Building C/C++ Projects with the Zig Build System

Before we start:

- Clone https://github.com/ziglang/zig-build-workshop
  - (Downloading the ZIP is fine if you don’t have Git)
    - (who doesn’t have Git?!)
- Install Zig 0.12.0
  - Very simple: just extract the self-contained tarball
  - Nightly builds will not work!
- If possible, install NASM

Hi!

- Andrew Kelley
  - https://andrewkelley.me/
  - Creator and BDFL of Zig
  - GitHub: @andrewrk
  - Mastodon: @andrewrk@mastodon.social
- Matthew Lugg
  - https://mlugg.co.uk
  - Zig “core team” member since 2023
  - GitHub: @mlugg
  - Mastodon: @mlugg@fosstodon.org

Workshop Structure

- Morning (09:30 – 13:00): guided exercises
  - Learn how to use `zig build`!
  - The repo you just cloned contains these exercises
- Afternoon (14:30 – 17:00): free-form section
  - Build a project of your choice!
  - We’re happy to help with any project if you get stuck

Zig Build System

- Zig: general-purpose programming language and toolchain
  - Easy to learn the basics: don’t worry if you don’t know Zig!
- Includes a build system with good support for building C and C++ code
- ...which you can use in a pure C/C++ project
- The build “script” is itself written in Zig

Zig Build System

- Manages a graph of “steps” which do different things
- When you run `zig build`, you pass names of step[s] to run
- e.g. `zig build install` – “install” your application
- e.g. `zig build test` – run unit tests
- e.g. `zig build run` – build and run your application

Build Script

- Contained in file `build.zig` in root of project tree
- Defines a `fn build` which declares your build steps
- The steps are then *executed* by the “build runner” which ships with Zig
- Steps can do all kinds of things:
  - Compile code
  - Run arbitrary commands
  - Write files
  - Copy files
  - ...etc

Build Script

- One more important concept: `LazyPath`
- Most file/directory paths are represented with this abstraction
- It could be a literal relative path...
  - `b.path("foo.c")`
- ...or a file generated by the build script
  - We'll look at this later on

Task 01: Hello World

- In the cloned repo, go to `01_hello/`
- Read the `README.md`, and do what it says
- We'll regroup in 15 minutes to discuss the solution
- Don't be afraid to ask for help if you need it!
- Also, use the standard library documentation!
  - https://ziglang.com/documentation/0.12.0/std
  - ...or, run `zig std`

- **Goal:** C Hello World built with `zig build`

Task 01: Hello World

- If all is well, you've just built your first C program with `zig build`! (Now that wasn't so hard, was it?)
- If you need it, a model solution can be found in:

  `<repo>/solutions/01/`

- Now let's move onto another task...

Task 02: Multiple Files

- This one is a pretty straightforward extension of the last task. Go to `02_multi` and try it out!
- We'll regroup in about 5 minutes

- **Goal:** Multi-file C program built with Zig

Undefined Behavior

- Okay, so, we tricked you here… guilty as charged!
- **A majority of C and C++ codebases in the wild are riddled with *undefined behavior*.**
- This code may work in practice, but the C spec allows the compiler to make it do anything, including (but not limited to):
  - Give a garbage result
  - Run unrelated functions
  - `rm -rf –no-preserve-root /`
  - Hack into the US government's missile control systems

Undefined Behavior

- Zig enables Clang’s Undefined Behavior Sanitizer (UBSan) by default
- Triggers an error when UB is detected
- When this happens, the correct thing to do is to submit a patch upstream to fix the faulty code, but that can be annoying. Maybe...
  - You need to use this software right now, or...
  - The upstream maintainer is slow at accepting patches, or...
  - You can’t be bothered

Undefined Behavior

- If you are unable to fix the issue upstream, and can’t or don’t want to patch the code in a fork, you can disable UBSan.
- When you add C source files to a compile step, you can pass flags to the compiler
- Passing `-fno-sanitize=undefined` will disable UBSan

Task 03: Creating Libraries

- Let’s move on. The third task focuses on generating static libraries.
  - (We don’t explicitly cover shared libraries, but the process is pretty much identical!)
- We’ll regroup in 5 minutes

