Durham University

COMP51915

attendance

Durham University

Build Systems & Containers

COMP51915 – *Collaborative Software Development*
Michaelmas Term 2024

Christopher Marcotte[1]

[1] For errata and questions please contact christopher.marcotte@durham.ac.uk

Outline

Outline

Learning Goals

- understand what a build system does,
- be able to build archetypal production software,
- understand how to deploy a containerized project,
- be able to assess these systems for your own projects.

Outline

Learning Goals

- understand what a build system does,
- be able to build archetypal production software,
- understand how to deploy a containerized project,
- be able to assess these systems for your own projects.

There will be a **task** given at the end of these slides.

What is a build system?

A Build System orchestrates multiple programs along with their respective inputs and outputs. Build Systems commonly use the output of a program as the input to other programs.

What is a build system?

A Build System orchestrates multiple programs along with their respective inputs and outputs. Build Systems commonly use the output of a program as the input to other programs.

This is very generic. In practice:

A *build system* produces a *software artifact* from *code* and *data*.

What is a build system?

A Build System orchestrates multiple programs along with their respective inputs and outputs. Build Systems commonly use the output of a program as the input to other programs.

This is very generic. In practice:

A *build system* produces a *software artifact* from *code* and *data*.

This is still very generic... let's consider some examples.

Our First Build System

The simplest build system is... a `bash` script!

```
g++ -fopenmp -O3 myCode.cpp -lm -o my.app
chmod +x my.app
./my.app -test all > err.log
```

This script

1. compiles `myCode.cpp` into an executable `my.app` with a fixed set of flags,
2. executes the executable with a testing flag, and
3. writes the errors to a file.

Our First Build System

The simplest build system is... a `bash` script!

```
g++ -fopenmp -O3 myCode.cpp -lm -o my.app
chmod +x my.app
./my.app -test all > err.log
```

This script

1. compiles `myCode.cpp` into an executable `my.app` with a fixed set of flags,
2. executes the executable with a testing flag, and
3. writes the errors to a file.

What would we do if we wanted to produce *different* executables?

This is getting out-of-hand...

```
typeExec=${1} #input to script
case $typeExec in
  "debug")
    g++ -fopenmp -O1 myCode.cpp -lm -o my.app;;
  "release")
    g++ -fopenmp -O3 myCode.cpp -lm -o my.app;;
  "test")
    ./my.app -test all > err_${typeExec}.log;;
  # and yet more ...
esac
```

This is getting out-of-hand...

```
typeExec=${1} #input to script
case $typeExec in
  "debug")
    g++ -fopenmp -O1 myCode.cpp -lm -o my.app;;
  "release")
    g++ -fopenmp -O3 myCode.cpp -lm -o my.app;;
  "test")
    ./my.app -test all > err_${typeExec}.log;;
  # and yet more ...
esac
```

Compilation scripts:

1. long, repetitive,
2. error-prone,
3. hard to maintain,
4. *not worth your time.*

This is getting out-of-hand...

```
typeExec=${1} #input to script
case $typeExec in
  "debug")
    g++ -fopenmp -O1 myCode.cpp -lm -o my.app;;
  "release")
    g++ -fopenmp -O3 myCode.cpp -lm -o my.app;;
  "test")
    ./my.app -test all > err_${typeExec}.log;;
  # and yet more ...
esac
```

Compilation scripts:

1. long, repetitive,
2. error-prone,
3. hard to maintain,
4. *not worth your time.*

You should use a dedicated build system for all but the simplest software.

Desirable properties of build systems

Build systems typically aim for certain properties:

- extensibility,
- maintainability,
- legibility, and
- reproducibility.

The goal is to empower programmers so that
one person can *write* the software, but hundreds can *use* it.

Desirable properties of build systems

Build systems typically aim for certain properties:

- extensibility,
- maintainability,
- legibility, and
- reproducibility.

The goal is to empower programmers so that
one person can *write* the software, but hundreds can *use* it.

The point of a build system is to *minimize effort* and *maximize productivity*.

Makefiles

Getting Started with `make`

The first thing we need to use `make` is a Makefile.

These contain a list of rules following a particular pattern:

```
target: prereqs
	recipe
```

- `target` is an file or command name
- `prereqs` is a list of inputs for `target`
- and `recipe` is a *tab-indented* command

Recipes are executed by `make`, using `prereqs` to form `target`.

