4
- 1.10 special targets . . . 4
- 1.11 Include other makefiles . . . 4

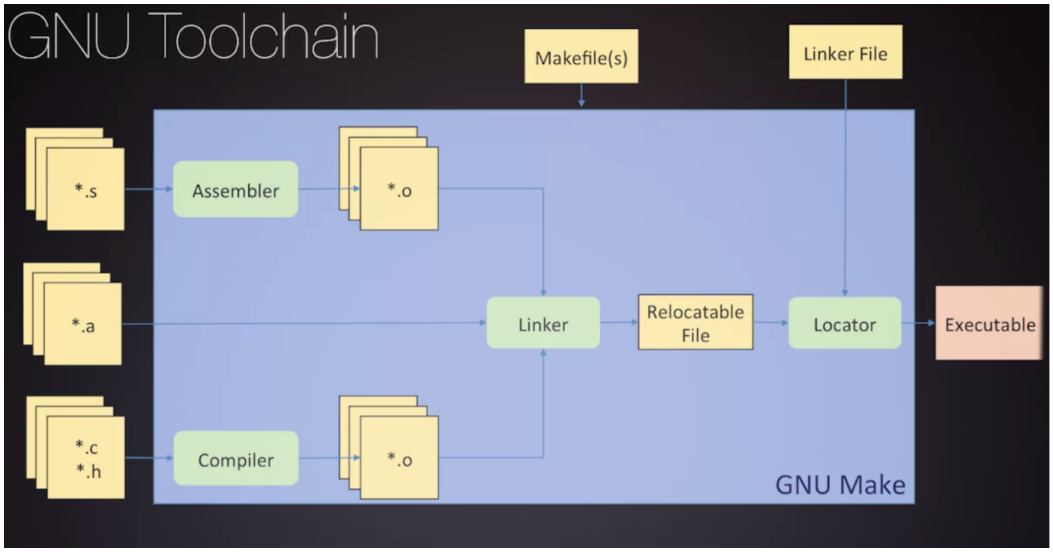
2 Using Ninja build system **5**
- 2.1 Simple Ninja buid . . . 5
- 2.2 Generating Ninja files from code . . . 6
  - 2.2.1 Use misc/ninja_syntax.py python script . . . 6
  - 2.2.2 Use meta-build system like cmake . . . 6

3 intro to cmake **6**
- 3.1 minimal cmake program script . . . 6
- 3.2 Adding sub-directories (Including other CMakeLists.txt) . . . 9
- 3.3 cmake generators . . . 9
  - 3.3.1 Using the cmake to generate for Visual Studio . . . 10

1 The make utility and the make files

It's an automatic build system, that drives the complier and generates an output executable files. and you need to know that in some situations in order to install a software in Linux, you just need to build it first from its source code.

1.1 The build process



1.2 makefile simple script

1.3 recipes, targets and dependencies

The building process is a set of (rules), every rule might be dependent on another rule and so on.
A recipe can be constructed by a **target**, **dependencies or prerequisites**, and **command or recipe**

Listing 1: make rule structure

```
[target ...] : [prerequisite ...]
    [recipe1]
    [recipe2]
```

Note: in *make*, each line of a recipe starts with a TAB, and make sure it's a TAB as the *make* util sometimes is sensitive to TABs and spaces while reading a recipe.

In order to invoke the **make.exe** to run, you can invoke with the following syntax:
make [option]... [target]... [macro=def]...

1.4 make implicit rules

1.5 make variables

In make, we have two flavors of variables:

- The value of a simply expanded variable is scanned once and for all, expanding any references to other variables and functions. It does not contain any references to other variables; it contains their values as of the time this variable was defined. (are expanded once at the time of the variable definition
- if it contains references to other variables, these references are expanded whenever this variable is substituted (in the course of expanding some other string). When this happens, it is called recursive expansion. (are expanded when the variable is substituted in) Example:

Listing 2: building and linking with libraries

```
foo = $(bar)
bar = $(ugh)
x := foo
y := $(x) bar

ARCH := (shell arch)
CWD  := (shell pwd)
OS   := (shell uname)
```

1.5.1 make built-in variables

1.6 make built-in functions

Note: You might need to visit the *make* documentation for any all the list of the built-in functions.

