Computing at Scale

Lecture 2: C++ Compilation Model and Build Systems

Jacob Merson

January 9, 2024

Logistics

Zulip

Moving forward, project announcements will be made on Zulip. Please join the Zulip chat for the class.

`https://computing-at-scale.zulipchat.com/join/edstxivbgnmyzobuljlriyoy/`

Other Logistics

- Review Homework 0 (due 1/16)
- Finish items from lecture 0

Introduction to cppreference

cppreference

`https://en.cppreference.com/w/`

- Go To resource for C++ documentation
- Select language version
- Compiler support tables

C++ Compilation Model

- C++ is a compiled language
- Closely maps to the hardware
- “Zero-Cost Abstractions”
- “You don’t pay for what you don’t use”
- static type checking

Compilation Overview

- Preprocessor fills macros, includes header files, etc.
- Each source file is compiled into it's own object file
- Object files are linked together to create an executable (or library)
- Object files and executables are not portable (they are specific to the platform and compiler, and sometimes the compiler version)

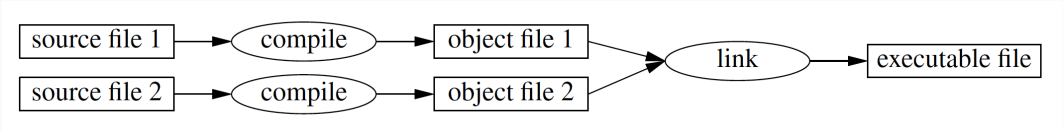

Figure 1: From Tour of C++

Demo

- Compile a simple C++ program to object file (g++ -c)
- Compile simple C program to object file (gcc -c)
- Library with math function, link object files (g++ -o, object files)

One Definition Rule

- ODR: an object or function can only have one definition
- Violating the ODR is undefined behavior
- Violating the ODR can happen when you define a function in a header file and include that header file in multiple source files

Undefined Behavior

undefined behavior: program behavior is not defined by C++ standard

Anything can happen! Anything.

- Program crashes
- Program doesn't crash and gives wrong results
- Program runs fine
- Program calls your mom and tells her your room is messy

Why? That seems dumb. Performance! (more on this later)

Preprocessor

- The preprocessor is a program that runs before the compiler
- Responsible for things like including header files, defining macros, and conditional compilation

Macros

- Macros can define symbols or functions.
- Avoid macros where possible as they make debugging challenging.
- C++ has better ways to deal with generics and constants.
- Can be useful for conditional compilation. But, need to be careful.

include guards

- header files are copy-pasted into source files
- Include guards are a way to prevent a header file from being included multiple times
- They are used to prevent the ODR from being violated
- They are implemented using #ifdef, #ifndef, #define, and #endif

Example Include Guard

```
#ifndef MY_HEADER_H
#define MY_HEADER_H

// header file contents

#endif
```

Include Guard Notes

- include guards need to be unique for each header file
- avoid using `#pragma once` as it is not standard
- Do not start include guards with underscores (items in global namespace starting with underscores are reserved see: `https://en.cppreference.com/w/cpp/language/identifiers`)

Demo

- Create a header file with include guards
- Create a source file that includes the header file twice
- Compile the source file

Static vs Dynamic Linking

- Static linking: the library code is copied into the executable
- Dynamic linking: the library code is loaded at runtime
- Static linking can lead to larger executables
- Dynamic linking can lead to more flexibility

C Linkage vs C++ Linkage

- C++ uses name mangling to encode the function signature in the function name
- C does not use name mangling
- C++ functions can be called from C code, but you need to use `extern "C"` to prevent name mangling

Demo

- compile library with c linkage and c++ linkage, show difference with `nm` and `nm --demangle`

Discussion

- Do you think you can have two functions with the same name, but different argument types in C? How about C++?
- Do you think you can have two functions with the same name, but different return types in C? How about C++?
- What potential problems do you see with mixing and matching compiler versions, flags, etc.?

Build Systems

Building Complex Projects

- For a project with a few files you may be content to compile everything by hand.
- For a large project with many files and dependencies, you will want to programatically build your project. Tools that do this are called build systems.

C++ Build Systems (worth mentioning)

- Make: base build system on Unix systems
- Ninja: faster than Make, not easy to hand code
- CMake: “meta” build system that generates Makefiles, Visual Studio projects, etc., cross platform
- Automake/autoconf: part of GNU build system that brings in platform specific details and dependencies. Used in old HPC codes. AVOID.

Makefiles

- Makefiles are a way to specify how to build a project
- They are a series of rules that specify how to build a target
- uses system time to determine if target is up to date

Makefile Syntax

- A rule is of the form `target: dependencies`
- The rule specifies how to build the target from the dependencies
- The rule is followed by a series of commands that are used to build the target
- The commands are indented with a tab

Example Makefile

```
all: hello

hello: hello.o
    g++ -o hello hello.o

hello.o: hello.cpp
    g++ -c hello.cpp

clean:
    rm -f hello hello.o
```

Demo

- Create a makefile for the math library we created earlier
- run `make` and `make clean`

CMake

- CMake is a “meta” build system (generates build files for other build systems)
- De-facto standard for C++ projects
- Cross platform
- Out of source builds
- Easy integration with dependencies

CMake Phases

- Configure: generate build files
- Build: compile and link (runs make or ninja)
- Install: copy files to install directory (only needed for libraries)
- Test: run tests

Example CMakeLists.txt

```
cmake_minimum_required(VERSION 3.22)
project(hello CXX)

# specify the C++ standard
set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED True)
set(CMAKE_CXX_EXTENSIONS False)

# add a library target
add_library(hellolib hellolib.cpp)

add_executable(hello hello.cpp)
target_link_libraries(hello PRIVATE hellolib)
```

Example CMakeLists.txt, target

```
cmake_minimum_required(VERSION 3.22)
project(hello CXX)

# add a library target
add_library(hellolib hellolib.cpp)
set_target_compile_features(hellolib PUBLIC cxx_std_17)

add_executable(hello hello.cpp)
target_link_libraries(hello PRIVATE hellolib)
```

Useful CMake Functions

Documentation: `https://cmake.org/cmake/help/latest/`

- `add_executable`: add an executable target
- `add_library`: add a library target
- `target_link_libraries`: link a target to a library
- `include_directories`: add include directories
- `find_package`: find a package (e.g., MPI, OpenMP, etc.)
- `add_subdirectory`: add a subdirectory

Running CMake

- Configure `cmake -S . -B build`
- Build: `cmake --build build` or `cd build && make -J 4`
- Install: `cmake --install build`
- Test: `cd build && ctest` or `cd build && make test`

Setting Configure Time Options

Use: `cmake -D<var>=<value>`

- `cmake -DCMAKE_BUILD_TYPE=Release`
- `cmake -DCMAKE_CXX_COMPILER=mpicxx`
- `cmake -DCMAKE_CXX_FLAGS="-Wall -Wextra -Wpedantic"`

Discussion

- What are the advantages of using makefiles or CMake over compiling by hand?
- Can you think of any disadvantages of using makefiles or CMake?