
Build Systems

Feb 08, 2019

CONTENTS

1	Introduction	3
1.1	Learning Objectives	3
1.2	Reading and Reference Material	3
1.3	What is a Build System?	3
2	Motivation	5
2.1	One Source File	5
2.2	Build System: Compiler Driver	5
2.3	Reusable Source File	5
2.4	Sharing Source Files	6
2.5	Build System: Shell Script	6
2.6	Sharing Source Files	7
2.7	Sharing Object Files	7
2.8	Sharing Multiple Sources	8
2.9	Static Libraries	8
2.10	Shared Libraries	9
2.11	Review of File Types	9
2.12	Build System: Shell Script	10
3	Build System: Make	11
3.1	Build Dependencies	11
3.2	Makefile	11
3.3	Run Make Tool	12
3.4	Implicit Dependencies	12
3.5	Makefile: Implicit Dependencies	13
3.6	Build System: Make	14
3.7	Build System: MSBuild	14
3.8	Example Build Systems	14
4	Generating Build Systems	15
4.1	Build System Generators	15
4.2	CMake	15
4.3	CMake Example Code	15
4.4	Running CMake	16
4.5	CMake-generated Makefiles	16
4.6	CMake GUI	17
4.7	CMake-generated VS Project	17
4.8	CMake Syntax Primer	18
5	Conclusion	19

5.1	Build Systems Summary	19
-----	---------------------------------	----

RPI Open Source Software - Spring 2019

Brad King, Kitware, Inc.

- Maintainer of CMake
- RPI '00 BS, '08 PhD

Licensed under: CC-BY-SA 4.0 <https://creativecommons.org/licenses/by-sa/4.0/>

Modifications by Wes Turner

INTRODUCTION

1.1 Learning Objectives

1. Learn the relationship between project complexity and build systems
2. Become familiar with and use the open source build utility “make”
3. Become familiar with and use the open source meta-build utility “cmake” for multiplatform development

1.2 Reading and Reference Material

- Overview of build systems
 - <https://medium.com/@julienjorge/an-overview-of-build-systems-mostly-for-c-projects-ac9931494444>
- Makefiles
 - <https://www.tutorialspoint.com/makefile/>
- CMake
 - <https://cmake.org/cmake/help/v3.13/index.html> – read cmake-buildsystem

1.3 What is a Build System?

- Specifies how to turn source files into useful programs.



- Organizes code to share among programs.
- Requirements vary by language, tools, and platform; here we focus on C and C++ languages.

MOTIVATION

2.1 One Source File

Consider a source file `hi.c`:

```
#include <stdio.h>
int main() {
    printf("hello\n");
    return 0;
}
```

Compile and run the program:

```
$ cc hi.c -o hi
$ ./hi
hello
```

2.2 Build System: Compiler Driver

The compiler driver is a simple build system.

It runs the compiler and linker internally:

```
$ gcc hi.c -o hi -###
cc1 hi.c -o /tmp/tmp1.o
collect2 -o hi /tmp/tmp1.o -lgcc ...
```

2.3 Reusable Source File

Declare a `hello()` function in a `hello.h` header:

```
void hello(void);
```

Implement the function in a `hello.c` source:

```
#include "hello.h"
#include <stdio.h>
void hello(void) {
```

(continues on next page)

(continued from previous page)

```
printf("hello\n");  
}
```

Copy `hi.c` to `hi1.c` and update the main program in the `hi1.c` source:

```
#include "hello.h"  
int main(void) {  
    hello();  
    return 0;  
}
```

Give both source files to the compiler driver:

```
$ cc hi1.c hello.c -o hi1  
$ ./hi1  
hello
```

The compiler driver runs the compiler and linker internally:

```
$ gcc hi1.c hello.c -o hi1 -###  
cc1 hi1.c -o /tmp/tmp1.o  
cc1 hello.c -o /tmp/tmp2.o  
collect2 -o hi1 /tmp/tmp1.o /tmp/tmp2.o -lgcc ...
```

