CS4414 Recitation 9

Cmake, Perfermance (gprof)

10/22/2021

Alicia Yang

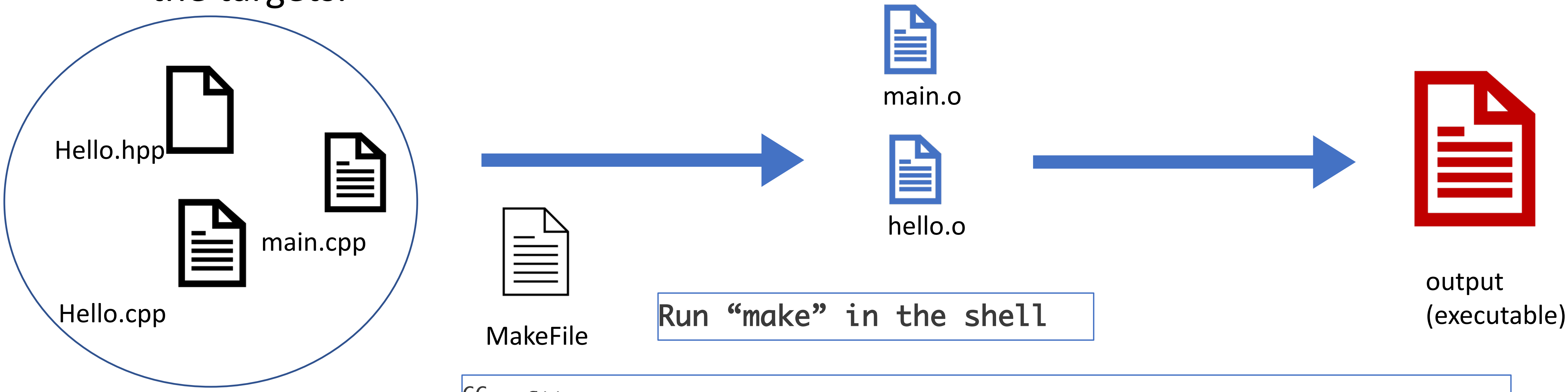
Cmake

- What is Cmake
- Simple Cmake
- Cmake with linked libraries
- Cmake with subdirectories

Code source:
https://github.com/aliciayuting/CS4414Demo.git

Build Files & Generate Executables --- MakeFile

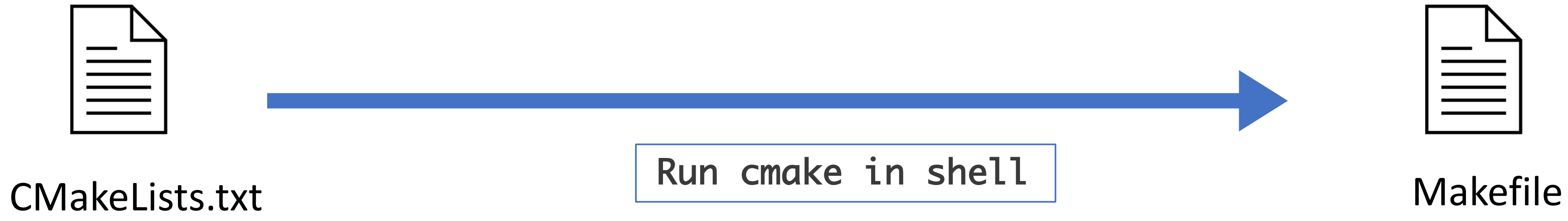
- Makefile is just a text file that is used or referenced by the ‘make’ command to build the targets.




```
CC = g++
CFLAGS = -g -Wall
TARGET = output
all: $(TARGET)
$(TARGET): main.o hello.o
          $(CC) $(CFLAGS) -o $(TARGET) main.o hello.o
main.o: main.cpp hello.hpp
          $(CC) $(CFLAGS) -c main.cpp
hello.o: hello.hpp hello.cpp
          $(CC) $(CFLAGS) -c hello.cpp
```

