

CS4414 Recitation 8

C++ Compilation & Performance(Gprof)

10/17/2025

Alicia Yang

How to write **good** system program in C++

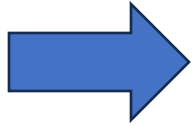
1. clean and correct code

2. Develop efficient system

Write clean and correct code

- The basics: C++ types, variable ... 
- Classes and functions 
- Memory management in C++, RAll principle 
- Smart pointers in C++ 
- C++ templates 
- Standard containers – `std::vector<T>`, `std::map<K,V>` 

Develop efficient system

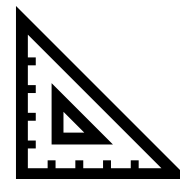


- Cmake for large system compilation management, gprof for program profiling
- Make efficient use of hardware
 - Hardware parallelism 
 - Multithreading and synchronization



Overview

- C++ compilation and linking
 - Linking review
 - Makefile and Cmake
- Performance optimization
 - Performance measurement
 - gprof



C++ Compilation

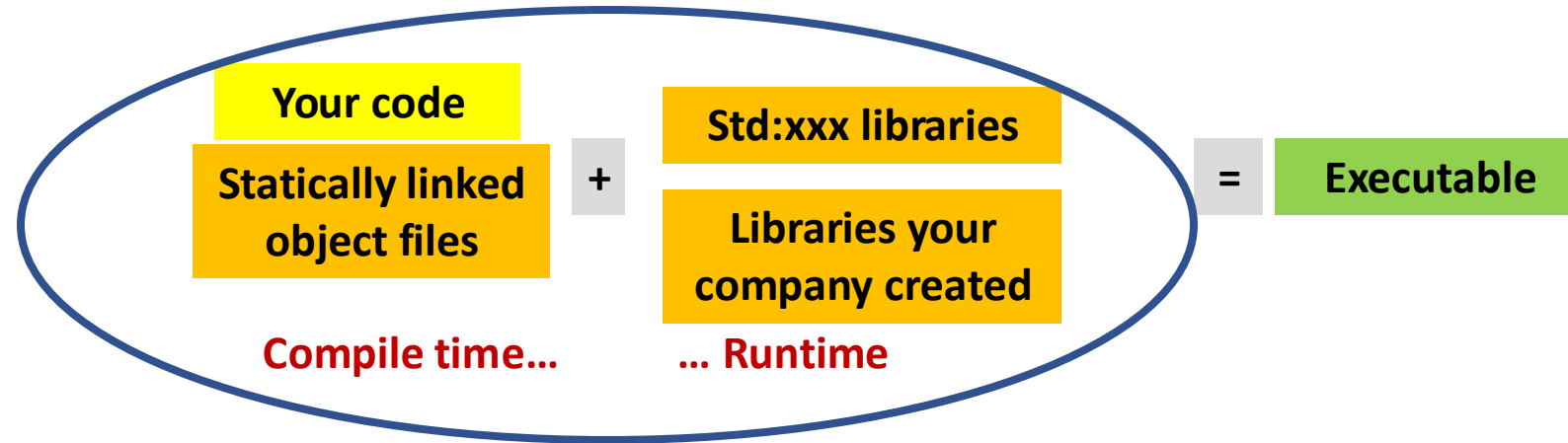
- Linking
- Statically linked library vs dynamically linked library
- Makefile & CMake

Linking

Linking is a technique that allows programs to be constructed from multiple object files.