2.4 Sharing Source Files

Now add a `hi2.c` executable sharing the `hello()` function:

```
#include "hello.h"  
int main(void) {  
    hello();  
    hello();  
    return 0;  
}
```

Use `hello.c` source file for both programs:

```
$ cc hi1.c hello.c -o hi1  
$ cc hi2.c hello.c -o hi2  
$ ./hi1  
hello  
$ ./hi2  
hello  
hello
```

2.5 Build System: Shell Script

List commands in a shell script, e.g. `build.sh`:

```
cc hi1.c hello.c -o hi1  
cc hi2.c hello.c -o hi2
```

Run the script to drive the build:

```
$ sh -x build.sh
+ cc hi1.c hello.c -o hi1
+ cc hi2.c hello.c -o hi2
```

2.6 Sharing Source Files

The compiler driver runs the compiler and linker internally:

```
$ gcc hi1.c hello.c -o hi1 -###
cc1 hi1.c -o /tmp/tmp1.o
cc1 hello.c -o /tmp/tmp2.o
collect2 -o hi1 /tmp/tmp1.o /tmp/tmp2.o -lgcc ...
$ gcc hi2.c hello.c -o hi2 -###
cc1 hi2.c -o /tmp/tmp1.o
cc1 hello.c -o /tmp/tmp2.o
collect2 -o hi2 /tmp/tmp1.o /tmp/tmp2.o -lgcc ...
```

- Compiles `hello.c` twice.
- Re-uses source file but not compiler output.

2.7 Sharing Object Files

- Compile `hello.c` to an *object file*.
- Use the object file to link each executable.
- Called “separate compilation”.

```
$ sh -x build.sh
+ cc -c hello.c -o hello.o
+ cc hi1.c hello.o -o hi1
+ cc hi2.c hello.o -o hi2
$ ./hi1
hello
$ ./hi2
hello
hello
```

The compiler driver runs the compiler and linker internally:

```
$ gcc hello.c -o hello.o -###
cc1 hello.c -o hello.o
$ gcc hi1.c hello.o -o hi1 -###
cc1 hi1.c -o /tmp/tmp1.o
collect2 -o hi1 /tmp/tmp1.o hello.o -lgcc ...
$ gcc hi2.c hello.o -o hi2 -###
cc1 hi2.c -o /tmp/tmp1.o
collect2 -o hi2 /tmp/tmp1.o hello.o -lgcc ...
```

- Compiles `hello.c` only once.

2.8 Sharing Multiple Sources

Split `hello.c` into `hello1.c`:

```
#include "hello.h"
extern void print_hello(const char *s);
void hello(void) {
    print_hello("world");
}
```

and `hello2.c`:

```
#include <stdio.h>
void print_hello(const char *s) {
    printf("hello: %s\n", s);
}
```

```
$ sh -x build.sh
+ cc -c hello1.c -o hello1.o
+ cc -c hello2.c -o hello2.o
+ cc -c hi1.c -o hi1.o
+ cc -c hi2.c -o hi2.o
+ cc hi1.o hello1.o hello2.o -o hi1
+ cc hi2.o hello1.o hello2.o -o hi2
$ ./hi1
hello: world
$ ./hi2
hello: world
hello: world
```

Callers of `hello()` function must use both `hello1.o` and `hello2.o` together, but should not have to know that.

2.9 Static Libraries

Create an archive of object files; use to link executables:

```
$ sh -x build.sh
+ cc -c hello1.c -o hello1.o
+ cc -c hello2.c -o hello2.o
+ ar qc libhello.a hello1.o hello2.o
+ cc -c hi1.c -o hi1.o
+ cc -c hi2.c -o hi2.o

.. nextslide::
+ cc hi1.o libhello.a -o hi1
+ cc hi2.o libhello.a -o hi2
$ ./hi1
hello: world
$ ./hi2
hello: world
hello: world
```

List the object files in the archive:

```
$ ar t libhello.a
hello1.o
hello2.o
```

