Using Make for Software Build Management

Benjamin S. Skrainka

University of Chicago
The Harris School of Public Policy
skrainka@uchicago.edu

May 23, 2012

Objectives

This talk's objectives:

- State Make's key features
- Use make to manage building software and documents of moderate complexity:
- List advanced features

Plan for this Talk

Getting Started

Basics

Typical Usage

Advanced Usage

Common Build Frustrations

Building software is frustrating:

- Specifying all of the compiler and linker options is impossible to remember and type correctly
- Using a script is unmaintainable:
  - Different options for different platforms
  - Only need to rebuild part of code which changed
  - Hard to read
  - Doesn't understand dependencies
- Difficult to build on a new platform
- Need to perform the same or similar tasks over and over

Benefits of Make

Make automates repetitive tasks such as building software, updating a paper, or running a data cleaning pipeline:

- Manage build process in platform-independent manner $\Rightarrow$ easy to port
- Easy to abstract build process $\Rightarrow$ easier to understand and maintain
- Easy to build different targets: e.g., optimized software, debug version, ...
- Rebuild only part of software which changed $\Rightarrow$ decreased build times
- Understands dependencies between different parts of what you are building
- Flexible: can be used with any programming language, LaTeX, websites, etc.
- Make models build as a directed acyclic graph

$\Rightarrow$ automate everything you can to avoid errors!

Building an Application with Make

Run `make` from command line with appropriate options and targets:

- By default, builds first target listed in Makefile
- Can specify arbitrary Makefile with -f option
- Can specify command line macros
- Typically support several common targets:
  - Optimized application
  - Version with debug symbols
  - clean – to delete all built files and start fresh

Flavors of Make

Most platforms support some form of make:

- GNU make (sometimes named gmake)
- MSVC nmake
- Unix make

Common options have subtle differences:

- Unless on Windows, use GNU make

imake and cmake are programs to generate Makefiles. For our purposes, they are probably more trouble than they are worth.

Disclaimer

At times, make will drive you insane:

- make is basically a programming language and is weird and quirky
- Designed by a summer intern at AT&T Bell Labs in the Pleistocene epoch
- No debugger (remake?) :-(
- Unfortunately, there is nothing better:
  - Live with it
  - Make is ubiquitous

Getting Started

First Steps

Let's start by discussing:

- Installation
- Configuration
- Getting Help

Installation

Make should already be installed on your computer:

- Linux/Unix: included with standard Unix utilities
- OS/X: included with Xcode
- Windows: nmake is included with VisualC++
- Can always download and build gmake via package manager (Synaptic on Ubuntu, apt-get on Debian, MacPorts on OS/X, ...)

Documentation

You have several documentation options:

- man make (a.k.a. RTFM)
- info make (a.k.a. RTFM)
- Online GNU make manual
- Managing Projects with GNU Make – do not purchase the pre-2009 version
- Software Carpentry's make lecture

Make Basics

Use make to build software:

- Write a Makefile describing what you want to build:
  - Create a *target* for each item you need to build
  - Specify relationships between objects (files, .o's, libraries, etc.)
  - Uses unique language to specify rules, actions, and macros
- Invoke make to build the desired target according to a Makefile:
  - By default make looks for `GNUMakefile`, `makefile`, and `Makefile` (in this order)
  - Can specify a arbitrary makefile with -f: e.g., `make -f myproj.mak`
  - Tip: always name your makefile `Makefile` so it is easy to see when you list a directory

First Makefile

Here is a Makefile to build a simple application:

```
# First Makefile - build PredPrey.cpp from homework 1.
PredPrey : PredPrey.cpp
  g++ -Wall -pedantic -Wextra -O2 PredPrey.cpp \
    -o PredPrey
```

This Makefile consists of:

- A comment which starts with '#'
- A rule which consists of *target : prerequisite1 prerequisite2 ...*
  - The target will be built if it does not exist or is older than any prerequisite
  - Target is built using actions specified on the lines after the rule
  - **Must indent actions with a tab!!!**

