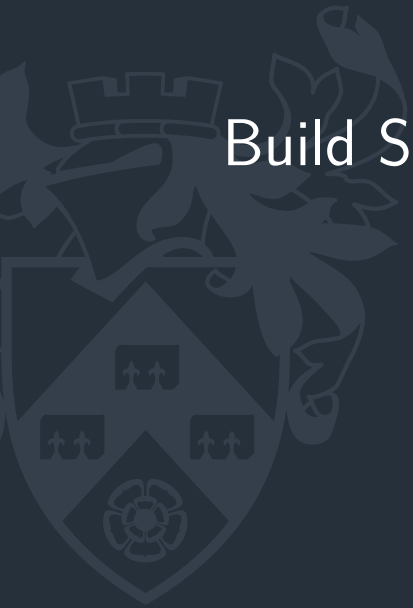

Build Systems and Packaging

Peter Hill

Build systems and packaging

- For Python, pretty easy: setuptools and pip
- For compiled languages, often trickier
- Compiling one file from the command line is easy
  - Likewise two files, maybe three
- Tens or hundreds of files unmanageable
- So we have build scripts
- Or fancier: Makefiles

Before we begin

Two kinds of build systems:

- Those that everyone complains about
- Those that no one has heard of

Portability

- What do you do when you need to use different compilers?

```
ifeq ($(COMPILER),gcc)
# gcc flags
endif
ifeq ($(COMPILER),intel)
# intel flags
endif
```

Portability

- Or different systems have the libraries you need in different places?

```
ifeq ($(SYSTEM),mymachine)
# york flags
endif
ifeq ($(SYSTEM),viking)
# archer flags
endif
```

Portability

- Make is suddenly not the right tool for the job
- What we need is a *build system* or *build system generator*
  - Confusingly, people use both terms to refer to different things
  - Let's not get bogged down in terminology!
- Something that's take what we *want* to build and work out *how* to do it

GNU Autotools

The grandmother of build systems

- You've seen it before:

```
$ ./configure
$ make
$ make install
```

- If you like shell scripts, you'll love Autotools
- Actually a family of related tools: autoheader, autoconf, automake, etc.
- Designed to generate Makefiles portable across POSIX systems
  - Not so useful if you want to also compile on Windows or other weird OSes
- Takes care of a whole bunch of standard things:
  - Different compilers, MPI, etc.
  - Install locations
  - `make clean`, `make install`, `make uninstall`
- Surprisingly easy to get started!

Hello World with Autotools

The basics

1. Write a simple "Hello World" program in your favourite compiled language

```
#include <iostream>
int main() {
  std::cout << "Hello, World!\n";
}
```

2. Try running `make hello` (assuming your file is called `hello.??` and is in C or C++)
3. Make `Makefile.am` with:

```
bin_PROGRAMS = hello
hello_SOURCES = hello.cpp
```

4. Run `autoscan`. This will automagically create a file called `configure.scan` – rename `configure.scan` to `configure.ac`

Hello World with Autotools

The basics

5. Open `configure.ac` and put:
   `AM_INIT_AUTOMAKE([-Wall -Werror foreign])`
   on the line after `AC_INIT`
   - For Fortran, you'll also need to add `AC_PROG_FC` on the line after `AM_INIT_AUTOMAKE` as well
6. Run `autoreconf -fvi`
7. Now run `./configure` then `make`
8. Out of source builds are automatically supported:
   1. `make distclean`
   2. `mkdir build && cd build`
   3. `../configure --prefix=$(pwd)/install`
   4. `make install`
   5. `install/bin/hello`

What have we done?