- **Goal:** Build a static library and link it to an executable

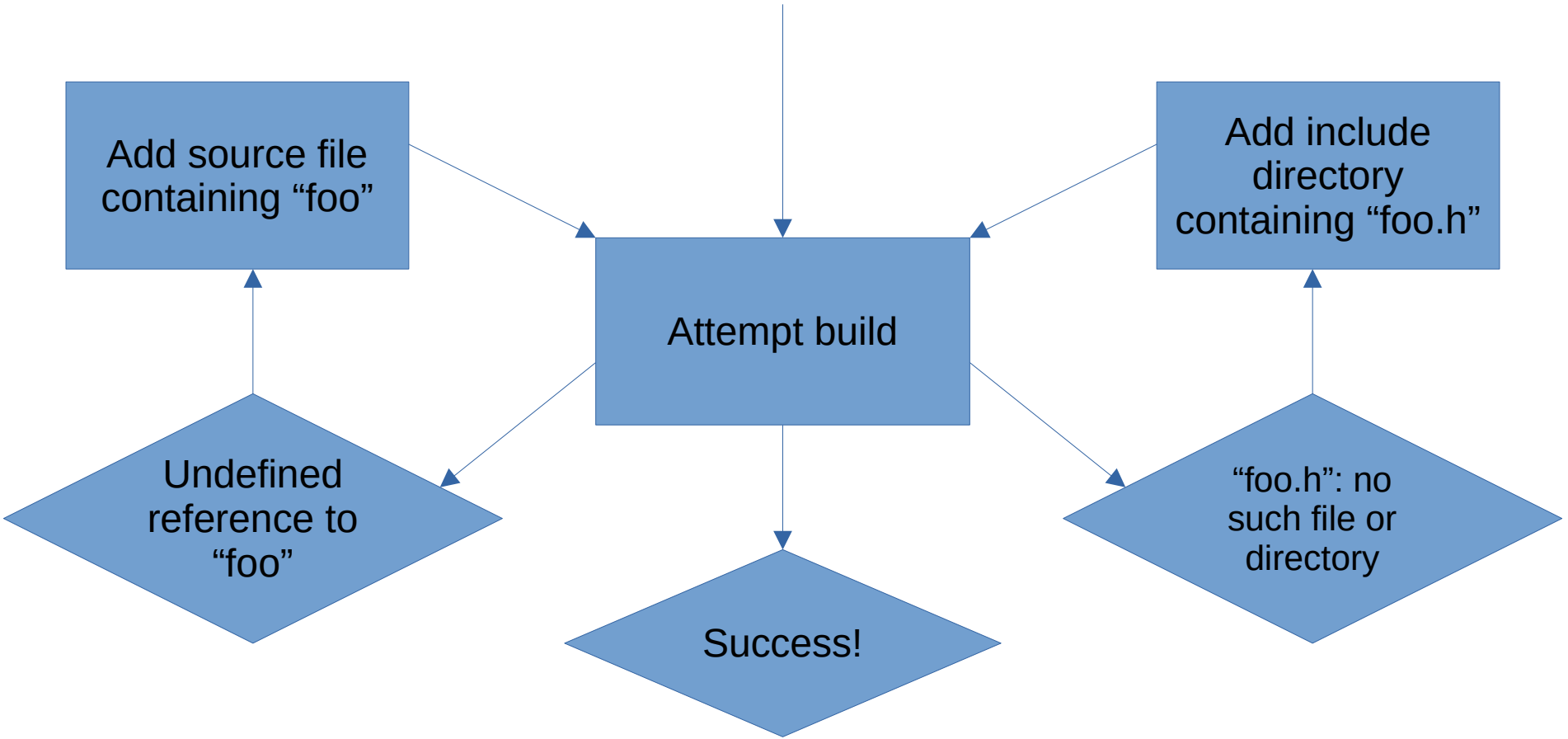
Next Steps

- You now understand the basics!
- Constructing `Compile` steps with…
  - `addExecutable`
  - `addStaticLibrary`
- Adding C source files
- Next, we'll look at building something a little more complicated

Building A Third-Party Project

- Building C/C++ isn’t as scary as it can seem!
- You need to figure out:
  - Which external dependencies to build?
  - Which source files to compile?
  - Which header directories to include?
- After figuring out external dependencies, you can do the rest easily...

Building A Third-Party Project



Task 04: zlib

- Your task is to build `zlib`, a general-purpose compression library written in C
- We'll give you around 30 minutes before we get back together to go over this one
- Good luck!

- **Goal:** Build `zlib` with Zig, and run the provided test code

Aside: External Library Sources

- The strategy we used here was effectively to fork zlib to integrate it with the build system
  - Advantages: straightforward to make required modifications (patches, extra build files, etc)
- Alternative strategy: *pristine tarball*
  - Directly depend on upstream tarball
  - Download and extract it on-demand, and build using only its original contents
  - Advantages: smaller repo, simpler upgrade process, better trust
  - We'll discuss this more later

Running Commands

- `zlib` is kind of the platonic ideal of a C program
  - No external dependencies
  - No build-time configuration needed
- What if our application's build process requires running external commands?
- We use a Run step!
  - `b.addSystemCommand(&.{“program-name”})`

Task 05: Running an Assembler

- Your task is to use NASM to build some x86 assembly and link the result into a C program
- More detailed explanation in `README.md`
- We'll regroup in around 20 minutes

- **Goal:** Build `test.asm` with NASM, and link it to `test.c` to get a Hello World

Running Commands

- What if your build environment doesn’t have NASM?
- Ideally, we would have minimal system dependencies
- This makes the build process easier, and improves cross-platform support
- What if we built NASM ourselves?

Task 06: NASM

- Your task is to build NASM from source using the Zig build system, like you did with `zlib`
- NASM is more complicated: don’t be afraid to ask for help!
- We’ll regroup in around 90 minutes
- **Goal:** Create a working build of NASM with Zig

Using NASM

- We have a working build of NASM, but how do we use it in another project (like task 05)?
- We don’t really want to vendor this dependency in every project that needs it!
- We could use git submodules or similar, but they’re a bit of a pain
- What else can we do?

Package Management

- The Zig build system includes a package manager
- Packages are decentralized: you give a URL for each dependency and the build system fetches it on-demand
- These dependencies can expose build artifacts to your own build script

Package Management

- **Idea:** let’s use our NASM build as a package!
- We can depend on it from our codebase, and have our build script compile and run it to build our project’s assembly code
- We eliminate our dependency on system NASM without bloating the repo!

Task 07: Package Management

- Putting it all together: use our NASM package in the build process of another codebase
- Use the package manager to reference the NASM build you just wrote

- **Goal:** Build `test.asm` and link with `test.c` without a dependency on system NASM