lrz

Leibniz Supercomputing Centre

of the Bavarian Academy of Sciences and Humanities

Introduction to GNU Make and CMake

Martin Ohlerich

Plan for Today ...

Introduction

Short Survey on Background Experience and Preferences

GNU Make

- Basic Usage
- Makefiles

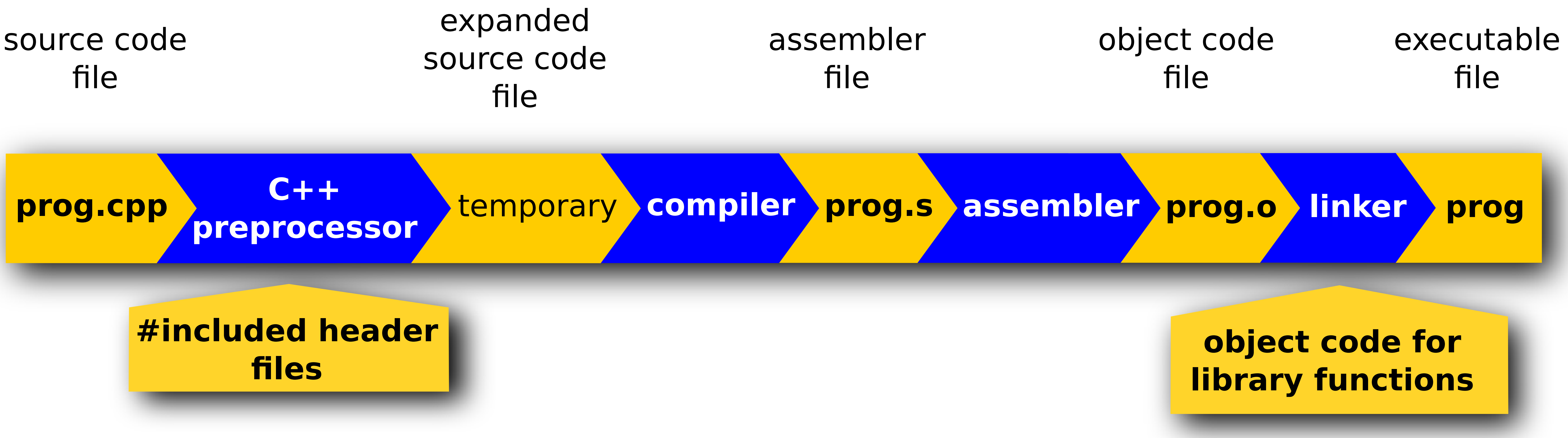
CMake

- Basic Usage
- CMakeLists.txt
- Advanced Features

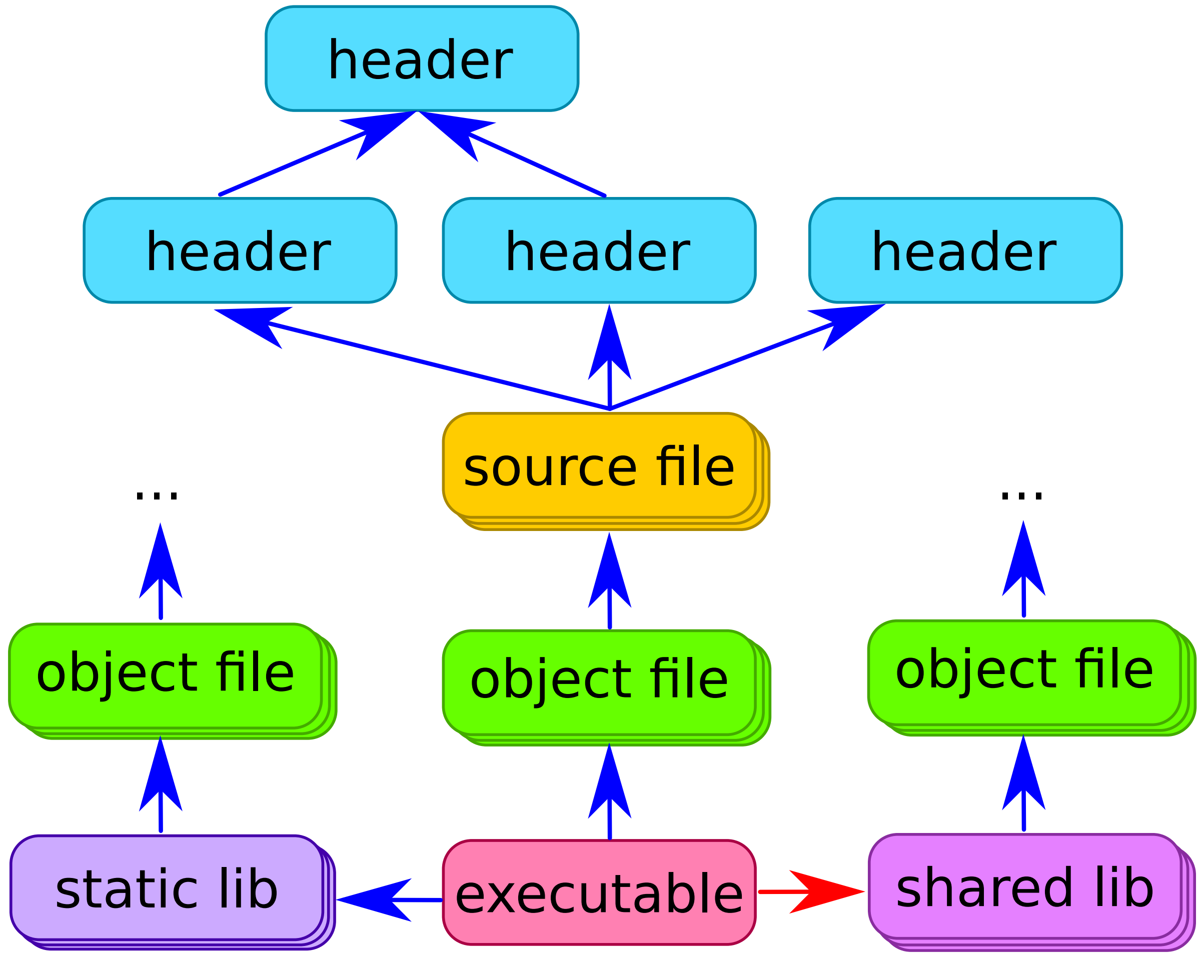
Survey

- Programming Experience? (For which OS? Which Programming Language?)
- Experiences with Build-Systems? (Make, CMake, Scons, IDE-inherent, self-written, ...)
- Knowledge about the Build-Process?

C/C++ Build Process



Build Dependency Tree



Build Commands (Linux/C)

Static Libraries

```
gcc -c -Wall a.c
gcc -c -Wall b.c
ar -cvq libmystaticlib.a a.o b.o
```

Shared Libraries

```
gcc -fPIC -c -Wall c.c
gcc -fPIC -c -Wall d.c
gcc shared -Wl,-soname,libmydynlib.so.1 -o libmydynlib.so.1.0.1 c.o d.o -lc
```

```
gcc -Wall -I include -o prog.exe -L. -lmystaticlib -lmydynlib proc.c           # usage
```

What we want is ...

- ... Simplicity!!!
- ... Flexibility!!!
- ... Robustness!!!
- ... Sytem-Independence!!!

GNU Make - Basic Usage

Example Programs

C program :

prog.c

```
#include <stdio.h>
int main() {
  prinf("Welcome to Parallel-Programming Course!\n");
}
```