2.10 Shared Libraries

Link object files into a shared library; link executables to it:

```
$ sh -x build.sh
+ cc -fPIC -c hello1.c -o hello1.o
+ cc -fPIC -c hello2.c -o hello2.o
+ cc -shared -o libhello.so hello1.o hello2.o
+ cc -c hi1.c -o hi1.o
+ cc -c hi2.c -o hi2.o
+ cc hi1.o libhello.so -o hi1 -Wl,-rpath='$ORIGIN'
+ cc hi2.o libhello.so -o hi2 -Wl,-rpath='$ORIGIN'
$ ./hi1
hello: world
$ ./hi2
hello: world
hello: world
```

For OSX, we need to use:

```
+ cc hi1.o libhello.so -o hi1 -Wl,-rpath .
+ cc hi2.o libhello.so -o hi2 -Wl,-rpath .
```

View dependency of executable on shared library:

```
$ readelf -d hi1 | grep NEEDED
0x0000000000000001 (NEEDED) Shared library: [libhello.so]
0x0000000000000001 (NEEDED) Shared library: [libc.so.6]
$ readelf -d hi1 | grep RPATH
0x000000000000000f (RPATH) Library rpath: [$ORIGIN]
```

For OSX, we need to use:

```
$ otool -l hi1
```

2.11 Review of File Types

Source files (*.c, *.cpp) Define “symbols” implementing functions and storage of global data.

Header files (*.h, *.hpp) Define interfaces shared among source files (e.g. function prototypes).

Object files (*.o, *.obj on Windows) Compiler output from source files.

Executables (no extension, *.exe on Windows) Object files linked together into programs with `main`.

Static libraries (*.a, *.lib with MS tools)

- Archives of object files.
- Searched by linker for objects implementing needed symbols.

- All symbols with “extern linkage” exposed publicly.

Shared libraries (`*.so`, `.dylib` on OSX, `*.dll` on Windows)

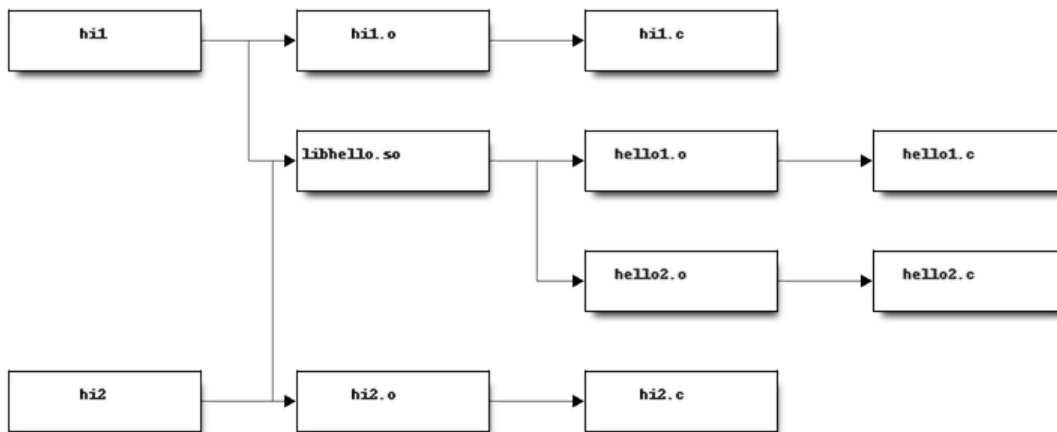
- Objects linked together into libraries loaded by programs at runtime.
- A subset of symbols with “extern linkage” exposed publicly via explicit markup.
- On Windows, associated “import library” (`.lib`).

2.12 Build System: Shell Script

- `build.sh` always runs all commands.
- No concurrency.
- No partial builds.
- No incremental rebuilds.
- Does not scale.
- Rarely used in practice.

