

AMath 483/583 — Lecture 9

This lecture:

- Multi-file Fortran codes
- Makefiles

Reading:

- class notes: Makefiles
- Software Carpentry lectures on Make

Notes:

Dependency checking

Makefiles give a way to recompile only the parts of the code that have changed.

Also used for checking dependencies in other build systems, e.g. creating figures, running latex, bibtex, etc. to construct a manuscript.

More modern build systems are available, e.g. [SCons](#), which allows expressing dependencies and build commands in Python.

But [make](#) (or [gmake](#)) are still widely used.

Notes:

Fortran code with 3 units

```
1  ! $UWHPSC/codes/fortran/multifile1/fullcode.f90
2
3  program demo
4      print *, "In main program"
5      call sub1()
6      call sub2()
7  end program demo
8
9  subroutine sub1()
10     print *, "In sub1"
11 end subroutine sub1
12
13 subroutine sub2()
14     print *, "In sub2"
15 end subroutine sub2
```

Notes:

Split code into 3 separate files...

```
1 ! $UWHPSC/codes/fortran/multifile1/main.f90
2
3 program demo
4   print *, "In main program"
5   call sub1()
6   call sub2()
7 end program demo
```

```
1 ! $UWHPSC/codes/fortran/multifile1/sub1.f90
2
3 subroutine sub1()
4   print *, "In sub1"
5 end subroutine sub1
```

```
1 ! $UWHPSC/codes/fortran/multifile1/sub2.f90
2
3 subroutine sub2()
4   print *, "In sub2"
5 end subroutine sub2
```

Notes:

Splitting Fortran codes into files

Compile all three and link together into single executable:

```
$ gfortran main.f90 sub1.f90 sub2.f90 \
  -o main.exe
```

Run the executable:

```
$ ./main.exe
In main program
In sub1
In sub2
```

Notes:

Splitting Fortran codes into files

Can split into separate compile....

```
$ gfortran -c main.f90 sub1.f90 sub2.f90
```

```
$ ls *.o
main.o sub1.o sub2.o
```

... and link steps:

```
$ gfortran main.o sub1.o sub2.o -o main.exe
```

```
$ ./main.exe > output.txt
```

Note: Redirected output to a text file.

Notes:

Splitting Fortran codes into files

Advantage: If we modify sub2.f90 to print "Now in sub2" we only need to recompile this piece:

```
$ gfortran -c sub2.f90

$ gfortran main.o sub1.o sub2.o -o main.exe

$ ./main.exe
In main program
In sub1
Now in sub2
```

When working on a big code (e.g. 100,000 lines split between 200 subroutines) this can make a big difference!

Use of **Makefiles** greatly simplifies this.

Notes:

Makefiles

A common way of automating software builds and other complex tasks with dependencies.

A Makefile is itself a program in a special language.

```
1 # $UWHPSC/codes/fortran/multifile1/Makefile
2
3 output.txt: main.exe
4     ./main.exe > output.txt
5
6 main.exe: main.o sub1.o sub2.o
7     gfortran main.o sub1.o sub2.o -o main.exe
8
9 main.o: main.f90
10    gfortran -c main.f90
11 sub1.o: sub1.f90
12    gfortran -c sub1.f90
13 sub2.o: sub2.f90
14    gfortran -c sub2.f90
```

Notes:

Makefiles

```
$ cd $UWHPSC/codes/fortran/multifile1
$ rm -f *.o *.exe # remove old versions

$ make main.exe
gfortran -c main.f90
gfortran -c sub1.f90
gfortran -c sub2.f90
gfortran main.o sub1.o sub2.o -o main.exe
```

Uses commands for making main.exe.

note: First had to make all the .o files.
Then executed the rule to make main.exe

Notes:

Structure of a Makefile

Typical element in the simple Makefile:

```
target: dependencies
<TAB> command(s) to make target
```

Important to use tab character, not spaces!!

Warning: Some editors replace tabs with spaces!