First Makefile Output

Here is the output

```
$ make
g++ -Wall -pedantic -Wextra -O2 PredPrey.cpp \
  -o PredPrey
$ make
make: 'PredPrey' is up to date.
```

- First invocation builds `PredPrey` because it does not exist
- Second invocation does nothing because `PredPrey` is newer than `PredPrey.cpp`

```
$touch PredPrey.cpp # make PredPrey.cpp newer
$ make
g++ -Wall -pedantic -Wextra -O2 PredPrey.cpp \
  -o PredPrey
```

Tabs

You must indent every action with a tab:

- This will drive you crazy
- If you forget, make will abort with a cryptic error message.
- Remember to unset automatic conversion of tabs to spaces in your editor → in vi, `:set noet`
- tabs are evil. . .

```
$ make    # run Makefile with spaces instead of tabs
Makefile2:4: *** missing separator.  Stop.
```

Basics

Basics

Makefiles consist of three parts:

1. Header: comments describing what the makefile does
2. Definition: macros which abstract build process:
   - Which commands to use (e.g., which compiler to use – g++ vs. g++-mp-4.7 vs. clang++ vs ...)
   - Compiler and linker options
   - Files to build
   - Locations of libraries and header files
3. Targets
4. Rules

By default, make will build the first target it finds in the Makefile.

Header

The header is simply the documentation at the start of your makefile:

- What the makefile builds
- Targets the makefile supports
- Options, such as command line macros, which configure the build

Example Header

```
#
# Makefile - builds housing model
#
# To build a target, enter:
#
#   make <target>
#
# Targets:
#
#   all    - builds everything
#   clean - deletes all .o, .a, binaries, etc.
#   exe    - optimized application
#   dbg    - application with debug symbols
#
```

```
#
# Options:
#
#  USE_MPI  - defaults to 0 (no MPI).  If defined to 1,
#             build application with MPI libraries for
#             execution on multiple processors
#
#  USE_DIAG - defaults to 0 (no diagnostic printing).
#             Set to 1 on the command line to enable
#             diagnostic printing.
#
```

```
#
# Example:
#
#   To build an optimized application with MPI:
#
#     make USE_MPI=1 exe
#
#   To build a optimized application for one
#   processor (sequential):
#
#     make exe
#
#   To build a sequential debug application:
#
#     make dbg
#
```

```
#
# modification history
# --------------------
# 12mar08 bss removed FC=... from make for
#             libraries because libraries
#             weren't building correctly
#             with mpiifort on IFS HPC.
# 26feb08 bss changed to use same compiler
#             for libraries and application.
# ...
```

Definitions

Make supports macros, which are like environment variables, whose contents are substituted when evaluated:

- Syntax: VAR_NAME = VALUE

```
CXXFLAGS = -Wall -Wextra -m64
```

- Macros are not resolved fully until substituted into a rule or action
- Evaluate using $(VAR_NAME)
  - Do not forgot parentheses
  - $VAR_NAME evaluates as the contents of $V followed by 'AR_NAME'
- Can append data using +=

```
CXXFLAGS += -O2 -DEIGEN_NO_DEBUG
CXXFLAGS += -I/home/skrainka/tools/include
```

Common Definitions

Some common definitions are:

- Commands for building software and performing actions
- Compiler and linker flags
- Location of include files
- Target application/objects to build

Put common definitions in a file and include it:

```
# make's include is like #include in C++
include /home/skrainka/tools/make/stddefs.inc
```

This provides portability:

- Just need to change macros to get code to build on a new platform
- Build process/logic remains the same!

