GWDG
Gesellschaft für wissenschaftliche Datenverarbeitung mbH Göttingen

GWDG AG-C

CMake

Dr. Freja Nordsiek

Table of contents

What is CMake

- https://cmake.org
- Build system
  - Find things on the system
  - Configure build
  - Setup build
  - Do build
  - Install, package, etc.
- Reasonably platform independent
- Can be used with any language, but has special support for specific ones

Starting Tiny

- Trivial C program

```c
int main(void)
{
    return 0;
}
```

- Trivial compilation

```
gcc -std=c11 -o trivial trivial.c
```

Less Tiny

- Simple C program

```c
#include <stdio.h>

#include "zlib.h"

int main(void)
{
    printf("Zlib version: %s\n", zlibVersion());
}
```

- Simple compilation

```
gcc -std=c11 -o simple simple.c -lz
```

Changing The Less Tiny

- A specific path for zlib

```
mkdir -p myzlib/include myzlib/lib
ln -f -s /usr/lib/libz.so myzlib/lib/libz.so
ln -f -s /usr/include/zlib.h myzlib/include/zlib.h
gcc -L myzlib/lib -I myzlib/include -std=c11 -o simple simple.c -lz
```

- Changing compiler to clang

```
clang -std=c11 -o simple simple.c -lz
```

- So far, this isn't too bad.

Two Code Files And A Header

small.h

```
#ifndef _SMALL_H
#define _SMALL_H
void printZlibVersion(void);
#endif
```

printv.c

```
#include <stdio.h>
#include "zlib.h"
#include "small.h"
void printZlibVersion()
{
    printf("Zlib version: %s\n", zlibVersion());
}
```

main.c

```
#include "small.h"
int main()
{
    printZlibVersion();
}
```

Compile All At Once

```
gcc -std=c11 -o small main.c printv.c -lz
```

Compile In Stages

```
gcc -std=c11 -c main.c
gcc -std=c11 -c printv.c
gcc -std=c11 -o small main.o printv.o -lz
```

Making Using zlib Optional

small.h

```c
#ifndef _SMALL_H
#define _SMALL_H
void printZlibVersion(void);
#endif
```

printv.c

```c
#include <stdio.h>
#include "zlib.h"
#include "small.h"
void printZlibVersion() {
#ifdef USE_ZLIB
    printf("Zlib version: %s\n", zlibVersion());
#else
    printf("NOT COMPILED WITH ZLIB!!!\n");
#endif
}
```

main.c

```c
#include "small.h"
int main() {
    printZlibVersion();
}
```

Compile Without zlib

```
gcc -std=c11 -o small_without main.c printv.c
```

Compile With zlib

```
gcc -std=c11 -DUSE_ZLIB -o small_with main.c printv.c -lz
```

Scaling Issues

- Gets more and more difficult as the following increase
  - Number of files
  - Number of dependencies
  - Number of configuration options
  - Complexity of source filesystem hierarchy
- We don’t want to compile with a single command (usually)
  - Use object files to speed up recompilation
  - What if different files need different options
- Want to make life easy if something changes
  - Only things that depend on changed file/s should be rebuilt
  - Remembering every step to compile is
    - tedius
    - error prone

Enter the Makefile

Makefile

```
1  CC      = gcc
2  CFLAGS  = -std=c11 -DUSE_ZLIB
3  LDFLAGS = -lz
4
5
6  TARGETS=small
7  OBJECTS=main.o printv.o
8
9  all: $(TARGETS)
10
11 .PHONY: clean
12 clean:
13         $(RM) $(TARGETS) $(OBJECTS)
14
15 small: $(OBJECTS)
16         $(CC) $(CFLAGS) -o $@ $^ $(LDFLAGS)
17
18 .SUFFIXES: .c .o
19
20 %.o: %.c small.h
21         $(CC) $(CFLAGS) -c -o $@ $<
```

Build

```
make
```

```
gcc -std=c11 -DUSE_ZLIB -c -o main.o main.c
gcc -std=c11 -DUSE_ZLIB -c -o printv.o printv.c
gcc -std=c11 -DUSE_ZLIB -o small main.o printv.o -lz
```

Re-Build

```
touch main.c
make
```

```
gcc -std=c11 -DUSE_ZLIB -c -o main.o main.c
gcc -std=c11 -DUSE_ZLIB -o small main.o printv.o -lz
```

In The Old Days (And Sadly Sometimes Today)

- A software package would come with only a `Makefile`
- User would need to edit variable definitions at the top
- User would need to know
  - The flags their compiler needs
  - Know which libraries and headers are on their system
  - Where each library, header, etc. is found on their system
  - Each define needed to indicate their specific platform
- Try building, re-edit, re-build, . . . until success (if ever)
- Hand crafted `Makefile` vary in quality
- And good luck in most hand crafted ones with
  - cross-compilation
  - out-of-tree builds

In The Less Olden Days

- Packages came with a script/program that generated the `Makefile` from a template based on the system and user supplied options
- A.K.A. the `./configure` script
- Absolute pain to write by hand
  - could be thousands of lines of portable shell (no bash-isms)
  - required arcane knowledge to be truly portable across unix-likes
  - brittle

Enter the Modern Build Systems

- Provide a high-level DSL to software developers for
  - Flags to enable/disable features
  - Check for dependencies
  - Configure/generate/template source code
  - What to build
  - How to build
  - Where to install
- Build system handles the hard parts
  - Consistent API for users
  - Use proper low level tools for the platform
  - Handle compiler, platform, hardware differences
  - How to look for dependencies on each platform

Some Common Build Systems

Build System	Platforms	Low-Level
Autotools	unix-like	`configure` script and `make`
Meson	unix-like, Windows	itself and `ninja`
Bazel	unix-like, Windows	itself
CMake	unix-like, Windows	itself and `make/ninja/..`

Why Choose CMake?

Strengths

- Very backwards compatible with its own DSL
- Turing complete for when it is needed
- Just need to know its DSL
- Very good at C++ including with Qt
- Supports unix-like (including Linux) and Windows

CMake Weaknesses

- DSL is weak for programming logic (not as bad as shell)
- Not purely declarative
- Can’t extend built in functionality to more languages
  - Must define custom targets and commands the harder way

Basic CMake

CMakeLists.txt

```
# Setup
cmake_minimum_required(VERSION 3.23)
project(
  small
  VERSION 1.0
  LANGUAGES C
)

# Option for user to decide what to build with
option(USE_ZLIB "Build with zlib support" OFF)
if(USE_ZLIB)
  find_package(ZLIB REQUIRED)
endif()

# Program to build.
add_executable(small main.c printv.c small.h)
target_compile_features(small PRIVATE c_std_11)

# Add zlib support if it was found
if(ZLIB_FOUND)
  target_link_libraries(small PRIVATE ZLIB::ZLIB)
  target_compile_definitions(small PRIVATE USE_ZLIB)
endif()

# Install the program
install(TARGETS small)
```

Build and Install

```
rm -rf build
mkdir build
cd build
cmake -DCMAKE_INSTALL_PREFIX=$(pwd)/inst_pref -DUSE_ZLIB=ON ../
cmake --build .
cmake --install .
```

```
    -- The C compiler identification is GNU 13.3.1
    -- Detecting C compiler ABI info
    -- Detecting C compiler ABI info - done
    -- Check for working C compiler: /usr/bin/cc - skipped
    -- Detecting C compile features
    -- Detecting C compile features - done
    -- Found ZLIB: /usr/lib64/libz.so (found version "1.2.13")
    -- Configuring done (0.2s)
    -- Generating done (0.0s)
    -- Build files have been written to: /home/fnordsi1/projects/hpc_coffee/2024_06_26_cmake/code/smallopt/build
    [ 33%] Building C object CMakeFiles/small.dir/main.c.o
    [ 66%] Building C object CMakeFiles/small.dir/printv.c.o
    [100%] Linking C executable small
    [100%] Built target small
    -- Install configuration: ""
    -- Installing: /home/fnordsi1/projects/hpc_coffee/2024_06_26_cmake/code/smallopt/build/inst_pref/bin/small
```

Run

```
./small
inst_pref/bin/small
```

```
    Zlib version: 1.2.13
    Zlib version: 1.2.13
```

Blow by Blow – Required CMake

```
cmake_minimum_required(VERSION 3.23)
```

- Sets the minimum required CMake version
  - `3.23` is when installing header files got easier
  - Definitely don't use anything before `3.0` for anything new
- Also sets language compatibility options in newer CMake
  - This is why it is so backwards compatible
  - The individual options (called "policies") can be set individually

Blow by Blow – Project

```
project(
  small
  VERSION 1.0
  LANGUAGES C
)
```

- Must always define the project
- Set version
- Indicate which programming language/s if any are used
  - Will look for the compilers
  - Can actually set later with `enable_language(<lang>)`

Blow by Blow – User Controlled Option

```
# Option for user to decide what to build with
option(USE_ZLIB "Build with zlib support" OFF)
```

- For flags that users can set when building
  - e.g. use OpenMP
- Arguments are
  - Option name
  - Help string for users
  - Default value
- Then the variable `USE_ZLIB` contains the set value
- User adjusts by `cmake -DUSE_ZLIB=<value>`

Blow by Blow – Conditional Logic and Looking for a Dependency

```
if(USE_ZLIB)
  find_package(ZLIB REQUIRED)
endif()
```

- Can branch on the value of a variable (here an option)
- `find_package` finds any package with a module for finding it
  - Many builtin modules
  - Not hard to write simple ones
  - Can make CMake build one for your package on installation so others don't have to write one
- Add the `REQUIRED` flag to indicate it must be present
- All should set `<package>_FOUND`

Blow by Blow – Define Program

```
# Program to build.
add_executable(small main.c printv.c small.h)
```

- Defines a program to build followed by its source files
- Defines a Target with the same name as the executable
  - Later commands operate on the target
- add_library does the same for libraries (more later)

Blow by Blow – Set The Language Standard

```
target_compile_features(small PRIVATE c_std_11)
```

- Specify the language standard to use for this target
- C11 in this case
- Works like many other `target_*` commands:
  - The `PRIVATE` flag means the value to use for building
  - A `PUBLIC` flag is the value to use for things depending on it
    - For headers and linking
    - Relevant for libraries, not programs
  - Can have different `PRIVATE` and `PUBLIC` values

Blow by Blow – Add Dependency

```
# Add zlib support if it was found
if(ZLIB_FOUND)
  target_link_libraries(small PRIVATE ZLIB::ZLIB)
  target_compile_definitions(small PRIVATE USE_ZLIB)
endif()
```

- `find_package` actually defines a Target for the found package
- Adding a target is a matter of specifying its name (often `NAME::NAME`)
- Easy to add the needed preprocessor definition

Blow by Blow – What to Install

```
# Install the program
install(TARGETS small)
```

- Takes form `install(<kind> <what>... [OPTIONS])`
- For targets, automatically handles programs and libraries
  - Need to use other options for public headers, etc.
- `install(FILES <files> DESTINATION <dir> [OPTIONS])`

Variables

- All variables in CMake are strings
- Other data types are just cleverly encoded strings
- String values specified similar to Bash
  - F00 is the string "F00" as long as it has no spaces
  - "F00" is the string "F00"
  - "${F00}" expands the value of variable F00
  - Escape with the \ character
- But there are some differences with Bash
  - Doesn't use single quotes ever
  - F00 BAR is the string "F00;BAR" (a list in CMake)
- Set a variable with `set(<name> <value>)`
  - Set the value in the parent function `set(<name> <value> PARENT_SCOPE)`
- Delete a variable with `unset(<name>)`

Booleans

True

- any non-zero number
- `ON`
- `YES`
- `Y`
- `TRUE`

False

- `0`
- `OFF`
- `NO`
- `N`
- `FALSE`
- `IGNORE`
- `NOTFOUND`
- anything ending in `-NOTFOUND`

Lists

- One of the most common "types" in CMake
- Encoded as a semicolon separated string
- Outside of double quotes, a space separator implies a new element
- Examples:

Code	Result	Number of Elements
`set(MYVAR a)`	`"a"`	1
`set(MYVAR a b c)`	`"a;b;c"`	3
`set(MYVAR "a b" c)`	`"a b;c"`	2
`set(MYVAR "a\\;b" c)`	`"a\;b;c"`	2
`set(MYVAR "a\;b" c)`	`"a\;b;c"`	2

Basic Logic

```
if(<cond>)
  <commands>
elseif(<cond>)
  <commands>
else()
  <commands>
endif()
```

- else blocks are optional
- Many forms of conditions:
  - <boolean>
  - <varname> (evaluate variable contents as boolean)
  - NOT <cond>
  - <cond1> AND <cond2>
  - <cond1> OR <cond2>
  - <var_or_value1> LESS <var_or_value2>
  - <var_or_value1> GREATER_EQUAL <var_or_value2>
  - <var_or_value1> STREQUAL <var_or_value2>
  - <var_or_value1> VERSION_LESS <var_or_value2>

Basic Loops

```
foreach(<loop_var> <over>)
  <commands>
endforeach()
```

- Can loop over many different things
- Many forms of <over>
  - `RANGE <stop>` – integers from 0 to `<stop>` inclusive
  - `RANGE <start> <stop> [<step>]`
  - `IN <item1> ...` – over the explicitly passed list
  - `IN LISTS <var1> ...` – over the elements in the list variables

With A Library – New CMakeLists.txt

```
# Setup
cmake_minimum_required(VERSION 3.23)
project(
  small
  VERSION 1.0
  LANGUAGES C
)

# Option for user to decide what to build with
option(USE_ZLIB "Build with zlib support" OFF)
if(USE_ZLIB)
  find_package(ZLIB REQUIRED)
endif()

# Library with printv.
add_library(small_lib SHARED printv.c)
target_sources(small_lib PUBLIC FILE_SET HEADERS FILES small.h)
target_compile_features(small_lib PRIVATE c_std_11)
if(ZLIB_FOUND)
  target_link_libraries(small_lib PRIVATE ZLIB::ZLIB)
  target_compile_definitions(small_lib PRIVATE USE_ZLIB)
endif()

# Program to build.
add_executable(small main.c)
target_compile_features(small PRIVATE c_std_11)
target_link_libraries(small PRIVATE small_lib)

# Install the library and program
install(TARGETS small_lib small FILE_SET HEADERS)
```

With A Library – Building and Running

```
rm -rf build
mkdir build
cd build
cmake -DCMAKE_INSTALL_PREFIX=$(pwd)/inst_pref -DUSE_ZLIB=ON ../
cmake --build .
cmake --install .
```

```
-- The C compiler identification is GNU 13.3.1
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working C compiler: /usr/bin/cc - skipped
-- Detecting C compile features
-- Detecting C compile features - done
-- Found ZLIB: /usr/lib64/libz.so (found version "1.2.13")
-- Configuring done (0.2s)
-- Generating done (0.0s)
-- Build files have been written to: /home/fnordsi1/projects/hpc_coffee/2024_06_26_cmake/code/withlib/build
[ 25%] Building C object CMakeFiles/small_lib.dir/printv.c.o
[ 50%] Linking C shared library libsmall_lib.so
[ 50%] Built target small_lib
[ 75%] Building C object CMakeFiles/small.dir/main.c.o
[100%] Linking C executable small
[100%] Built target small
-- Install configuration: ""
-- Installing: /home/fnordsi1/projects/hpc_coffee/2024_06_26_cmake/code/withlib/build/inst_pref/lib/libsmall_lib.so
-- Installing: /home/fnordsi1/projects/hpc_coffee/2024_06_26_cmake/code/withlib/build/inst_pref/include/small.h
-- Installing: /home/fnordsi1/projects/hpc_coffee/2024_06_26_cmake/code/withlib/build/inst_pref/bin/small
-- Set runtime path of "/home/fnordsi1/projects/hpc_coffee/2024_06_26_cmake/code/withlib/build/inst_pref/bin/small" to ""
```

Blow by Blow – Library

```
add_library(small_lib SHARED printv.c)
target_sources(small_lib PUBLIC FILE_SET HEADERS FILES small.h)
target_compile_features(small_lib PRIVATE c_std_11)
```

- Libraries are SHARED or STATIC
  - advanced: there are others
- Must set public headers that will install in a weird way:

```
target_srouces(
  <target>
  PUBLIC
  FILE_SET HEADERS
  FILES <file1> ...
)
```

- Must add FILE_SET HEADERS to install:

```
install(TARGETS small_lib small FILE_SET HEADERS)
```

- As bad as setting headers it install was
  - Had to be done manually before CMake 3.23

Where to Go From Here

- Just scratched the surface
  - list and string manipulation with `list()` and `string()`
  - sub-directories
  - low-level finding
  - writing functions and macros
  - writing modules
  - `add_custom_target()` and `add_custom_command()`
  - templating files with `configure_file()`
  - other languages
  - testing (`ctest`) and packaging (`cpack`)
- See offical documentation: `https://cmake.org/documentation`
- See the last GöHPC Coffee on CMake: `https://pad.gwdg.de/iQkDoVwqT6qglbIMU6AxJQ`