Build systems

Has this ever happened to you?

"Hmm 🤔 how do I build this project? <up> <up> <up> <up> <up> <up> <Enter>"

"Hey Tom, use this command to build the project. Wait, no, this one 👍"

"Can't believe I've been forgetting to compile myfile.c all day 💀💀💀"

Manually typing out g++ every time you compile your project	
Copy-pasting the command every time you compile	
Using the up arrow to navigate your shell history every time you compile	
Writing a shell script to build your project for you	
Using a build system like Make	

What does a build system provide?

- Consistency
  - One simple command runs arbitrarily complex build rules
  - Build happens identically for everyone working on the project
  - Rules can be developed alongside code, tracked in version control
- Efficiency
  - Build is split up into multiple *rules*, which transform inputs into outputs
  - Each rule only needs to run if its inputs have changed
  - Build system figures out the minimal set of rules that need to run

How do you use a build system?

1. Define build rules by writing code
    - Each build system has its own language for rules
    - Some, like Make, have totally custom languages
    - Others build on top of existing languages (e.g. Ruby or Python)
    - Generally the build system looks for rules in a file with a specific name
2. Run the build
    - Invoke the build system using its command-line tool (or an IDE integration)
    - Generally takes few or no arguments—configuration comes from the build rules instead

What kinds of build systems exist?

- Language-agnostic (usually represent rules as shell commands)
  - **Make**
  - **Rake**
  - **SCons**
  - **Ninja**
- Language-specific (often have common operations like compilation hardcoded)
  - **CMake** (C/C++)
  - **npm** (JavaScript/Node.js)
  - **setuptools** (Python)
  - **Cargo** (Rust)
  - **`go build`** (Go)
  - **Cabal** (Haskell)
  - and many more
- Language-specific, with support for multiple languages
  - **Meson**
  - **Bazel**
  - **Buck**

If your project's language comes with a build system, use it! Don't pick a different one just because you happen to already know it.

Why not scripts?

You can replicate a build system's functionality with sufficiently complex scripts. But it won't be fun:

- You'll reinvent functionality that build systems already have
- Your rule definitions will likely be up much more verbose
- Others won't know how to write build rules for your project

What questions do you have?

Build rules: a closer look

- A *rule* represents a single (generally idempotent) operation that transforms one or more *input files* into one or more *output files*.
  - Example: producing a `.o` file from a `.cpp` file
  - "Building" doesn't just mean compilation—rules can do anything
- Build systems don't generally run rules themselves, but instead call external utilities (e.g. `clang++`) or libraries
- If one build rule's input is another build rule's output, the first build rule *depends* on the second.
  - Forms a DAG (like Git!)
- By knowing what output file you want, the build system can figure out all the build rules transitively depended on by that output.

Introducing Make

Make: pros and cons

Pros:

- Immediately relevant to you at Tufts
- Isn't language-specific
- Widely used in the real world
- Knowledge applicable to other build systems

Cons:

- The Make language can be unintuitive
- Not the best choice for many projects
- Missing features that more modern build system provide

Wen Kokke
@wenkokke

Hot take, but I kinda like Make?

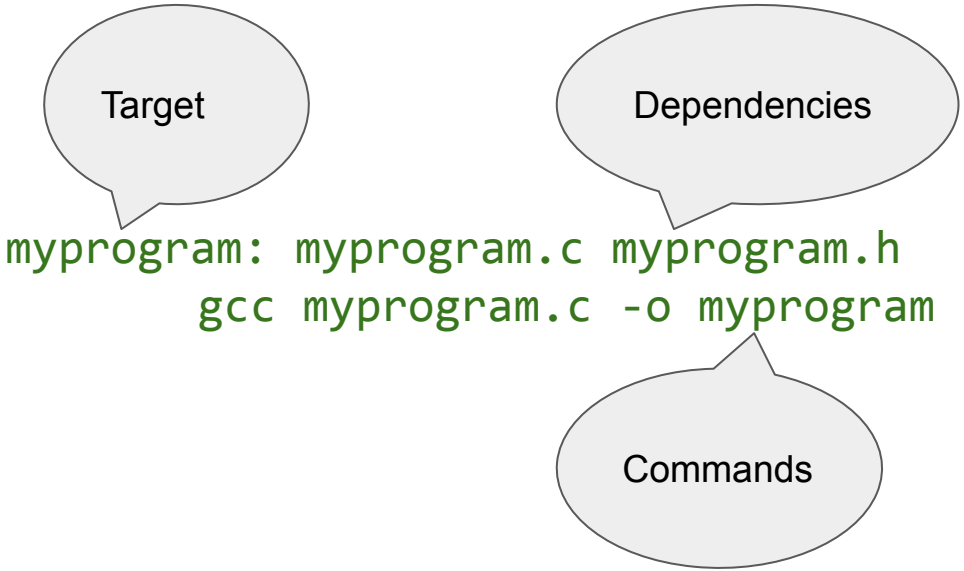
5:00AM · Sep 6, 2021 · Twitter Web App

Make

Reads the following from a file called Makefile

- Variables
- Targets
  - "recipes" for building things
- Commands
  - steps in a given target

Sample Makefile



demo time!