Makefiles and You

James Biddle

June 21, 2019

1 Part A

1. Create a file called 'makefile', 'Makefile' or 'GNUmakefile'.
2. Make a target called 'target'.
3. Make it echo some text when run, highlight the fact that all you're doing with the makefile is executing bash commands.
4. Note that to write a command it must be preceeded by a TAB.
5. Now make a new target called 'program.x', to compile 'program.f90'. Do not include dependency yet.
6. Edit 'program.f90' to write another line of text.
7. Type make and show that it doesn't recompile because there is no dependency specified.
8. Add dependency to 'program.f90'.
9. Look at what happens if we just type make.
10. Add dummy 'all' target to run all targets.
11. Make 'clean' target to remove generated files.
12. Make a file called 'clean' and then type 'make clean'. Show that we get undesired behaviour.
13. Introduce a '.PHONY' target to resolve this behaviour.
14. We've been using gfortran as our compiler, so instead of writing this out we can ensure consistency by putting this in a variable 'F90C'.
15. Variables are used by writing

    ```
    ${varname}
    ```

16. We've written 'program' a lot as well, which introduces concerns for typos. We can resolve this with inbuilt variables:

```
$@ to represent the full target name of the current target
$? returns the dependencies that are newer than the current
    target
$* returns the text that corresponds to % in the target
$< returns the name of the first dependency
$^ returns the names of all the dependencies with space as
    the delimiter
```

Solution:

```
F90C=gfortran
F90FLAGS=-O2

all: betterprogram.x program.x target

target:
	@echo "Hello World"

program.x: program.f90
	@echo "Compiling program.f90"
	gfortran program.f90 -o program.x

betterprogram.x: program.f90
	@echo "Compiling" $<
	@${F90C} ${F90FLAGS} $< -o $@

clean:
	rm -f *.x

.PHONY: target all clean
```

2 PartB

1. Now we're going to look at a more complicated example, where we use variables to simplify and streamline our makefile.

2. We have a main program that is also dependent on two modules, with one of the modules dependent on the other.

3. We need to compile our module files into object files, then compile our main program into the executable, ensuring that everything is done in the correct order.

4. First let's compile the module files into object files that can then be linked when we make the executable.

5. We could explicitly type out the rule for each module, or we could use pattern matching. This is because the rule to make '.o' files is the same every time.

6. We also have to add the '.o' files to the dependency list of 'program.x', as well as change our inbuilt variable to reference all the dependencies.

7. Finally, we need to set the dependencies of the modules, in this case modB depends on modA.

8. Demonstrate that swapping the dependency order doesn't change the compile order due to the dependencies.

9. Move objects into 'OBJS' variable.

10. Demonstrate automatic object variable creation via makefile and shell commands.

11. Demonstrate including dependencies through an include statement.

Solution:

```
F90C=gfortran
F90FLAGS=-O2

OBJS=modB.o modA.o

# Alternative way to auto-generate object list
# Get all .f90 files
F90FILES=$(shell ls *.f90)
# List non-object files
PROG=program.f90
# Remove non-object files from list
# OBJS=$(filter-out ${PROG} ,${F90FILES})
# Replace .f90 with .o for object files
```

```
# OBJS:=$(OBJS:.f90=.o)

all: program.x

# modA.o: modA.f90
#  @echo "Making object file for " $<
#  ${F90C} $< -c

# modB.o: modB.f90
#  @echo "Making object file for " $<
#  ${F90C} $< -c

%.o: %.f90
	@echo "Making␣object␣file␣for␣" $<
	${F90C} ${F90FLAGS} $< -c

program.x: program.f90 ${OBJS}
	@echo "Compiling" $<
	${F90C} ${F90FLAGS} $^ -o $@
	@echo "Done"

#modB.o: modA.o

clean:
	rm -f *.x *.o *.mod

.PHONY: clean all

include depends.mk
```