- Compile time (when a program is compiled)
- Load time (when a program is loaded into memory)
- Run time (while a program is executing)

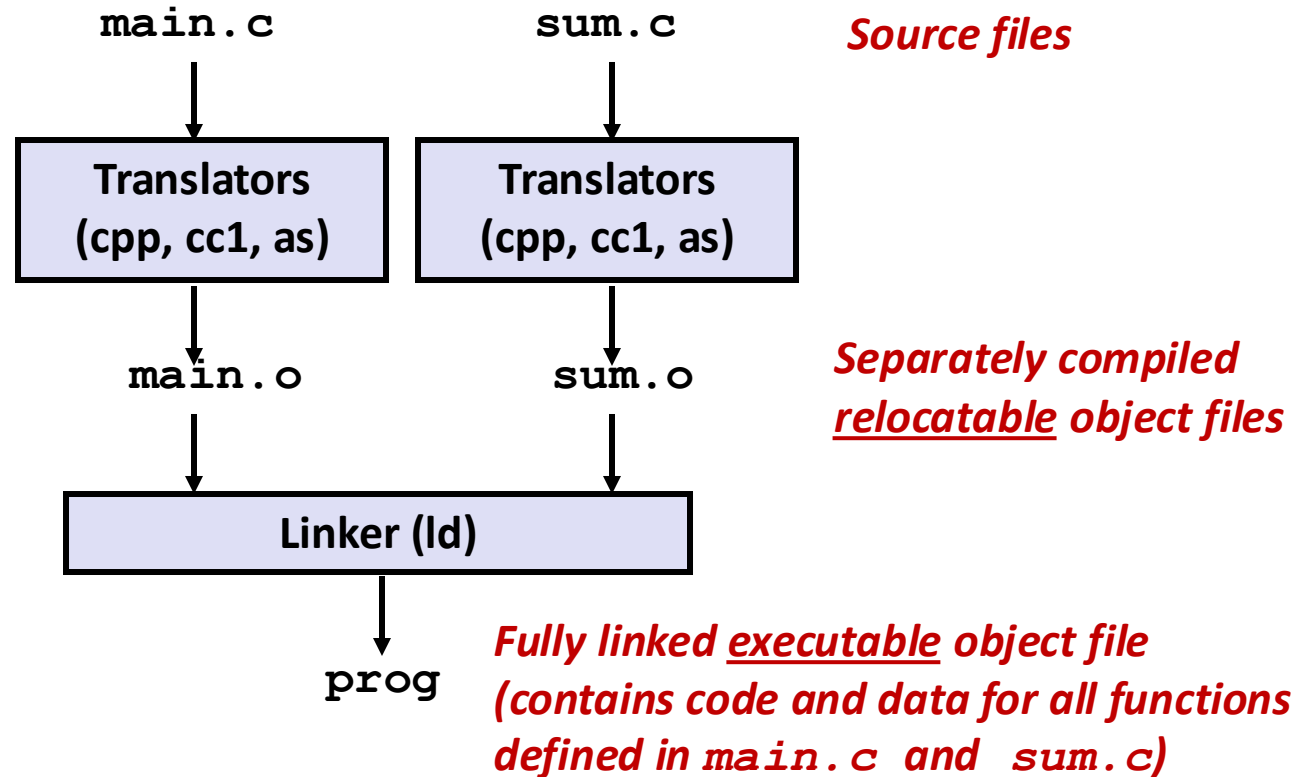
Linking



- A linker takes a collection of object files and combines them into an object file. But this object file will still depend on libraries.
- Next it cross-references this single object file against libraries, resolving any references to methods or constants in those libraries.
- If everything needed has been found, it outputs an executable image.

Linking

- Gcc is really a “*compiler driver*”: It launches a series of sub-programs
 - `linux> gcc -Og -o prog main.c sum.c`
 - `linux> ./prog`



C++ libraries

- Why use library?
 - The C++ libraries are **modular components of reusable code**. Using class libraries, you can integrate blocks of code that have been previously built and tested.
- What are in C++ library?
 - A C++ library consists of **header files** and **an object library**.
 - **The header files** provide class and other definitions that the library **exposes** (offers) to the programs using its.
 - **The object library(precompiled binary)** contains **compiled implementation** of functions and data that are linked with your program to produce an executable program.

Static-linked libraries

- contains code that is **linked to users' programs at compile time.**
(.a(archive) in linux, or .lib in windows)
- **Compiled and linked directly** into the program
- a copy of the library becomes part of every executable that uses it, this can cause a lot of wasted space. (Suppose building 100 executables, each one of them will contain the whole library code, which increases the code size overall)

Dynamic(shared) library

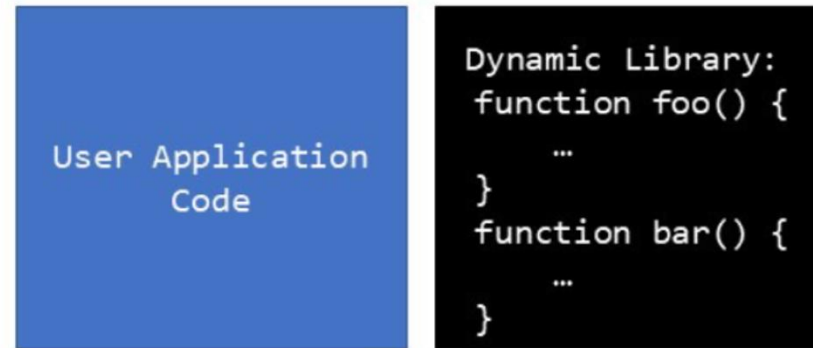
- contains code designed to be **shared by multiple programs**. (.so in linux, or .dll in windows, .dylib in OS X files)
- Loaded into your application **at run time**
- many programs can share one copy, which saves space. (All the functions are in a certain place in memory space, and every program can access them, without having multiple copies of them)

Library Types in C++

--- compile time



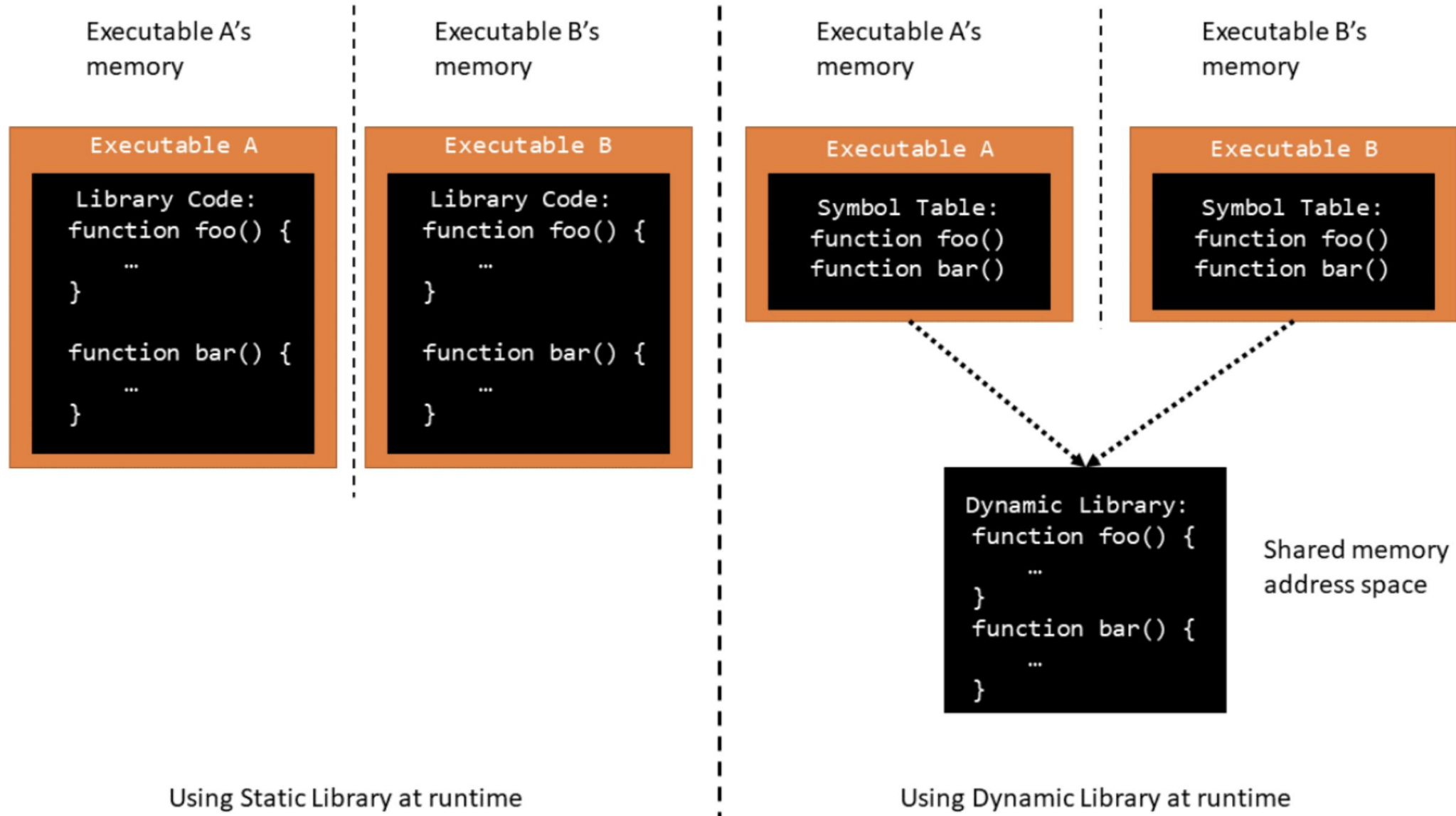
Using Static Library



Using Dynamic Library

Library Types in C++

--- run time



When to use **static linking** vs. dynamic linking



- Static linking disadvantages
 - **Duplication** in the **stored** executables
 - **Duplication** in the **running** executables
 - Minor bug fixes in system libraries? Must **rebuild** everything!

When to use **static linking** vs. dynamic linking



- Static linking advantages
 - Executable is **complete** and **self-contained**. **No runtime dependencies**
 - **Predictable** behavior
 - Requires **minimal** operating system
- When to use
 - Commonly used by embedded systems, like microcontroller, IoT devices, ...

When to use static linking vs. **dynamic linking**



- Dynamic linking advantages
 - **Runtime dependency:** at execution, the dynamic linker does need to be able to find the library file (a “.so” file) If a dynamically linked executable is launched on a machine that lacks the DLL, you will get an error message (usually, on startup, but there are some obscure cases where it happens later, when the DLL is needed)
 - **Compatibility issues:** version conflict

When to use static linking vs. **dynamic linking**



- Dynamic linking advantages
 - Reduced memory usage, smaller executable size: a single copy is shared
 - Easier for update and maintenance
 - Version flexibility. If the library updates, simply only need update the library itself (if the APIs remain the same)
- When to use it
 - Commonly used for open-sourced libraries (boost, opencv, grpc..)

Linking

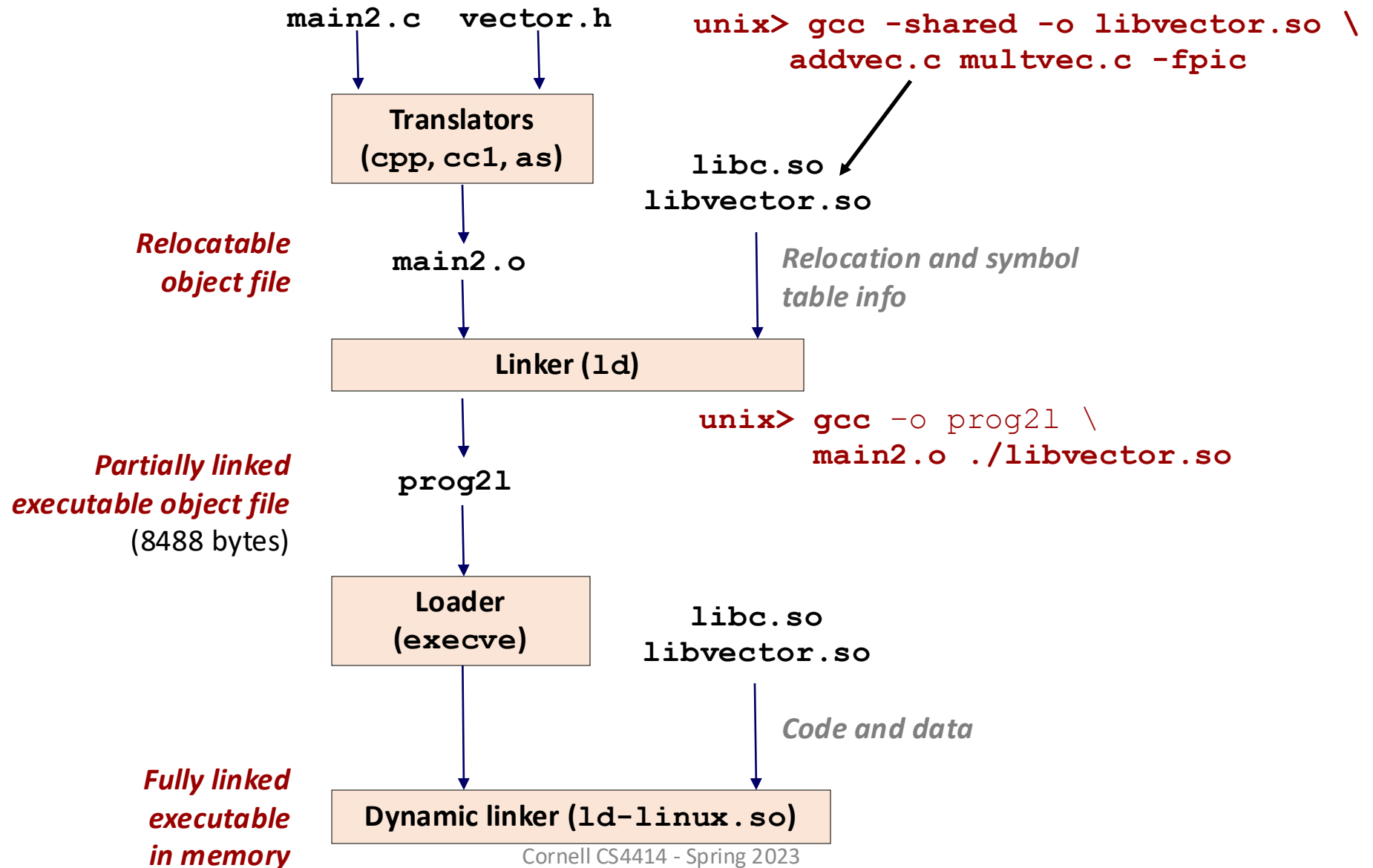
- Linking is the process of combining various object files (and libraries) into a single executable or library.
- Linking happens either at **compile time (static linking)** or at **runtime** (dynamic linking).

```
$ ldd my_exec
```

Loading

- Loading is the process of bringing an executable (and its dependencies) into memory to run it.
- Loading happens at runtime
 - For **statically linked** programs: the operating system **directly loads** the entire binary into memory. No loading or dynamic linking involved.
 - For **dynamically linked** programs: dynamic linker(part of OS) **finds, loads,** and **links** shared libraries into memory.

Dynamic Linking at Load-time



Makefile & Cmake

- What is Makefile and CMake
- Simple CMake
- CMake with linked libraries
- CMake with flags

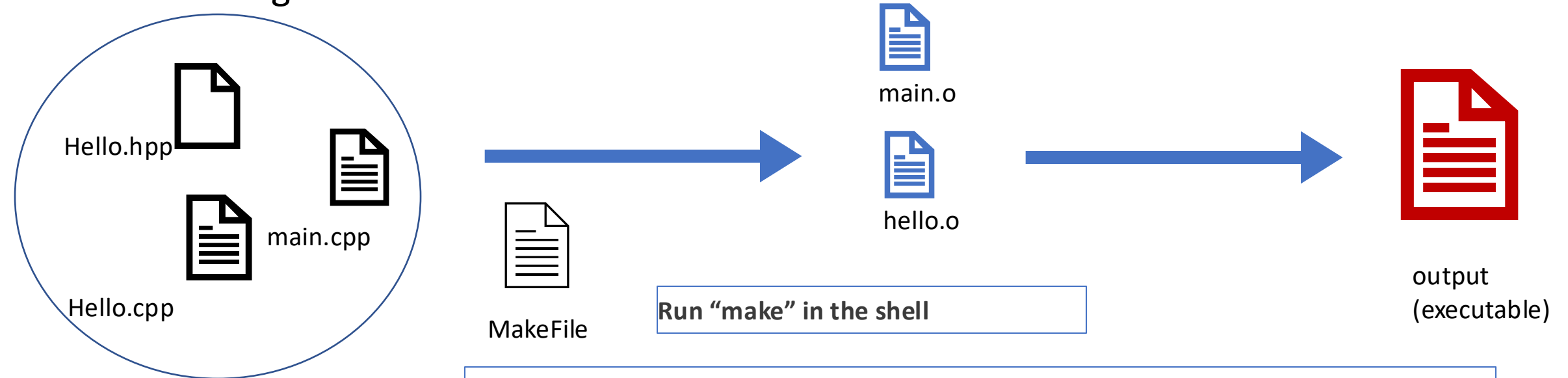
Code source:

<https://github.com/aliciavuting/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.



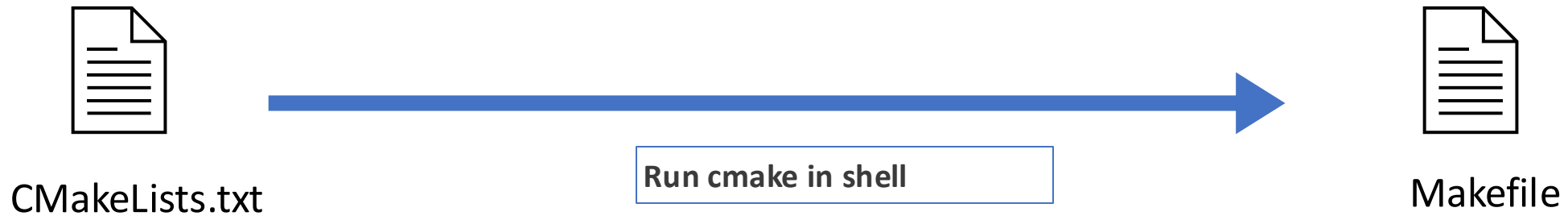
```
CC = g++
CFLAGS = -g -Wall
TARGET = output
all: $(TARGET)
$(TARGET): main.o hello.o
        $(CC) $(CFLAGS) -o $(TARGET) main.o hello.o
main.o: main.cpp hello.hpp
        $(CC) $(CFLAGS) -c main.cpp
hello.o: hello.hpp hello.cpp
        $(CC) $(CFLAGS) -c hello.cpp
```

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.