Our First Makefile

```
# makefile for app
app : main.o
      g++ -o app main.o
main.o : main.c defs.h
      g++ -c main.c
docs.html : docs.md
      pandoc -o docs.html docs.md
test : app
      ./app --test
clean :
      rm main.o docs.html
clean-$@ : docs.html
      rm docs.html
```

Our First Makefile

```
# makefile for app
app : main.o
      g++ -o app main.o
main.o : main.c defs.h
      g++ -c main.c
docs.html : docs.md
      pandoc -o docs.html docs.md
test : app
      ./app --test
clean :
      rm main.o docs.html
clean-$@ : docs.html
      rm docs.html
```

With this Makefile:

- calling `make` → `main.o` → `app`
- calling `make docs.html` → `pandoc` → `docs.html`
- calling `make test` runs `app --test`
- calling `make clean` deletes `main.o` and `docs.html`

Our First Makefile

```
# makefile for app
app : main.o
      g++ -o app main.o
main.o : main.c defs.h
      g++ -c main.c
docs.html : docs.md
      pandoc -o docs.html docs.md
test : app
      ./app --test
clean :
      rm main.o docs.html
clean-$@ : docs.html
      rm docs.html
```

With this Makefile:

- calling `make` → `main.o` → `app`
- calling `make docs.html` → `pandoc` → `docs.html`
- calling `make test` runs `app --test`
- calling `make clean` deletes `main.o` and `docs.html`

Makefiles standardize your builds!

Ignoring existing files and `.PHONY:`

Sometimes `make` will refuse to run an action if

1. the target already exists; or
2. there's no action associated with the target.

But, *sometimes* we want a rule to *always* run.

Ignoring existing files and `.PHONY:`

Sometimes `make` will refuse to run an action if

1. the target already exists; or
2. there's no action associated with the target.

But, *sometimes* we want a rule to *always* run.

Marking the rule as `.PHONY: rule` in our Makefile will prevent messages like:

`make: target is up to date` or

`make: Nothing to be done for target`

Ignoring existing files and `.PHONY:`

Sometimes `make` will refuse to run an action if

1. the target already exists; or
2. there's no action associated with the target.

But, *sometimes* we want a rule to *always* run.

Marking the rule as `.PHONY: rule` in our Makefile will prevent messages like:

`make: target is up to date` or

`make: Nothing to be done for target`

The typical use-case is `.PHONY: clean` – `make clean` will always run rule `clean:`

Advanced Makefiles

Most intuition from `bash` scripting also holds for Makefiles. You can:

- Create named variables,
- Have implicit build rules,
- Group commands by prereq,
- Invoke other programs, like `git`...

Advanced Makefiles

Most intuition from `bash` scripting also holds for Makefiles. You can:

- Create named variables,
- Have implicit build rules,
- Group commands by prereq,
- Invoke other programs, like `git`...

GNU `make` is an extremely flexible and legible build system —
it should be your first stop when assessing build systems for a project.

```
# The final build step.
$(BUILD_DIR)/$(TARGET_EXEC): $(OBJS)
    $(CXX) $(OBJS) -o $@ $(LDFLAGS)

# Build step for C source
$(BUILD_DIR)/%.c.o: %.c
mkdir -p $(dir $@)
    $(CC) $(CPPFLAGS) $(CFLAGS) -c $< -o $@

# Build step for C++ source
$(BUILD_DIR)/%.cpp.o: %.cpp
mkdir -p $(dir $@)
    $(CXX) $(CPPFLAGS) $(CXXFLAGS) -c $< -o $@
```

Advanced Makefiles

The syntax can be obscure

- `%`: matches strings
- `$<`: first prereq
- `$@`: target name
- `*`: matches filenames on your filesystem
- `$^`: all prereqs

```
# The final build step.
$(BUILD_DIR)/$(TARGET_EXEC): $(OBJS)
    $(CXX) $(OBJS) -o $@ $(LDFLAGS)

# Build step for C source
$(BUILD_DIR)/%.c.o: %.c
mkdir -p $(dir $@)
    $(CC) $(CPPFLAGS) $(CFLAGS) -c $< -o $@

# Build step for C++ source
$(BUILD_DIR)/%.cpp.o: %.cpp
mkdir -p $(dir $@)
    $(CXX) $(CPPFLAGS) $(CXXFLAGS) -c $< -o $@
```

Advanced Makefiles

The syntax can be obscure

- `%`: matches strings
- `$<`: first prereq
- `$@`: target name
- `*`: matches filenames on your filesystem
- `$^`: all prereqs

The Makefile Cookbook is a great resource for writing a Makefile.

Makefile limitations & pitfalls

One of the first things people try with `make` is automating the building of large documents—e.g. books—with LaTeX.[2]

```
book.pdf : $(texfiles)
    pdflatex book.tex
    bibtex book.aux
    pdflatex book.tex
    pdflatex book.tex
```

Your recipe ***should not*** be:

- Iterative, or
- Interactive, or
- Output to another program

LaTeX is all of these – it can be very tricky to use it with `make`!