CMake

- Why CMake?
  - Makefiles are low-level, clunky creatures
  - CMake is a higher level language to automatically generate Makefiles
  - CMake contains more features, such as finding library, files, header files; it makes the linking process easier, and gives readable errors
- What is CMake?
  - CMake is an extensible, open-source system that manages the build process in an operating system and in a compiler-independent manner.
- CMakeLists.txt files in each source directory are used to generate Makefiles



Cmake

1.simple CMake

- Helloworld demo example

cmakelists.txt

```
cmake_minimum_required(VERSION 3.10) # set the project
name project(MyProject) # add the executable
add_executable(output main.cpp)
```

- Build and Run
  - Navigate to the source directory, and create a build directory

    $ cd ./myproject & $ mkdir build
  - Navigate to the build directory, and run Cmake to configure the project and generate a build system

    $ cd build &. $ cmake ..
  - Call build system to compile/link the project

    either run. $ make

    or run. $ cmake –build .

Cmake

2. Cmake with libraries

- Demo: main.cpp with hello library

- Declare a new library
  - Library name : say-hello
  - Source files: hello.hpp, hello.cpp
  - Can add library type: STATIC (default), SHARED

- Tell cmake to link the library to the executable(output)
  - Private link
  - Public link
  - interface

cmakelists.txt

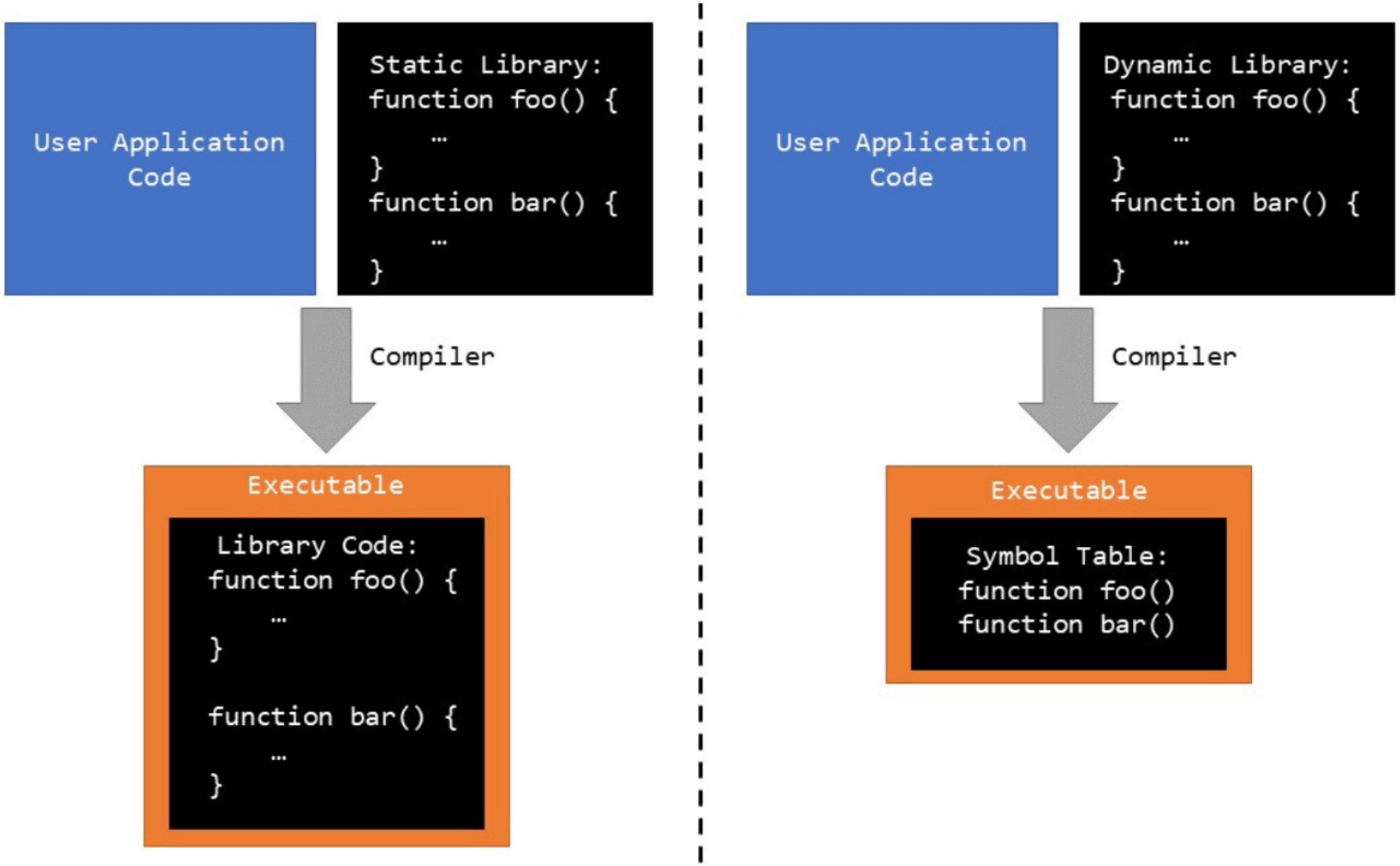
```
cmake_minimum_required(VERSION 3.12)
project(MyProject VERSION 1.0.0)

add_library{
        say-hello          [library type](optional)
        hello.hpp
        hello.cpp
}


add_executable(output main.cpp)

target_link_libraries(output PRIVATE say-hello)
```

Library Types in C++

- Static-linked library:
  - contains code that is linked to users' programs at compile time.
  - The executable produced is standalone and you don't access to the library file at runtime
  - Suppose building 100 executables, each one of them will contain the whole library code, which increases the code size overall
  - Longer to execute, because loading into the memory happens every time while executing.
- Shared library:
  - contains code designed to be shared by multiple programs. (.so in linux, or .dll in wondows, .dylib in OS X files)
  - The executable produced is not standalone and you need access to the library file at runtime
  - All the functions are in a certain place in memory space, and every program can access them, without having multiple copies of them.
  - Faster to execute, because shared library code is already in the memory; and don't need to be loaded if not required

Library Types in C++

--- compile time

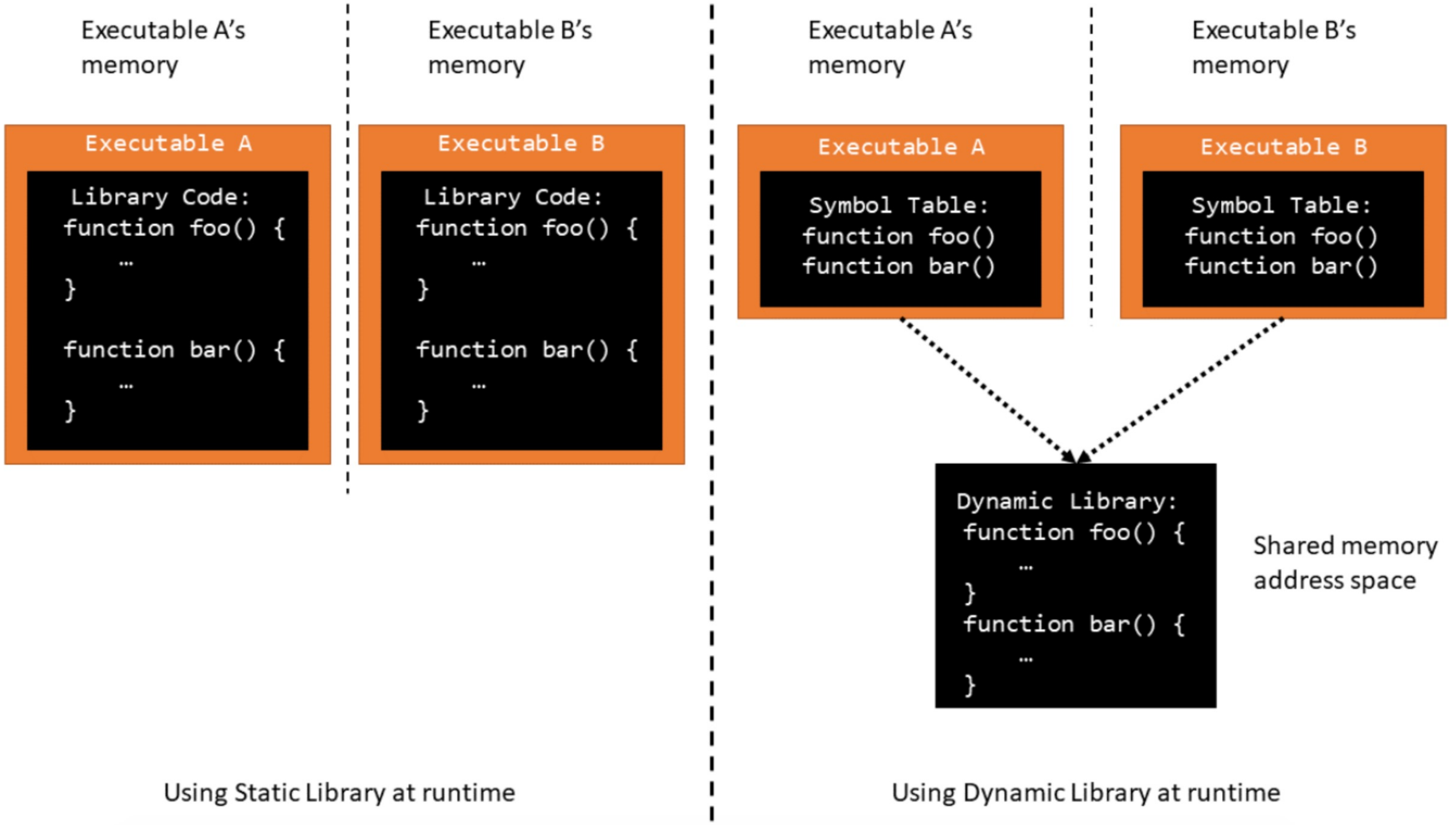


Using Static Library

Using Dynamic Library

Library Types in C++

--- run time



Cmake

2. Cmake with libraries

- Demo: main.cpp with hello library

- Declare a new library
  - Library name : say-hello
  - Source files: hello.hpp, hello.cpp
  - Can add library type: STATIC (default), SHARED

- Tell cmake to link the library to the executable(output)
  - Private link
  - Public link
  - interface

cmakelists.txt

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

add_library{
        say-hello        [library type](optional)
        hello.hpp
        hello.cpp
}


add_executable(output main.cpp)

