MAE 5032 High Performance Computing: Methods and Practices

Lecture 7: Makefile and CMake

Ju Liu
Department of Mechanics and Aerospace Engineering
liuj36@sustech.edu.cn

1. Makefile

Reference

https://www.gnu.org/software/make/manual/make.html

GNU make

Short Table of Contents

Makefile

- Scientific codes tend towards the small end
  - Dozens to hundreds of files
  - O(10K) to O(100K) lines of code
  - Single developer or small team

- Code management
  - Recompiing whole code is prohibitive
  - Assume each developer has all source files

- make: project management tool
  - generate targeted at codes with multiple source files, library dependencies, etc.
  - strives to recompile only things that have changed
  - Available on Linux, mac, and windows
  - prerequisite for other build tools (Cmake, autoconf, etc.)
  - Suitable for light-weight projects
  - http://savannah.gnu.org/projects/make/

Make

- Make works by reading a Makefile which describes
  - files to be created
  - their dependencies
  - instructions to create the specific files

- Makefile describes a DAG (Directed Acyclic Graph) of the dependencies
  - Make a file (“parent”) requires making other files (“children”), for instance library from object files
  - Also: if a child has changed, its parents needs to be updated

- make works by the dependency graph building files until the goal file is up-to-date
  - only builds files whose dependencies are newer than the goal file itself

Basic usage

- In the directory that contains your source files
  - create a file named Makefile or makefile
  - uppercase is preferred so that it comes near the top of a directory listing, but this is not required
  - put the instructions for building your code in the Makefile

- Type make to build your program, i.e.,

```
make
```

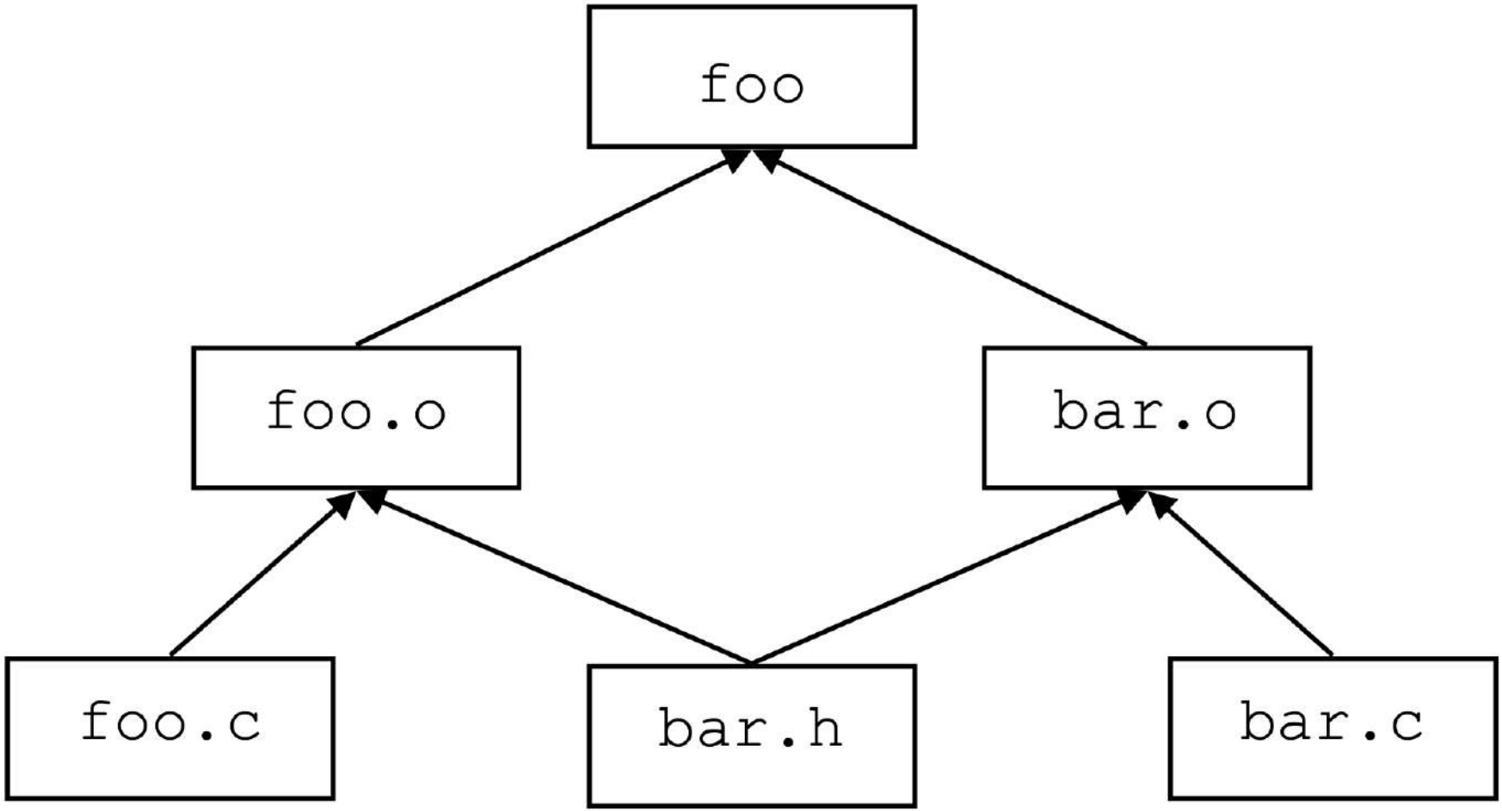
Simple example

```c
#include "bar.h"
int c=3;
int d=4;

int main()
{
   int a=2;
   return (bar(a*c*d));
}
```

```c
#include "bar.h"
int bar(int a)
{
   int b=10;
   return (b*a);
}
```

```c
int bar(int);
```

