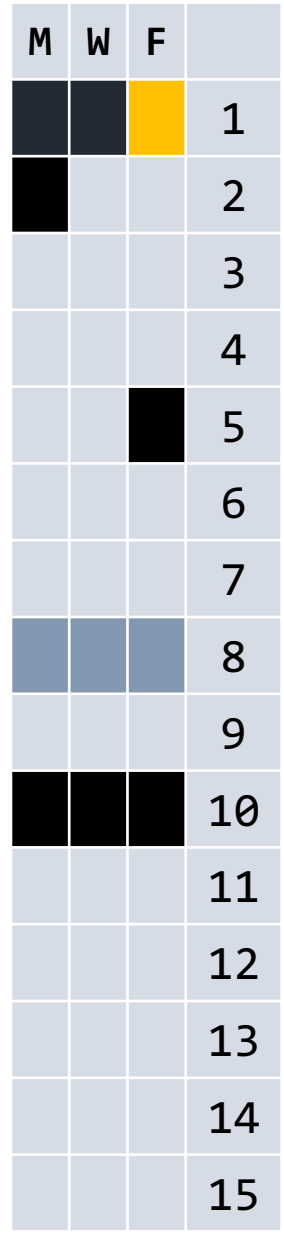
CMOR 421/521
Compilation and Julia



Compiling

Compiling happens in several stages

1. **Preprocessing/precompiling** => Expanded text files (.cpp)
2. **Compilation** => Assembly code (function and symbol tables)
3. **Assembling** => Object code (machine instructions, .o files)
4. **Linking** => **Executable file (.exe) or library (.a)**

These steps can be done all at once for small projects

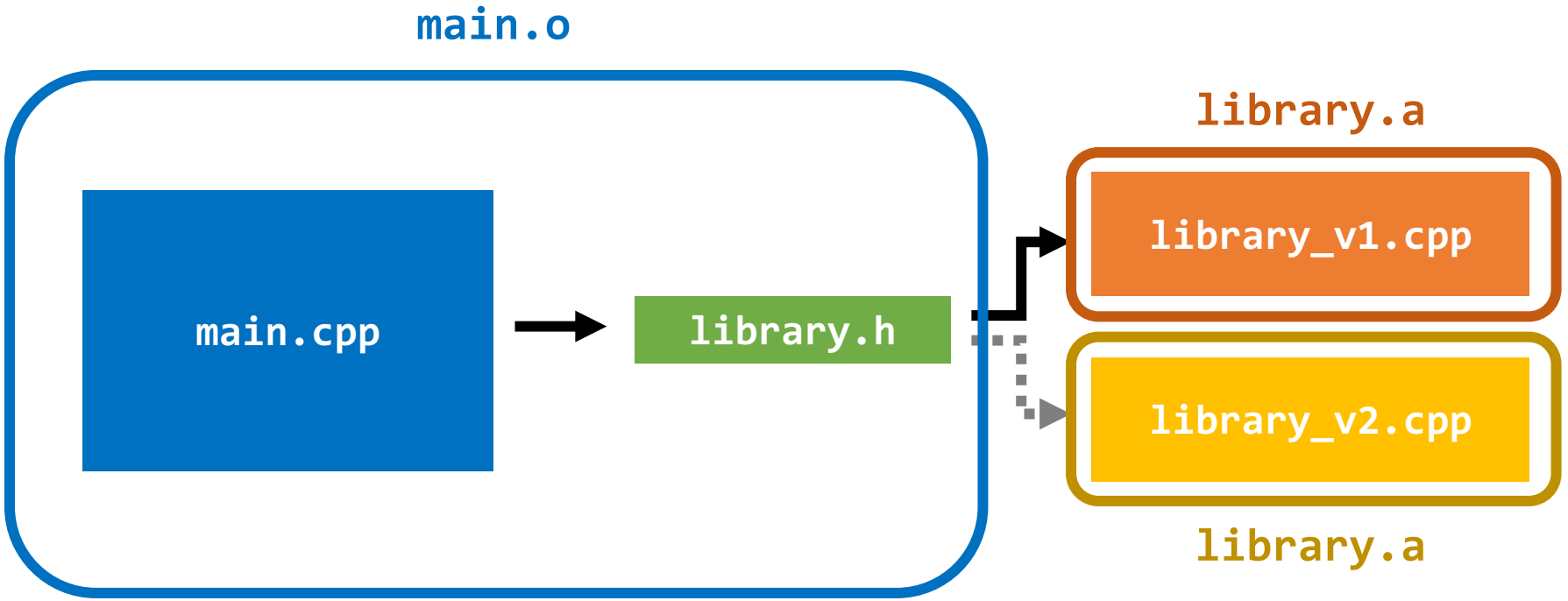
Why Header Files?

