

AMath 483/583 — Lecture 8 — April 13, 2011

Today:

- Makefiles

Friday:

- Computer architecture
- Cache considerations
- Optimizing Fortran codes

Read: Class notes and references

Notes:

Splitting Fortran codes into files

Single file program with 2 subroutines:

```
! $CLASSHG/codes/fortran/multifile1/fullcode.f90
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
```

Notes:

Splitting Fortran codes into files

Split into 3 files:

Main program...

```
! $CLASSHG/codes/fortran/multifile1/main.f90
program demo
  print *, "In main program"
  call sub1()
  call sub2()
end program demo
```

and two separate files (for $N = 1, 2$):

```
! $CLASSHG/codes/fortran/multifile1/subN.f90
subroutine subN()
  print *, "In subN"
end subroutine subN
```

Notes:

Splitting Fortran codes into files

Compile all three and link together into single executable:

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

Run the executable:

```
$ ./fullcode.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 fullcode.exe
```

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

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 fullcode.exe
```

```
$ ./fullcode.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!

Next lecture: Make this easier with **Makefiles**.

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.

```
# $CLASSHG/codes/fortran/multifile1/Makefile

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

Notes:

Makefiles

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

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

Uses commands for making `fullcode.exe`.

Note: First had to make all the `.o` files.

Then executed the rule to make `fullcode.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 # first target in file if none specified
gfortran -c sub2.f90
gfortran main.o sub1.o sub2.o -o fullcode.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 27 16:05 sub1.f90
-rw-r--r-- 1 rjl staff 936 Apr 27 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 27 17:10 sub1.f90

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

Notes:

Implicit rules

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

```
# $CLASSHG/codes/fortran/multifile1/Makefile2

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

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

Making fullcode.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.

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 `$CLASSHG/codes/fortran/multifile1` contains several sample makefiles.

Notes:

Makefile variables or macros

```
# $CLASSHG/codes/fortran/multifile1/Makefile3

OBJECTS = main.o sub1.o sub2.o

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

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

Notes:

Makefile variables

```
# $CLASSHG/codes/fortran/multifile1/Makefile4

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

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

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

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 fullcode.exe
```

Can specify variables on command line:

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

Notes:

Phony targets — don't create files

```
# $CLASSHG/codes/fortran/multifile1/Makefile5
OBJECTS = main.o sub1.o sub2.o
.PHONY: clean

fullcode.exe: $(OBJECTS)
    gfortran $(OBJECTS) -o fullcode.exe
%.o : %.f90
    gfortran -c $<

clean:
    rm -f $(OBJECTS) fullcode.exe
```

Note: No dependencies, so always do commands

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

Fancier things are possible...

```
# $CLASSHG/codes/fortran/multifile1/Makefile6

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
```

Notes:

make help

```
# $CLASSHG/codes/fortran/multifile1/Makefile6

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

... as in Makefile5

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

Notes:

Other makefile examples

The html version of the class notes are created by typing

```
make html
```

in the the directory `$CLASSHG/sphinx/`

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: