Make: the universal project management tool

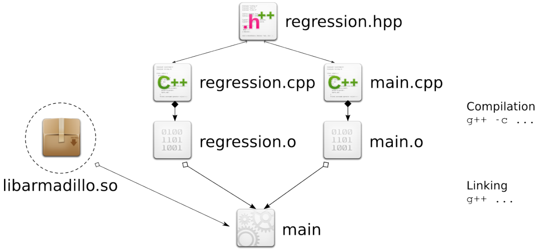
Jamie Fairbrother

August 11, 2014

Motivation

- Computer projects, such as building software or large documents, may often involve processing several different files
- Terminal commands for this processing may be complicated
- These projects may have complicated dependency structures
- Running all required build commands is tedious and error-prone

A simple C++ project



regression.hpp

regression.cpp

main.cpp

Compilation
`g++ -c ...`

regression.o

main.o

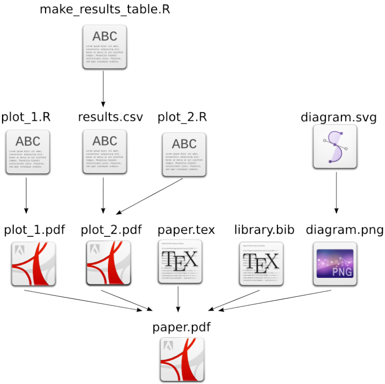
libarmadillo.so

Linking
`g++ ...`

main

```
g++ -c regression.cpp -o regression.o
g++ -c main.cpp -o main.o
g++ main.o regression.o -larmadillo -o main
```

A LaTeX project



```
Rscript --no-slave plot_1.R
Rscript --no-slave make_results_table.R
Rscript --no-slave plot_2.R
inkscape diagram.svg --export-png diagram.png
latexmk -pdf paper.tex
```

Make

- (GNU) Make is a dependency management tool used to build and maintain projects
- It has been around since the 1970s
- It is by far the most popular tool for building software on Unix-like systems
- It is extremely portable: Linux operating systems come with Make installed

Basic usage

- To use Make in your project, you first must create a text file called *Makefile* in your project directories
- This *Makefile* will contain the dependency structure of your project, and instructions on how to make each component
- To build a target you simply run make with the name of the target, for example:

```
make paper.pdf
```

- On its invocation, Make parses the Makefile for the project dependency structure and runs the required commands to build the project
- Make is economical: it only rebuilds those components that need rebuilding

Writing Makefiles

- A Makefile consists of a list of rules
- Each rule specifies a *target*, *prerequisites*, and build commands for building the target from the prerequisites

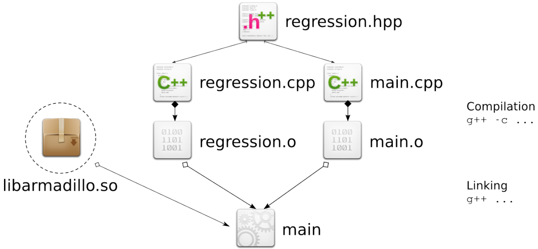
```
target: prereq1 prereq2 ...
      # build commands
```

An example Makefile

```
main: main.o regression.o
    g++ main.o regression.o -larmadillo -o main

main.o: main.cpp regression.hpp
    g++ -c main.cpp -o main.o

regression.o: regression.cpp regression.hpp
    g++ -c regresion.cpp -o regression.o
```



regression.hpp

regression.cpp

main.cpp

regression.o

main.o

libarmadillo.so

main

Compilation
`g++ -c ...`

Linking
`g++ ...`

- When Make is run the following things happen:
  1. Make looks for a rule to build the specified target
  2. Make checks whether the target needs building or rebuilding
  3. A target needs rebuilding if it doesn't already exist, it is older than one or more of its prerequisites, or its prerequisites need rebuilding - make works recursively

Using variables

- Variables can be set and used in similar manner to how they are used in BASH scripts
- Variables are often used to store:
    - A set of compiler flags commonly used in a Makefile
    - Specify include/library directories required for compilation/linkage
    - Lists of libraries required for linkage

```
CPLEX_DIR=/opt/ibm/ILOG/CPLEX_Studio126
LIB_DIR=-L $(CPLEX_DIR)/cplex/lib/x86-64_linux/static_pic -L $(CPLEX_DIR)/concert/lib/x86-64_
INC_DIR=-I $(CPLEX_DIR)/cplex/include -I $(CPLEX_DIR)/concert/include
LIBS=-lilocplex -lconcert -lcplex -lm -lpthread
FLAGS=-DIL_STD -std=c++11

network: network.cpp
	g++ $(INC_DIR) $(FLAGS) $(LIB_DIR) network.cpp $(LIBS) -o network
```