Typing “make target” means:

- 1 Make sure all the dependencies are up to date (those that are also targets)
- 2 If target is **older** than any dependency, **recreate** it using the specified commands.

These rules are applied **recursively!**

Notes:

Make examples

```
$ rm -f *.o *.exe

$ make sub1.o
gfortran -c sub1.f90

$ make main.o
gfortran -c main.f90

$ make main.exe
gfortran -c sub2.f90
gfortran main.o sub1.o sub2.o -o main.exe
```

Note: Last make required compiling sub2.f90
but **not** sub1.f90 or main.f90.

Notes:

Age of dependencies

The last modification time of the file is used.

```
$ ls -l sub1.*
-rw-r--r-- 1 rjl staff 111 Apr 18 16:05 sub1.f90
-rw-r--r-- 1 rjl staff 936 Apr 18 16:56 sub1.o

$ make sub1.o
make: `sub1.o' is up to date.

$ touch sub1.f90; ls -l sub1.f90
-rw-r--r-- 1 rjl staff 111 Apr 18 17:10 sub1.f90

$ make main.exe
gfortran -c sub1.f90
gfortran main.o sub1.o sub2.o -o main.exe
```

Notes:

Makefiles

First version of Makefile has 3 rules that are very similar

```
1 # $UWHPSC/codes/fortran/multifile1/Makefile
2
3 output.txt: main.exe
4     ./main.exe > output.txt
5
6 main.exe: main.o sub1.o sub2.o
7     gfortran main.o sub1.o sub2.o -o main.exe
8
9 main.o: main.f90
10    gfortran -c main.f90
11 sub1.o: sub1.f90
12    gfortran -c sub1.f90
13 sub2.o: sub2.f90
14    gfortran -c sub2.f90
```

Replace these with a [pattern](#) rule...

Notes:

Implicit rules

General rule to make the `.o` file from `.f90` file:

```
1 # $UWHPSC/codes/fortran/multifile1/Makefile2
2
3 output.txt: main.exe
4     ./main.exe > output.txt
5
6 main.exe: main.o sub1.o sub2.o
7     gfortran main.o sub1.o sub2.o -o main.exe
8
9 %.o : %.f90
10    gfortran -c $<
```

Making `main.exe` requires `main.o sub1.o sub2.o` to be up to date.

Rather than a rule to make each one separately, the implicit rule (lines 9-10) is used for all three.

Notes:

Specifying a different makefile

To use a makefile with a different name than `Makefile`:

```
$ make sub1.o -f Makefile2
gfortran -c sub1.f90
```

The rules in `Makefile2` will be used.

The directory `$UWHPSC/codes/fortran/multifile1` contains several sample makefiles.

See [class notes: Makefiles](#) for a summary.

Notes:

Implicit rules

We have to repeat the list of `.o` files twice:

```
1 # $UWHPSC/codes/fortran/multifile1/Makefile2
2
3 output.txt: main.exe
4     ./main.exe > output.txt
5
6 main.exe: main.o sub1.o sub2.o
7     gfortran main.o sub1.o sub2.o -o main.exe
8
9 %.o : %.f90
10     gfortran -c $<
```

Simplify and reduce errors by defining a macro.

Notes:

Makefile variables or macros

```
1 # $UWHPSC/codes/fortran/multifile1/Makefile3
2
3 OBJECTS = main.o sub1.o sub2.o
4
5 output.txt: main.exe
6     ./main.exe > output.txt
7
8 main.exe: $(OBJECTS)
9     gfortran $(OBJECTS) -o main.exe
10
11 %.o : %.f90
12     gfortran -c $<
```

By convention, all-caps names are used for Makefile macros.

Note that to use `OBJECTS` we must write `$(OBJECTS)`.