Makefile.am

- This file tells `automake` what you want to build, and what is needed to build it
- `bin_PROGRAMS`: A list of `PROGRAMS` to install in `bin`
- `hello_SOURCES`: The list of `SOURCES` needed to build `hello`
- You can add other "normal makefile" stuff here too
  - e.g. Fortran dependency generation
- `automake` takes care of all the "usual" targets

autoscan

- Makes a bare-bones `configure.scan` based on your project layout

What have we done?

configure.ac

- These `AC_*` variables are *macros*
- They get replaced by some literal text, possibly after doing something with their arguments
- Important ones:
  - `AC_PROG_CC`/`AC_PROG_CXX`/`AC_PROG_FC`: Look for a C/C++/Fortran compiler and check it works!
  - `AC_CONFIG_FILES([Makefile])`: Create a file called `Makefile` from a file `Makefile.in`
- Other macros for searching for libraries and checking they work
- Find MPI, check your compiler supports C++11, F2008, etc.

What have we done?

`autoreconf`

- `autoreconf` looks for all the important input files and runs all the important autotools programs on them in the correct order
- Creates `Makefile.in` from `Makefile.am`
- Creates `configure` from `configure.ac`
- Neither of these generated files are supposed to be human-readable!
- Also brings in a whole bunch of other files that we won't get into

What have we done?

`configure`

- Takes `Makefile.in` and makes `Makefile` for your actual system, your compilers, libraries, etc., along with where you want to build and install it

`Makefile`

- The thing we actually want!
- Now we can finally compile the program

CMake

- CMake is newer than Autotools, but has been around since 2000
- CMake 3.0 introduced some nicer features in 2014
  - "Modern" CMake
- CMake is a "build system generator"
  - Can make Makefiles as well as a whole bunch of other types, e.g. Ninja
  - Works very well with a huge range of IDEs
- Works well with dependencies, especially if they also use CMake

Hello world with CMake

1. Copy your simple hello world program to a new directory
2. We need a `CMakeLists.txt` file with three lines:

```
cmake_minimum_required(VERSION 3.10)
project(my_hello VERSION 0.1 LANGUAGES CXX) # Or C or Fortran
add_executable(hello hello.cpp)
```

3. Now make a `build` directory and `cd` into it
   - Prefer out-of-source builds
4. `cmake ..` instead of `configure`
5. `make && ./hello` as usual
6. Alternatively, `cmake --build . && ./hello`

What have we done?

CMakeLists.txt

- This is the equivalent of autotool's `configure` script, only in the CMake language
- `cmake_minimum_required` sets the minimum version of CMake. You should try to use the most recent version if you can
  - `pip3 install --user cmake`!
- `project` defines a project and the languages it uses. CMake will find the compilers.
- `add_executable` defines an executable and its source files

Something a bit more complicated

libsay

1. Move the "Hello, World" bit of your program into a new function in a separate file
2. Organise your project a bit like this:

```
+-- CMakeLists.txt
+-- include
|   +-- libsay
|       +-- say.hpp
+-- src
    +-- hello.cpp
    +-- libsay
        +-- say.cpp
```

   - `hello.cpp` and `say.cpp` should both `#include "libsay/say.hpp"`
   - Fortran doesn't need the `include` directory

Something a bit more complicated

libsay

We need to add a few lines to our `CMakeLists.txt`:

```
add_library(say src/libsay/say.cpp include/libsay/say.hpp)
target_include_directories(say PUBLIC include)

add_executable(hello src/hello.cpp)
target_link_libraries(hello PRIVATE say)
```

Something a bit more complicated

Installing

CMake needs to be told what to install and where

```
set_property(TARGET say
    PROPERTY PUBLIC_HEADER include/libsay/say.hpp)

install(TARGETS hello say
    EXPORT libsay
    ARCHIVE DESTINATION lib
    LIBRARY DESTINATION lib
    RUNTIME DESTINATION bin
    PUBLIC_HEADER DESTINATION include)
```

Something a bit more complicated

libsay

- CMake takes options with `-D`, such as `-DCMAKE_INSTALL_PREFIX=$(pwd)/install` (to install files under `./install`) or `-DCMAKE_BUILD_TYPE=Debug` (for debug flags)
- List all options with `cmake -LH`
- Try just running `make install` again from your build directory!
- `ccmake` is a slightly fancier TUI