Automatic variables

- Automatic variables are in-rule variables which make build commands easier write and maintain

Automatic Variable	Meaning
$@	the target
$<	the first prerequisite
$ˆ	all prerequisites
$?	only prerequisites which have changed

```
main: main.o regression.hpp
    g++ main.o regression.o -larmadillo -o main

main.o: main.hpp regression.hpp
    g++ -c main.cpp -o main.o

regression.o: regression.cpp regression.hpp
    g++ -c regresion.cpp -o regression.o
```

Automatic variables

- Automatic variables are in-rule variables which make build commands easier write and maintain

Automatic Variable	Meaning
$@	the target
$<	the first prerequisite
$^	all prerequisites
$?	only prerequisites which have changed

```
main: main.o regression.hpp
    g++ $^ -larmadillo -o $@

main.o: main.cpp regression.hpp
    g++ -c $< -o $@

regression.o: regression.cpp regression.hpp
    g++ -c $< -o $@
```

Pattern rules

- Often a set of targets will follow the exact same build pattern
- In this case the user can specify a *pattern rule*
- In a pattern rule, the *base* of a file is represented by a %
- Pattern rules help scale Makefile to large projects

```
Simulation_Base.o: Simulation_Base.cpp Simulation_Base.hpp
    g++ -c $< -o $@

Simulation_Serial.o: Simulation_Serial.cpp Simulation_Serial.hpp
    g++ -c $< -o $@

Simulation_Parallel.o: Simulation_Parallel.cpp Simulation_Parallel.hpp
    g++ -c $< -o $@

libSimulation.a: Simulation_Base.o Simulation_Serial.o SimulationParallel.o
    ar rv $@ $<

main: main.cpp libSimulation.a
    g++ $^ -o $<
```

Pattern rules

- Often a set of targets will follow the exact same build pattern
- In this case the user can specify a *pattern rule*
- In a pattern rule, the *base* of a file is represented by a %
- Pattern rules help scale Makefile to large projects

```
%.o: %.cpp %.hpp
    g++ -c $< -o $@

libSimulation.a: Simulation_Base.o Simulation_Serial.o SimulationParallel.o
    ar rv $@ $?

main: main.cpp libSimulation.a
    g++ $^ -o $<
```

Utilities: clean

- Beside building projects, rules can be used to create general utilities
- A common utility is *clean* which is used to remove all files generated in the build process

```
.PHONY:
clean:
    rm -f *.o *.a *~ main
```

A clean command for a C++ project

Utilities: dist

- Another common utility is a rule to package a project up for distribution
- Typically projects are distributed in tarballs or zip files

```
PRJ=C++_project

.PHONY:
dist: clean
	(cd ..; tar cvzf $(PRJ).tgz $(PRJ))
```

A dist command for a C++ project

- Note that utilities like `clean` and `dist` should be declared `.PHONY` so that make does not search for files of the same name

A more advanced Makefile

```
PRJ=latex_project
SVG_IMAGES = diagram.png

paper.pdf: paper.tex $(SVG_IMAGES) plot_1.pdf plot_2.pdf
  latexmk $< -pdf $@

$(SVG_IMAGES):
%.png: %.svg
  inkscape $<  --export-png $@

plot_1.pdf: plot_1.R
  Rscript --slave $< $@

plot_2.pdf: plot_2.R results.csv
  Rscript --slave $< $@

results.csv: make_results_table.R
  Rscript --slave $<

.PHONY:
clean:
  latexmk -C
  rm -f *.pdf *.csv *~ *.bbl *.png

.PHONY:
dist: clean
  (cd ..; tar cvzf $(PRJ).tgz $(PRJ))
```

Why use Makefiles?

- Facilitates build process by running all required build commands automatically
- Users of your project do not need to understand, or be given detailed instructions on how to build your project - they simpley run make
- Make can be used to provide useful utilities

Why not just use an IDE for C/C++ projects?

- Make is more portable (and readable) than IDE project files
- Make is more flexible
- Writing Makefiles gives you a better understanding of the compilation process

Getting more help

- Ask someone who uses it (Emma, Christian, Terry, Jamie, etc.)
- Use Google
- Download and use the official Make texinfo page:

```
sudo apt-get install make-doc
info make
```

- Note that this can be used inside Emacs

Any questions?