target_link_libraries(output PRIVATE say-hello)
```

Cmake

--- Target_link_libraries

- target_link_libraries(<target>

  <PRIVATE|PUBLIC|INTERFACE> <lib> ...])
- <target> is the name of generated executable/library
- Each <lib> may be:
  - a library target name (The named target must be created by add_library() or as an IMPORTED library.)
  - a full path to a library file (e.g. /usr/lib/libfoo.so)
  - a plain library name (e.g. foo becomes -lfoo or foo.lib)

Cmake

--- Target_link_libraries/Target_include_directories

- target_link_libraries(<target>

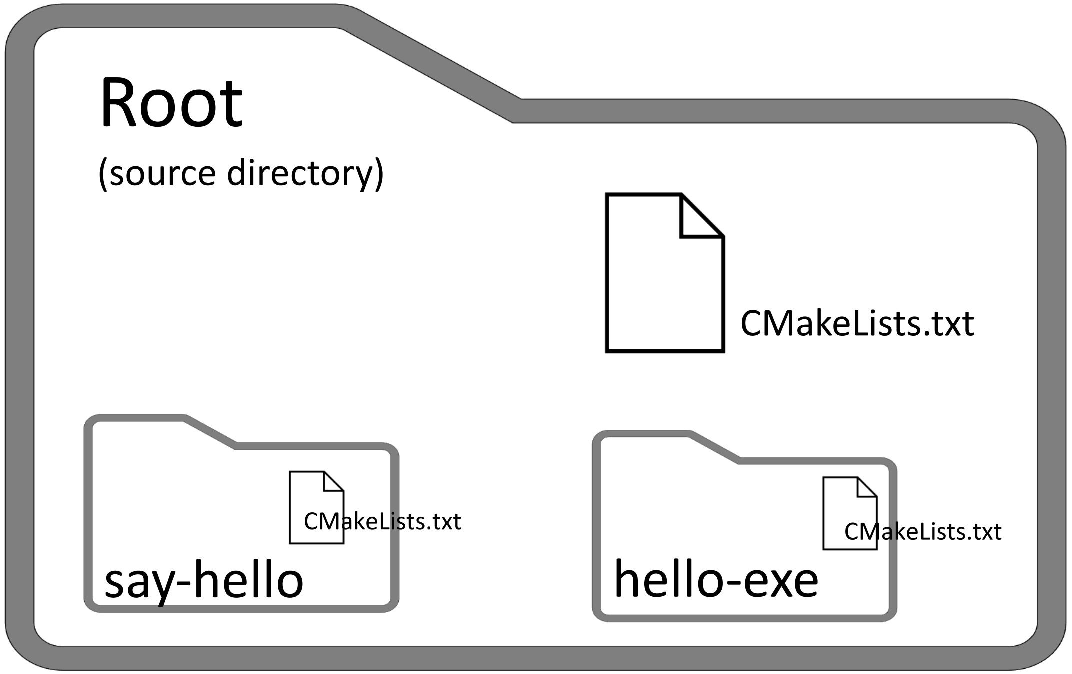
  <PRIVATE|PUBLIC|INTERFACE> <lib> ...])

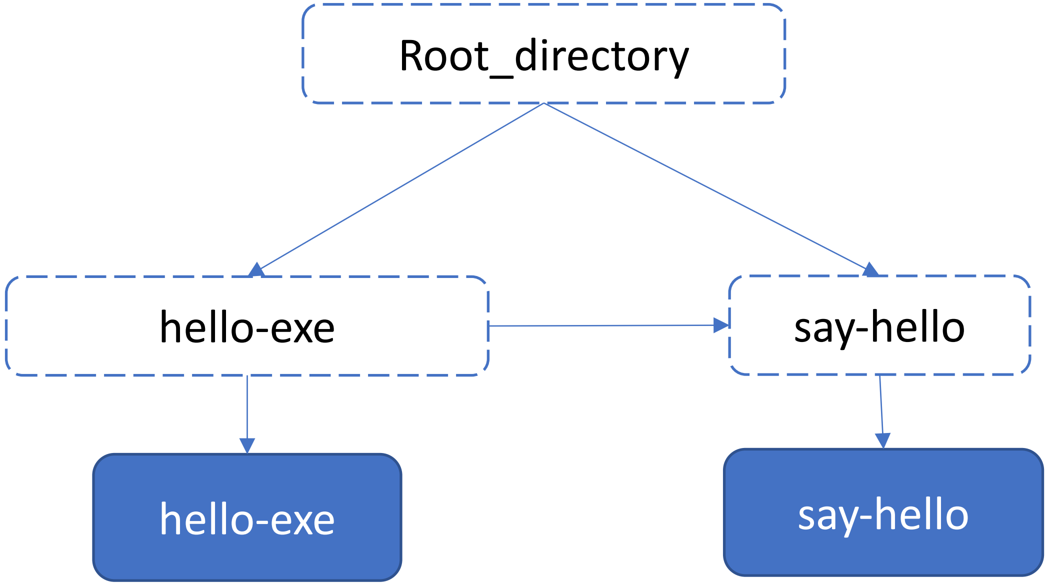
- The PUBLIC, PRIVATE and INTERFACE keywords can be used to specify both the link dependencies and the link interface in one command.
  - PUBLIC: Libraries and targets following PUBLIC are linked to, and are made part of the link interface.
  - PRIVATE: Libraries and targets following PRIVATE are linked to, but are not made part of the link interface.