Directed Acyclic Graph



Example Makefile

- In this example:
  - bar.c + bar.h -> bar.o
  - foo.c + bar.h -> foo.o
  - foo.o + bar.o -> foo (the executable)

- We just described a flatten version of the DAG

- To build this without make, you may execute

```
gcc -c foo.c
gcc -c bar.c
gcc -o foo foo.o bar.o
```

Example Makefile

```
foo: foo.o bar.o
  gcc -o foo foo.o bar.o

foo.o: foo.c bar.h
  gcc -c foo.c

bar.o: bar.c bar.h
  gcc -c bar.c
```

Basic Makefile Rules

```
target: prerequisite
[TAB]command
[TAB]command
```

- Targets are the files to be created/updated

```
make target
```

- Prerequisites are the files which must be up-to-date before the target can be updated.
- Commands are shell scripts used to update the target
- make only executes the command if the target is out-of-date, meaning either it does not exist or its modification is older than any of its dependencies.
- prerequisite can be empty and the command will be executed when it is first invoked to build the targets.

```
clean:
    rm *.o
type make clean
```

More syntax

- make generally assumes that a line describes a target and its dependencies unless the line starts with a TAB.

- Lines that begin with a TAB are considered commands belonging to the most recent rule definition (do not use spaces to replace TAB!)
  - ➢ each line is a separate command invoked in a shell
  - ➢ unless you use `\` shell continuation to tell make otherwise

- If make gets confused, it stops and prints an error message like

```
localhost$ make
Makefile:14: *** mising separator. Stop.
```

- Most editors (vi, emacs, etc.) can tell when you are editing a Makefile and know to use an actual TAB character than than expanding it to SPACES.

Running make

```
-> make
gcc -c foo.c
gcc -c bar.c
gcc -o foo foo.o bar.o
```

- Compiles foo.c and bar.c to foo.o and bar.o
- Links foo.o and bar.o to foo, the executable
- Echos each command as it goes
  - prefix the command with an `@` to suppress this

A slightly improved example

```
CC = gcc
foo: foo.o bar.o
  $(CC) -o $@ $^

foo.o: foo.c bar.h
  $(CC) -c $<

bar.o: bar.c bar.h
  $(CC) -c $<
```

More syntax

- Assigns a **variable** with the compiler name
  - ➢ used in the compile and linking commands
  - ➢ different syntax than the shell
  - ➢ to get to a shell variable, you must escape the `$` by using `$$`

```
library.a: foo.o bar.o
        for f in $^; do \
            ar rc $@ $$f ; done
```

- Use some **automatic variables**
  - **$@** is the target of the current rule
  - **$^** is all the prerequisites of the current rule
  - **$<** is the first prerequisite of the current rule
  - https://www.gnu.org/software/make/manual/html_node/Automatic-Variables.html

More complex Makefile

```
# Files
EXEC := foo
SRC  := $(wildcard *.c)
OBJ  := $(patsubst %.c,%.o,$(SRC))

# Options
CC       := gcc
CFLAGS   := -O3
LDFLAGS  := -L/usr/lib
LDLIBS   := -lm

# Rules
$(EXEC): $(OBJ)
  $(CC) $(LDFLAGS) $(LDLIBS) -o $@ $^

%.o: %.c
  $(CC) $(CFLAGS) -c $<

foo.o bar.o: bar.h

# Useful phony targets
.PHONY: clobber clean neat echo
clobber: clean
  $(RM) $(EXEC)
clean: neat
  $(RM) $(OBJ)
neat:
  $(RM) *~ .*~
echo:
  @echo $(OBJ)
```

- Good for lots of source files
- Puts the frequently modified parts at the top
- Define some convenience targets
- Uses some make built-in features

Comments

```
# Files
EXEC := foo
SRC  := $(wildcard *.c)
OBJ  := $(patsubst %.c,%.o,$(SRC))

# Options
CC       := gcc
CFLAGS   := -O3
LDFLAGS  := -L/usr/lib
LDLIBS   := -lm

# Rules
$(EXEC): $(OBJ)
  $(CC) $(LDFLAGS) $(LDLIBS) -o $@ $^

%.o: %.c
  $(CC) $(CFLAGS) -c $<

foo.o bar.o: bar.h

# Useful phony targets
.PHONY: clobber clean neat echo
clobber: clean
  $(RM) $(EXEC)
clean: neat
  $(RM) $(OBJ)
neat:
  $(RM) *~ .*~
echo:
  @echo $(OBJ)
```

Files

- `#' anywhere on a line denotes the start of a comment which continues until the end of the line (just like in shell scripts)

- If you need to use a `#' in your script somewhere, you must escape it:

```
echo:
    echo foo \# bar
```