C++ program :

prog.cpp

```
#include <iostream>
int main() {
  std::cout << "Welcome to Parallel-Programming Course!\n";
}
```

Fortran program :

prog.f

```
program hello
  print *,"Welcome to Parallel-Programming Course!"
end program hello
```

Shortest Makefile ever

```
make prog
```

Helpful Make Command-Line Options

```
make -h                              # help
make -v                              # version
make -p                              # pre-defined rules
make -j 4                            # parallel 4 threads
make -f MyMakefile                   # if deviating from default
make -n                              # dry-run
make -C path                         # goto path before make
```

Different Settings (Compiler/Flags/...)

```
make CC=gcc      CFLAGS="-std=c11 -O3 -g -Wall" prog         # C
make CXX=g++     CXXFLAGS="-std=c++17 -O3 -g -Wall" prog     # C++
make FC=gfortran FFLAGS="-std=f95 -ffree-form" prog          # Fortran
```

Makefiles

GNUmakefile, makefile, Makefile

```
CC=gcc
CFLAGS=-std=c11 -O3 -g -Wall
INCS=-I./include
prog: prog.c
	$(CC) $(CFLAGS) $(INCS) prog.c -o prog
```

Just enter **`make`**

Rules - Explanation

```
<target>: <dependencies>
<tab>     <shell command>
```

Tab character = mandatory (character can be changed via **`.RECIPEPREFIX`**)

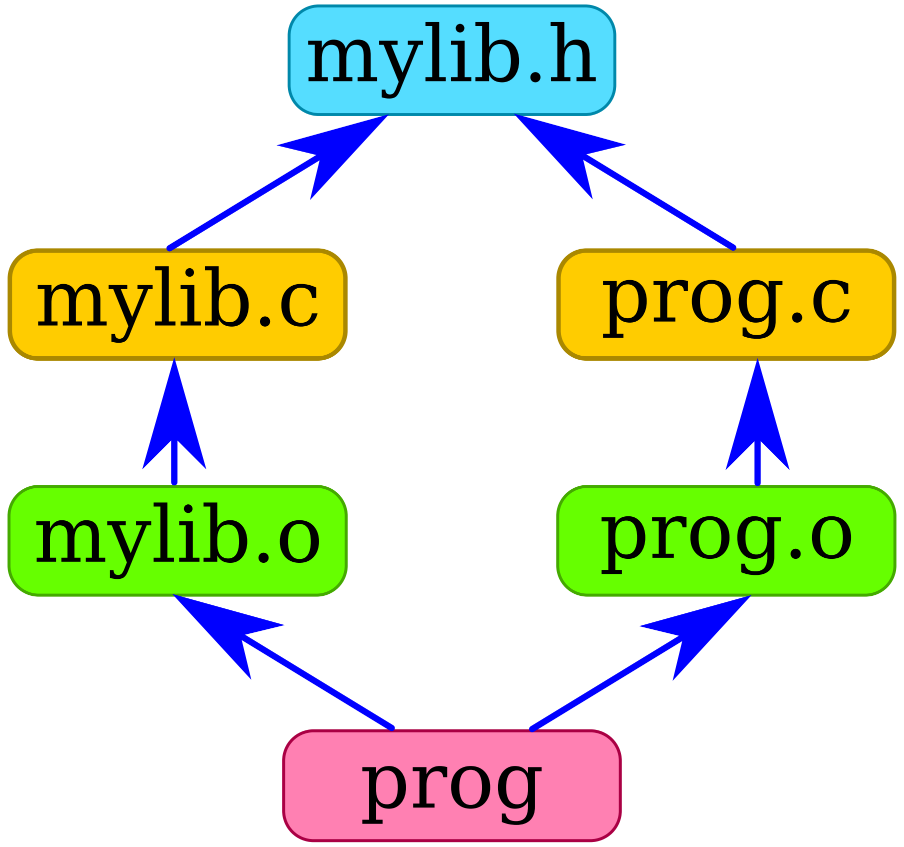
Splitting the Build Process

```
CC=gcc
CFLAGS=-std=c11 -O3 -g -Wall
INCS=-I./include
LIBS=-lm                                     # as harmless example
prog: prog.o
        $(CC) prog.o $(LDFLAGS) $(LIBS) -o prog
prog.o: prog.c
        $(CC) $(CFLAGS) $(INCS) -c prog.c -o prog.o
```

Some Simplification (DRY)

```
[...]
prog: prog.o
        $(CC) $^ $(LDFLAGS) $(LIBS) -o $@
prog.o: prog.c
        $(CC) $(CFLAGS) $(INCS) -c $< -o $@
```

More Complex Project



```
[...]
prog: prog.o mylib.o
        $(CC) $^ $(LDFLAGS) $(LIBS) -o $@
prog.o: prog.c mylib.h
        $(CC) $(CFLAGS) $(INCS) -c $< -o $@
mylib.o: mylib.c mylib.h
        $(CC) $(CFLAGS) $(INCS) -c $< -o $@
```

- 1 rule for each object that is generated out of some dependencies
- first target (*prog*) is default target
- can also call **`make <target>`**

Exercise

Write a small program with few (at least 3 including main) implementation (`.c`, `.cpp`, `.f`) and header files (`.h`, `.hpp`, or module files for fortran). You can also include some external (system) library if available to practice the inclusion of libraries.
Think of how the dependency tree must look like (that's possibly more difficult within Fortran when using modules)!
Write a *Makefile*, which compiles this program! Test the different flags and options of `make`! Also change the compiler and linker flags when calling `make` in order to observe the effect!

10 minutes

Convenience Feature: .PHONY targets

```
CC=gcc
CFLAGS=-std=c11 -O3 -g -Wall
INCS=-I./include
LIBS=-lm
.PHONY: all clean
all: prog
prog : prog.o
        $(CC) $^ $(LDFLAGS) $(LIBS) -o $@
prog.o : prog.c
        $(CC) $(CFLAGS) $(INCS) -c $< -o $@
clean :
        rm -rf prog.o prog *~
```

- *all* is the default target
- *all* could be used to default build several independent libraries and executables
- *.PHONY targets* can be used for built-up of a secondary (internal) dependency logic

Convenience Feature: Implicit (Generic) Rules

```
CC=gcc
CFLAGS=-std=c11 -O3 -g -Wall
INCS=-I./include
LIBS=-lm
.PHONY: clean
prog: prog.o mylib.o
        $(CC) $^ $(LDFLAGS) $(LIBS) -o $@
%.o: %.c
        $(CC) $(CFLAGS) $(INCS) -c $< -o $@
clean:
        rm -rf *.o prog *~
```

- Helpful for non-standard source file endings (e.g. `.cxx` in C++)
- Header dependency tree is more difficult to realize;
  simplest solution: if headers change → `make clean && make`

Convenience Feature: Functions and @ Operator

```
[...]
SRC = $(wildcard *.c)
OBJS = $(SRC:.c=.o)                       # same as OBJS = $(patsubst %.c,%.o,$SRC)
.PHONY: clean
prog: $(OBJS)
        $(CC) $^ $(LDFLAGS) $(LIBS) -o $@
%.o: %.c
        $(CC) $(CFLAGS) $(INCS) -c $< -o $@
clean:
        @rm -rf $(OBJS) prog *~
        @echo "Everything clean"
```

- Next to `wildcard`, lot of more functions available (→ Docu)
- With @, shell command is not printed to screen
- That's so far most generic Makefile (w/o header dependency)

And a lot of more advanced Features ...

Conventions and Standard Targets ...

Exercise

Improve, i.e. shorten, your former attempt of a Makefile!

10 minutes

CMake - Basic Usage

1st Hands-On CMake Exercise (Warm-Up)

Go to https://cmake.org/download and download the latest sources!

```
tar xf cmake-3.16.3.tar.gz
mkdir build && cd build                     # out-of-source build
cmake ../cmake-3.16.3                       # Configuration
make -j 4                                   # Check how many cores you have!!
##make VERBOSE=1 -j 4                       # maybe helpful
##make install                              # !!! Careful !!!
make help                                   # can help
```

- `cmake` executable configures the build, and creates a `Makefile`
- build is done *out-of-source*!

5 minutes

Useful cmake Command-Line Options

```
cmake -h                   # Help!
cmake --version            # Version
cmake -G ...               # Show/Specify Generator
cmake -D ...               # Specify variables (Compilers, Flags, ...)
```

- Generators determine which build system will be supported (Default Linux: Unix Makefiles)

```
cmake -G "CodeBlocks - Unix Makefiles"   \
      -DCMAKE_CXX_COMPILER=$(which icpc) \
      -DCMAKE_INSTALL_PREFIX=/usr        \
      path-to-source
```

- *path-to-source* can be absolute/relative; must contain a *CMakeLists.txt* file

"What the Hell ...!", you may say

"How should I remember all these flags and variables!?"

You don't need to!

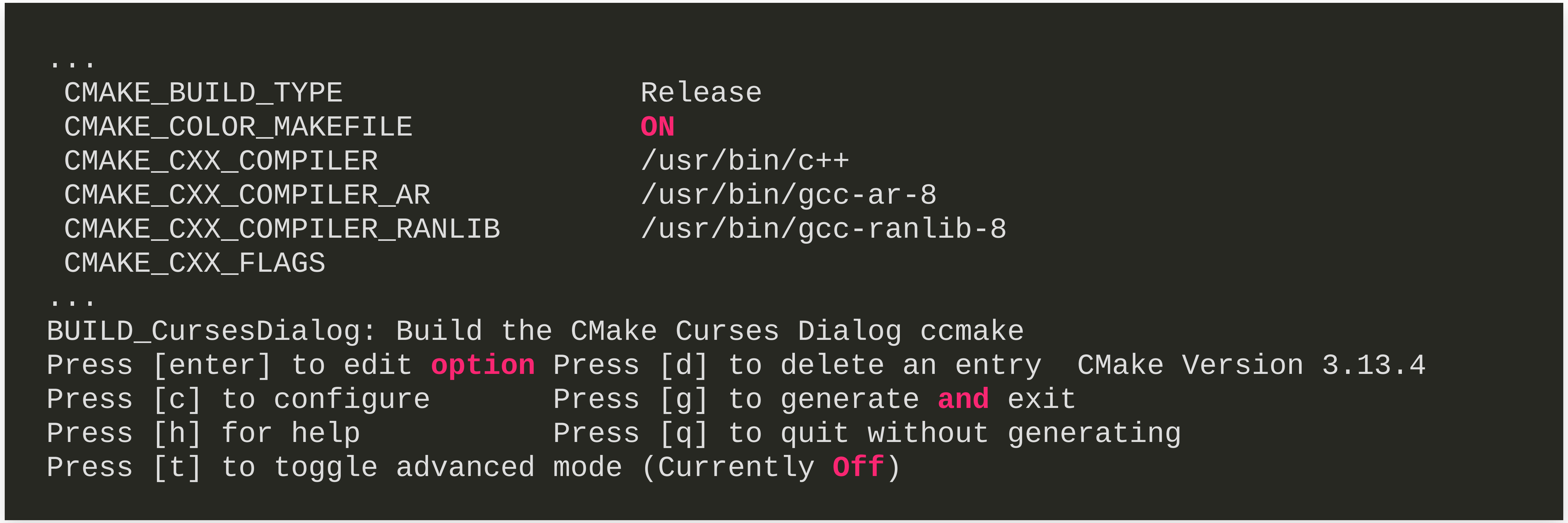
(see also CMake Docu: cmake-variables)