[2] Consider `tectonic` for your offline TeX document processing needs, otherwise stick with Overleaf – and please attend my LaTeX workshop in Term 2.

Implicit Rules & C

`make` can also use *implicit rules* – recipes which are not explicitly in the Makefile but inferred by the `make` program.

These are mostly related to *C* compilation and very useful, but difficult to parse.

```
CC = gcc     # Flag for implicit rules
CFLAGS = -g # Flag for implicit rules

blah: blah.o

blah.c:
  echo "int main() { return 0; }" > blah.c
```

Note that `blah.o` is not an output for any rule – it is inferred by `make` implicitly from `blah.c`

I wouldn't use implicit rules.

CMake

CMake: In Theory & In Practice

CMake differs from GNU `make` in two important ways:

CMake: In Theory & In Practice

CMake differs from GNU `make` in two important ways:

1. *CMake* can act as preprocessor for a build system, e.g. `make`.
2. *CMake* is truly cross-platform, so you only specify the build *once*.

CMake: In Theory & In Practice

CMake differs from GNU `make` in two important ways:

1. *CMake* can act as preprocessor for a build system, e.g. `make`.
2. *CMake* is truly cross-platform, so you only specify the build *once*.

You process and build your project with:

```
cmake .
cmake --build . # or make
```

Specifying `CMakeLists.txt`

CMake is configured by one-or-more `CMakeLists.txt` files.

Specifying `CMakeLists.txt`

CMake is configured by one-or-more `CMakeLists.txt` files.

`CMakeLists.txt` contains (at least) three statements:

- `cmake_minimum_required(<version>)`
- `project(<name>)`
- `add_executable(<name> <source file>)`

And may specify: the C++ standard,

Specifying `CMakeLists.txt`

CMake is configured by one-or-more `CMakeLists.txt` files.

`CMakeLists.txt` contains (at least) three statements:

- `cmake_minimum_required(<version>)`
- `project(<name>)`
- `add_executable(<name> <source file>)`

And may specify: the C++ standard, common source files,

Specifying `CMakeLists.txt`

CMake is configured by one-or-more `CMakeLists.txt` files.

`CMakeLists.txt` contains (at least) three statements:

- `cmake_minimum_required(<version>)`
- `project(<name>)`
- `add_executable(<name> <source file>)`

And may specify: the C++ standard, common source files, optional libraries (and their requirements),

Specifying `CMakeLists.txt`

CMake is configured by one-or-more `CMakeLists.txt` files.

`CMakeLists.txt` contains (at least) three statements:

- `cmake_minimum_required(<version>)`
- `project(<name>)`
- `add_executable(<name> <source file>)`

And may specify: the C++ standard, common source files, optional libraries (and their requirements), system-dependencies,

Specifying `CMakeLists.txt`

CMake is configured by one-or-more `CMakeLists.txt` files.

`CMakeLists.txt` contains (at least) three statements:

- `cmake_minimum_required(<version>)`
- `project(<name>)`
- `add_executable(<name> <source file>)`

And may specify: the C++ standard, common source files, optional libraries (and their requirements), system-dependencies, compilation flags, etc...

CMake Example

`ls` reveals:

- `CMakeLists.txt`,
- `main.cpp`, and
- `build_dir/`

CMake Example

`ls` reveals:

- `CMakeLists.txt`,
- `main.cpp`, and
- `build_dir/`

In a shell we run:

```
cd build_dir
cmake ../
cmake --build .
```

Invoking these commands:

- moves you into the build directory `Example/build_dir/`,
- configures the build according to `../CMakeLists.txt`, and
- then builds the project in `build_dir`.

Advanced *CMake*

A very useful *CMake* feature[3] is automatic finding of shared libraries.

[3] Most we can't cover here – it's a very expansive build system...

Advanced *CMake*

A very useful *CMake* feature[4] is automatic finding of shared libraries.

Here is an example of usage with Boost:

```
find_package(Boost 1.36.0)
if(Boost_FOUND)
  include_directories(${Boost_INCLUDE_DIRS})
  add_executable(foo foo.cc)
endif()
```

[4] Most we can't cover here – it's a very expansive build system...

How do we ensure that, when the user runs the program we just built, it works?

How do we ensure that, when the user runs the program we just built, it works?

How do we take responsibility for development ***and*** *deployment?*

Containers

A *container* is a sandboxed process running an **image** containing your application code together with all required dependencies.

Containers & Images

A *container* is a sandboxed process running an **image** containing your application code together with all required dependencies.

An *image* is a file containing an isolated file system, and all files needed to run your application in a **container** environment.

Containers & Images

A *container* is a sandboxed process running an **image** containing your application code together with all required dependencies.

