makefiles

RJ LeVeque, UWHPSC

This lecture:

- Multi-file Fortran codes
- Makefiles

Reading:

- Software Carpentry lectures on Make

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.

/hands_on/fortran/multifile/

```
program demo
    print *, "In main program"
    call sub1()
    call sub2()
end program demo

subroutine sub1()
    print *, "In sub1"
end subroutine sub1

subroutine sub2()
    print *, "In sub2"
end subroutine sub2
```

fullcode.f90

/hands_on/fortran/multifile/

main.f90

```
program demo
    print *, "In main program"
    call sub1()
    call sub2()
end program demo
```

sub1.f90

```
subroutine sub1()
    print *, "In sub1"
end subroutine sub1
```

sub2.f90

```
subroutine sub2()
    print *, "In sub2"
end subroutine sub2
```

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
```

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.

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.

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

A Makefile is itself a program in a special language.

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

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

main.o: main.f90
	gfortran -c main.f90
sub1.o: sub1.f90
	gfortran -c sub1.f90
sub2.o: sub2.f90
	gfortran -c sub2.f90
```

/hands_on/fortran/multifile/Makefile

What does it do(make)?

```
$ 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`

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!

```
$ 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`.

The last modification time of the file is used.

```
$ ls -l sub1.*

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

```
$ touch sub1.f90;   ls -l sub1.f90
```

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

Rules

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

Makefile2

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

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

%.o : %.f90
        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 is used for all three.

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.

Macros

Makefile3

```
OBJECTS = main.o sub1.o sub2.o

output.txt: main.exe
	./main.exe > output.txt

main.exe: $(OBJECTS)
	gfortran $(OBJECTS) -o main.exe

%.o : %.f90
	gfortran -c $<
```

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

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

Variables

Makefile4

```
FC = gfortran
FFLAGS = -O3
LFLAGS =
OBJECTS = main.o sub1.o sub2.o

output.txt: main.exe
	./main.exe > output.txt

main.exe: $(OBJECTS)
	$(FC) $(LFLAGS) $(OBJECTS) -o main.exe

%.o : %.f90
	$(FC) $(FFLAGS) -c $<
```

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

```
$ 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
```

Phony objects

Makefile5

```
OBJECTS = main.o sub1.o sub2.o
.PHONY: clean

output.txt: main.exe
	./main.exe > output.txt

main.exe: $(OBJECTS)
	gfortran $(OBJECTS) -o main.exe

%.o : %.f90
	gfortran -c $<

clean:
	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
```

Make help

Makefile6

```
OBJECTS = main.o sub1.o sub2.o
.PHONY: clean help

output.txt: main.exe
	./main.exe > output.txt

main.exe: $(OBJECTS)
	gfortran $(OBJECTS) -o main.exe

%.o : %.f90
	gfortran -c $<

clean:
	rm -f $(OBJECTS) main.exe

help:
	@echo "Valid targets:"
	@echo "  main.exe"
	@echo "  main.o"
	@echo "  sub1.o"
	@echo "  sub2.o"
	@echo "  clean:  removes .o and .exe files"
```

Wildcard!

Makefile7

```
SOURCES = $(wildcard *.f90)
OBJECTS = $(subst .f90,.o,$(SOURCES))

.PHONY: test

test:
        @echo "Sources are: " $(SOURCES)
        @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!