Convenience Tool: ccmake

```
ccmake path-to-source                    # or "ccmake ." if cmake already passed once
```

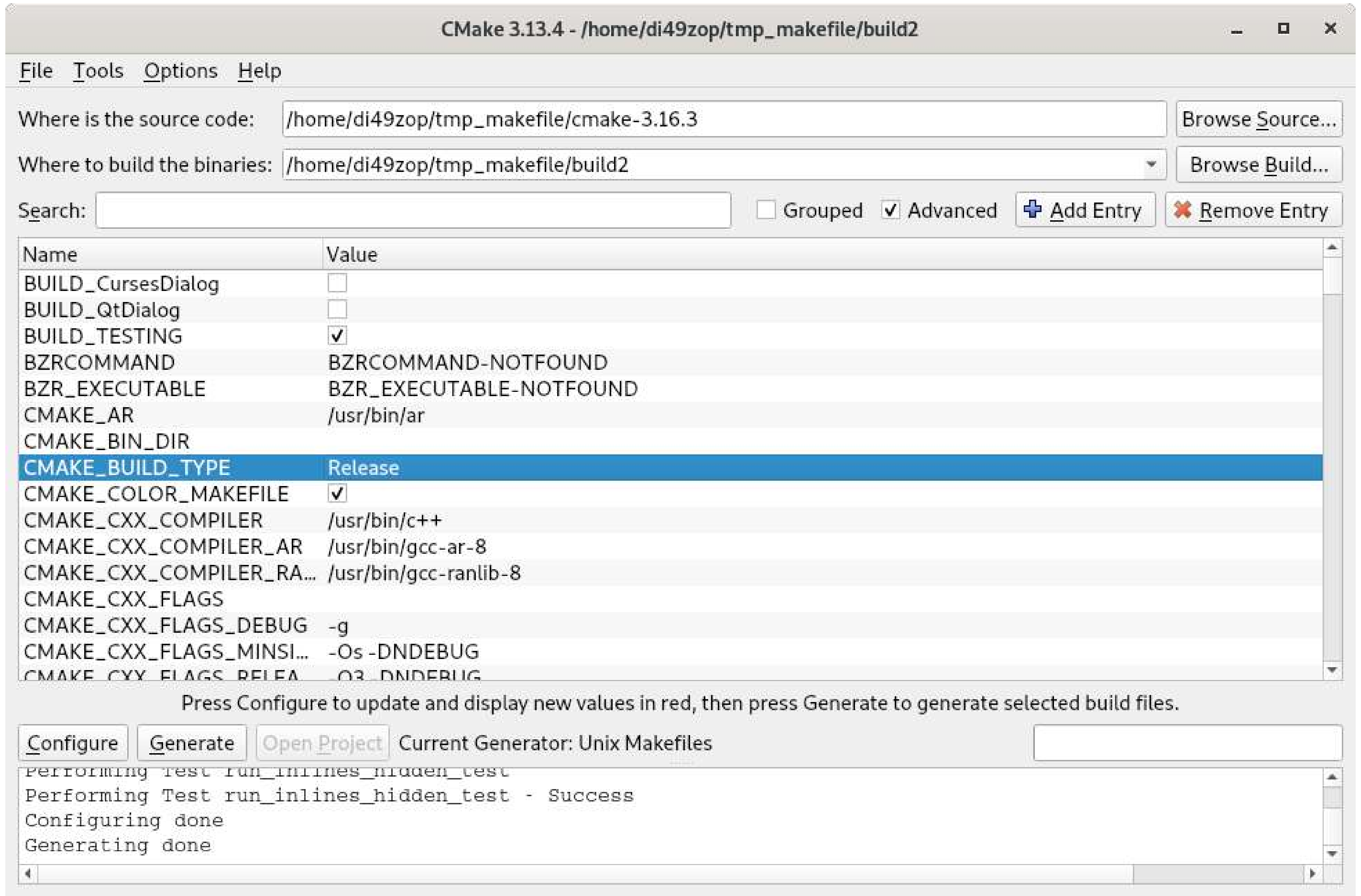
```
                                                   Page 0 of 1
EMPTY CACHE

[...]

EMPTY CACHE:
Press [enter] to edit option Press [d] to delete an entry  CMake Version 3.13.4
Press [c] to configure
Press [h] for help           Press [q] to quit without generating
Press [t] to toggle advanced mode (Currently Off)
```

Convenience Tool: ccmake (cont'd)



```
...
 CMAKE_BUILD_TYPE                 Release
 CMAKE_COLOR_MAKEFILE             ON
 CMAKE_CXX_COMPILER               /usr/bin/c++
 CMAKE_CXX_COMPILER_AR            /usr/bin/gcc-ar-8
 CMAKE_CXX_COMPILER_RANLIB        /usr/bin/gcc-ranlib-8
 CMAKE_CXX_FLAGS
...
BUILD_CursesDialog: Build the CMake Curses Dialog ccmake
Press [enter] to edit option Press [d] to delete an entry  CMake Version 3.13.4
Press [c] to configure       Press [g] to generate and exit
Press [h] for help           Press [q] to quit without generating
Press [t] to toggle advanced mode (Currently Off)
```

Convenience Tool: `cmake-gui`



Where is the source code: /home/di49zop/tmp_makefile/cmake-3.16.3

Where to build the binaries: /home/di49zop/tmp_makefile/build2

Search:

☐ Grouped ☑ Advanced

Name	Value
BUILD_CursesDialog	☐
BUILD_QtDialog	☐
BUILD_TESTING	☑
BZRCOMMAND	BZRCOMMAND-NOTFOUND
BZR_EXECUTABLE	BZR_EXECUTABLE-NOTFOUND
CMAKE_AR	/usr/bin/ar
CMAKE_BIN_DIR	
CMAKE_BUILD_TYPE	Release
CMAKE_COLOR_MAKEFILE	☑
CMAKE_CXX_COMPILER	/usr/bin/c++
CMAKE_CXX_COMPILER_AR	/usr/bin/gcc-ar-8
CMAKE_CXX_COMPILER_RA...	/usr/bin/gcc-ranlib-8
CMAKE_CXX_FLAGS	
CMAKE_CXX_FLAGS_DEBUG	-g
CMAKE_CXX_FLAGS_MINSI...	-Os -DNDEBUG
CMAKE_CXX_FLAGS_RELEA...	-O3 -DNDEBUG

Press Configure to update and display new values in red, then press Generate to generate selected build files.

Current Generator: Unix Makefiles

```
Performing Test run_inlines_hidden_test
Performing Test run_inlines_hidden_test - Success
Configuring done
Generating done
```

The more important Variables

```
CMAKE_INSTALL_PREFIX                # path to install after build via "make install"
CMAKE_BUILD_TYPE                    # none, Debug, Release, RelWithDebInfo, MinSizeRel, ...
CMAKE_<LANG>_COMPILER               # compiler (CC, CXX, FC)
CMAKE_<LANG>_FLAGS                  # compiler flags (CFLAGS, CXXFLAGS, FFLAGS)
BUILD_SHARED_LIBS                   # build shared libraries (.so, .dll) if ON
```

- `CMAKE_<LANG>_COMPILER` names the compiler; you can't change the language! <LANG> can be `C`, `CXX`, `Fortran`, `CUDA`, ...
  Can be used to e.g. set `mpicc` for MPI programs, or `scorep` for profiling/tracing instrumentation
- Developer can add project specific variables (→ CMakeLists.txt)

Now the Fun-Part: CMakeLists.txt

For those who want to go to a higher level
https://cmake.org/documentation → latest → CMake Tutorial

One File Project

Exercise

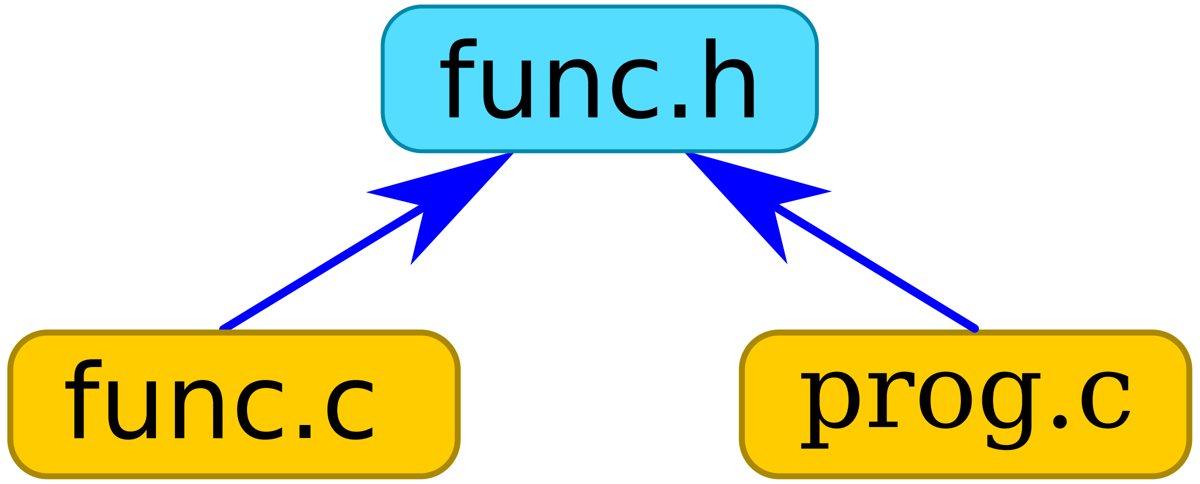
Pick your "Hello, World!" example from the *Makefile* part! Build one or all examples using cmake/make!

CMakeLists.txt

```
cmake_minimum_required (VERSION 3.5)
project (Tutorial C CXX Fortran)
add_executable (progcpp prog.cxx)
add_executable (progc prog.c)
add_executable (progf prog.f)
```

- Source file and CMakeLists.txt in the same directory
- You can select one language; C/C++ as default can be omitted
- Several targets (executables/libraries) possible

Standard Project - C/C++



- How to extend to many files should be immediately clear!
- Good Thing: Header dependency tree is automatic!

```
#ifndef FUNC_
#define FUNC_
void print_text(const char* text);
#endif
```

```
#include "func.h"
int main() {
  print_text("Hello");
}
```

```
#include "func.h"
#include <iostream>
void print_text(const char* text) {
  std::cout << text << '\n';
}
```

Standard Project - C/C++ (cont'd)

CMakeLists.txt for C++ (C same)

```
cmake_minimum_required (VERSION 3.5)
project (Tutorial)
include_directories("${PROJECT_SOURCE_DIR}")
add_executable (prog prog.cxx func.cxx)
```

in the project's top directory

Exercise

- Put the header files in a *include* subfolder, and the other sources into a *src* subfolder! Adapt the *CMakeLists.txt* accordingly, and build the executable!
- Add more header and source files!
- Or use example projects!

Standard Project - Fortran

```
program hello
use func
  call print_text("Hello!")
end program hello
```

```
module func
  implicit none
  contains
  subroutine print_text(a)
    CHARACTER (LEN=*), intent(in) :: a
    write(*,*) a
  end subroutine print_text
end module func
```

CMakeLists.txt for Fortran

```
cmake_minimum_required (VERSION 3.5)
project (Tutorial Fortran)
set(CMAKE_Fortran_MODULE_DIRECTORY
                  ${CMAKE_BINARY_DIR}/modules)
add_executable (prog prog.f func.f)
```

Exercise

- Put the source into a *src* folder, and change *CMakeLists.txt*!
- Add more source files!

External Libraries

Free Style (not portable)

```
[...]
include_directories("path-to-header-files")
add_executable (prog ...)
target_link_libraries(prog fftw3)
```

- `C_INCLUDE_PATH` and `CPLUS_INCLUDE_PATH` can be used
- finds `libfftw3.so`

Preferred Style

```
[...]
find_package(Boost 1.72.0 EXACT REQUIRED regex)
include_directories(${Boost_INCLUDE_DIRS})
add_executable(prog ... )
target_link_libraries(prog Boost::regex)
```

- add boost path to `CMAKE_PREFIX_PATH` (→ CMake Docu: find_package)
- for header-only modules, `regex` and `Boost::regex` can be omitted

Project Internal Libraries

Top Level CMakeLists.txt

```
include_directories (
            "${PROJECT_SOURCE_DIR}/mylib")
add_subdirectory (mylib)
set (EXTRA_LIBS ${EXTRA_LIBS} mylib)
add_executable (prog ...)
target_link_libraries (prog ${EXTRA_LIBS})
```

- add_subdirectory includes another source directory with a CMakeLists.txt file

Sub-Directory CMakeLists.txt

```
add_library(mylib mylib.cxx)
```

- mylib.cxx and mylib.h both in sub-directory mylib relative to project top-level folder
- Exercise: Realize a library solution with the standard project code from above! Test BUILD_SHARED_LIBS!

Advanced CMake Features/Tools

CMake Configure-Time Code-Modification

CMake Tutorial: Steps 1/2

```
cmake_minimum_required(VERSION 3.5)
project(Tutorial VERSION 1.0)
[...]
option(USE_MYLIB
       "Use MYLIB implementation" ON)
[...]
configure_file(
       TutorialConfig.h.in TutorialConfig.h)
[...]
```

