# Software Systems

Day 9 - Build Systems, Structs

# Demoscene

# Today's Topic: Helpful Project Tips

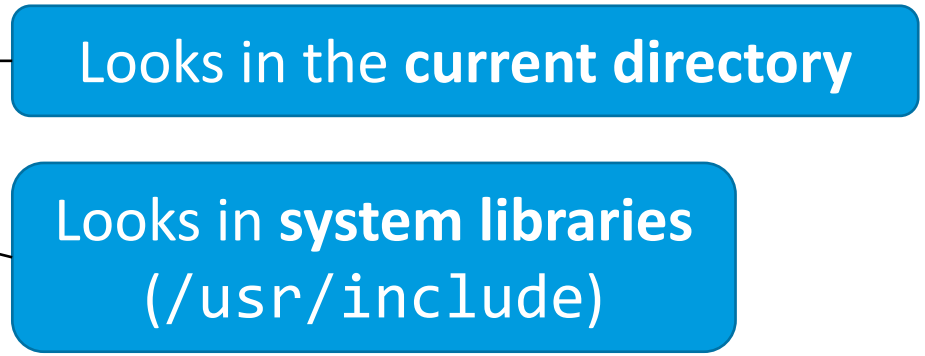
- As everyone gets their projects off the ground, there are a few tips and tricks that will be useful in planning and executing your project.
- We'll talk about how to structure projects in terms of files and directories.
- We'll talk about CMake and how to use it to simplify compilation.
- We'll talk about structs and how to use them in your projects.

# C File Structure: Sources and Headers

- C code typically comes in a .c (source) file or .h (header) file.
- You only ever include header files:

```
#include "foo.h"
#include <stdio.h>

int main(void) {
  return 0;
}
```




- You can include .c files, but it's not recommended.

# C File Structure: Sources and Headers

- Typically, the declaration is in a .h file:
```
double div(int dividend, int divisor);
```
- And the implementation/definition is in a .c file:
```
double div(int dividend, int divisor) {
  return 1.0 * dividend / (1.0 * divisor);
}
```
- The compiler only needs to know the parameter/return types of a function to generate machine code, so you only need to include a .h file.

# C File Structure: Sources and Headers

- The source/header separation means you also don't have to recompile if all that changes is the implementation.
- The interface of a function (name, return type, parameters) changes much less frequently than the implementation (code body) does.

# C File Structure: Sources and Headers

- Exercise:
  - In the course repo, under the folder for today's class session, you will find a directory called `header-exercise`.
  - Place the appropriate code in `main.c` to get the code to compile properly.
  - Use CMake to compile your code and make sure that it works.
  - Once compiled, in the build directory, you can use `./src/main` to run the program.

# C File Structure: Include Guards

- Functions can only be declared once in C, so this won't work:

```
int multiply(int x, int y);
char* multiply(int x, char* y);
// -> Error: multiply already defined
```

- Even if the types are exactly the same, it won't work:

```
int multiply(int x, int y);
int multiply(int x, int y);
// -> Error: multiply already defined
```

# C File Structure: Include Guards

- This applies to `#includes` as well.
- So this won't work:
  - foo.h:
    ```
    #include "bar.h"
    ```
  - main.c:
    ```
    #include "foo.h"
    #include "bar.h"
    ```
- Counting on foo.h to include bar.h isn't good practice, either.

# C File Structure: Include Guards

- Include guards prevent the preprocessor from including the same file twice.
- foo.h:

```
#ifndef FOO_H_
#define FOO_H_
// Header contents go here.
#endif  // FOO_H_
```

- Or, on *most* modern compilers, you can replace all of that with:

```
#pragma once
```

# C File Structure: Include Guards

- Exercise:
  - In the course repo, under the folder for today's class session, you will find a directory called `include-guard-exercise`.
  - Place the appropriate code in the files to get the code to compile properly.
  - Use CMake to compile your code and make sure that it works.
  - Once compiled, in the build directory, you can use `./src/main` to run the program.

# Build Systems: CMake

- If your code is split across files, you could do this in gcc:
```
gcc -o main main.c square.c square.h powmod.c powmod.h
```
- To avoid unnecessary recompilations:
```
gcc -o main.o -c main.c square.h powmod.h
gcc -o square.o -c square.c square.h
gcc -o powmod.o -c powmod.c powmod.h square.h
gcc -o main main.o square.o powmod.o
```
- Or you can use Makefiles to only rebuild what's necessary, but maintaining Makefiles can be tough.

# Build Systems: CMake

- CMake can be difficult, but takes away a lot of the underlying complexity.
- Essentially, CMake generates a Makefile for you, based on a higher-level configuration.
- The typical workflow separates source and build directories to make cleaning up easy.
  - This means you have a `src/` directory and a `build/` directory.
  - You may also have a `test/` directory for unit tests (see assignments on how to set these up).

# Build Systems: CMake

- CMake's configuration files are called CMakeLists.txt and found in every directory of your project containing code.
- Minimal example:
  - Root directory:
    ```
    cmake_minimum_required(VERSION 3.22)
    project(MyProject VERSION 1.0 LANGUAGES C)
    add_subdirectory(src)
    ```
  - `src/` directory:
    ```
    add_executable(main main.c)
    ```

# Build Systems: CMake

- An executable has a main method, while a library does not:
```
add_executable(foo foo.c)
add_library(bar bar.c bar.h)
```
- When you compile files separately, they need to be linked together so their implementations are available.
- In CMake, link libraries like this:
```
target_link_libraries(foo PRIVATE bar)
```
- The different words indicate whether you use the library in the implementation only (PRIVATE), in the header only (INTERFACE), or both (PUBLIC).

# Structs

- Use structs to combine multiple types:

```
typedef struct {
  int real;
  int imaginary;
} complex;
```

- Typically, arrange from largest type to smallest.