Makefiles: autotools (automake & autoconf) and CMake

David Miller

23 February 2018

Table of Contents

Plan

What does a makefile do?

Three core parts to a make file

- Variable declarations
- File creation rules
- Phony rules

Variable Declarations

- Typically variables are declared at the top
- These could be standard makefile variables (like shown below) or custom ones like PROG or a list of source files

```
1 CC = gcc
2 CXX = g++
3 CXXFLAGS = -O0 -g
4 LFLAGS = -llapack
```

Listing 1: Sample variable declaration

- **CC** and **CXX** are default variables for the C compiler and C++ compiler
- **CXXFLAGS** (or **CFLAGS**) is similarly the compilation flag for C++ (or C)
- **LFLAGS** is the flag for linking

Object and File Rules

- A makefile works by having a rule (*'all'* by default) which must be met
- Each rule has prerequisites and a method (potentially empty)
- For a required file the prerequisites are typically other files and then some build/link command

```
1 required file: prerequisites
2       method for generating the required file from the
    prerequisites
```

Listing 2: Generic rule format

Rule for compilation

```
1 $(SDIR)/%.o: $(SDIR)/%.cpp
2       $(CC) $(CFLAGS) -o $@ $<
```

Listing 3: Sample rule for compilation

- **$@** is the target (the .o file in this case)
- **$<** is the first prereq (**$^** can be used for all prereqs)
- **$(VARNAME)** allows you to reference a variable
- **%** is a wildcard character

So this is basically:

```
1 src/jacobi.o: src/jacobi.cpp
2       g++ -O0 -g -o jacobi.o jacobi.cpp
```

Listing 4: More explicit rule for compilation

Phony Rules

- A phony rule is a rule which doesn't actually generate a file
- It can be called using *'make rule-name'*
- By default when make is called it looks for the phony rule *'all'*
- Usually you want *'all'* and *'clean'*

```
1 .PHONY: all clean
2
3 all: $(PROG_NAME)
4
5 clean:
6     rm -f $(SDIR)/*.o $(SDIR)/*~
```

Listing 5: More explicit rule for compilation

Sample Makefile

```
1  CXX=g++
2  CXXFLAGS=-O0 -g
3  LFLAGS=-llapack
4
5  SDIR=src
6  PROG=jacobi.x
7
8  .PHONY: all clean
9
10 all: $(PROG)
11
12 clean:
13     rm -f $(SDIR)/*.o $(SDIR)/*~
14
15 $(SDIR)/%.o: $(SDIR)/%.cpp
16     $(CXX) $(CXXFLAGS) -o $@ $<
17
18 $(PROG): $(SDIR)/jacobi.o  $(SDIR)/householder.o  $(SDIR)/other.o
19     $(CXX) $(CXXFLAGS) -o $@ $(LFLAGS) $^
```

Listing 6: Sample Makefile

Plan

1. makefiles
2. autotools: automake & autoconf
   - autoconf
   - automake
3. CMake

automake & autoconf

- Tools developed to simplify the makefile process
- Help with portability of programs between different systems
- Typically overkill for very small projects
- More powerful control while doing less work for large projects

autoconf Process

- Initialization by running the following (I have a script called *autogen.sh* which does this)
  - `aclocal`
  - `autoconf`
  - `automake -a`
- Run `./configure` which uses *configure.ac* and *Makefile.am* to generate a makefile
- Run `make`
- (optional) Run `make install`

configure.ac

- This file checks for the required settings, header files, and libraries
- You can set the desired language here
- This also tells the system what makefiles you need to create

Sample configure.ac file

```
1  AC_INIT([PHY480 Automake Example], [1.0.0],
2      [mill2723@msu.edu], [phy480-automake-example])
3
4  AM_INIT_AUTOMAKE([subdir-objects foreign])
5
6  AC_PROG_CXX
7  AC_PROG_CXX_C_O
8  AC_LANG(C++)
9  AC_PREREQ([2.59])
10
11 AC_CHECK_HEADERS([cmath] [iostream] [fstream] [iomanip],,
       AC_MSG_ERROR(Standard library headers are required))
12
13 AC_CONFIG_FILES([Makefile])
14 AC_OUTPUT
```

Listing 7: Sample configure.ac file without linking checks

Using automake

- automake is meant to simplify the makefile creation by only requiring you to declare what needs to be built and automake will figure out how to do it
- Thus, for highly customized rules you may need to still declare it separate
- This can be placed at the end of *Makefile.am* or in a separate file *makefile* (Case matters!)

Variables in automake

- Many variables are the same (**CC**, **CXX**)
- Some are passed via an AM prefix (**AM_CFLAGS**, **AM_CXXFLAGS**)
- Use **LDADD** instead of **LFLAGS** or **LDFLAGS** (changes where in the command the linking flag is placed)
- Any library or binary can be given it's own custom flags
  - **name-of-binary_LDADD** or **name-of-library_CFLAGS** for example
- You must declare what libraries or binaries you want to build
  - This is done with **lib_LIBRARIES** and bin_PROGRAMS
- Then you just declare all the source files for each
  - **name-of-program_SOURCES** where any '/' in the path for the binary or library are replaced with '_'

Sample Makefile.am file

```
1  SUFFIXES = .c .h .cpp
2  CLEANFILES = *~ src/*~
3  SUBDIRS = src/UnitTestscpp/JacobiTests src/UnitTestscpp/
       ToeplitzTests
4
5  CC=gcc
6  CXX=g++
7
8  WARN_FLAGS = -Wall -Wextra
9  AM_CXXFLAGS = $(WARN_FLAGS) -O0 -pg -g3 -Isrc
10 LDADD = -larmadillo -llapack -lblas
11
12 bin_PROGRAMS = bin/TridiagToeplitz.x
13 bin_TridiagToeplitz_x_SOURCES = src/TridiagToeplitz.cpp
```

Listing 8: Sample Makefile.am file

Mixing makefiles and automake

- You can declare subdirs which you want built (but don't need to generate a Makefile for) using the **SUBDIRS** variable
- The corresponding directories should have an existing makefile
- This makefile needs at minimum the *'all'* and *'clean'* rules
- It can be useful to make *'distclean'* rule as well
- Since the Makefile generated by automake has a uppercase 'M' it is smart to use a lowercase 'm' for a custom makefile

Plan

Why use CMake?

- If you really want to get fancy, `cmake` can be used to setup and build programs and libraries
- Only uses a single file to declare all sources, binaries, libraries, etc.
- Keeps build objects separate from source directories
- Can use CTest to run tests after compilation (I won't be talking much about this)
- More modern feel and custom tools compared to autotools (automake and autoconf)

Using CMake

- Single file (at least per directory) which defines everything: *'CMakeLists.txt'*
- Reads more like a scripting language than a makefile

```
1 set (CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -O3 -g")
2
3 add_executable("Tridiag.x" ${SOURCES})
4
5 target_link_library("Tridiag.x" lapack)
```

Listing 9: Some sample CMake commands

- Similar general flow as autoconf & automake
  1. Declare the project
  2. Set the compiler flags
  3. Check for header files, libraries
  4. Declare source files and build objects (binaries and libraries)
  5. Link external libraries
- Offers much more extensive scripting and external build objects (although sometimes this can be more work)

Sample CMakeLists.txt file

```
1  cmake_minimum_required (VERSION 2.8)
2  set (CMAKE_CXX_COMPILER "g++")
3  project (PHY480_CMAKE_EXAMPLE LANGUAGES CXX Fortran C)
4  set (CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -O0 -g")
5
6  include (CheckIncludeFileCXX)
7  CHECK_INCLUDE_FILE_CXX (armadillo   HAVE_ARMADILLO)
8
9  include (FindLAPACK)
10 include (FindBLAS)
11 find_library (ARMA_LIBRARIES  armadillo)
12
13 set (SOURCES src/TridiagToeplitz.cpp)
14 set (CMAKE_RUNTIME_OUTPUT_DIRECTORY bin)
15 add_executable ("Tridiag.x" ${SOURCES})
16
17 target_link_libraries ("Tridiag.x"
18     ${ARMA_LIBRARIES} ${LAPACK_LIBRARIES}  ${BLAS_LIBRARIES})
```

Listing 10: Sample CMakeLists.txt file