ENGR 301 *Project Management*

Lecture 12 — Build Management

James Quilty

School of Engineering and Computer Science
Victoria University of Wellington

Introduction

Build management concerns management of the processes for producing final artefacts from sources.

- Has been around since artefacts were built
- Manual processes are “default”
- Automated processes have become available

Today’s lecture looks at build management with examples using Make.

Manual build processes

Manual build processes are:

- Error-prone
- Repetitive
- Time-consuming

Create *toil* and the economy of manual processes involves the “monetisation of toil” (Alan Geer, 2019).

Automated build processes

Automated build processes are:

- Consistent
- Reproducible
- Fast[er than people]

History: Make

Automated software build management was codified in the C era by Make but there are now many packages:

- CMake, SCons, MSBuild, Ant, Maven or Gradle
- Very often targeted at specific contexts
- “CMake is not a build system itself; it generates another system’s build files.”

Make is not as context-specific as the other and consequently is still prevalent. Try searching GitHub for `Makefile`: `https://bit.ly/3TsKqO1`;`https://bit.ly/3x0rFdn`.

Introduction: Make

The GNU Autoconf Manual summarises make very well:

> “The ubiquity of make means that a makefile is almost the only viable way to distribute automatic build rules for software, but one quickly runs into its numerous limitations. Its lack of support for automatic dependency tracking, recursive builds in subdirectories, reliable timestamps (e.g., for network file systems), and so on, mean that developers must painfully (and often incorrectly) reinvent the wheel for each project. Portability is non-trivial, thanks to the quirks of make on many systems. On top of all this is the manual labor required to implement the many standard targets that users have come to expect (make install, make distclean, make uninstall, etc.). ... Into this mess steps Automake.”

`https://bit.ly/3vjwLky`

Make Concepts

Originally motivated by someone forgetting to recompile their source code to object code after fixing a bug. Make characteristics:

- Rule-based system
- Internally represented as a directed acyclic graph
- Seeks to fulfil rules by following the graph

Makefiles

A simple Makefile contains *rules*

```
target ... : prerequisites ...
        recipe
        ...
        ...
```

- **target**: usually the name of a file that is generated by a program. A target can also be the name of an action to carry out, such as *clean* (a *Phony Target*).
- **prerequisite**: a file that is used as input to create the target. A target often depends on several files.
- **recipe** an action that Make carries out in a shell process.

Makefiles

What Makefiles often contain:

- *explicit rules*
- *implicit rules*
- *variable definitions*
- *directives*
- *comments*

Pitfalls

There are many:

- Mixes declarative and imperative language (!)
- Tab character indentation by default (!)
- Implicit rules (!)
- Different “styles” of declarative rules (!)
- Cryptic syntax (!)

Remake: An Improved Version of Make

Remake `https://bashdb.sourceforge.net/remake` is available via many package managers. It offers a number of helpful features compared to Make:

- additional CLI options
- improved error reporting
- better tracing
- build profiling
- debugger

Further Reading

- GNU Make Manual
  `https://bit.ly/4a1TPU0`
- Managing Projects with GNU Make
  `https://bit.ly/3PwFVkn`
  `https://www.oreilly.com/openbook/make3/book/`
- Rules of Makefiles
  `https://bit.ly/4ajYUXs`