GNU make

How to use GNU make

D.S. Hwang

Department of Software Science
Dankook University

Outline

Direct Compilation

C codes:

- multiply_array.c has function main() and multiplyTwoArrays().
- common.c defines the dynamic memory allocation functions.
- common.h describes the headers of functions in common.c. Those functions will be used in multiply_array.c.

Compilation process:

```
Ex0% gcc -c common.c -I.
Ex0% gcc -c multiply_array.c -I.
Ex0% gcc -o gobhagi common.o multiply_array.o
Ex0% ./gobhagi
```

- The option `-I.` is included so that gcc will look in the current directory for the include file common.h.

GNU make

- GNU `make` provides a simple way to organize code compilation.
- GNU `make` is a tool which controls the generation of executables and other non-source files.
- Check for more details in http://www.gnu.org/software/make/

Make Rules and Targets

- A rule tells `make` how to execute a series of commands in order to build a target file from source files.
- A list of dependencies of the target file includes all files which are used as inputs to the commands in the rule.

```
target: dependencies ...
        commands
        ...
```

Make Example 1

Direct compilation approach causes two troubles.

P1 If you lose the compile command or switch computers you have to retype it from scratch.

P2 If you are only making changes to one .c file, recompiling all of them every time is very time-consuming and inefficient.

The use of `Makefile` or `makefile` provides an efficient way to avoid these downfalls.

```
% cat Makefile
gobhagi: multiply_array.o common.o
        gcc -o gobhagi multiply_array.o common.o

multiply_array.o: multiply_array.c common.h
        gcc -c multiply_array.c -I.

common.o: common.c common.h
        gcc -c common.c -I.
% make
```

Make Example 1

```
% cat Makefile
gobhagi: multiply_array.o common.o
        gcc -o gobhagi multiply_array.o common.o

multiply_array.o: multiply_array.c common.h
        gcc -c multiply_array.c -I.

common.o: common.c common.h
        gcc -c common.c -I.
% make
```

- The rule `gobhagi` needs to be executed if any of dependent files change.
- P1 is solved but the system is still not being efficient in terms of compiling only the latest changes.

Make Example 2

```
% cat Makefile
CC = gcc
CFLAG = -I.

gobhagi: multiply_array.o common.o
        $(CC) -o gobhagi multiply_array.o common.o

multiply_array.o: multiply_array.c common.h
        $(CC) -c multiply_array.c $(CFLAG)

common.o: common.c common.h
        $(CC) -c common.c $(CFLAG)

clean:
        rm -f *.o gobhagi
% make
```

- There are special constants that communicate to make how we want to compile sources and generate the executables.
- The macro `CC` is the C compiler to use, and `CFLAG` is the list of flags to pass to the compilation command.

Make Example 3

The problem of example 2 is

- If you were to make a change to common.h, `make` would not recompile the .c files.
- Need to tell make that all .c files depend on certain .h files.

Some built-in macros are used for brevity.

Macros	Meaning
$@	The name of the current target
$?	The list of dependencies that have changed recently than the current target
$<	The name of the current dependency
$ˆ	A space-separated list of all dependencies without duplications

Make Example 3

```
% cat Makefile
CC = gcc
CFLAG = -I.

gobhagi: multiply_array.o common.o
        $(CC) -o $@ $?

multiply_array.o: multiply_array.c common.h
        $(CC) -c $? $(CFLAG)

common.o: common.c common.h
        $(CC) -c $? $(CFLAG)

clean:
        rm -f *.o gobhagi
% make
```

Make Example 3

```
CC = gcc
CFLAG = -I.
DEPS = common.h

%.o: %.c $(DEPS)
	$(CC) -c -o $@ $< $(CFLAG)

gobhagi: multiply_array.o common.o
	$(CC) -o $@ $?

clean:
	rm -f *.o gobhagi
```

- The macro `DEPS`, which is the set of .h files on which the .c files depend.
- In order to generate the .o file, `make` needs to compile the .c file using the compiler defined in the CC macro.
- The -c flag says to generate the object file.
- The special macros $@ and $< are the left and right sides of the :.

Make Example 4

Make the overall compilation rule more general by using macros.

- All of the include files are listed as part of the macro DEPS.
- All of the object files are listed as part of the macro OBJ.

Make Example 4

```
% cat  Makefile
CC = gcc
CFLAG = -I .
DEPS = common.h
OBJ = multiply_array.o common.o
EXE = gobhagi

%.o: %.c $(DEPS)
	$(CC) -c -o $@ $< $(CFLAG)

$(EXE): $(OBJ)
	$(CC) -o $@ $^

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

Make Example 5

When we want to manage all related files in different directories:

- .h files in an `include` directory
- source codes in a `src` directory
- object files in a `obj` directory
- executable file in a `proj` directory

```
proj
 |- src/*.c
 |- include/*.h
 |- obj/*.o
```

Make Example 5

```
IDIR= include
SDIR= src
ODIR= obj

CC= gcc
CFLAG= -I$(IDIR)
LIB= -lm

_DEPS= common.h
DEPS = $(patsubst %,$(IDIR)/%,$(_DEPS))

_OBJ = multiply_array.o common.o
OBJ = $(patsubst %,$(ODIR)/%,$(_OBJ))

EXE = gobhagi

$(ODIR)/%.o: $(SDIR)/%.c $(DEPS)
	@echo $(DEPS)
	@echo $(OBJ)
	@echo $@
	$(CC) -c -o $@ $< $(CFLAG)

$(EXE): $(OBJ)
	$(CC) -o $@ $^ $(CFLAG) $(LIB)

clean:
	rm -f $(EXE) $(ODIR)/*.o
```

How to execute make tool

- (default file name) makefile or Makefile

```
% make
% make clean
```

- (user defined file name) any file name(ex: run)

```
% make -f run
% make -f run clean
```

References

- http://www.gnu.org/software/make/
- A Simple Makefile Tutorial, http://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/
- Brian W. Kernighan and Dennis M. Ritchie, *C Programming Language*(2nd Edition), Prentice Hall, 1988
- Robert Mecklenburg, *Managing Projects with GNU Make*, O'Reilly Media, 2004
- Andy Oram and Mike Loukides, *Programming with GNU Software*, O'Reilly Media, 1996