Variables

```
# Files
EXEC := foo
SRC  := $(wildcard *.c)
OBJ  := $(patsubst %.c,%.o,$(SRC))

# Options
CC       := gcc
CFLAGS   := -O3
LDFLAGS  := -L/usr/lib
LDLIBS   := -lm

# Rules
$(EXEC): $(OBJ)
  $(CC) $(LDFLAGS) $(LDLIBS) -o $@ $^

%.o: %.c
  $(CC) $(CFLAGS) -c $<

foo.o bar.o: bar.h

# Useful phony targets
.PHONY: clobber clean neat echo
clobber: clean
  $(RM) $(EXEC)
clean: neat
  $(RM) $(OBJ)
neat:
  $(RM) *~ .*~
echo:
  @echo $(OBJ)
```

EXEC := foo

- Defines the variable EXEC with the value foo, i.e. the name of the program we want to build.
  - there is nothing special about the name EXEC
  - there are many special make variables
    - predefined values

- := immediately evaluates the RHS expression and assigns its value to the LHS
  - almost always the better choice
  - unique to GNU make, so not portable

Variables

- = assigns the unevaluated RHS to the LHS
  - the expression now stored in the variable is evaluated anew every time the variable is used
  - need to take care that you get what you are expecting

```
bar = World
foo := $(bar)
bar = Hello
all:
  @echo $(foo)
```

- SPACEs before and after the assignment operators are ignored
  - However, SPACEs after the value are kept
  - be careful not to get a variable whose value is: fooSPACE

Variables

- Variables can be defined in three places
  - in the Makefile, like the example above
  - in your shell environment: `export FOO="foo"`
  - on the command line `make FOO="foo"`
- All variables used the same way: **$(FOO)**
- Precedence: command line, then makefile, then environment

Variables used by implicit rules

```
$(CC) -c $(CFLAGS)
```

- Some predefined variables
  - CC : compiler for C codes; default ‘cc’
  - CXX: compiler for C++ codes; default `g++’
  - CFLAGS : extra flags to give the C compiler; default empty string
  - FFLAGS : extra flags to give the Fortran compiler; default empty string
  - LDFLAGS : extra flags to give the linker; default empty string
  - LDLIBS : Library flags or names given to compilers ; default empty string
- Use `make -p` to see the complete list of predefined variables

Functions - Wildcard

```
# Files
EXEC := foo
SRC  := $(wildcard *.c)
OBJ  := $(patsubst %.c,%.o,$(SRC))

# Options
CC      := gcc
CFLAGS  := -O3
LDFLAGS := -L/usr/lib
LDLIBS  := -lm

# Rules
$(EXEC): $(OBJ)
  $(CC) $(LDFLAGS) $(LDLIBS) -o $@ $^

%.o: %.c
  $(CC) $(CFLAGS) -c $<

foo.o bar.o: bar.h

# Useful phony targets
.PHONY: clobber clean neat echo
clobber: clean
  $(RM) $(EXEC)
clean: neat
  $(RM) $(OBJ)
neat:
  $(RM) *~ .*~
echo:
  @echo $(OBJ)
```

SRC := $(wildcard *.c)

- Assigns to the variable SRC all of the files in the current directory matching the glob pattern `*.c'.
- See https://man7.org/linux/man-pages/man7/glob.7.html
- Evaluates the wildcard function now
- Sets SRC to “bar.c foo.c”

Functions – Pattern substitution

```
# Files
EXEC := foo
SRC  := $(wildcard *.c)
OBJ  := $(patsubst %.c,%.o,$(SRC))

# Options
CC      := gcc
CFLAGS  := -O3
LDFLAGS := -L/usr/lib
LDLIBS  := -lm

# Rules
$(EXEC): $(OBJ)
  $(CC) $(LDFLAGS) $(LDLIBS) -o $@ $^

%.o: %.c
  $(CC) $(CFLAGS) -c $<

foo.o bar.o: bar.h

# Useful phony targets
.PHONY: clobber clean neat echo
clobber: clean
  $(RM) $(EXEC)
clean: neat
  $(RM) $(OBJ)
neat:
  $(RM) *~ .*~
echo:
  @echo $(OBJ)
```

OBJ := $(patsubst %.c,%.o,$(SRC))

- Changes every space-separated thing in SRC that ends in '.c' to end in '.o'
  - '%' is the pattern matching operator in make
  - foo.o matches '%.o' with the % foo and the remainder .o
  - foo.out would not match the pattern
  - be careful with the spaces around the commas.
- Evaluates immediately
- Uses the value in SRC
- Sets OBJ to "bar.o foo.o"

Main goal rule with variables

```
# Files
EXEC := foo
SRC  := $(wildcard *.c)
OBJ  := $(patsubst %.c,%.o,$(SRC))

# Options
CC      := gcc
CFLAGS  := -O3
LDFLAGS := -L/usr/lib
LDLIBS  := -lm

# Rules
$(EXEC): $(OBJ)
  $(CC) $(LDFLAGS) $(LDLIBS) -o $@ $^

%.o: %.c
  $(CC) $(CFLAGS) -c $<

foo.o bar.o: bar.h

# Useful phony targets
.PHONY: clobber clean neat echo
clobber: clean
  $(RM) $(EXEC)
clean: neat
  $(RM) $(OBJ)
neat:
  $(RM) *~ .*~
echo:
  @echo $(OBJ)
```