1.7 automatic variables

These variables have values computed afresh for each rule that is executed, based on the target and prerequisites of the rule. Its used to reduce the amount of typing in for a rule

Listing 3: building and linking with libraries

```
\$\@    the current target
$^, $<  each of the prerequisites

Example:
%.o: %.c
   $(CC) -c $^ $(CFLAGS) -o $@
```

1.8 Passing arguments to a makefile (Conditional Processing)

Make allows you to use if conditions to change the build process and this means more flexibility

Listing 4: building and linking with libraries

```
If ($(OS), Windows\_NT)
    RM = del /s
    MV = move /Y
else
    RM = rm  f
    MV = mv  force
Endif

ifeq ( $(PLATFORM), MSP )
    CPU = cortex-m4
endif

From commandline:
Then you can use this:  "make all PLATFORM=MSP"
```

1.9 Static pattern rules

Static pattern rules are rules which specify multiple targets and construct the prerequisite names for each target based on the target name.

[target] : target-pattern : [prerequisite-patterns]

The target specifies the targets the rules applies to. The target-pattern and prerequisite-patterns specify how to compute the prerequisites of each target. Each target is matched against the target-pattern to extract a part of the target name, called the stem. This stem is substituted into each of the prerequisite-patterns to make the prerequisite names (one from each prerequisite-pattern).

Listing 5: building and linking with libraries

1.10 special targets

1.11 Include other makefiles

You can include other make files: usually a file for sources, and includes is made for the project, you can just use the *include* directive

2 Using Ninja build system

Another types of build systems like **make** which is fast and lightweight and uses parallel builds.

Why Ninja: Ninja, a small build system with a focus on speed. It's built to be fast build system, and you will might notice that the speed will increase based on the CPU cores.

2.1 Simple Ninja buid

Listing 6: Simple build.ninja file

```
# build.ninja
cc     = g++
cflags = -Wall
lflags = -lstdc++

rule compile
  command = $cc $cflags -c $in -o $out

rule link
  command = $cc $lflags $in -o $out

build main.o: compile main.cpp
build output: link main.o

default output
```

Listing 7: Ouput from Ninja build

```
$ ls
build.ninja main.cpp

$ cat main.cpp
#include <iostream>

int main(int argc, char** argv){

    std::cout << "Hello World!" << std::endl;

    return 0;
}

$ ninja
[2/2] g++ -lstdc++ main.o -o output

$ ls
```

```
build.ninja main.cpp main.o output.exe

$ ./output.exe
Hello World!
```

2.2 Generating Ninja files from code

2.2.1 Use misc/ninja_syntax.py python script

2.2.2 Use meta-build system like cmake

3 intro to cmake

I think now you might asking what is really the difference between the cmake and the make environment.
The answer is the *make* is a build system that drives the complier and the linker till the output executable is generated, but the *cmake* is a cross-platform build system generator that can generate the makefiles themselves, then the role of the *make* utility comes to build the software.

Simply **cmake** is meta-build system which is a build system that generates other build systems. (Note: Other meta-build systems like Gyn, and Gn).

cmake is not only used for generating builds, but also it's used for packaging and installation!.

3.1 minimal cmake program script

Listing 8: Simple cmake script

```
cmake\_minimum\_required(VERSION 3.12)
project(SimpleProject VERSION 1.0.0)

add_executable(cmake-test main.cpp)
```

Generating and building the project:

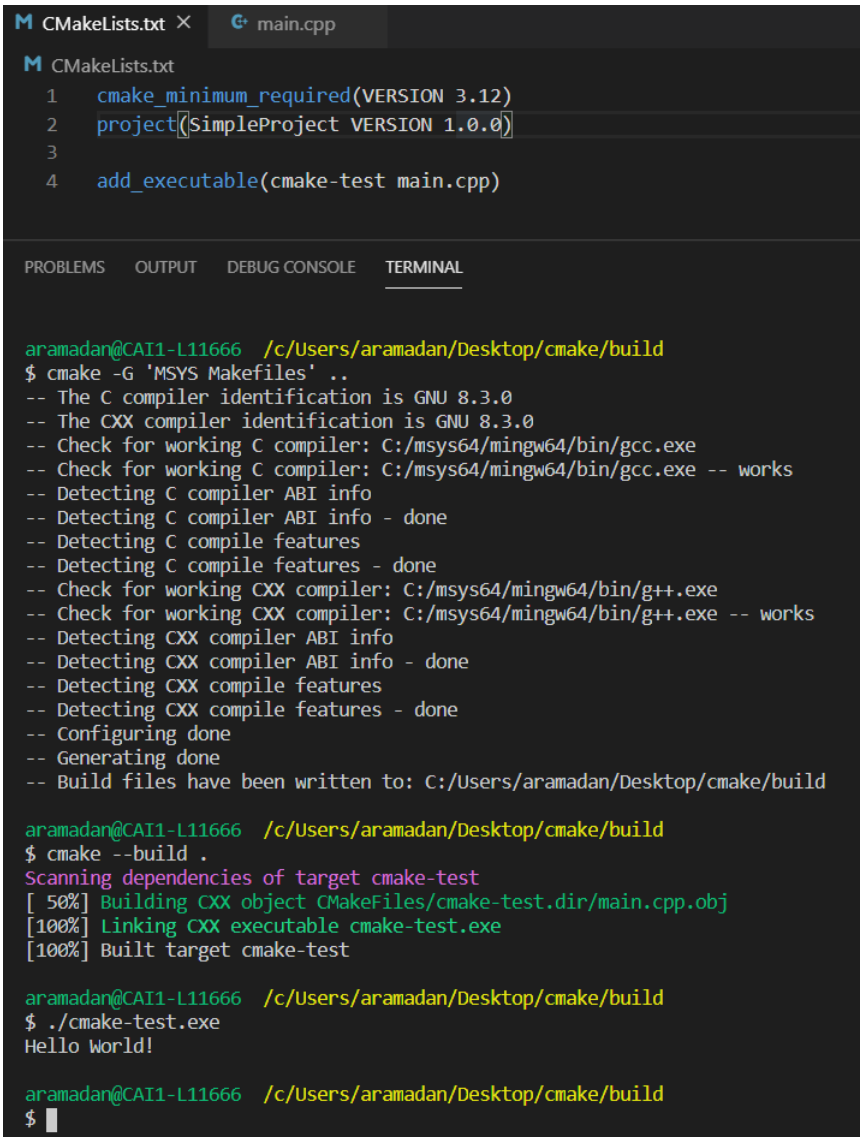
Listing 9: Simple cmake script

```
cmake\_minimum\_required(VERSION 3.12)
project(SimpleProject VERSION 1.0.0)

add_executable(cmake-test main.cpp)
```

Listing 10: Building a simple cmake

```
aramadan@CAI1-L11666 /c/Users/aramadan/Desktop/cmake
```

```
$ ls
build  CMakeLists.txt  main.cpp

aramadan@CAI1-L11666 /c/Users/aramadan/Desktop/cmake
$ cd build/

aramadan@CAI1-L11666 /c/Users/aramadan/Desktop/cmake/build
$ cmake -G 'MSYS Makefiles' ..
-- The C compiler identification is GNU 8.3.0
-- The CXX compiler identification is GNU 8.3.0
-- Check for working C compiler: C:/msys64/mingw64/bin/gcc.exe
-- Check for working C compiler: C:/msys64/mingw64/bin/gcc.exe -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: C:/msys64/mingw64/bin/g++.exe
-- Check for working CXX compiler: C:/msys64/mingw64/bin/g++.exe -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to:
    C:/Users/aramadan/Desktop/cmake/build

aramadan@CAI1-L11666 /c/Users/aramadan/Desktop/cmake/build
$ cmake --build .
Scanning dependencies of target cmake-test
[ 50%] Building CXX object CMakeFiles/cmake-test.dir/main.cpp.obj
[100%] Linking CXX executable cmake-test.exe
[100%] Built target cmake-test

aramadan@CAI1-L11666 /c/Users/aramadan/Desktop/cmake/build
$ ./cmake-test.exe
Hello World!
```



You can also build and link with libraries, let's consider that we have a new files **"libfoo.hpp, and libfoo.cpp"** and let's print the hello world from inside.

Listing 11: building and linking with libraries

```
aramadan@CAI1-L11666 /c/Users/aramadan/Desktop/cmake/build
$ cmake --build .
-- Configuring done
-- Generating done
-- Build files have been written to:
    C:/Users/aramadan/Desktop/cmake/build
```

```
[ 50%] Built target foo
Scanning dependencies of target cmake-test
[ 75%] Linking CXX executable cmake-test.exe
[100%] Built target cmake-test

aramadan@CAI1-L11666 /c/Users/aramadan/Desktop/cmake/build
$ ./cmake-test.exe
Hello CMake from lib!
```

the output as expected, but let's see how can we edit the CMakeLists file.

Listing 12: building and linking with libraries

```
cmake_minimum_required(VERSION 3.12)
project(SimpleProject VERSION 1.0.0)

#added line to create libfoo with the respective hpp and cpp files
add_library(foo
    libfoo.cpp
    libfoo.hpp
)

add_executable(cmake-test main.cpp)

#added line to link with the libfoo
target_link_libraries(cmake-test PRIVATE foo)
```

3.2 Adding sub-directories (Including other CMakeLists.txt)

You can add sub-directory to be combined with the current while building. you can achieve that by adding creating a new sub-directory and creating a separate CMakeLists for it, then using the **add_subdirectory(directory_name)** inside the main CMakeLists.txt.

Interface Modes

- Public In **PUBLIC** interface, Any one links with the library will be have the includes also the all the directives visible to him.
- Private In **PRIVATE** only us has the ability to see our definitions
- Interface In **INTERFACE** We can't see the definitions, only our consumers.

3.3 cmake generators

cmake is a cross-platform generator which allows generating a complete build environment, and you can not only using it for generating a makefiles but you can use also to generate other build environments templates for XCode, Visual

Studio, Ninga build, ... etc. and you can achieve using a certain generate by specifying it using **'-G'** option.

Listing 13: Using a specific generator

```
aramadan@CAI1-L11666 /c/Users/aramadan/Desktop/cmake/build
$ cmake -G 'MSYS Makefiles' ..
# Here we are using a makefiles under the MSYS environment
```

Note: for a list of all the generators use ***cmake -G***

3.3.1 Using the cmake to generate for Visual Studio

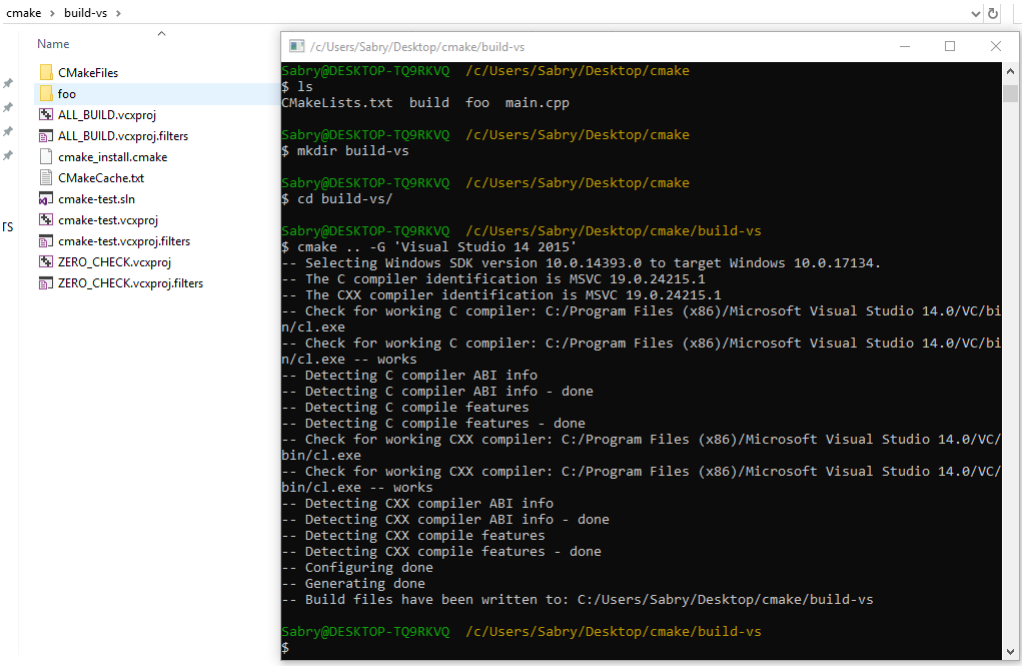
As we just discussed you can use *cmake* to generate Visual Studio projects by specifying a certain generator. the next example will show that.