Software Engineering: *Independent interface and implementation*

- We can swap out different implementations of the same library
- Header files are easy to digest; they sometimes serve as documentation for the code

Compiling

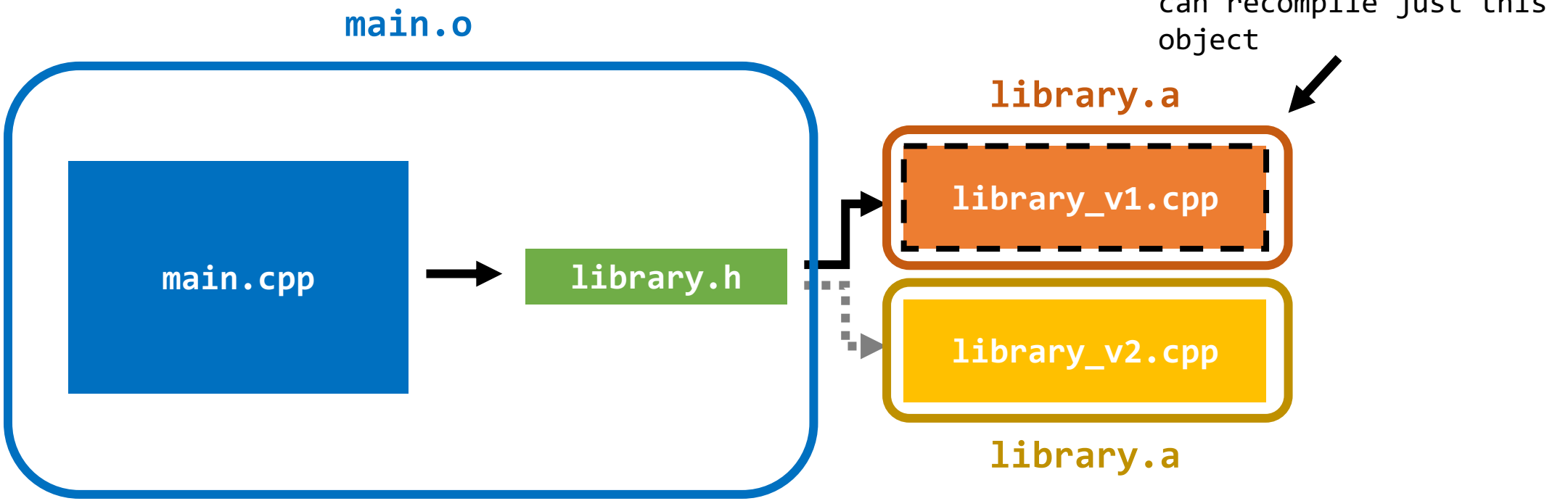
- The object files are step above executable code
- They contain references to functions that may be implemented elsewhere, allowing us to swap implementations



Linking

Software Engineering: *Modularity*

- Changes do not propagate to the whole program; you only need to recompile the objects whose files were changed
- We can swap out implementations



Compiling with GNU

Terminal

```
Basic compiling command:
> g++ -o <exe name> <.cpp files>
```

- This compiles everything all at once (i.e. we don’t have modular compilation), but is fine for small projects
- For larger projects we use a Makefile
  - The Makefile automates compilation rules for several targets
  - Targets can be the main program, its dependencies, testing code, and/or code for use with the debugger, and deleting objects and programs
  - Compiling/building w Makefiles: `> make <target(s)>`

Compiling cont.

- There are a lot of things you can do with compilers
- You can stop the compiler at different stages:
  - Stop after preprocessing (-E): > g++ -E <.cpp files>
  - Stop after compiling (symbol tables, etc) (-S): > g++ -S <.cpp files>
  - Stop after assembling (-c): > g++ -c <.cpp files>
- You can pass flags to the compiler to set different options:
  - -Werror: turns warnings into errors
  - -Wall: show all warnings (some are hidden by default)
  - Optimization settings: -O1 to –O3
  - Version control: -std=c++11

Running a program

Terminal

```
Basic command:
> ./<program name>
```

- Computers have what is called a PATH variable
- The PATH variable contains directories (i.e. file folders)
- These folders tell the computer where to find programs (.exe)
- Your executables likely aren’t in any of those folders
  - They are likely in the folder you are currently in
  - The “./” tells the computer to look for the program in your current directory; “.” means the current directory, “..” means its parent dir
  - You can modify your PATH variable to include “.”, but this can be bad

AOT, JIT, and “JAOT”?

- C/C++ are “Ahead-Of-Time” (AOT) compiled languages
- Python operates on a mix of pre-compiled and “Just-In-Time” (JIT) compilation
- Julia is a “Just-Ahead-Of-Time” (JAOT) language
  - Julia functions are not typed, but Julia compiles type-aware functions “just ahead of” their usage.

Julia and compilation

Julia performs all the main steps of compilation

1. **Preprocessing/precompiling** => Julia code gets converted to an Abstract Syntax Tree (AST) representation
2. **Compilation** => AST to assembly/machine code
3. **Assembling** => Object code (machine instructions, .o files)
4. **Linking** => Julia tracks and runs compiled functions

Julia and multiple dispatch

- Multiple dispatch: similar to function overloading in C/C++
  - Can define multiple instances of a function with different combinations of inputs
- Example: foo(x), foo(x, y), foo(x::Int) define three distinct “instances” of a function “foo”
- Multiple dispatch is **efficient** in Julia

Julia and fast types

- C/C++ is *statically* typed (the type of each variable must be declared)
  - Typed code is easier to optimize by the compiler
- Python is *dynamically* typed, tends to be slower
- Julia is in-between
  - Doesn’t require specifying variable types
  - When a function is run **for the first time**, Julia acts as a compiler → creates fast assembly code. The second time the function is run, it reuses the compiled code.
  - Consequence: Julia functions run **slow** the first time

Julia and efficient code

- Good: because Julia is statically typed like C/C++, it *can* achieve high levels of performance.
- Bad: naïve Julia code may not achieve -03 speeds
- Ugly: Julia requires being able to *infer* each variable's type. If a variable is not inferable at *compile-time*, it generates a "type instability".

Julia and efficient code

- Good: because Julia is statically typed like C/C++, it *can* achieve high levels of performance.
- Bad: naïve Julia code may not achieve -03 speeds
- Ugly: Julia requires being able to *infer* each variable's type. If a variable is not inferable at *compile-time*, it generates a "type instability".
  - Example:

```
if x > 0 # need value of x to infer return type
    return 1 # Int type
else
    return "1" # String type
end
```

Metaprogramming

- Ugly, cont: Julia metaprogramming/macros allow you to manually apply hardware-specific optimizations directly to Julia-generated native code
- Can semi-automate this process; see LoopVectorization.jl for a blazingly fast (but kind of brittle) example.