$(EXEC) : $(OBJ)
        $(CC) $(LDFLAGS) $(LDLIBS) –o $@ $^

- Makes the value of EXEC depend on the value of OBJ
  - i.e. foo depends on foo.o and bar.o
- Uses a number of automatic variables to construct the command
  - CC points to the C compiler
  - LDFLAGS and LDLIBS for library paths and library themselves
  - $@ expands to the target of the rule
  - $^ expands to the list of prerequisites

Pattern-based rules

```
%.o: %.c
     $(CC) $(CFLAGS) -c $<
```

- Creates a rule for creating object files from source files with the same name
- Uses the automatic variables again
  - CFLAGS contain any compiler options needed at run time
  - $< expands to the first prerequisite

```
foo.o bar.o: bar.h
```

- Adds a dependency for foo.o and bar.o on bar.h
- Especially useful for C/C++ header files and Fortran includes.

Phony targets

```
.PHONY: clobber clean neat echo
clobber: clean
  $(RM) $(EXEC)
clean: neat
  $(RM) $(OBJ)
neat:
  $(RM) *~ .*~
echo:
  @echo $(OBJ)
```

- Targets listed as the dependencies of .PHONY do not reference files
- Are always treated as out-of-date

`make clobber`

- invokes the commands for neat and clean and then its own command
  - useful for cleaning up a directory
  - clean or neat could also be invoked to clean up less files
- `echo` useful in debugging the Makefile

Running the improved Makefile

```
-> make
gcc -O3 -c bar.c
gcc -O3 -c foo.c
gcc -L/usr/lib  -lm -o foo bar.o foo.o
```

- Now has our extra options
- Order is different

```
-> make bar.o
gcc -O3 -c bar.c
```

- Can be used with any target (of list of targets)
- Useful when you just want to check syntax

Running the improved Makefile

```
-> make clobber
rm -f *~ .*~
rm -f bar.o foo.o
rm -f foo
```

- Good for cleaning and starting over
- Tilde files (*~, .*~) are usually backup files from your text editor (be careful there aren't any spaces in there!)

More improved Makefile

```
# Files
EXEC := foo
SRC  := $(wildcard *.c)
OBJ  := $(patsubst %.c,%.o,$(SRC))

# Options
CC       := gcc
CFLAGS   := -O3
LDFLAGS  := -L/usr/lib
LDLIBS   := -lm

# Rules
$(EXEC): $(OBJ)
  $(LINK.o) $(LDLIBS) -o $@ $^

%.o: %.c
  $(COMPILE.c) $<

foo.o bar.o: bar.h

# Useful phony targets
.PHONY: clobber clean neat echo
clobber: clean
  $(RM) $(EXEC)
clean: neat
  $(RM) $(OBJ)
neat:
  $(RM) *~ .*~
echo:
  @echo $(OBJ)
```

```
$(LINK.o) = $(CC) $(LDFLAGS) $(TARGET_ARCH)

$(COMPILE.c) = $(CC) $(CFLAGS) $(CPPFLAGS)
               $(TARGET_ARCH) -c
```

- Note the use of '=' rahter than ':='
  - allows you to change each of the internal variables (like $(CC)) before you use it
- $(TARGET_ARCH) is empty by default
- `make -n -p | more` to tell make print all rules and variables, so you can search through the output to see the built-in stuff.

More

- make looks for Makefile and makefile in the current directory by default

- You can give a file with any name you like using `make -f mymakefilename`

```
ifdef DEBUG_MODE
     CFLAGS := -g
else
     CFLAGS := -O3
endif
```

- Looks to see if DEBUG_MODE is defined (i.e., has any value)

```
make DEBUG_MODE=ASDFSA
gcc -g -c -o foo.o foo.c ……
```

- Sets CFLAGS accordingly
- Space between ifdef and its arguments is important

More

- `ifeq(arg1,arg2)`
- `ifneq(arg1,arg2)`
- `ifdef variable-name`
- `ifndef variable-name`

```
ifneq ($(DEBUG_MODE),yes)
  CFLAGS  := -O3
else
  CFLAGS := -g
endif
```

```
-> make DEBUG_MODE=ASSDF
gcc -O3   -c bar.c
gcc -O3   -c foo.c
gcc -L/usr/lib   -lm -o foo bar.o foo.o
juliu::Kolmogorov {~/MAE5032/week_07/example-04 }
-> make clobber
rm -f *~ .*~
rm -f bar.o foo.o
rm -f foo
juliu::Kolmogorov {~/MAE5032/week_07/example-04 }
-> make DEBUG_MODE=yes
gcc -g   -c bar.c
gcc -g   -c foo.c
gcc -L/usr/lib   -lm -o foo bar.o foo.o
```

Includes

- make can include pieces of Makefiles to build up the makefile it is working with

```
# Files
EXEC := foo
SRC  := $(wildcard *.c)
OBJ  := $(patsubst %.c,%.o,$(SRC))

# Options
include Makefile_options.inc

# Rules
$(EXEC): $(OBJ)
  $(LINK.o) $(LDLIBS) -o $@ $^

%.o: %.c
  $(COMPILE.c) $<

foo.o bar.o: bar.h
```

Makefile_options.inc:

```
# Options
CC       := gcc
ifneq ($(DEBUG_MODE),yes)
  CFLAGS  := -O3
else
  CFLAGS := -g
endif
LDFLAGS := -L/usr/lib
LDLIBS  := -lm
```

Make libraries

- Recall that to make a *shared* library, we need to **compile** source codes into position-independent code (PIC) using the **`-fpic`** flag;
- And to turn the object file into a shared library, we need to use the **`-shared`** flag.
- Use LD_LIBRARY_PATH to direct the loading path of the shared libraries
- you may pass the loading path at linking time by –Wl,-rpath

Make libraries

- Recall that to make a *shared* library, we need to **compile** source codes into position-independent code (PIC) using the **`-fpic`** flag;

```
gcc -c -Wall -fpic hello.c
```

- And to turn the object file into a shared library, we need to use the **`-shared`** flag.

```
gcc -shared -o libhello.so hello.o
```

- Use LD_LIBRARY_PATH to direct the loading path of the shared libraries; or you may pass the loading path at linking time by –Wl,-rpath

```
gcc calc.c -o calc -L/home/user/lib/ -lmylib
      -Wl,-rpath=/home/user/lib/mylib
```

Make libraries

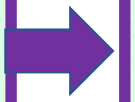
- To make a static library, we need **ar rs**

```
ar rs libfoo.a bar.o foo.o
```

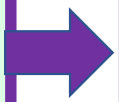
- Compilers search the directoires in the following order
  1. command-line options –I and –L from left to right
  2. directories specified by environment variables such as C_INCLUDE_PATH and LIBRARY_PATH
  3. default system directories

Make libraries

```
.
├── Makefile
├── foo.c
├── include
│   ├── bar.h
│   └── hello.h
└── src
    ├── bar.c
    └── hello.c
```



```
EXEC := foo
LIB  := libbar.a
DIR  := ./lib
SRC  := $(wildcard ./src/*.c)
OBJ  := $(patsubst ./src/%c, %o, $(SRC))
CFLAGS := -I./include
LDFLAGS := -L./lib
LDLIBS := -lbar
CC := gcc
AR := ar rs
RM := rm -rf

$(LIB) : $(OBJ)
  $(AR) $@ $^

%.o : ./src/%.c ./include/%.h
  $(CC) $(CFLAGS) -c $<

$(EXEC) : foo.c
  $(CC) $(CFLAGS) $(LDFLAGS) $(LDLIBS) -o $@ $^

.PHONY: clean install
clean:
  $(RM) $(EXEC) $(OBJ) $(LIB) $(DIR)

install:
  mkdir -p $(DIR)
  mv $(LIB) $(DIR)
```



```
.
├── Makefile
├── bar.o
├── foo
├── foo.c
├── hello.o
├── include
│   ├── bar.h
│   └── hello.h
├── lib
│   └── libbar.a
└── src
    ├── bar.c
    └── hello.c
```

Summary

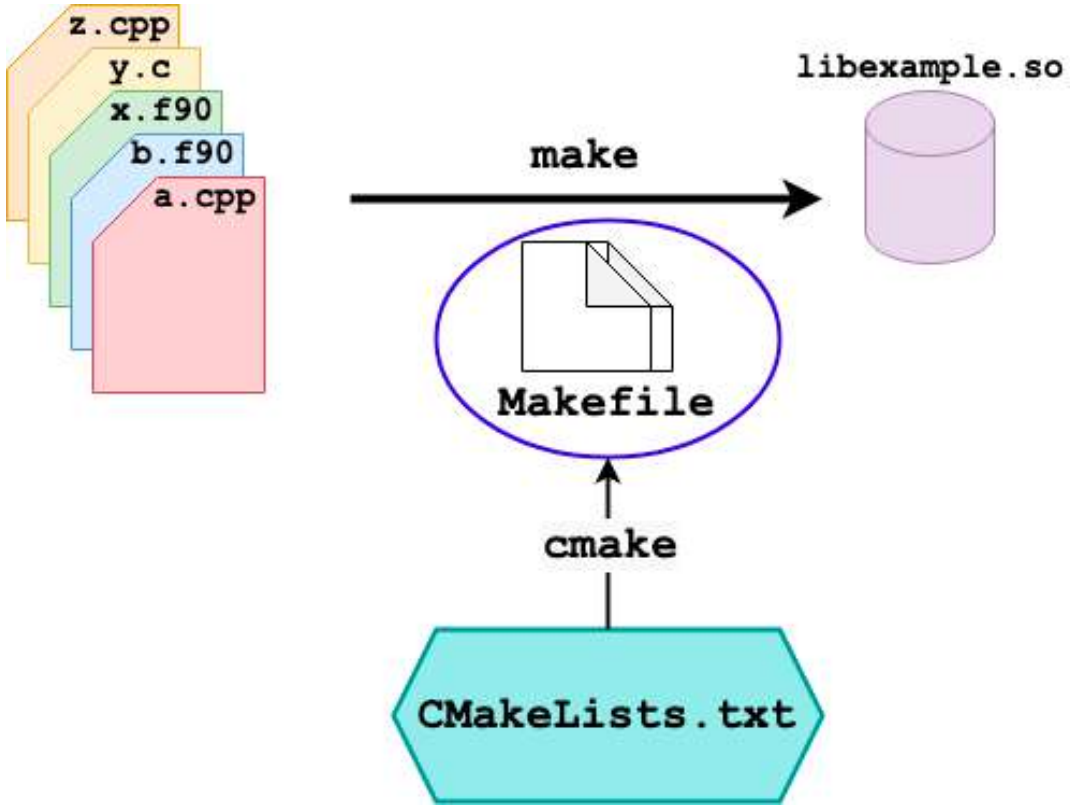
- Make is an automatic tool for the generation of executables from source files.
- https://www.gnu.org/software/make/
- A good tutorial https://makefiletutorial.com/