An *image* is a file containing an isolated file system, and all files needed to run your application in a **container** environment.

To run a program in a container, we must:

1. Build the application image, and then
2. Run the container.

These are the same steps for the build systems – first specify, then run.

Docker

Containers, as a technology, are agnostic to any particular product.

Docker is the de-facto standard — it's unavoidable…

Docker

Containers, as a technology, are agnostic to any particular product.

Docker is the de-facto standard — it's unavoidable…

Related software:

- Podman container software by the RedHat developers
- Apptainer, neé Singularity,
- containerd a container runtime for managing system calls
- Kubernetes for management of large-scale container deployment

Containerization

```
# Use an official base image
FROM gcc:latest

# Set the working directory
WORKDIR /app

# Copy your source code to the container
COPY . /app

# Compile the C++ application
RUN g++ -o myapp main.cpp

# Define the container start command
CMD ["./myapp"]
```

Docker uses a Dockerfile to specify the image build.

Containerization

```
# Use an official base image
FROM gcc:latest

# Set the working directory
WORKDIR /app

# Copy your source code to the container
COPY . /app

# Compile the C++ application
RUN g++ -o myapp main.cpp

# Define the container start command
CMD ["./myapp"]
```

Docker uses a Dockerfile to specify the image build.

- starts `FROM` *base image*
- sets directory access
- **compiles** the application
- defines run command

Deployment with Containers

To build your image:

```
docker build -t my-cpp-app .
```

To run your container:[5]

```
docker run -it my-cpp-app
```

- Communication is handled by exposing a network port.
- Dual server-client applications can be packaged together.

[5] `-i` for interactivity, and `-t` for an allocated terminal emulator (TTY).

Deployment with Containers

To build your image:

```
docker build -t my-cpp-app .
```

To run your container:[6]

```
docker run -it my-cpp-app
```

- Communication is handled by exposing a network port.
- Dual server-client applications can be packaged together.

[6] `-i` for interactivity, and `-t` for an allocated terminal emulator (TTY).

Limitations of Containers

Containers promise a consistent environment for the development and deployment of an application – write *once*, run *anywhere*.[7]

Containers are *lighter* than a virtual machine, but not *as efficient* as an optimized application running on the OS.

Containers are siloed and may be difficult to have interact with the rest of the system effectively.

Containers are still not ubiquitous on clusters, especially University ones, so you may need to still engage with bespoke build environs.[1]

[7] How reliably this promise is actually achieved is another story, e.g. tsunami model in Docker on Apple Silicon.

Further Reading

There is *a lot* of build system & container advice online.

We have covered only a few sensible options.

Consider these references for future usage:

- *GNU* `make` manual
- Software Carpentry: *Automation and Make*
- The *Mastering CMake* book
- Docker *Getting Started* Guide

Build Systems Workshop Tasks

mfem is a project aimed at the solution of partial differential equations using h-adaptive refinement strategies with Finite Elements, written in C++.

deal.ii is a project aimed at the solution of partial differential equations using h^p-adaptive refinement strategies with Finite Elements, written in C++.

Tasks

1. Run any of the mfem examples.
2. Run any of the deal.ii tutorials.

You have *just* learned about build systems – not expecting you to write one!

Task Guidance

We encourage you to:

- work together,
- consult the web,
- find creative solutions,
- document your process,
- ask questions...

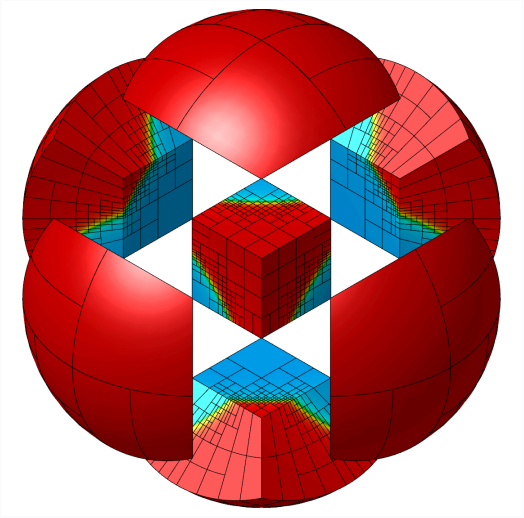

Figure 1: mfem logo

This task is meant to prepare you for the workshop coursework.

Bibliography

[1] J. Watada, A. Roy, R. Kadikar, H. Pham, and B. Xu, “Emerging trends, techniques and open issues of containerization: A review,” *IEEE Access*, vol. 7, pp. 152443–152472, 2019, [Online]. Available: https://ieeexplore.ieee.org/iel7/6287639/8600701/08861307.pdf