Example Definitions

```
# Define commands for this platform
RM  = rm -f          # Delete command
AR  = ar cru         # Create archive
RANLIB = ranlib      # Index archive
CXX = g++            # C++ Compiler
LD  = g++            # Linker

# Define compiler and linker flags
CXXFLAGS = -Wall -Wextra -O2 -DEIGEN_NO_DEBUG \
           $(INCLUDES)
LDFLAGS  = -Wall -Wextra -O2

# Setup
IPOPT_DIR = /home/skrainka/public/sw
EIGEN_DIR = /home/skrainka/public/sw/include
INCLUDES = -I$(EIGEN_DIR) \
           -I$(IPOPT_DIR)/include/coin
```

```
# Link line for shared libraries
LIBS=-L$(IPOPT_DIR)/lib          \
      -lipopt                    \
      -lcoinblas                 \
      -lcoinlapack               \
      -lcoinmetis                \
      -lcoinmumps                \
      -rdynamic -ldl -lpthread

# Application files
TGT  = logit          # application to build
OBJS = logitDriver.o \
       logitLib.o    \
       logitNLP.o    \
       solverLib.o
DBGS = $(patsubst %.o, %.d, $(OBJS) )
```

Targets

Common make targets include:

- default: build default object (usually equivalent to exe target)
- all: build everything
- exe or $(TGT): build optimized executable
- dbg: build executable with debug symbols
- doc: build documentation
- clean: delete all derived files
- test: run (unit) tests to verify code works correctly
- install: install code
- pdf: build pdf from LaTeX
- $\Rightarrow$ choose a sensible name based on what you are building

Rules

A rule specifies how to build a target and the resources it depends on. The syntax is:

```
target : [0 or more prerequisites]
  [ 0 or more lines of commands to build the target]
```

- Note: a prerequisite will be either a target of another rule (e.g., a .o) or a precious resource (e.g., a .cpp or .hpp)
- Warning: You must start the actions lines with a tab!!!

Rules ⇒ DAG

Make analyzes rules to build a directed acyclic graph of all dependencies:

- Based on timestamps
- Determines what make needs to build
- Decreases build times because make only builds what is necessary

Actions are any valid bash commands – so you can use sed, awk, if, for, etc. as well as standard compiler commands

Example Rule

Here is a simple rule to build Homework 1:

```
default : PredPrey

PredPrey : PredPrey.cpp
  g++ PredPrey.cpp -o PredPrey -O2 -m64 \
    -Wall -pedantic -Wextra

clean :
  rm -f PredPrey PredPrey.o
```

Debugging Tips

Makefiles are usually a nightmare to debug:

- No debugger
- Weird syntax
- Infernal tabs
- Can use `make -n` to perform a dry run (i.e. see how the commands execute without actually running them)
- Add echo to the front of each action to print out the action or test your understanding of some aspect of Make
- Change access and modification times of an object to current time with

  ```
  touch myPrerequisite.cpp
  ```

  to force make to rebuild a target.

Typical Usage

Basics

But, make allows us to create much more sophisticated and maintainable Makefiles to manage a build:

- Abstract out all duplicate information using automatic variables and macros – i.e. no redundant information or cut & paste which is error prone (a.k.a. evil evil evil)
- Create platform-independent rules and actions
- Use patterns to specify similar rules and actions once

All of these features ease maintenance, improve portability, and, thus, increase productivity

Macros

Use macros sensibly:

- For all commands used in build process
- For build options, such as compiler and linker flags, locations of directories
- For files to build, such as .o's and the final application

Automatic Variables

Make uses special built-in *automatic variables* as short-hand for names of targets and prerequisites:

VARIABLE	RESOURCE
$@	target of the current rule
$^	All prerequisites of current rule
$<	First prerequisite in the list
$?	All out of date prerequisites

Note: these are impossible to remember unless you write makefiles everyday ...

