Programming Techniques for Scientific Simulations I

CMake - a cross-platform build system generator

CMake: https://cmake.org

- Is a cross-platform build system generator
  - Unix/Linux
  - MacOS X
  - Windows
- Build instructions in a file called CMakeLists.txt get translated into the build system of your choice
  - Makefiles
  - Microsoft Visual Studio projects
  - Xcode projects
- We will just show a few examples. Read tutorials for more
  - https://cmake.org/examples/
  - https://cmake.org/cmake/help/latest/guide/tutorial/index.html

Compile an application

- Create a file called CMakeLists.txt

```
# require minimum version of CMake
cmake_minimum_required(VERSION 3.1)

# set name of project
project(Square)

# add executable to the project using the specified source files
add_executable(square main.cpp square.cpp)

# specify install rules
install(TARGETS square DESTINATION bin)
```

Running CMake to create makefiles

- Either use a CMake GUI or the command line tool
- Run cmake GUI in the build directory

```
$ cd build-directory
$ cmake source-directory
```

- Optionally specify where files should be installed

```
$ cmake –DCMAKE_INSTALL_PREFIX=install-path source-directory
```

- Note that absolute paths must be given.
- Use ccmake or the GUI to adjust build settings, compiler flags, ...

```
$ ccmake source-directory
```

Building after creating makefiles

- Now just build it like with makefiles:
  - Build all: `$ make`
  - Build a specific target: `$ make square`
  - Cleaning the build: `$ make clean`
  - Installing: `$ make install`
- Set the environment variable VERBOSE to see more details

```
$ make VERBOSE=1
```

Variables you might want to customize

- `CMAKE_BUILD_TYPE` can be set to
  - `Release`
  - `Debug`
  - …
- `CMAKE_*_FLAGS*` can be set to control compiling and linking
- `CMAKE_INSTALL_PREFIX` sets where files are installed
- Setting the compilers
  - `CMAKE_C_COMPILER, CMAKE_CXX_COMPILER`
  - `$ CC=gcc-6.6.6 CXX=g++-6.6.6 cmake`

Creating and using a static library

```
# add library using the specified source files
add_library(squareLib STATIC square.cpp)

# add include directories to our target library
# (PUBLIC because users of the library will need the
# include file square.hpp)
target_include_directories(squareLib PUBLIC ./)

# specify install rules (for our square library squareLib)
install(TARGETS squareLib
        ARCHIVE DESTINATION lib
        LIBRARY DESTINATION lib
        RUNTIME DESTINATION bin
       )

# specify install rules for the header
install(FILES square.hpp DESTINATION include)
```