- adds a project specific CMake variable USE_MYLIB, which can be ON or OFF

Source Code

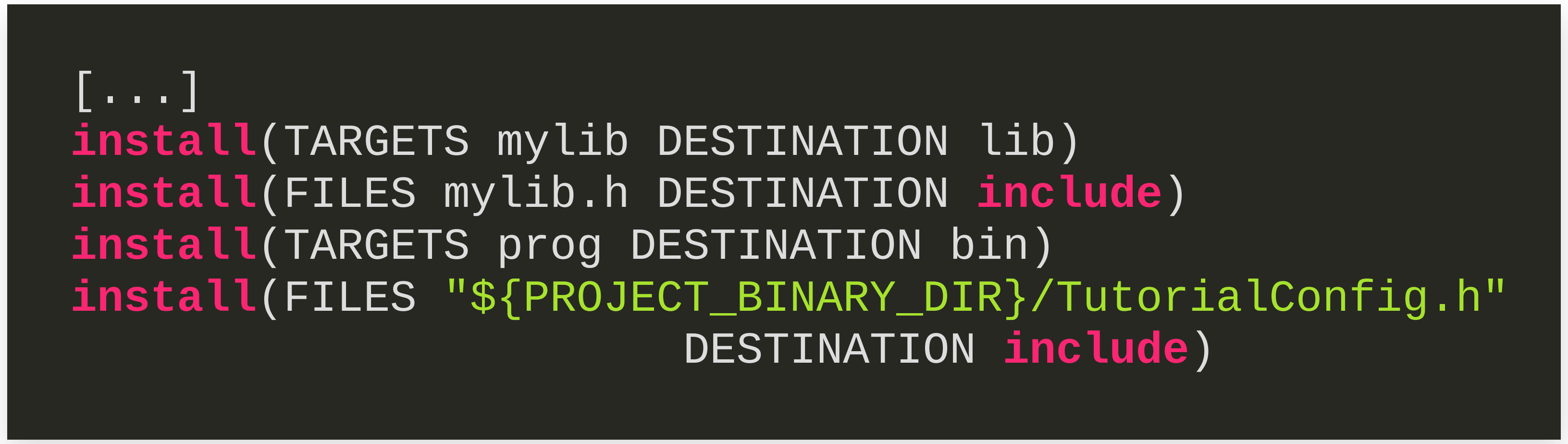
```
#ifdef USE_MYLIB
#include "MyLib.h"
#endif
[...]
#ifdef USE_MYLIB
  const double outputValue = mylib_func();
#endif
```

TutorialConfig.h.in

```
#define Tutorial_VERSION_MAJOR @Tutorial_VERSION_MAJOR@
#define Tutorial_VERSION_MINOR @Tutorial_VERSION_MINOR@
```

Install Targets

CMake Tutorial: Step 4



```
[...]
install(TARGETS mylib DESTINATION lib)
install(FILES mylib.h DESTINATION include)
install(TARGETS prog DESTINATION bin)
install(FILES "${PROJECT_BINARY_DIR}/TutorialConfig.h"
                       DESTINATION include)
```

- `DESTINATION` is relative to `CMAKE_INSTALL_PREFIX`
- gets relevant when executing `make install`
- `CMAKE_BUILD_WITH_INSTALL_RPATH=ON` can be used to set RPATH for dynamic library dependencies

Testing Support

CMake Tutorial: Step 4

```
[...]
enable_testing()
add_test(NAME Runs COMMAND Tutorial 25)
add_test(NAME Usage COMMAND Tutorial)
set_tests_properties(Usage
  PROPERTIES PASS_REGULAR_EXPRESSION "Usage:.*number")
[...]
```

after successful build:

```
make test        # or
ctest
```

- For unit, integration and general function tests

CPack - Creating Install Packages CMake Tutorial: Step 7

```
include(InstallRequiredSystemLibraries)
set(CPACK_RESOURCE_FILE_LICENSE "${CMAKE_CURRENT_SOURCE_DIR}/License.txt")
set(CPACK_PACKAGE_VERSION_MAJOR "${Tutorial_VERSION_MAJOR}")
set(CPACK_PACKAGE_VERSION_MINOR "${Tutorial_VERSION_MINOR}")
include(CPack)
```

- `install(TARGET ...)` must be set

```
cpack                            # Build in all Generators available in CPackConfig.cmake
cpack -G TGZ                     # Build in TGZ (Tarball); ZIP, RPM, DEB
cpack --config CPackSourceConfig.cmake # Source Tarball (out of source!!!)
```

- `CPackConfig.cmake` and `CPackSourceConfig.cmake` can/must be edited
- RPM and DEB require additional actions/tools (`rpmbuild`)

Setting Specific CMake Variables

For instance, requiring C++ 17 standard conform compiler:

```
set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED True)
```

CMake Scripting Language

Learn CMake Scripting Language in 15 Minutes

```cmake
set(world "World")
message("Hello, ${world}!")
file(GLOB sources "src/*.cxx")
message("Source Files: ${sources}")
```

- variables, lists
- `if-elseif-else` constructs
- `while`, `for_each` loops
- functions
- arithmetics is possible (`math`)

References

- GNU Make Documentation (PDF)
- CMake Documentation
- CMake Tutorial
- Mastering CMake (PDF)
- CMake Fortran Use
- CMake Fortran Support
- C++ Build Process
- Shared Libraries: Understanding Dynamic Loading (Linux)