LUND
UNIVERSITY

Tools for modern Fortran development

Maintaining Fortran code

- Compilation and maintainance issues grow as project becomes larger
- Requirements for building on multiple platforms
- Shell scripts not suitable for larger projects
  - Can't handle source dependencies
  - Require complete rebuild for every change
- Make and CMake can solve some of these problems

MAINTAINING FORTRAN PROJECTS WITH MAKE

Make

- Builds software from using a set of rules
- Automatically handles dependencies in source files
  - Only rebuilds files affected by the change in the source files
- Available on all platforms (GNU Make)
- Can be non-intuitive in the beginning

Basic make file syntax

```
target: [dependencies]
→ system command
```

TAB

This is the target of the rule

Myfile.gz depends on myfile.txt

```
myfile.gz: myfile.txt
→ cat myfile.txt | gzip > myfile.gz
```

TAB

This is the "recipe" for creating myfile.gz

Running make

```
$ ls
Makefile myfile.txt
$ make
cat myfile.txt | gzip > myfile.gz $ ls
Makefile myfile.gz myfile.txt
```

Running make again

```
$ make
make: 'myfile.gz' is up to date.
```

Running make with updated myfile.txt

```
$ touch myfile.txt
$ make
cat myfile.txt | gzip > myfile.gz $
```

Compiling code

- Building an executable requires
  - Compilation of source code to object-files
  - Linking of object-files to executable
- Object-files depend on source files
- Executable depends on object-files
- Case for make ;)

Example

```
myprog : myprog.o
   gfortran myprog.o -o myprog

myprog.o : myprog.f90
   gfortran -c myprog.f90
```

Link executable with object-files

Compile source code to object file

Running make

```
$ ls
Makefile myprog.f90
$ make
gfortran -c myprog.f90
gfortran myprog.o -o myprog
```

Example 2

```
myprog : mymodule.o myprog.o
   gfortran mymodule.o myprog.o -o myprog

myprog.o : myprog.f90
   gfortran -c myprog.f90

mymodule.o : mymodule.f90
   gfortran -c mymodule.f90
```

Running make

No compilation of myprog.o

```
$ touch mymodule.f90
$ make
gfortran -c mymodule.f90
gfortran myprog.o mymodule.o -o myprog
```

Module dependencies

- In Fortran 9x, compilation order is important
- A module used by another source file needs to be built earlier in the build
- Make can solve this for us again ;)

Example 3

```
myprog : module main.o module_truss.o
    gfortran module main.o module_truss.o -o myprog

module_main.o : module_main.f90
    gfortran -c module_main.f90

module_truss.o: module_truss.f90
    gfortran -c module_truss . f90
```

Running make

```
$ make
module_main.f90:3.5:

use truss
    1

Fatal Error: Can't open module file 'truss.mod' for reading at (1):
No such file or directory
make: *** [module_main.o] Error 1
```

Module_truss needs to built before module_main

Example 3 - Modified

```
myprog : module main.o module_truss.o
    gfortran module main.o module_truss.o -o myprog

module_main.o : module_main.f90 module_truss.o
    gfortran -c module_main.f90

module_truss.o: module_truss.f90
    gfortran -c module_truss . f90
```

Module_truss is now build before module_main

Running make

```
$ make
gfortran -c module_truss.f90
gfortran -c module_main.f90
gfortran module_main.o module_truss.o -o myprog
```

Variables in make

- Reduce rewrite of common commands
- Increase configuratbility and portability
- Easier to read
- Works in principle as bash variables
- Assignment
  - [Variablename] = value
- Using value of variable
  - $([Variablename])

Example 4

```
FC=gfortran
FFLAGS=-c
EXECUTABLE=myprog

$(EXECUTABLE) : myprog.o mymodule.o
    $(FC) myprog.o mymodule.o -o myprog

Myprog.o : myprog.f90
    $(FC) $(FFLAGS) myprog.f90

Mymodule.o : mymodule.f90
    $(FC) $(FFLAGS) mymodule.f90

clean :
    rm -rf *.o *.mod $(EXECUTABLE)
```