BUILD SYSTEM: MAKE

3.1 Build Dependencies



3.2 Makefile

A Makefile expresses build dependencies:

```
all: hi1 hi2
hi1: hi1.o libhello.so
hi2: hi2.o libhello.so
hi1.o: hi1.c
hi2.o: hi2.c
libhello.so: hello1.o hello2.o
hello1.o: hello1.c
hello2.o: hello2.c
```

A Makefile also specifies build commands:

```
all: hi1 hi2
hi1: hi1.o libhello.so
    cc hi1.o libhello.so -o hi1 -Wl,-rpath='$$ORIGIN'
hi2: hi2.o libhello.so
```

(continues on next page)

(continued from previous page)

```
cc hi2.o libhello.so -o hi2 -Wl,-rpath='$$ORIGIN'
hi1.o: hi1.c
cc -c hi1.c -o hi1.o
hi2.o: hi2.c
cc -c hi2.c -o hi2.o
libhello.so: hello1.o hello2.o
cc -shared -o libhello.so hello1.o hello2.o
hello1.o: hello1.c
cc -fPIC -c hello1.c -o hello1.o
hello2.o: hello2.c
cc -fPIC -c hello2.c -o hello2.o
```

3.3 Run Make Tool

Run make tool to drive build process:

```
$ make
cc -c hi1.c -o hi1.o
cc -fPIC -c hello1.c -o hello1.o
cc -fPIC -c hello2.c -o hello2.o
cc -shared -o libhello.so hello1.o hello2.o
cc hi1.o libhello.so -o hi1 -Wl,-rpath='$ORIGIN'
cc -c hi2.c -o hi2.o
cc hi2.o libhello.so -o hi2 -Wl,-rpath='$ORIGIN'
$ ./hi1
hello: world
$ ./hi2
hello: world
hello: world
```

The make tool checks timestamps, follows dependencies:

```
$ make
make: Nothing to be done for 'all'.
$ touch hello2.c
$ make
cc -fPIC -c hello2.c -o hello2.o
cc -shared -o libhello.so hello1.o hello2.o
cc hi1.o libhello.so -o hi1 -Wl,-rpath='$ORIGIN'
cc hi2.o libhello.so -o hi2 -Wl,-rpath='$ORIGIN'
$ make
make: Nothing to be done for 'all'.
```

3.4 Implicit Dependencies

Header files (*.h) are *implicit* dependencies of compilation:

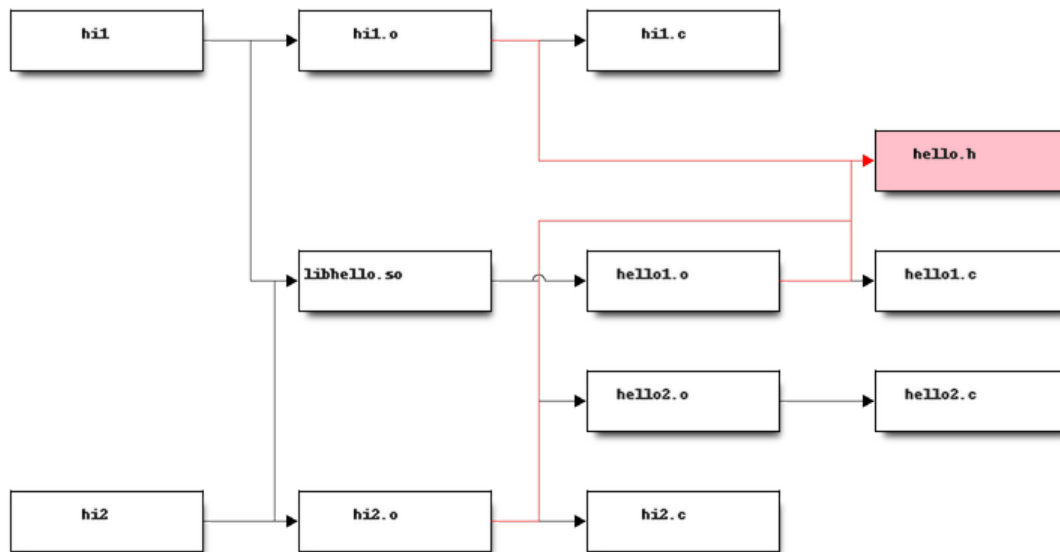
```
$ grep hello.h *.c
hello1.c:#include "hello.h"
hi2.c:#include "hello.h"
hi1.c:#include "hello.h"
```