  - INTERFACE: Libraries following INTERFACE are appended to the link interface and are not used for linking <target>.

Cmake

3. Cmake with subdirectory

- CMakeLists.txt files placed in each source directory are used to generate standard build files (e.g., makefiles on Unix and projects/workspaces in Windows MSVC).
- CMake supports in-place and out-of-place builds, and can therefore support multiple builds from a single source tree.





Demo

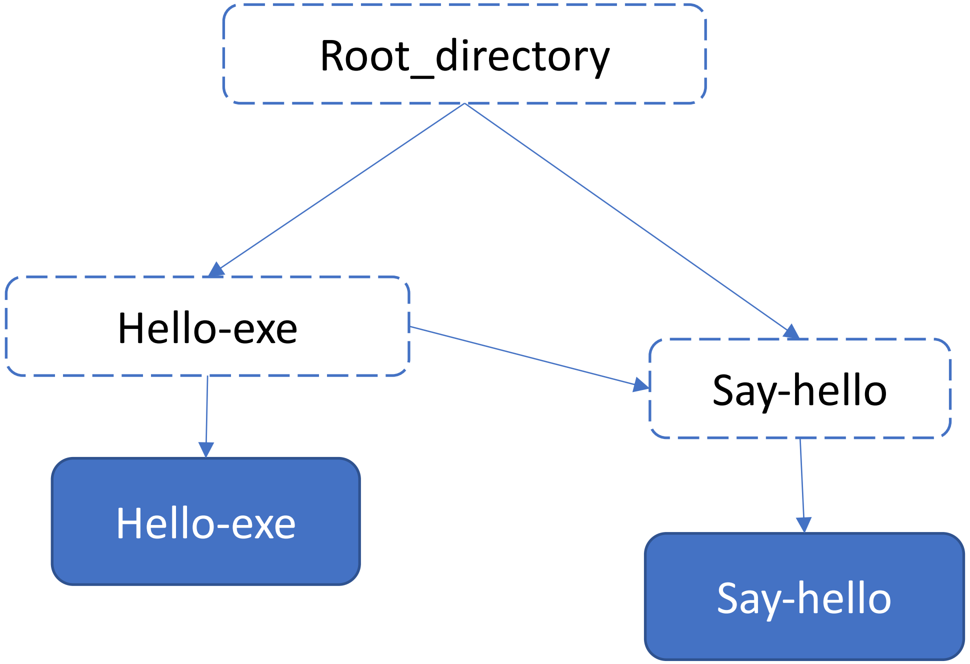
Demo

Cmake

3. Cmake with subdirectory

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

add_subdirectory(say-hello)
add_subdirectory(hello-exe)
```

```
add_executable(hell
o_exe main.cpp)

target_link_librari
es(hello_exe
PRIVATE say-hello)
```




```
add_library(
    say-hello
    hello.hpp
    hello.cpp
)

target_include_directories
(say-hello PUBLIC
"${CMAKE_CURRENT_SOURCE_DI
R}")

target_compile_definitions
(say-hello PUBLIC
SAY_HELLO_NUM=5)
```