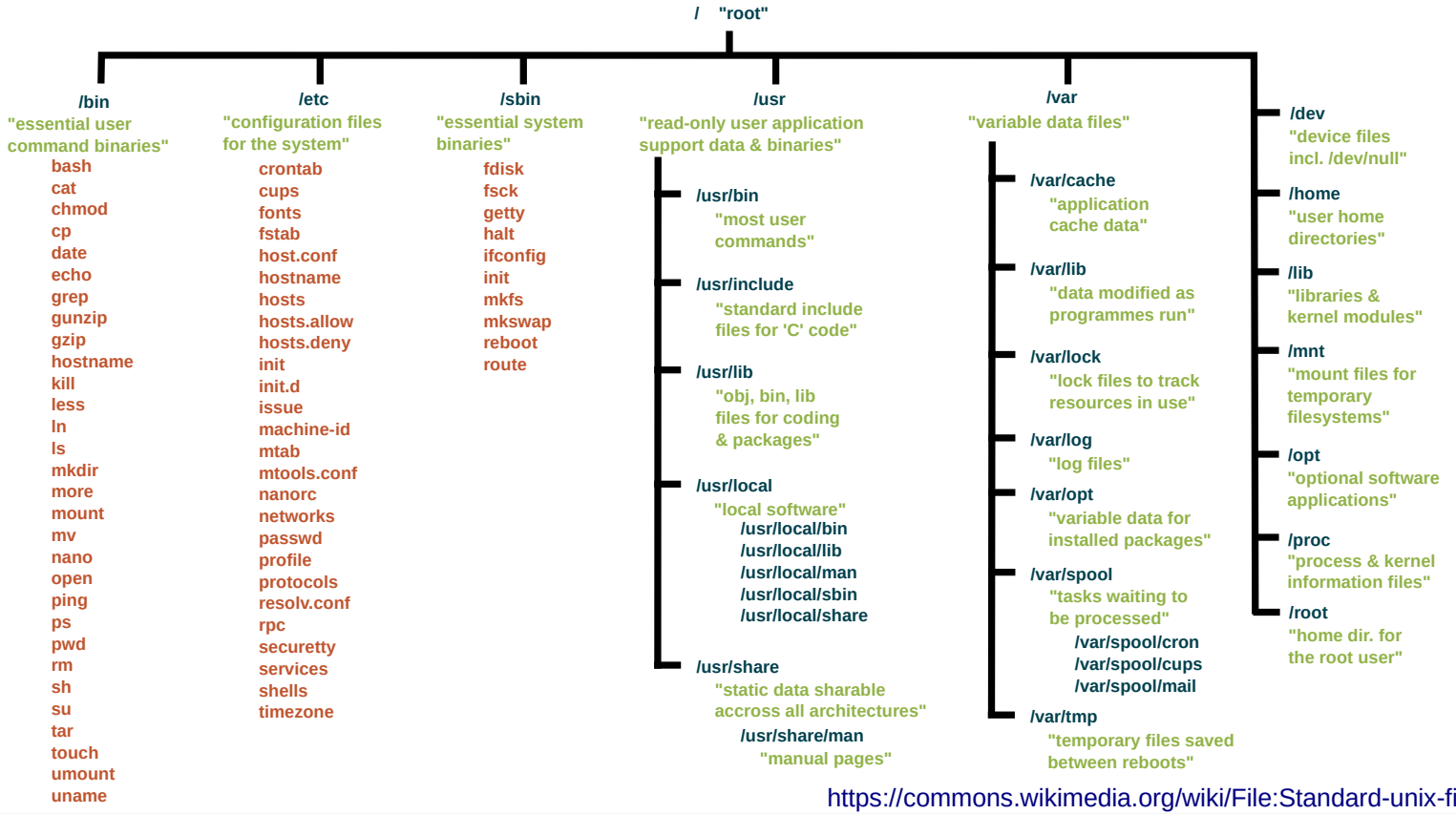
Creating and using a shared library

```
# add library using the specified source files
add_library(squareLib SHARED square.cpp)

# add include directories to our target library
# (PUBLIC because users of the library will need the
# include file square.hpp)
target_include_directories(squareLib PUBLIC ./)

# specify install rules (for our square library squareLib)
install(TARGETS squareLib
        ARCHIVE DESTINATION lib
        LIBRARY DESTINATION lib
        RUNTIME DESTINATION bin
       )

# specify install rules for the header
install(FILES square.hpp DESTINATION include)
```

Directory Structure

- The organization of files on a filesystem
- This is highly operating system dependent (and we will focus on Unix-like)
- See for more information, e.g.:
  - https://en.wikipedia.org/wiki/Directory_structure
  - https://en.wikipedia.org/wiki/Unix_filesystem

Directory Structure (Unix-like)



https://commons.wikimedia.org/wiki/File:Standard-unix-filesystem-hierarchy.svg

Linking against the library

```
# require minimum version of CMake
cmake_minimum_required(VERSION 3.1)

# set name of project
project(Square)

# add a subdirectory to the build (here lib contains the
# CMakeLists.txt with the square library)
add_subdirectory(lib)

# add executable to the project using the specified source files
add_executable(square main.cpp)

# specify that we need the square library (squareLib)
target_link_libraries(square squareLib)

# specify install rules
install(TARGETS square DESTINATION bin)
```

Choosing between static and dynamic

```
# CMakeLists.txt
# ...
# option to build STATIC or SHARED library
option(BUILD_SQUARE_SHARED "Build the square library shared." OFF)
if(BUILD_SQUARE_SHARED)
  set(SQUARE_LIBRARY_TYPE SHARED)
else()
  set(SQUARE_LIBRARY_TYPE STATIC)
endif()
# ...

# lib/CMakeLists.txt
# ...
# add library using the specified source files
add_library(squareLib ${SQUARE_LIBRARY_TYPE} square.cpp)
#...
```

Setting compiler flags & C++ standard

```
# ...
# setting special compiler options
if(CMAKE_CXX_COMPILER_ID MATCHES "(C|c?)lang")
  add_compile_options(--special-clang-option)
else()
  add_compile_options(--alternative-option)
endif()

endif()
# ...

# ...
# set a certain C++ standard level, require it, disable extensions, ...
set(CMAKE_CXX_STANDARD 11)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_CXX_EXTENSIONS OFF)
# ...
```

There is much more

- There are many more features:
  - Finding libraries (e.g. BLAS, LAPACK, MPI, HDF5, Boost, …)
  - Defining functions
  - Setting build options
  - Looping over variables
  - Creating installer packages
  - Creating and running unit tests
- Some of these we will encounter over the next weeks…
- For others, read the manuals and tutorials
  - `cmake –help, cmake --help-command-list, ...`
  - https://cmake.org/examples/
  - https://cmake.org/cmake/help/latest/guide/tutorial/index.html