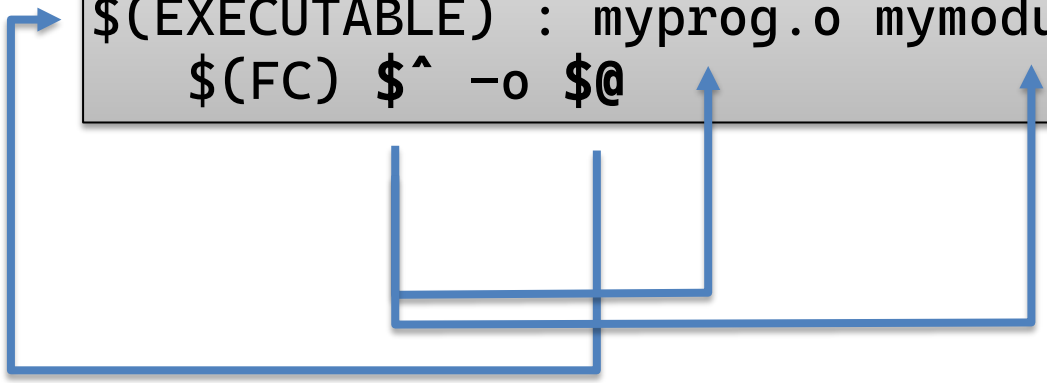
All configurable options located in a single position in the makefile

Internal macros

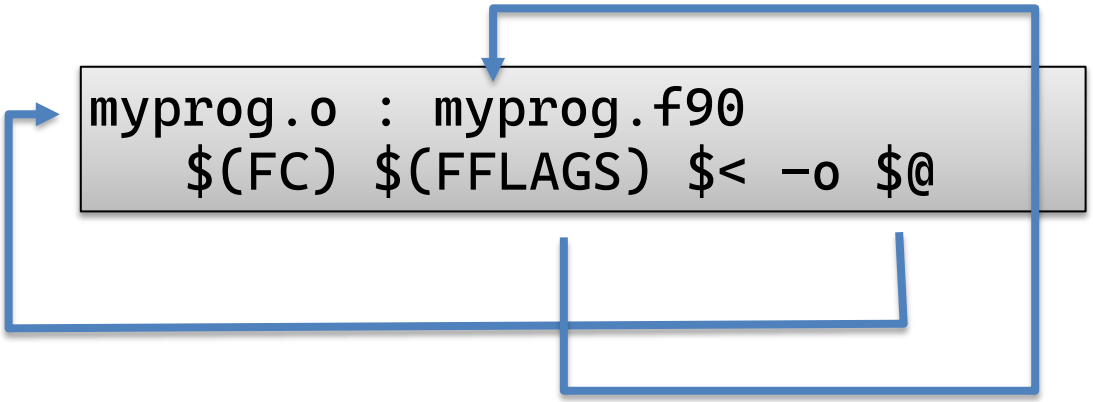
- Create even more generic makefiles
- Most important macros
  - $@ - Target of the current rule executed
  - $^ - Name of all pre-requisites
  - $< - Name first pre-requisite

Example 5

```
FC=gfortran
FFLAGS=-c
EXECUTABLE=myprog
$(EXECUTABLE) : myprog.o mymodule.o
   $(FC) $^ -o $@
```



Example 5 – cont…



Suffix rules

- In larger projects it can be difficult to maintin a large number of rules for single source files
- Suffix rules are ”recipies” for converting a source suffix to an object suffix
  - Same rule applies to all source code in makefile

Example 6 – cont…

If an executable depends on **main.o** this rule will automatically try to build it using **main.f90**

```
.f90.o:
	$(FC) $(FFLAGS) $< –o $@

.SUFFIXES: .f90 .f03 .f .F
```

A suffix must be supported by make. Additional suffices can be added using the **.SUFFIXES** rule.

Wildcard expansion and more

- Wildcard expansion using the wildcard-function can be used to create lists of files
- Substitutions can also be made using the patsubst-function in make
- This enables even more generic makefiles

Example 7

```
F90_FILES := $(wildcard *.f90)
```

Finds all .f90 files in the current directory and assigns them to the list F90_FILES.

NOTE, the := assignment operator must be used in make-functions

```
OBJECTS := $(patsubst %.f90, %.o, $(F90_FILES))
```

Replaces all .f90 suffixes with .o assigns this list to the variable OBJECTS.

Example 7 - continued

```
$(EXECUTABLE) : $(OBJECTS)
    $(FC) $^ -o $@
```

Here we have a generic reusable rule for linking an executable

Combined with the suffix.rules no source files needs to be specified in the makefile

Pattern rules

- Similar to suffix rules.
- No need to use .SUFFIXES for additional supported suffixes
- The recommended way in GNU Make
- Uses the % operator in the same way as in the wildcard functions

Example 8 – Generic make

```
FC=gfortran
FFLAGS=-c
EXECUTABLE=myprog
F90_FILES := $(wildcard *.f90)
OBJECTS := $(patsubst %.f90, %.o, $(F90_FILES))

$(EXECUTABLE) : $(OBJECTS)
    $(FC) $^ -o $@

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

mymodule.o : myutils.o

clean :
    rm -rf *.o *.mod $(EXECUTABLE)
```

No explicit specification of source files or object-files

Modules dependencies still needs to be handled.

MAINTAINING FORTRAN PROJECTS WITH CMAKE

CMake

- CMake is a cross-platform build system
- Generates different types of build files
  - Makefiles of different flavors
  - Project files for Eclipse, Visual studio and Xcode
- Is a language

Example 1

```
cmake_minimum_required(VERSION 3.0)

project(simple)

enable_language(Fortran)

add_executable(simple myprog.f90)
```

CMake version control

Name of project (not executable)

Activate Fortran support. Not on by default.

Create an executable target, simple, that uses myprog.f90 as source.

Example 1 – cont…

```
$ ls
CMakeLists.txt myprog.f90
$ cmake .
-- The C compiler identification is GNU 4.2.1
-- The CXX compiler
    ...
-- Configuring done
-- Generating done
-- Build files have been written to: /Users/.../simple
$ ls
CMakeCache.txt
CMakeFiles Makefile myprog.f90
```

Running make

```
$ make
Scanning dependencies of target simple
[100%] Building Fortran object
CMakeFiles/simple.dir/myprog.f90.
o
Linking Fortran executable simple
[100%] Built target simple
```

Example 1 – Using a build directory

CMake generates a large number of build files. To avoid mixing these with existing source. A build, directory is often used.

```
$ mkdir build
$ cd build
$ cmake ..
-- The C compiler identification is GNU 4.2.1 .
.
-- Generating done
-- Build files have been written to: /Users/.../simple/build
```

Debug and release versions

- Cmake supports building for debugging and a release version
- The CMAKE_BUILD_TYPE variable controls the current build type
  - Set by using the -D option on the command line

```
$ cmake -D CMAKE_BUILD_TYPE=Release ..
-- Configuring done
-- Generating done
-- Build files have been written to: /Users/lindemann/
Development/progsci_book/source/cmake_examples/simple/b
```

Library dependencies

```
cmake minimum required(VERSION 2.6)

project(simple)

enable language(Fortran)

add_executable(simple myprog.f90)
target_link_libraries(simple blas m)
```

target_link_libraries associates a target with its library dependencies. Same notation used for both Unix and Windows builds.

Conditional builds

- To control configuration and build options for different platforms. Conditional build statements can be used.
- Predefined variables for different platforms exists (UNIX, LINUX, APPLE, WIN32)

```
if (UNIX)
    message("This is a Unix build.")
endif (UNIX)
```

Running CMAke on the above code on a Unix based system will print the message on standard output.

Conditional builds

```
if (UNIX)
    add_executable(multiple myprog.f90 mymodule.f90)
    target_link_libraries(multiple blas m)
else (UNIX)
    if (WIN32)
        add_executable(multiple myprog.f90 mymodule.f90)
        target_link_libraries(multiple blas32)
    else (WIN32)
        message("Not supported configuration.")
    endif (WIN32)
endif (UNIX)
```

Variables

- Variables are defined using the set command
  - set(MYVAR "This is a variable")
- Using the variables value requires it to be enclosed in ${…}
  - message(${MYVAR})
- Lists are also created using the set command
  - set(MYLIST a b c)

Iterating over lists

```
set(MYLIST a b c)
Foreach(i ${MYLIST})
    message(${i})
Endforeach(i)
```

Running CMake

```
$ cmake ..
a
b
c
-- Configuring done
-- Generating done
-- Build files have been written to: ...
```

Controlling optimisation

- The optimisation options for Fortran are controlled using the variables:
  - CMAKE_Fortran_FLAGS_RELEASE
  - CMAKE_Fortran_FLAGS_DEBUG
- The chosen Fortran compiler can be set using the variable
  - CMAKE_Fortran_COMPILER

CMake and compiler options

```
get_filename_component (Fortran_COMPILER_NAME ${CMAKE_Fortran_COMPILER}
NAME)

if (Fortran_COMPILER_NAME STREQUAL "gfortran")
  set (CMAKE_Fortran_FLAGS_RELEASE "-funroll-all-loops -fno-f2c -O3")
  set (CMAKE_Fortran_FLAGS_DEBUG   "-fno-f2c -O0 -g")
elseif (Fortran_COMPILER_NAME STREQUAL "ifort")
  set (CMAKE_Fortran_FLAGS_RELEASE "-f77rtl -O3")
  set (CMAKE_Fortran_FLAGS_DEBUG   "-f77rtl -O0 -g")
elseif (Fortran_COMPILER_NAME STREQUAL "g77")
  set (CMAKE_Fortran_FLAGS_RELEASE "-funroll-all-loops -fno-f2c -O3 -m32")
  set (CMAKE_Fortran_FLAGS_DEBUG   "-fno-f2c -O0 -g -m32")
else (Fortran_COMPILER_NAME STREQUAL "gfortran")
  message ("No optimized Fortran compiler flags are known.")
  set (CMAKE_Fortran_FLAGS_RELEASE "-O2")
  set (CMAKE_Fortran_FLAGS_DEBUG   "-O0 -g")
endif (Fortran_COMPILER_NAME STREQUAL "gfortran")
```

CMake – Project generation

- CMake supports the generation of project files for most development environments
  - Visual Studio
  - Eclipse
  - NetBeans
  - Xcode
  - And more
- The –G command line option is used to select the used generator (default is makefiles)

CMake – Project generation

```
mkdir build_eclipse
$ cd build_eclipse/
$ cmake -G "Eclipse CDT4 - Unix Makefiles" ../multiple/
-- The C compiler identification is GNU 4.2.1
-- The CXX compiler identification is Clang 4.0.0
-- Could not determine Eclipse version, assuming at least 3.6 (
Helios). Adjust CMAKE_ECLIPSE_VERSION if this is wrong. ...
-- Generating done
-- Build files have been written to: ...
$ ls -la
total 112
drwxr-xr-x   8 lindemann  staff    272 Aug 29 20:07 .
drwxr-xr-x  13 lindemann  staff    442 Aug 29 20:06 ..
-rw-r--r--   1 lindemann  staff  14343 Aug 29 20:07 .cproject
-rw-r--r--   1 lindemann  staff   5527 Aug 29 20:07 .project
-rw-r--r--   1 lindemann  staff  17808 Aug 29 20:07 CMakeCache.txt
drwxr-xr-x  21 lindemann  staff    714 Aug 29 20:07 CMakeFiles
-rw-r--r--   1 lindemann  staff   4770 Aug 29 20:07 Makefile
-rw-r--r--   1 lindemann  staff   1562 Aug 29 20:07 cmake_install.cmake
```

These files can be opened directly in Eclipse