CMake

CMakeLists.txt files in each source directory are used to generate Makefiles



4.x

6 Branches 126 Tags

Go to file

Add file

Code

About

Open Source Computer Vision Library

opencv.org

opencv c-plus-plus computer-vision
deep-learning image-processing

Readme

Apache-2.0 license

Security policy

Activity

Custom properties

78.6k stars

2.7k watching

55.8k forks

Report repository

Releases 63

OpenCV 4.10.0 Latest
on Jun 3

+ 62 releases

Sponsor this project

opencv OpenCV

Sponsor

asmorkalov	Merge pull request #26281 from kallaballa:clgl_device_discovery	e026a5a · 2 hours ago	34,601 Commits
.github	Force contributors to define Apache 2.0 license for the n...		4 months ago
3rdparty	Merge pull request #26216 from hanliutong:rvv-hal-mer...		last week
apps	Merge pull request #25582 from fengyuentau:dnn/dump...		5 months ago
cmake	Merge pull request #26234 from zachlowry:apply-gcc6-...		last week
data	Merge pull request #22727 from su77ungr:patch-1		2 years ago
doc	Merge pull request #26260 from sturkmen72:upd_doc_...		last week
include	exclude opencv_contrib modules		4 years ago
modules	Merge pull request #26281 from kallaballa:clgl_device_d...		2 hours ago
platforms	Merge pull request #25901 from mshabunin:fix-riscv-aa...		last month
samples	Merge pull request #26212 from jamacias:feature/TickM...		4 days ago
.editorconfig	add .editorconfig		6 years ago
.gitattributes	cmake: generate and install ffmpeg-download.ps1		6 years ago
.gitignore	Merge pull request #17165 from komakai:objc-binding		4 years ago
CMakeLists.txt	build: set cmake policy for if(IN_LIST) support		2 days ago
CONTRIBUTING.md	migration: github.com/opencv/opencv		8 years ago

Why CMake?

- Compilation tool that helps to **generate build file** in a standard way
 - specify build order and dependencies
 - prevent creating cyclic dependencies and common bugs
 - Good at scaling to **large projects**

Cmake with one simple file

- Helloworld demo example

cmakelists.txt

```
cmake_minimum_required(VERSION 3.12) # 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 with libraries

- Demo: main.cpp with hello library
- **add_executable:**
 - create an **executable target** from source files
 - generate the final program that can be run on the system

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
}

target_include_directories(say-hello
PUBLIC ${CMAKE_SOURCE_DIR})

add_executable(output main.cpp)

target_link_libraries(output PRIVATE say-
hello)
```

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**

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
}
```

```
target_include_directories(say-hello
PUBLIC ${CMAKE_SOURCE_DIR})
```

```
add_executable(output main.cpp)
```

```
target_link_libraries(output PRIVATE say-
hello)
```

C++ libraries

- What are in C++ library?
 - A C++ library consists of **header files** and **an object library**.
 - **The header files** provide class and other definitions that the library **exposes** (offers) to the programs using its.
 - **The object library(precompiled binary)** contains **compiled implementation** of functions and data that are linked with your program to produce an executable program.

Cmake with libraries

- Demo: main.cpp with hello library
- 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
}

target_include_directories(say-hello
PUBLIC ${CMAKE_SOURCE_DIR})

add_executable(output main.cpp)

target_link_libraries(output PRIVATE say-
hello)
```


- `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>

```
target_include_directories(<target> [SYSTEM] [AFTER|BEFORE]  
                           <INTERFACE|PUBLIC|PRIVATE> [items1...]  
                           [<INTERFACE|PUBLIC|PRIVATE> [items2...] ...])
```

- Specifies include directories to use when compiling a given target.
- Tells the compiler where to look for header files (e.g., .h, .hpp files) that define functions, classes, or other declarations.

Example of PRIVATE PUBLIC INTERFACE link libraries

```
add_library(my_lib STATIC my_lib.cpp)           # Setting include directories for my_lib

target_include_directories(my_lib                # Link libraries for my_lib
    PRIVATE ${CMAKE_SOURCE_DIR}/include/private # Only my_lib will use this
    PUBLIC  ${CMAKE_SOURCE_DIR}/include/public
    INTERFACE ${CMAKE_SOURCE_DIR}/include/interface )

target_link_libraries(my_lib PRIVATE private_lib PUBLIC public_lib INTERFACE
interface_lib )                                # Add the executable

add_executable(my_app main.cpp)
target_link_libraries(my_app my_lib)           # Link my_app to my_lib
```