Cmake

---add_subdirectory

- add_subdirectory(source_dir [binary_dir] [EXCLUDE_FROM_ALL])

- Adds a subdirectory to the build. The source_dir specifies the directory in which the source CMakeLists.txt and code files are located.

Cmake

- target_include_directories(<target> [SYSTEM] [AFTER|BEFORE]
  <INTERFACE|PUBLIC|PRIVATE> [items1...])

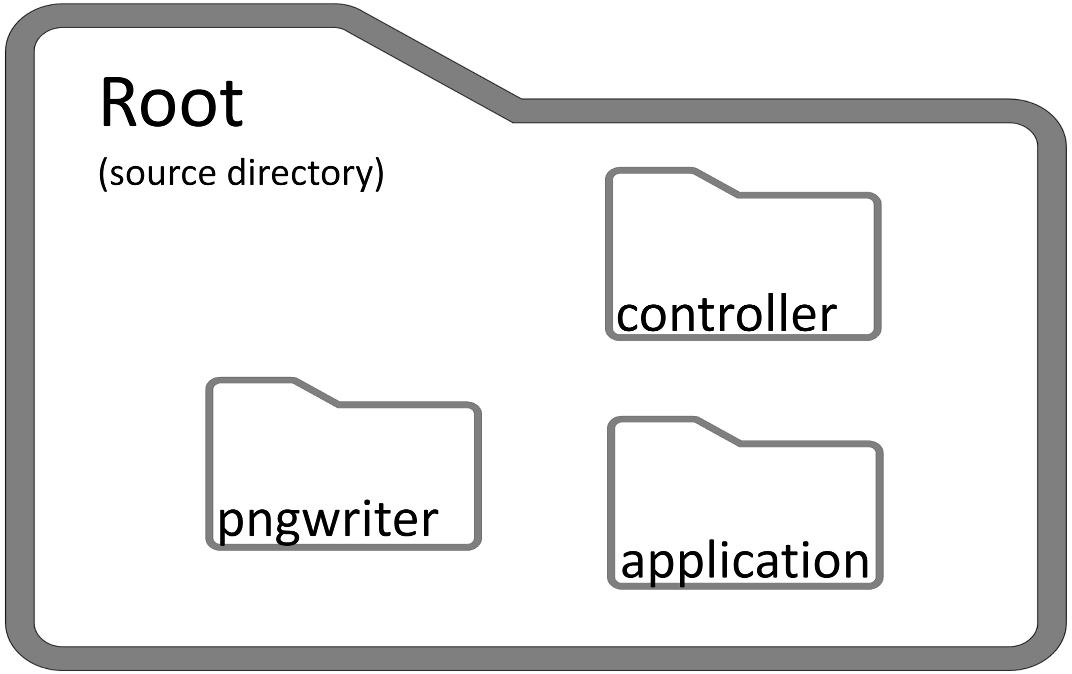
- Set include directory properly

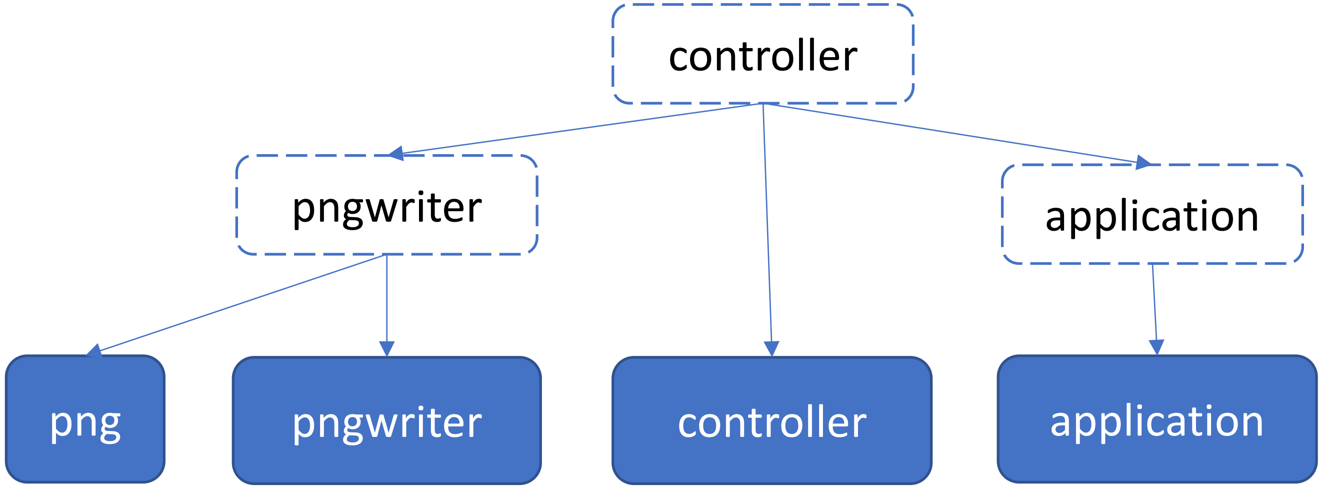
- The PUBLIC, PRIVATE and INTERFACE keywords can be used to specify both the link dependencies and the link interface in one command.
  - PUBLIC(default): All the directories following PUBLIC will be used for the current target and the other targets that have dependencies on the current target
  - PRIVATE: All the include directories following PRIVATE will be used for the current target only
  - INTERFACE: All the include directories following INTERFACE will NOT be used for the current target but will be accessible for the other targets that have dependencies on the current target