Listing 14: building and linking with libraries

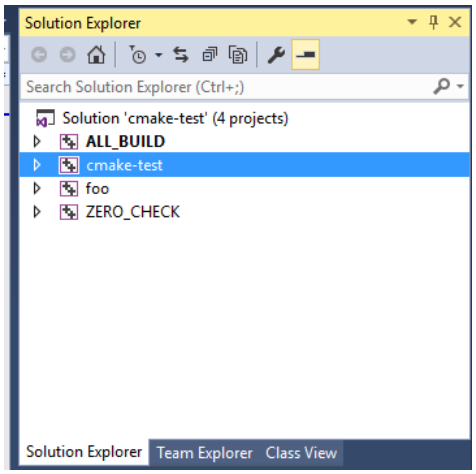
```
Sabry@DESKTOP-TQ9RKVQ /c/Users/Sabry/Desktop/cmake
$ ls
CMakeLists.txt build foo main.cpp

Sabry@DESKTOP-TQ9RKVQ /c/Users/Sabry/Desktop/cmake
$ mkdir build-vs

Sabry@DESKTOP-TQ9RKVQ /c/Users/Sabry/Desktop/cmake
$ cd build-vs/

Sabry@DESKTOP-TQ9RKVQ /c/Users/Sabry/Desktop/cmake/build-vs
$ cmake .. -G 'Visual Studio 14 2015'
-- Selecting Windows SDK version 10.0.14393.0 to target Windows
    10.0.17134.
-- The C compiler identification is MSVC 19.0.24215.1
-- The CXX compiler identification is MSVC 19.0.24215.1
-- Check for working C compiler: C:/Program Files (x86)/Microsoft Visual
    Studio 14.0/VC/bin/cl.exe
-- Check for working C compiler: C:/Program Files (x86)/Microsoft Visual
    Studio 14.0/VC/bin/cl.exe -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: C:/Program Files (x86)/Microsoft
    Visual Studio 14.0/VC/bin/cl.exe
-- Check for working CXX compiler: C:/Program Files (x86)/Microsoft
    Visual Studio 14.0/VC/bin/cl.exe -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
```

```
-- Configuring done
-- Generating done
-- Build files have been written to:
    C:/Users/Sabry/Desktop/cmake/build-vs
```

The output is a normal visual studio 2015 solution as follow:



You can normally open the solution with the visual studio, and you might notice that the *cmake* has generated four projects as follow;



two of them are our foo library and the other is the cmake-test project, but what about the **ALL_BUILD**, and **ZERO_CHECK** projects.

- ALL_BUILD project is used to build every signal project in the solution, you can think of it as *make all*
- ZERO_CHECK is used to sync with the original CMakeLists.txt file, so if you changed anything in the CMakeLists.txt you should build this project to sync with the visual studio solution.

cmake can be used to generate a lot of build systems, such that *make* build files, or XCode projects, even visual studio projects ... etc.