The compiler can tell us about dependencies:

```
$ gcc -MM hello1.c -MT hello1.o
hello1.o: hello1.c hello.h
```

Implicit dependencies not yet expressed in our Makefile:

```
$ touch hello.h
$ make
make: Nothing to be done for 'all'.
```



3.5 Makefile: Implicit Dependencies

Extend our Makefile with implicit dependencies:

```
hi1.o: hello.h
hi2.o: hello.h
hello1.o: hello.h
```

```
$ make
cc -c hi1.c -o hi1.o
cc -fPIC -c hello1.c -o hello1.o
cc -shared -o libhello.so hello1.o hello2.o
cc hi1.o libhello.so -o hi1 -Wl,-rpath='$ORIGIN'
cc -c hi2.c -o hi2.o
cc hi2.o libhello.so -o hi2 -Wl,-rpath='$ORIGIN'
```

Everything but `hello2.o` rebuilds when `hello.h` changes.

3.6 Build System: Make

- Features:
 - Dependencies enable efficient, concurrent (re-)builds.
- Limitations:
 - Tricky to maintain implicit dependencies.
 - Platform- and tool-specific tables of commands.
 - Build rules do not re-run when commands change.
 - Need manual rules for “install” and “clean” operations.
 - Not reusable with IDEs like Visual Studio and Xcode.

3.7 Build System: MSBuild

Underlies Visual Studio 2010+ builds.

```
<Project DefaultTargets="Build" ToolsVersion="12.0" .\.\.>
...
<ItemGroup>
  <ClCompile Include="hi1.c" />
</ItemGroup>
<ItemGroup>
  <ProjectReference Include="hello.vcxproj">
    <Project>158CE2ED-F99F-4D09-A981-CF4C46D9A63B</Project>
  </ProjectReference>
</ItemGroup>
...
</Project>
```

- Features:
 - Create and update through Visual Studio IDE.
 - Handles implicit dependencies automatically.
 - Built-in “clean” operations.
- Limitations:
 - Platform- and tool-specific. Not portable.
 - Need manual rules for “install” operations.
 - Difficult to merge version control branches.

3.8 Example Build Systems

- **Make**: Canonical dependency-based build system.
- **Ninja**: An “assembly language for build systems”. Designed to be generated.
- **MSBuild**: Underlies Visual Studio 2010+ builds.
- **Waf**, **Scons**: Python-based build system frameworks.

GENERATING BUILD SYSTEMS

4.1 Build System Generators

Transform a common input specification into platform- and tool-specific build files. Examples:

- **GNU Build System (autotools)**: Generates `configure` script for distribution with source code to generate `GNU make` build files for local system.
- **CMake**: Generates for Make, Ninja, Visual Studio, or Xcode build files for local system.
- **Premake, GYP**: Generate re-distributable GNU Make, Visual Studio, and Xcode build files.

4.2 CMake

- Created by **Kitware** in 2000 to support cross-platform builds for the **Insight Toolkit**. Sponsored originally by the **US NLM**.
- Generalized incrementally over time.
- **KDE** (K Desktop Environment) switched to CMake in 2006; kicked off widespread adoption.
- Now de-facto standard for cross-platform C, C++, and **Fortran** projects.
- Homepage: <https://cmake.org>
- Documentation: <https://cmake.org/documentation>

4.3 CMake Example Code

Create a `CMakeLists.txt` file for our example:

```
cmake_minimum_required(VERSION 3.0)
project(Hello C)

add_library(hello SHARED hello1.c hello2.c hello.h)

add_executable(hi1 hi1.c)
target_link_libraries(hi1 hello)

add_executable(hi2 hi2.c)
target_link_libraries(hi2 hello)
```

4.4 Running CMake

Make an *out-of-source* build directory and run `cmake` tool:

```
$ mkdir build && cd build
$ cmake ..
...
-- Build files have been written to: ../../build
$ ls
CMakeCache.txt
CMakeFiles/
cmake_install.cmake
Makefile
```

Run `make` tool to drive the actual build:

```
$ make
Scanning dependencies of target hello
[ 14%] Building C object CMakeFiles/hello.dir/hello1.c.o
[ 28%] Building C object CMakeFiles/hello.dir/hello2.c.o
[ 42%] Linking C shared library libhello.so
[ 42%] Built target hello
Scanning dependencies of target hi1
[ 57%] Building C object CMakeFiles/hi1.dir/hi1.c.o
[ 71%] Linking C executable hi1
[ 71%] Built target hi1
Scanning dependencies of target hi2
[ 85%] Building C object CMakeFiles/hi2.dir/hi2.c.o
[100%] Linking C executable hi2
[100%] Built target hi2
```

Inspect results:

```
$ ls
CMakeCache.txt
CMakeFiles/
cmake_install.cmake
hi1
hi2
libhello.so
Makefile
$ ./hi1
hello: world
$ ./hi2
hello: world
hello: world
```

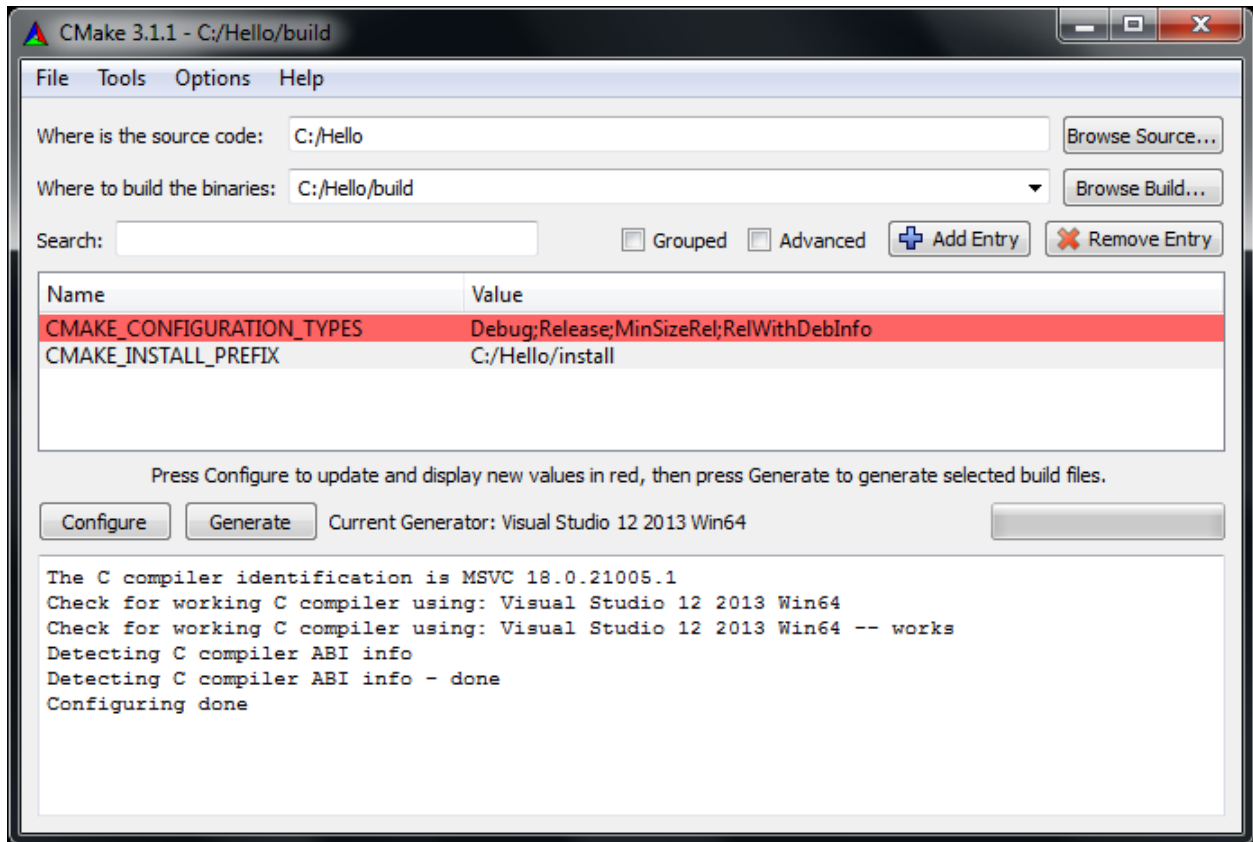
4.5 CMake-generated Makefiles

- Use platform- and tool-specific commands.
- Handle implicit dependencies automatically.
- Provide rules for “install” and “clean” operations.
- Display description of each step with progress percentage.

- Maintain pristine source with *out-of-source* builds.

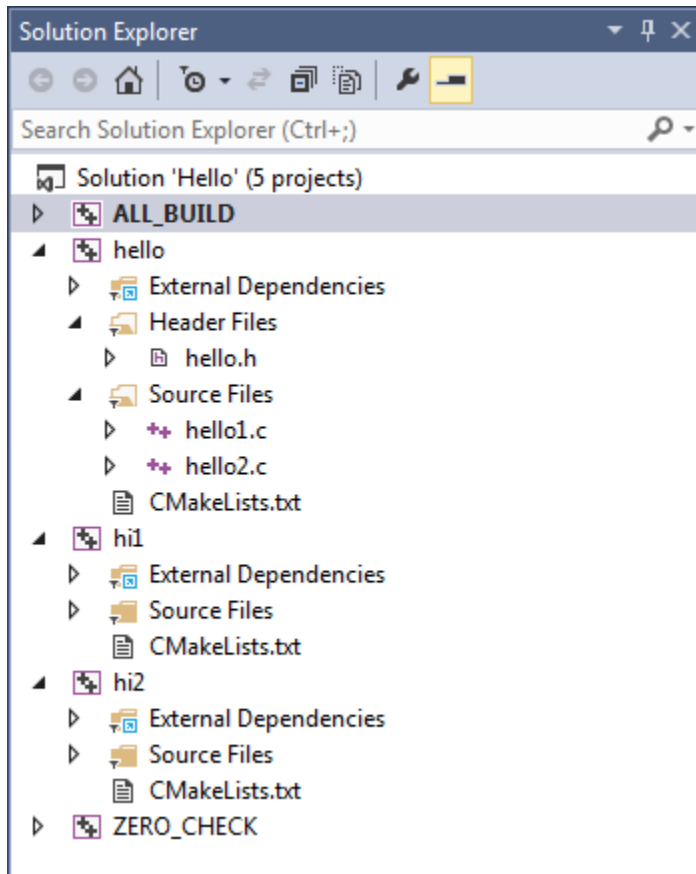
4.6 CMake GUI

Optionally use a GUI instead of a command prompt:



4.7 CMake-generated VS Project

Generated Visual Studio IDE project:



4.8 CMake Syntax Primer

- See the `cmake-language(7)` manual.
- `CMakeLists.txt` files denote source directories.
- `*.cmake` files implement modules and scripts.

```
# line comment
#[[bracket comment]]
set(VAR1 a) # "a"
set(VAR2 a b c) # "a;b;c"
message(${VAR2} "\n" # "abc" (unquoted)
        "${VAR2}" "\n" # "a;b;c" (quoted)
        [[${VAR2}]] "\n" # "${VAR2}" (bracket)
)
```

CONCLUSION

5.1 Build Systems Summary

- Turn sources into programs.
- Organize code to share among programs.
- Encode build dependencies.
- Generated for portability and scale.
- CMake used widely for C, C++, and Fortran.

-
- Your next lab session will focus on CMake.
 - Thank You