Cmake

3. Cmake with subdirectory

- Demo of Traffic Controller Simulator in Sagar’s HW1 solution





Performance Optimization

- 5 steps to improve runtime efficiency
- Time study
- How to use gprof
- Demo

Improve Execution Time Efficiency

1. Do timing studies
2. Identify hot spots
3. Use a better algorithm or data structure
4. Enable compiler speed optimization
5. Tune the code

Time the program

--- Unix 'time' command

- Run $ time ./output

  real 0m12.977s

  user 0m12.860s

  sys 0m0.010s

- Real: Wall-clock time between program invocation and termination
- User: CPU time spent executing the program
- System: CPU time spent within the OS on the program's behalf

Identify hot spots

- Gather statistics about your program’s execution
- Runtime profiler: gprof (GNU Performance Profiler)
- How does gprof work?
  - By randomly sampling the code as it runs, gprof check what line is running, and what function it’s in

Gprof

- Compile the code with flag –pg
  - g++ –pg helloworld.cpp –o output
- Run the program
  - $ ./output
  - Running the application produce a profiling result called gmon.out
- Create the report file
  - gprof output > myreport
- Read the report
  - vim myreport

Flat Profile

```
Each sample counts as 0.01 seconds.
  %      cumulative   self              self     total
 time      seconds   seconds   calls  us/call  us/call  name
 13.22       0.28     0.28  50045000     0.01     0.01   void std::__cxx11::basic_string<char, std::char_traits<char>, …
 10.39       0.50     0.22  100000000    0.00     0.00   std::vector<Entity, std::allocator<Entity> >::operator[](unsigned long)
 6.85        0.65     0.15  50005000     0.00     0.00   __gnu_cxx::__normal_iterator<Entity const*,std::vector<Entity,…
 5.67        0.77     0.12  100030000    0.00     0.00   __gnu_cxx::__normal_iterator<Entity const*, std::vector<Entity, …
 5.67        0.89     0.12  50045000     0.00     0.01   std::iterator_traits<char*>::difference_type std::distance<char*>(char*,…
 5.43        1.00     0.12  50005000     0.00     0.00   __gnu_cxx::__normal_iterator<Entity const*,std::vector<Entity, …
…
…
```

- name: name of the function
- %time: percentage of time spent executing this function
- cumulative seconds: [skipping, as this isn’t all that useful
- self seconds: time spent executing this function
- calls: number of times function was called (excluding recursive)
- self s/call: average time per execution (excluding descendents)
- total s/call: average time per execution (including descendents)

Improve Execution Time Efficiency

1. Do timing studies
2. Identify hot spots
3. Use a better algorithm or data structure
4. Enable compiler speed optimization. (compile flag with -O3)
5. Tune the code

Where to find the resources?

- CMake tutorials:
  - https://www.youtube.com/watch?v=LMP_sxOaz6g
  - https://subscription.packtpub.com/book/programming/9781786465184/1/ch01lvl1sec6/creating-type-aliases-and-alias-templates
  - https://cmake.org/cmake/help/latest/guide/tutorial/A%20Basic%20Starting%20Point.html#build-and-run
- Library Linking:
  - https://domiyanyue.medium.com/c-development-tutorial-4-static-and-dynamic-libraries-7b537656163e
  - https://leimao.github.io/blog/CMake-Public-Private-Interface/
- Code:
  - https://github.com/aliciayuting/CS4414Demo.git
- Gprof:
  - https://www.cs.princeton.edu/courses/archive/fall13/cos217/lectures/07Performance.pdf