3 Part C

1. Now we want to look at a multi-directory structure, with three folders: bin, src, and lib.
2. bin contains our final programs source code, and will contain our executables at the end of compilation.
3. src contains our modules.
4. lib, if we have enough time, will contain library files that allow us to bundle modules.
5. First, let's set up our top level makefile to go into each directory and type 'make'.
6. Make variables for each subdirectory, and make a 'SUBDIR' variable to hold the full collection.
7. Introduce the 'MAKE' variable and use it to enter the subdirectories and type make.
8. Make bin depend on src
9. Make a 'clean' target that loops through each directory and runs 'make clean'
10. Add our .PHONY target.
11. Now we simply make use of our makefile from the previous section in the src directory. Here we can just make an OBJS variable from all the files in the directory.
12. To compile our main program, we need to get a list of the object files, as well as add an include flag to tell the compiler where it can find the .mod files.
13. Now we will talk about dynamic and static libraries as well as how we can compile and link to them.
14. Libraries are useful for bundling large code bases into easy to use blocks. We can then reference the modules, functions and subroutines contained within them.
15. Once compiled, we don't need to worry about cross-dependencies within the library, so we are free to use the modules we want without worrying about their individual dependencies.
16. Static libraries are compiled with the code, so if the library changes the code must be recompiled. They add to the size of the executable as it bundles all used and unused routines. They are however faster as they don't require calls to external sources.

17. Dynamic libraries are accessed at runtime, meaning that if the library changes the executable doesn't have to. This can save space and recompilation time. It does introduce another possible source of corruption or error into the code, as the program may run even if the library is faulty.

18. Static libraries are easier to compile, dynamic libraries are somewhat trickier.

19. First we compile all our .o files, but with two new flags -fpic and -shared.

20. Then bundle them into a library file with the same flags.

21. In the bin folder we need to include the lib directory, as well as add the -L flag to link to the library specified with the -l command. Note that we omit 'lib' and '.so' from the library name.

22. Because it's used at runtime, we also need to tell our program where to find the library.

23. Showcase the fact that if we change the library, the program changes without recompiling.

Solution (top):

```
BIN = ./bin/
SRC = ./src/
LIB = ./lib/
SUBDIRS = ${BIN} ${SRC} ${LIB}

all: ${SUBDIRS}

${SUBDIRS}:
	${MAKE} -C $@

${BIN}: ${SRC} ${LIB}

clean:
	$(foreach dir, ${SUBDIRS}, ${MAKE} -C ${dir} clean;)

.PHONY: clean all ${SUBDIRS}
```

Solution (bin):

```
F90C = gfortran
F90FLAGS = -O2
LIBDIR = ./../lib
SRCDIR = ./../src
LINK = -L${LIBDIR} -Wl,-rpath,'$$ORIGIN/../lib'
LIB = -ltest
INCLUDE = -I${LIBDIR} -I${SRCDIR}
```

```
OBJS=$(shell ls ${SRCDIR}/*.o)

all: program.x

program.x: program.f90 ${OBJS}
	${F90C} ${F90FLAGS} ${LINK} $^ ${LIB} ${INCLUDE} -o $@

clean:
	rm -f *.x
```

Solution (src):

```
F90C = gfortran
F90FLAGS = -O2
SRC=$(shell ls *.f90)
OBJS=$(SRC:.f90=.o)

all: ${OBJS}

%.o: %.f90
	${F90C} ${F90FLAGS} $< -c

modB.o: modA.o

clean:
	rm -f *.o *.mod

.PHONY: all clean
```

Solution (lib):

```
F90C = gfortran
LDFLAGS = -fpic -shared

SRC = $(shell ls *.f90)
OBJS = $(SRC:.f90=.o)

all: libtest.so

%.o: %.f90
	${F90C} ${LDFLAGS} $< -c

libtest.so: ${OBJS}
	${F90C} ${LDFLAGS} $^ -o $@

libmodA.o: libmodB.o
```

```
clean:
	rm -f *.mod *.a *.o *.so*

.PHONY: clean all
```