2. CMake

Introduction

- **CMake** is a cross-platform, open-source build system.
- **CMake** generates native makefiles that can be used in the compiler environment of your choice.
- Idea: you write a single configuration file that Cmake understands, Cmake takes care that it works on all compilers and all platforms.
- Compiler and platform test results are cached in **CMakeCache.txt**.
  - You may modify this file and rerun cmake.
- Cmake also provides graphical cache editors named **ccmake**.

Simple example



- On Linux, the native build system is a collection of **Makefile**. The make build tool uses the makefiles to transform sources to executables and libraries
- CMake abstracts the process of generating the Makefile into a generic domain-specific language
- CMake bring your code closer to being platform- and compiler-agnostic

Simple example

main.c

```c
#include <stdio.h>
int main()
{
  printf("hello world.\n");
  return 0;
}
```

CMakeLists.txt

```
PROJECT (HELLO)
ADD_EXECUTABLE(hello main.c)
```

- CMakeLists.txt file has two commands
  - use upper case here for the commands
  - Upper, lower, and mixed commands are supported by Cmake
  - **CMakeLists.txt file** name is **case sensitive**!
- PROJECT(project_name) sets the name of the project and stores it in the variable PROJECT_NAME
- It also sets two implicit variables
  <PROJECT NAME>_SOURCE_DIR
  <PROJECT NAME>_BINARY_DIR

Simple example

main.c

```c
#include <stdio.h>
int main()
{
  printf("hello world.\n");
  return 0;
}
```

CMakeLists.txt

```
PROJECT (HELLO)
ADD_EXECUTABLE(hello main.c)
```

- PROJECT(project_name) sets the name of the project and stores it in the variable PROJECT_NAME
- It also sets two implicit variables
  <PROJECT NAME>_SOURCE_DIR
  <PROJECT NAME>_BINARY_DIR
- Same are stored in
  PROJECT_SOURCE_DIR
  PROJECT_BINARY_DIR
- You can use PROJECT(app LANGUAGES C CXX) to specify the language.

Simple example

main.c

```
#include <stdio.h>
int main()
{
  printf("hello world.\n");
  return 0;
}
```

CMakeLists.txt

```
PROJECT (HELLO)
ADD_EXECUTABLE(hello main.c)
```

- ADD_EXECUTABLE instructs to generate an executable using the listed source file

ADD_EXECUTABLE(binaryname source1 source2 …)

- You may add “” for main.c. If there are spaces in the source name, you have to use “”.

Simple example

- `cmake [options] <path>`
  - if the specified path contains a **CMakeCache.txt**, it is treated as a build directory where the build system is reconfigured and regenerated.
  - otherwise, the specified path is treated as the source directory and the current directory is treated as build directory.

1. Create a build folder
mkdir build

2. Enter the build folder
cd build

3. Run cmake
cmake ..

4. Run make
make

```
juliu::Kolmogorov {~/MAE5032/week_07/cmake-01/build }
-> cmake ..
-- The C compiler identification is AppleClang 12.0.0.12000032
-- The CXX compiler identification is AppleClang 12.0.0.12000032
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working C compiler: /Applications/Xcode.app/Contents/Dev
eloper/Toolchains/XcodeDefault.xctoolchain/usr/bin/cc - skipped
-- Detecting C compile features
-- Detecting C compile features - done
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: /Applications/Xcode.app/Contents/D
eveloper/Toolchains/XcodeDefault.xctoolchain/usr/bin/c++ - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: /Users/juliu/MAE5032/week_07/cma
ke-01/build
juliu::Kolmogorov {~/MAE5032/week_07/cmake-01/build }
-> make
Scanning dependencies of target hello
[ 50%] Building C object CMakeFiles/hello.dir/main.c.o
[100%] Linking C executable hello
[100%] Built target hello
```

A slightly better one

```
CMAKE_MINIMUM_REQUIRED(VERSION 3.10)
PROJECT (HELLO)
MESSAGE(STATUS "This is BINARY dir " ${HELLO_BINARY_DIR})
MESSAGE(STATUS "This is SOURCE dir " ${HELLO_SOURCE_DIR})
SET(src main.c)
ADD_EXECUTABLE(hello ${src})
```

- VERSION is a special keyword for CMAKE_MINIMUM_REQUIRED
- The version dictates the cmake policies (defining behaviors)
- This makes sure the cmake is backward compatible; upgrading cmake will not break or change anything at all!
- Do not set this older than your compiler.