Cmake with Flags

- C++ standard (equivalent to -std=c++20)
CMAKE_CXX_STANDARD

cmakelists.txt

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

```
set(CMAKE_CXX_STANDARD 20)
```

```
set(CMAKE_BUILD_TYPE Release)
if(CMAKE_BUILD_TYPE STREQUAL "Release")
    set(CMAKE_CXX_FLAGS_RELEASE
        "${CMAKE_CXX_FLAGS_RELEASE} -O3")
    set(CMAKE_C_FLAGS_RELEASE
        "${CMAKE_C_FLAGS_RELEASE} -O3")
endif()
```

```
add_executable(output main.cpp)
```

Cmake with Flags

- Build Type

```
set(CMAKE_BUILD_TYPE Release)
```

```
set(CMAKE_BUILD_TYPE Debug) // gdb
```

- Optimization level

```
set(CMAKE_CXX_FLAGS_RELEASE  
"${CMAKE_CXX_FLAGS_RELEASE} -O1")
```

```
set(CMAKE_CXX_FLAGS_RELEASE  
"${CMAKE_CXX_FLAGS_RELEASE} -O3")
```

cmakelists.txt

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

```
set(CMAKE_CXX_STANDARD 20)
```

```
set(CMAKE_BUILD_TYPE Release)  
if(CMAKE_BUILD_TYPE STREQUAL "Release")  
    set(CMAKE_CXX_FLAGS_RELEASE  
        "${CMAKE_CXX_FLAGS_RELEASE} -O3")  
    set(CMAKE_C_FLAGS_RELEASE  
        "${CMAKE_C_FLAGS_RELEASE} -O3")  
endif()
```

```
add_executable(output main.cpp)
```

Cmake commands

- Scope of execution

Additional files can be run (added to the scope) using the `add_subdirectory()` command

cmakelists.txt

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

```
set(CMAKE_CXX_STANDARD 20)
```

```
set(CMAKE_BUILD_TYPE Release)
```

```
add_subdirectory(src/rectangle)
```

```
add_subdirectory(src/test)
```

```
add_executable(output main.cpp)
```

Performance Optimization

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

Improve Execution Time Efficiency

1. Performance measurement (timing breakdown analysis)
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`

Gprof by CMake

- Compile the code with flag `-pg` set in **CMakeLists**
- Run the program
 - `$ ./output`
- Create the report file
 - `gprof output > myreport`
- Read the report
 - `vim myreport`

cmakelists.txt

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

```
# Enable gprof profiling
set(CMAKE_CXX_FLAGS
"${CMAKE_CXX_FLAGS} -pg")
set(CMAKE_EXE_LINKER_FLAGS
"${CMAKE_EXE_LINKER_FLAGS} -pg")
```

```
add_executable(output main.cpp)
```

Flat Profile

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	us/call	total 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:** This is the cumulative total number of seconds the computer spent executing this functions, plus the time spent in all the functions above this one in this table.
- **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. Performance measurement (timing breakdown analysis)
2. Identify hot spots
3. Use a better algorithm or data structure
4. Enable compiler speed optimization. (Compile with -O3)
5. Tune the code

Reasoning about system performance

- Which algorithm? A system can be very complex with many features



Fairly optimized code

Highly inefficient code

- A = processing files, B = printing 1 million lines of output

Reasoning about system performance

- Which algorithm? A system can be very complex with many features



Fairly optimized code

Highly inefficient code

- What if step A takes about 99% of the total time? We need to profile and understand performance characteristics of code we write

Where to find the resources?

- Linking and Compilation

- <https://www.cs.cornell.edu/courses/cs4414/2024fa/Schedule.htm> Lecture 13
- CPPCON linker and loaders: <https://www.youtube.com/watch?v=enXulxuNV4>

- Makefile & Cmake

- <https://cmake.org/cmake/help/book/mastering-cmake/chapter/Converting%20Existing%20Systems%20To%20CMake.html>

- Gprof

- GNU gprof manual: https://ftp.gnu.org/old-gnu/Manuals/gprof-2.9.1/html_mono/gprof.html