Example with Automatic Variables

Here is a simple rule to build Homework 1:

```
TGT = PredPrey
default : $(TGT)

$(TGT) : $(TGT).cpp
  g++ $^ -o $@ -O2 -m64 \
    -Wall -pedantic -Wextra

clean :
  rm -f $(TGT) *.o
```

Pattern Rules

Often in more complex projects you need to build multiple intermediary objects in order to create the final application:

- Example: an application which depends on multiple .cpp files:
  1. First compile each .cpp file into a .o
  2. Then link all the .o's into the final application
- One option, is to specify a rule for each .cpp:
  - But, is error prone and hard to maintain
  - Effectively, this is programming via cut and paste
- Best solution is to use a pattern rule to specify how to build a general class of targets for a corresponding class of objects:
  - A pattern rule is like a template where the individual files are the type T
  - Typically, use a pattern rule to specify how to build individual object files

Using a Pattern Rule

A pattern rule looks like a regular rule, except uses a % as a wildcard in the target:

- Use pattern rule in place of explicit rule:

  ```
  [target-pattern] : [prereq-pattern]
  ```

- Make substitutes what % matches in target pattern for % in prerequisite pattern

```
OBJS = file1.o file2.o file3.o ...
# Rule to build application from multiple files
$(TGT) : $(OBJS)
  $(LD) $^ -o $@ $(LDFLAGS)
# Pattern rule to build .o's
%.o : %.cpp %.hpp
  $(CXX) -c $< -o $@ $(CXXFLAGS)
```

- Can also use syntax

  ```
  [target list] : [target-pat] : [prereq-pat]
  ```

Phony Targets

Some targets do not correspond to actual files – i.e., a concrete object is never built. In this case, specify that the target is phony with .PHONY so make will always build it:

```
.PHONY : clean

clean:
  rm -f $(OBJS) $(DBGS) $(TGT) $(TGT).dbg
```

Advanced Usage

Advanced Usage

GNU make is incredibly powerful and flexible, if cryptic. Common advanced usage includes:

- Invoke make on another Makefile:
  - To descend a directory tree (e.g. build Ipopt)
  - To invoke a platform-specific Makefile
  - To process a Makefile recursively (e..g, to run make to generate dependencies and then build software accordingly)
- Auto-generate file dependencies using g++ `-M`
- include platform specific macro definitions for portability
- Write debug and optimized .o's and executables to Debug and Release subdirectories (MSFT convention)

Portability & Conditional

Often you need to build code on different platforms:

- Abstract build process using macros so it is platform independent
- Define appropriate macros for each platform in a platform-specific include file
- Include the appropriate platform-specific definitions for each platform using
  - `ifeq / else / endif` commands
  - Can determine OS time, time zone, hostname, hostid, etc. using

    ```
    OS_TYPE = $(shell uname)
    ```

    to run the `uname` command and capture the results in a macro named `OS_TYPE`
  - Then test results of shell commands to select correct platform definitions

Portability Example

```
OS_TYPE = $(shell uname)
# Select options based on OS type
ifeq ("$(OS_TYPE)","Linux")
# Linux Configuration
include defs.linux.inc

else
# OS/X Configuration
include defs.mac_osx.inc
endif
```

Pattern Substitution

Often, you need to specify lists of files which differ in a well-defined way, such as different suffixes.

- Make's command `patsubt` allows you to perform pattern substitution
- Syntax: `$(patsubst` *`find-pattern`*`,` *`replacement`*`,` *`input-text`* `)`
- % serves as a wildcard just like the pattern rules
- Example:

```
SRC  = logitDriver.cpp \
       logitLib.cpp    \
       logitNLP.cpp    \
       solverLib.cpp

OBJS = $(patsubst  %.cpp, %.o, $(SRC) )
DBGS = $(patsubst  %.cpp, %.d, $(SRC) )
```

Dependency Trick

Sometimes a file depends on several prerequisites, only some of which need to be compiled or processed as part of the build.

- Example: `summary.dat` depends on a Python script , `stats.py`, which builds it as well as the input data, `data*.dat`.[1]
- To avoid running `stats.py` on itself, we specify the dependency of `summary.dat` on `stats.py` as a separate rule:

```
summary.dat : stats.py
summary.dat : data1.dat data2.dat data3.dat
  stats.py $^ > $@
```

- Now, make will build `summary.dat` if either the script or input file change, which would not have worked if `stats.py` were in the same list of prerequisites as the `data*.dat` files.

[1] This example was stolen from Software Carpentry.

Example Makefile

```
# Makefile to build homework 3.
#
# To build any of the targets, run
#
# $ make <Target>
#
# The targets are:
#
#   default : build default target, which is exe
#   exe     : compile, assemble, and link application
#             into executable. You can run the
#             executable by typing ./logit
#   dbg     : compile, assemble, and link application
#             into executable. Builds with debug
#             symbols. You can run the executable
#             by typing ./logit.dbg
#   clean   : delete all derived files.
```

```
#
# Do not forget to update your LD_LIBRARY_PATH
# variable to point to the location
# where the Ipopt libraries are installed.
#
# You will need to modify some of the paths to
# suit your setup. I have marked these sections
# with XXX to make them easy to search for.
#
```

```
#-----------------------------------------------------------
# Determine if we are running on OS/X or Linux
OS_TYPE = $(shell uname)

# Select options based on OS type
ifeq ("$(OS_TYPE)","Linux")

# Linux Configuration
# Ipopt Installation location XXX
IPOPT_DIR=/home/skrainka/public/sw

# Eigen installation location XXX
EIGEN_DIR=/home/skrainka/public/sw/include

# Compiler Toolchain
CXX = g++
LD  = g++
RM=rm -f
```

```
# Link line for shared libraries
LIBS=-L$(IPOPT_DIR)/lib          \
      -lipopt                    \
      -lcoinblas                 \
      -lcoinlapack               \
      -lcoinmetis                \
      -lcoinmumps                \
      -rdynamic -ldl -lpthread
```

```
else

# OS/X configuration -- assumes MacPorts
# If you are compiling on a Windows/Cygwin machine,
# then you will need to change all these definitions
# as appropriate. XXX

# Ipopt installation location XXX
IPOPT_DIR=/Users/bss/Tools/ipopt/Ipopt-3.10.2/build

# Eigen installation location XXX
EIGEN_DIR=/Users/bss/Tools/Eigen/eigen-3.0.5

# Compiler Toolchain
# Change this to the compiler you are using XXX
CXX = g++-mp-4.5
LD  = g++-mp-4.5
RM=rm -f
```

```
# Link line for shared libraries
LIBS=-L$(IPOPT_DIR)/lib           \
      -lipopt                     \
      -lcoinblas                  \
      -lcoinlapack                \
      -lcoinmetis                 \
      -lcoinmumps                 \
      -ldl -lpthread              \
      -lgfortran
endif
```

```
# Application to build
# Change to the name of your application XXX
TGT = logit

# Change to the list of the object files you want
# built, i.e. the .o's corresponding to the .cpp
# files you need to compile and link together. XXX

OBJS= logitDriver.o logitLib.o logitNLP.o solverLib.o
DBGS = $(patsubst  %.o, %.d, $(OBJS) )

# Note: -pedantic is incompatible with Eigen.
CXXFLAGS=-Wall -Wextra -O2 -m64 \
    -I$(IPOPT_DIR)/include/coin \
    -DEIGEN_NO_DEBUG -I$(EIGEN_DIR)

CXX_DEBUG_FLAGS=-Wall -Wextra -g -m64 \
    -I$(IPOPT_DIR)/include/coin -I$(EIGEN_DIR)
```

```
#-----------------------------------------------------------
# Rules to build targets
default : exe
exe : $(TGT)

# Build Problem 2
$(TGT) : $(OBJS)
  $(LD) $(CXXFLAGS) $^ -o $(TGT) $(LIBS)

dbg : $(DBGS)
  $(LD) $(CXX_DEBUG_FLAGS) $^ -o $(TGT).dbg $(LIBS)

clean :
  $(RM) $(OBJS) $(DBGS) $(TGT) $(TGT).dbg ipopt.out
```

```
#--------------------
# Pattern rules to compile individual files
$(OBJS) : %.o : %.cpp
  $(CXX) $(CXXFLAGS) -c $< -o $@

$(DBGS) : %.d : %.cpp
  $(CXX) $(CXX_DEBUG_FLAGS) -c $< -o $@
```