A slightly better one

```
CMAKE_MINIMUM_REQUIRED(VERSION 3.10)
PROJECT (HELLO)
MESSAGE(STATUS "This is BINARY dir " ${HELLO_BINARY_DIR})
MESSAGE(STATUS "This is SOURCE dir " ${HELLO_SOURCE_DIR})
SET(src main.c)
ADD_EXECUTABLE(hello ${src})
```

- 3.4: the bare minimum. Never set less
- 3.8: C++ meta features, CUDA, etc.
- 3.10: Ubuntu 18.04
- 3.16: Ubuntu 20.04
- 3.18: lots more CUDA support

A slightly better one

```
CMAKE_MINIMUM_REQUIRED(VERSION 3.10)
PROJECT (HELLO)
MESSAGE(STATUS "This is BINARY dir " ${HELLO_BINARY_DIR})
MESSAGE(STATUS "This is SOURCE dir " ${HELLO_SOURCE_DIR})
SET(src main.c)
ADD_EXECUTABLE(hello ${src})
```

- MESSAGE([mode], “message to display”)
  - mode: FATAL_ERROR ; error and stop processing
  - mode: WARNING ; warning and continue processing
  - mode: STATUS ; informative message to display (starting with --)

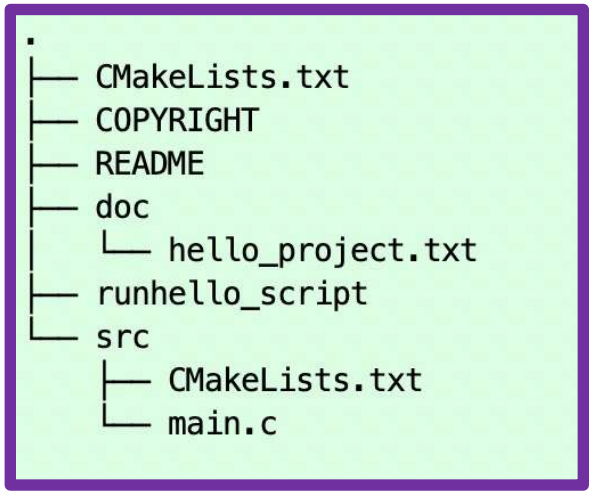
A slightly better one

```
CMAKE_MINIMUM_REQUIRED(VERSION 3.10)
PROJECT (HELLO)
MESSAGE(STATUS "This is BINARY dir " ${HELLO_BINARY_DIR})
MESSAGE(STATUS "This is SOURCE dir " ${HELLO_SOURCE_DIR})
SET(src main.c)
ADD_EXECUTABLE(hello ${src})
```

- SET(MY_VARIABLE “VALUE”)
  - The names of variables are usually all caps, followed by its value.
  - You may access a variable by ${}.
- SET(MY_LIST “one” “two”)
  - You may list variables that are separated by spaces or semicolon.

Setup CMake for a project

- need a src folder for source files.
- need a doc folder for project documents.
- need files COPYRIGHT and README
- need a job script runhello_script



- we will be able to install the targets to a specified destination
- we will be able to install doc folder as well as the COPYRIGHT & README files to a specified destination.

Setup CMake for a project

```
.
├── CMakeLists.txt
├── COPYRIGHT
├── README
├── doc
│   └── hello_project.txt
├── runhello_script
└── src
    ├── CMakeLists.txt
    └── main.c
```

```
CMAKE_MINIMUM_REQUIRED(VERSION 3.10)
PROJECT (HELLO)
ADD_SUBDIRECTORY(src)
INSTALL(FILES COPYRIGHT README DESTINATION doc)
INSTALL(PROGRAMS runhello_script DESTINATION bin)
INSTALL(DIRECTORY doc/ DESTINATION doc)
INSTALL(TARGETS hello DESTINATION bin)
```

```
SET(src main.c)
ADD_EXECUTABLE(hello ${src})
```

Setup CMake for a project

```
CMAKE_MINIMUM_REQUIRED(VERSION 3.10)
PROJECT (HELLO)
ADD_SUBDIRECTORY(src bin)
```

- ADD_SUBDIRECTORY tells CMake that we have an extra directory.
- `bin` tells the location for the compiled targets.
- ADD_SUBDIRECTORY(src) will also work, it will put the compiled targets to folder `src`.

Setup CMake for a project

```
INSTALL(FILES COPYRIGHT README DESTINATION doc)
INSTALL(PROGRAMS runhello_script DESTINATION bin)
INSTALL(DIRECTORY doc/ DESTINATION doc)
INSTALL(TARGETS hello DESTINATION bin)
```

- INSTALL generates installation rules for a project. You will be able to run make install to put the files, binaries, libraries into appropriate locations.
  - FILES specifies regular documentary files;
  - PROGRAMS specifies files with executable permission (e.g. job scripts);
  - DIRECTORY specifies a folder to be installed;
  - TARGETS specifies binaries or libraries to be installed.

- DESTINATION specifies the location for the installed files relative to CMAKE_INSTALL_PREFIX, whose default value is /usr/local

Setup CMake for a project

There are ways to change the value of CMake variables.

1. You may explicitly set its value in CMakeLists.txt;

2. You may run cmake first and edit its value in CMakeCache.txt;

3. You may set its value by using the GUI interface ccmake;

4. You may set its value from command line when running cmake,
   cmake --DCMAKE_INSTALL_PREFIX=….

Setup CMake for a project with libraries

```
.
├── CMakeLists.txt
├── COPYRIGHT
├── README
├── doc
│   └── hello_project.txt
├── runhello_script
└── src
    ├── CMakeLists.txt
    └── main.c
```

```
-> tree
.
├── CMakeLists.txt
├── COPYRIGHT
├── README
├── doc
│   └── hello_project.txt
├── include
│   └── hello.h
├── main.c
├── runhello_script
└── src
    ├── CMakeLists.txt
    └── hello.c
```

Setup CMake for a project with libraries

```
-> tree
.
├── CMakeLists.txt
├── COPYRIGHT
├── README
├── doc
│   └── hello_project.txt
├── include
│   └── hello.h
├── main.c
├── runhello_script
└── src
    ├── CMakeLists.txt
    └── hello.c
```

```
PROJECT(HELLO)
CMAKE_MINIMUM_REQUIRED(VERSION 3.10)

ADD_SUBDIRECTORY(src)
INCLUDE_DIRECTORIES(include)
ADD_EXECUTABLE(driver main.c)
TARGET_LINK_LIBRARIES(driver hello)

INSTALL(FILES COPYRIGHT README DESTINATION doc)
INSTALL(PROGRAMS runhello_script DESTINATION bin)
INSTALL(DIRECTORY doc/ DESTINATION doc)
INSTALL(DIRECTORY include/ DESTINATION include)
INSTALL(TARGETS driver DESTINATION bin)
INSTALL(TARGETS hello DESTINATION lib)
```

```
SET(LIBSRC hello.c)
INCLUDE_DIRECTORIES(../include)
ADD_LIBRARY(hello SHARED ${LIBSRC})
SET_TARGET_PROPERTIES(hello PROPERTIES VERSION 0)
```

Setup CMake for a project with libraries

```
PROJECT(HELLO)
CMAKE_MINIMUM_REQUIRED(VERSION 3.10)

ADD_SUBDIRECTORY(src)
INCLUDE_DIRECTORIES(include)
ADD_EXECUTABLE(driver main.c)
TARGET_LINK_LIBRARIES(driver hello)

INSTALL(FILES COPYRIGHT README DESTINATION doc)
INSTALL(PROGRAMS runhello_script DESTINATION bin)
INSTALL(DIRECTORY doc/ DESTINATION doc)
INSTALL(DIRECTORY include/ DESTINATION include)
INSTALL(TARGETS driver DESTINATION bin)
INSTALL(TARGETS hello DESTINATION lib)
```

- **INCLUDE_DIRECTORIES** tells the CMake to locate the header files.
- **TARGET_LINK_LIBRARIES** tells the Cmake to link to the libraries.

Setup CMake for a project with libraries

```
SET(LIBSRC hello.c)
INCLUDE_DIRECTORIES(../include)
ADD_LIBRARY(hello SHARED ${LIBSRC})
SET_TARGET_PROPERTIES(hello PROPERTIES VERSION 0)
```

- **ADD_LIBRARY** instruct CMake to create a library from the source code.
- You may instruct it to build a static library by replacing SHARED by STATIC.
- **SET_TARGET_PROPERTIES** here instruct the name of the library to have a version number.

Setup CMake for a project with libraries

```
-> make install
[ 50%] Built target hello
[100%] Built target driver
Install the project...
-- Install configuration: ""
-- Installing: /Users/juliu/MAE5032/test/doc/COPYRIGHT
-- Installing: /Users/juliu/MAE5032/test/doc/README
-- Installing: /Users/juliu/MAE5032/test/bin/runhello_script
-- Up-to-date: /Users/juliu/MAE5032/test/doc
-- Installing: /Users/juliu/MAE5032/test/doc/hello_project.
-- Installing: /Users/juliu/MAE5032/test/include
-- Installing: /Users/juliu/MAE5032/test/include/hello.h
-- Installing: /Users/juliu/MAE5032/test/bin/driver
-- Installing: /Users/juliu/MAE5032/test/lib/libhello.0.dyl
-- Installing: /Users/juliu/MAE5032/test/lib/libhello.dylib
```

Some environmental variables

- **CMAKE_BUILD_TYPE**: specifies the build type. Typical values include Debug, Release, RelWithDebInfo, and MinSizeRel.
  - Release: high optimization level, no debug info;
  - Debug: No optimization, asserts enabled;
  - RelWithDebInfo: optimized with debug info;
  - MinSizeRel: same as Release but optimizing for size rather than speed.

- **CMAKE_C_COMPILER**: the full path to the compiler for C. You may replace C by the language name (CXX, Fortran, etc.)

- **CMAKE_C_FLAGS**: the flags for all build types. Initialized from CFLAGS in the shell environmental variable.

References

- Cmake official help
  - https://cmake.org/cmake/help/latest/
- An introduction to Mordern Cmake
  - https://cliutils.gitlab.io/modern-cmake/
- Cmake hands-on workshop
  - https://enccs.github.io/cmake-workshop/