What have we done?

add_library

- Creates a new library as a target. We can control whether its built as a shared or static library either with the explicit SHARED/STATIC keywords or with BUILD_SHARED_LIBS option

target_include_directories

- Sets the "include directories" property of its target, and whether we only need them to build the target itself (PRIVATE) or if we also need them when we want to use the target (PUBLIC)

What have we done?

target_link_libraries

- Tells CMake to link the target against the listed libraries. This can be another CMake target or an external library
- This adds all the information about the library to the target, e.g. the include directories

set_property

- Sets further properties on a target or other object

What have we done?

install

- Just lists what targets should be installed and where to
- `ARCHIVE` for static libraries
- `LIBRARY` for shared libraries
- `RUNTIME` for binaries
- `PUBLIC_HEADER` for headers

Meson

- New comer, first release 2013
- Python-like syntax
- Very fast, simple things are very simple
- Uses Ninja build system rather than Makefiles
- Can automatically fetch and compile dependencies through its “wrap” system

Hello World with Meson

1. Start off with your simple “Hello world” single file
2. Make `meson.build` with the following two lines:

```
project('hello', 'cpp') # or 'c' or 'fortran'
executable('hello', 'hello.cpp')
```

3. Create a build directory:

```
$ meson build
```

4. From the build directory, run ninja:

```
$ ninja && ./hello
```

libhello

1. Copy your librarified “hello world”
2. Update your `meson.build` file

```
incdir = include_directories('include')
lib = shared_library('say', 'src/say/say.cpp',
                     include_directories: incdir)
executable('hello', 'hello.cpp', link_with: lib,
           include_directories: incdir)
```

What have we done?

`project`

- Defines a project and what languages it uses

`executable`

- Defines an executable and its source files
- Targets in Meson are immutable: you have to define all their properties when you create them
  - What libraries to link against
  - What directories to include

What have we done?

`include_directories`

- Defines directories to be included

`shared_library`

- Defines a shared library

Python

setup.py

- Python packaging (mostly) a lot simpler
- Has it's own complications
- Write a `setup.py` at the top-level
- Enables installing with `pip`

Hello World with Python

Project layout

```
+-- setup.py
+-- hello
    +-- __init__.py
    +-- hello.py
```

Files

```
# __init__.py
from .hello import hello

# hello.py
def hello():
    print("Hello, World!")
```

Hello World with Python

setup.py

```
from setuptools import setup

setup(name="hello",
      version="0.1",
      packages=["hello"],
)
```

- Now you can install with `pip install --user -e .`
- `-e` argument makes it “editable”: no need to reinstall while you develop

Slightly fancier Python Package

Package + executable

```
setup(name="hello",
      version="0.1",
      packages=["hello"],
      entry_points={
          "console_scripts": [
              "hello = hello.hello:hello"]},
)
```

Other setup options

Requirements

- `install_requires`: other packages and their versions
- `extras_require`: optional packages
- `python_requires`: which versions of Python are required

Metadata

- `author`
- `description`, `long_description`
- `url`
- `classifiers`

Python distribution

Make it installable from (almost) anywhere in the world

- `pip3 install --user --upgrade setuptools wheel twine`
- `python3 setup.py sdist bdst_wheel`
  - Makes “wheel” and tarball for distribution
- `twine upload dist/my-package-0.1.0*`
  - Uploads package to PyPI
  - You’ll need account first!

Further reading

Autotools

- https://www.lrde.epita.fr/~adl/dl/autotools.pdf

CMake

- https://cliutils.gitlab.io/modern-cmake/

Meson

- https://mesonbuild.com/

Python

- https://setuptools.readthedocs.io/en/latest/setuptools.html
- https://packaging.python.org/tutorials/packaging-projects/