Notes:

Makefile variables

```
1 # $UWHPSC/codes/fortran/multifile1/Makefile4
2
3 FC = gfortran
4 FFLAGS = -O3
5 LFLAGS =
6 OBJECTS = main.o sub1.o sub2.o
7
8 output.txt: main.exe
9     ./main.exe > output.txt
10
11 main.exe: $(OBJECTS)
12     $(FC) $(LFLAGS) $(OBJECTS) -o main.exe
13
14 %.o : %.f90
15     $(FC) $(FFLAGS) -c $<
```

Here we have added for the name of the Fortran command and for compile flags and linking flags.

Notes:

Makefile variables

```
$ rm -f *.o *.exe
$ make -f Makefile4

gfortran -O3 -c main.f90
gfortran -O3 -c sub1.f90
gfortran -O3 -c sub2.f90
gfortran -O3 main.o sub1.o sub2.o -o main.exe
./main.exe > output.txt
```

Can specify variables on command line:

```
$ rm -f *.o *.exe
$ make main.exe FFLAGS=-g -f Makefile4

gfortran -g -c main.f90
gfortran -g -c sub1.f90
gfortran -g -c sub2.f90
gfortran -g main.o sub1.o sub2.o -o main.exe
```

Notes:

Phony targets — don't create files

```
1 # $UWHPSC/codes/fortran/multifile1/Makefile5
2
3 OBJECTS = main.o sub1.o sub2.o
4 .PHONY: clean
5
6 output.txt: main.exe
7     ./main.exe > output.txt
8
9 main.exe: $(OBJECTS)
10     gfortran $(OBJECTS) -o main.exe
11
12 %.o : %.f90
13     gfortran -c $<
14
15 clean:
16     rm -f $(OBJECTS) main.exe
```

Note: No dependencies, so always do commands

```
$ make clean -f Makefile5
rm -f main.o sub1.o sub2.o main.exe
```

Notes:

Common Makefile error

Using spaces instead of tab...

If we did this in the `clean` commands, we'd get:

```
$ make clean -f Makefile5
```

```
Makefile5:14: *** missing separator. Stop.
```

Notes:

make help

```
1 # $UWHPSC/codes/fortran/multifile1/Makefile6
2
3 OBJECTS = main.o sub1.o sub2.o
4 .PHONY: clean help
5
6 output.txt: main.exe
7     ./main.exe > output.txt
8
9 main.exe: $(OBJECTS)
10     gfortran $(OBJECTS) -o main.exe
11
12 %.o : %.f90
13     gfortran -c $<
14
15 clean:
16     rm -f $(OBJECTS) main.exe
17
18 help:
19     @echo "Valid targets:"
20     @echo "  main.exe"
21     @echo "  main.o"
22     @echo "  sub1.o"
23     @echo "  sub2.o"
24     @echo "  clean: removes .o and .exe files"
```

`echo` means print out the string.

`@echo` means print out the string but don't print the command.

Notes:

Fancier things are possible...

```
1 # $UWHPSC/codes/fortran/multifile1/Makefile7
2
3 SOURCES = $(wildcard *.f90)
4 OBJECTS = $(subst .f90,.o,$(SOURCES))
5
6 .PHONY: test
7
8 test:
9     @echo "Sources are: " $(SOURCES)
10    @echo "Objects are: " $(OBJECTS)
```

This gives:

```
$ make test -f Makefile6
Sources are: fullcode.f90 main.f90 sub1.f90 sub2.f
Objects are: fullcode.o main.o sub1.o sub2.o
```

Note this found `fullcode.f90` too!

Notes:

Other makefile examples

The html version of the class notes are created by typing

```
make html # OR: make latex
```

in the the directory `$UWHPSC/notes/`

See the Makefile in that directory.

Each `.rst` (ReStructured Text) file is turned into an html file corresponding to one webpage.

Changing one `.rst` file and redoing `make html` only "recompiles" this one file.

But try modifying the configuration file `conf.py` and all files will be regenerated.

Note: This is not a great example because the dependency checking is actually done by the program `sphinx-build`.

Notes: