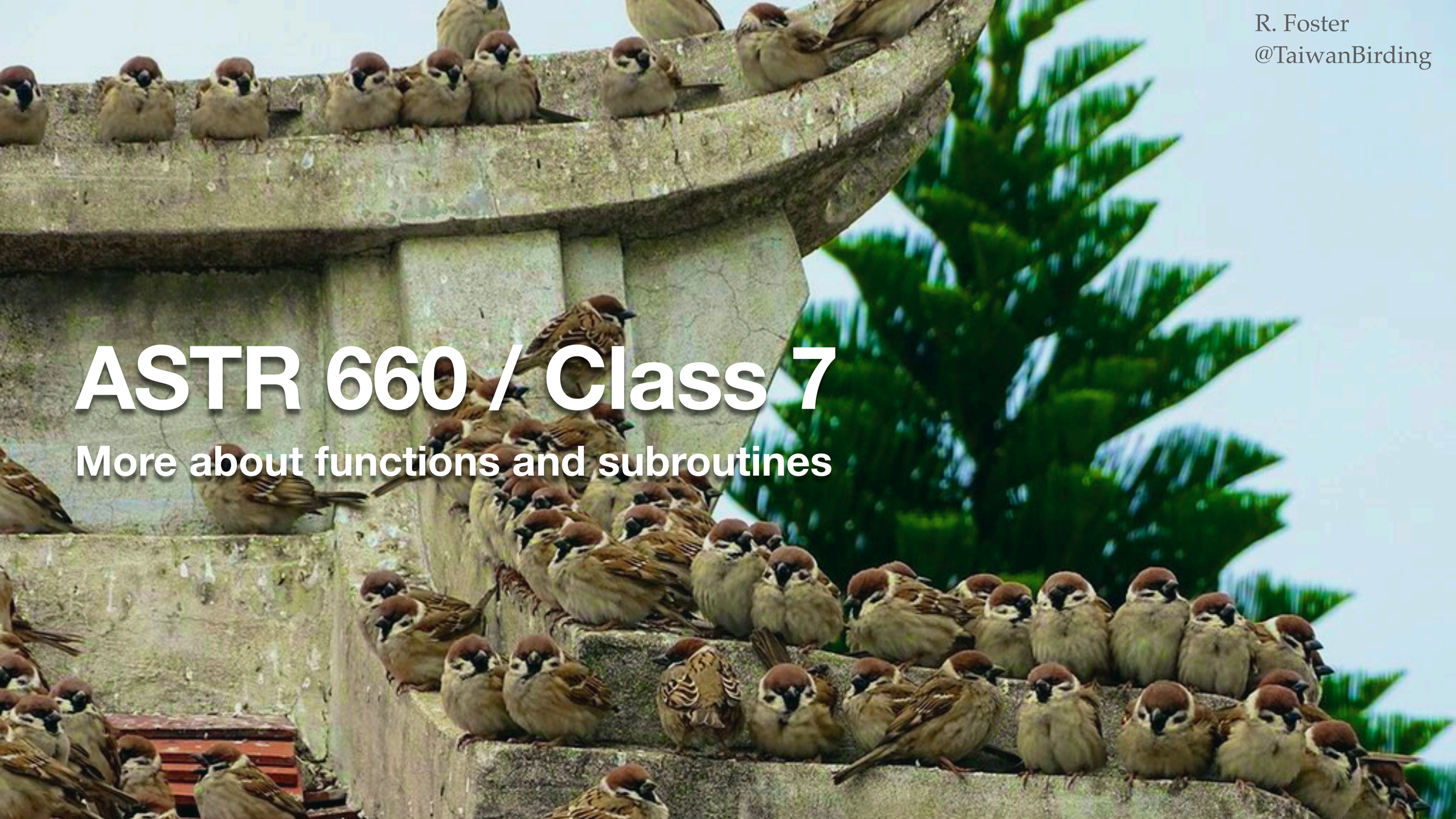

ASTR 660 / Class 7

More about functions and subroutines

R. Foster
@TaiwanBirding

Procedures (functions and subroutines)

Last week we introduced subroutines, the fundamental principle by which Fortran programs are organised. We had a couple of golden rules:

> **Use functions where there is a single clear "return" value and the procedure has no side effects. Use subroutines for everything else.**
>
> **Use `intent(IN)`, `intent(OUT)` and `intent(INOUT)` to make it clear what functions and subroutines do with their arguments**

In common with Python, I'll often use the word "functions" to mean either functions or subroutines in Fortran. The correct word is really "procedures", but nobody ever says "procedures". I will try to make it clear where the difference between functions and subroutines matters.

Procedure declarations and arguments

We will wrap up the discussion of procedures by considering the following problem:

Imagine a subroutine that takes a function as an argument. For example, the subroutine might be an integrator, and the function might be the integrand, like in the Sersic homework problem.

In python this is trivial, we just happily pass the function around like the other arguments. It's nice that we can do this, it lets us express our idea of what an "integrator" is in a natural way.

```
def integrator(integrand, dx, x0, nsteps):
    sum = 0
    for istep in range(0,nsteps):
      sum += integrand(x0+dx*istep)*dx
    return sum
```

Procedures calling other procedures

Here is a program that calls a subroutine, passing a function. The subroutine then uses the function.

```
program main
  implicit none
  real :: beta

  ! Call the subroutine alpha, passing the
  ! function beta as an argument
  call alpha(beta)

end program main
```

```
real function beta(a)
  implicit none
  real :: a
  ! A simple function
  beta = a**2
end function beta

subroutine alpha(f)
  implicit none
  ! We need to define the function type here
  real :: f
  real :: a
  ! Call whatever function has been passed.
  a = f()
end subroutine alpha
```

Procedures calling other procedures

Here is a program that calls a subroutine, passing a function. The subroutine then uses the function.

```
program main
  implicit none

  ! Since beta is a function, do we need to
  ! define it here?
  real :: beta

  ! Call the subroutine alpha, passing the
  ! function beta as an argument
  call alpha(beta)

end program main
```

```
real function beta(a)
  implicit none
  real :: a
  ! A simple function
  beta = a**2
end function beta

subroutine alpha(f)
  implicit none
  ! We need to define the function type here
  real :: f
  real :: a
  ! Call whatever function has been passed.
  a = f()
end subroutine alpha
```

Functions calling functions

Yes, define `beta` in the `program` block!

```
functions_calling_functions.f90:25:13:

   25 |   call alpha(beta)
      |             1
Error: Expected a procedure for argument 'f' at (1)
```

No, don't define `beta` in the `program` block!

```
functions_calling_functions.f90:25:17:

   25 |   call alpha(beta)
      |                 1
Error: Symbol 'beta' at (1) has no IMPLICIT type
```

Is there a right answer?

If we define `beta` in the normal way, as `real`, it conflicts with what `alpha` expects as an argument.

If we don't define it, it seems the compiler doesn't know about the function definition.

That's reasonable too: the function is a separate "unit", which we would have to declare if we want to refer to it.

External functions

To make it clear what we want, we can add `external` to the declaration of `beta`.

```
program main
  implicit none

  ! Define an external function
  real, external :: beta

  ! Call the subroutine alpha, passing the
  ! function beta as an argument
  call alpha(beta)

end program main
```

```
real function beta(a)
  implicit none
  real :: a
  ! A simple function
  beta = a**2
end function beta

subroutine alpha(f)
  implicit none
  ! We need to define the function type here
  real :: f
  real :: a
  ! Call whatever function has been passed.
  a = f()
end subroutine alpha
```

External functions

We can add external every time we declare a function inside another function. It seems like **a good idea that does no harm!**

```
program main
  implicit none

  ! Define an external function
  real, external :: beta

  ! Call the subroutine alpha, passing the
  ! function beta as an argument
  call alpha(beta)

end program main
```

```
real function beta(a)
  implicit none
  real :: a
  ! A simple function
  beta = a**2
end function beta

subroutine alpha(f)
  implicit none
  ! We need to define the function type here
  real, external :: f
  real :: a
  ! Call whatever function has been passed.
  a = f()
end subroutine alpha
```

Summary: procedures

Use functions where there is a single clear "return" value and the procedure has no side effects. Use subroutines for everything else.

Use `intent(IN)`, `intent(OUT)` and `intent(INOUT)` to make it clear what functions and subroutines do with their arguments.

To distinguish function declarations from regular variables, use `external`. Always do this for clarity, even if it isn't strictly needed.

Inspired by this last point, we will move on to some more practical information about compiling and organising Fortran code (which also apply to C and C++).

R. Foster
@TaiwanBirding

Collared Bush-Robin
栗背林鴝

ASTR 660 / Class 7

Compiler options

Compiler warnings

If it's such a good idea to always declare external functions, maybe the compiler can help us? If we don't use the `external` but instead use the following compiler option:

```
gfortran -Wimplicit-procedure -o test functions_calling_functions.f90
```

```
functions_calling_functions.f90:25:18:

   25 |   call alpha(beta)
      |                  1
Warning: Procedure 'alpha' called at (1) is not explicitly declared [-Wimplicit-procedure]
```

This is an example of a compiler warning. We can choose how "helpful" we want the compiler to be in telling us about things that **may or may not** be real problems.

Note: "procedure" is the proper name for functions and subroutines. Apparently what we're talking about here is called an "implicit procedure".

Compiler warnings

Since we are still learning Fortran, maybe we want as much advice from the compiler as possible. We can turn on "all" the warnings with `-Wall`.

```
gfortran -Wall -o test functions_calling_functions.f90
```

Unfortunately, this is just all the common warnings, not literally all the possible warnings.

It doesn't include either of the two warnings we've used so far! What does it include?

https://gcc.gnu.org/onlinedocs/gfortran/Error-and-Warning-Options.html

> Enables commonly used warning options pertaining to usage that we recommend avoiding and that we believe are easy to avoid. This currently includes -Waliasing, -Wampersand, -Wconversion, -Wsurprising, -Wc-binding-type, -Wintrinsics-std, -Wtabs, -Wintrinsic-shadow, -Wline-truncation, -Wtarget-lifetime, -Winteger-division, -Wreal-q-constant, -Wunused and -Wundefined-do-loop.

Compiler warnings

If you want to see what warnings are available, you can use the following (no need to compile anything, this just asks `gfortran` for the information):

```
gfortran --help=warnings,F
```

There are a lot of other warning options available! Here are all the others that aren't activated by `-Wall -Wextra`

(I'm not sure about -Wconversion, the manual seems to be wrong about that one!)

```
-Walign-commons -Warray-temporaries -Wcharacter-truncation -Wcompare-reals -Wconversion
-Wconversion-extra -Wdate-time -Wdo-subscript -Wfunction-elimination -Wimplicit-interface
-Wimplicit-procedure -Wmissing-include-dirs -Wopenacc-parallelism -Woverwrite-recursive
-Wpedantic -Wrealloc-lhs -Wrealloc-lhs-all -Wunderflow -Wunused-dummy-argument -Wuse-without-
only -Wzerotrip
```

Environment variables

Of course that's too tedious to type in every time you compile something. There are several solutions. One simple one is to define a shell environment variable.

In bash, environment variables are defined with the shell command `export`. For example:

```
export MY_FAVORITE_PATH=/cluster/software/admin
```

In the case of our compiler warnings, what we want is:

```
export ALL_WARNINGS=(-Wall -Wextra -Walign-commons -Warray-temporaries -Wcharacter-
truncation -Wcompare-reals -Wconversion -Wconversion-extra -Wdate-time -Wdo-subscript
-Wfunction-elimination -Wimplicit-interface -Wimplicit-procedure -Wmissing-include-dirs
-Wopenacc-parallelism -Woverwrite-recursive -Wpedantic -Wrealloc-lhs -Wrealloc-lhs-all
-Wunderflow -Wunused-dummy-argument -Wuse-without-only -Wzerotrip)
```

Notice the brackets, they are important! (Some shells might have problems with this; if so, try quotes).

Environment variables

```
gfortran $ALL_WARNINGS -o test functions_calling_functions.f90

functions_calling_functions.f90:14:6:

   14 |   a = f()
      |       1
Warning: Procedure 'f' called with an implicit interface at (1) [-Wimplicit-interface]
functions_calling_functions.f90:25:18:

   25 |   call alpha(beta)
      |                  1
Warning: Procedure 'alpha' called with an implicit interface at (1) [-Wimplicit-interface]
```

Compiler optimisation

https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

We already saw `-g`, to include debugging symbols.

The next most commonly used option is `-O` (capital letter O) which enables optimisation.

Different levels of optimisation ('aggressiveness') are specified by numbers after the `O`:

`-O1` (basic; same as -O), `-O2` (normal), `-O3` (aggressive).

The compiler tries to make the code faster using common strategies like inlining procedures to avoid function calls, unrolling loops etc.

Each level allows the compiler to take more risks in how it interprets what your code is supposed to do. With optimisation, the code that gets compiled is no long exactly the code you wrote.

Compiler optimisation

https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

General strategy:

Higher levels can make your program harder (or impossible) to debug by inspection. Don't use any optimisation while you are developing your code.

If the code is too slow to run and debug without some basic optimisation, use `-Og` (only optimisations that don't interfere with debugging) or `-O1` (same as -O).

When your code is running smoothly, use `-O2`.

Only use `-O3` if it makes the code significantly faster compared to `-O2` and you are sure it produces the same results. Do not take that for granted.

Premature optimisation of any kind is usually a bad idea. Optimisation is only worth it if you can also profile the code.

Bounds checks

It is usually hard for the compiler to catch out-of-bounds access to arrays.

This *might* not crash your program or change your results, but it could.

The option `-fcheck=bounds` tells the compiler to add explicit bounds checking to all array access. This is expensive (i.e. it slows down the code a lot).

There are other `-fcheck` *options.*

```
program bounds
  implicit none
  real, allocatable :: A(:)
  real :: sum
  integer :: n, i

  n = 10
  allocate(A(n))

  A(:) = 2

  sum = 0.0
  do i=1,100
    sum = sum + A(i)
  end do
  write(*,*) sum
end program bounds
```

R. Foster
@TaiwanBirding

White-eared Sibia
白耳畫眉

ASTR 660 / Class 7

Fortran modules

Modules

Modules are the Fortran equivalent of “libraries” or “packages”.

They are used to organise long and complicated programs into separate files, grouping together functions and subroutines that have a common purpose.

They allow code to be re-used in different projects without cut-and-pasting.

A module has two parts: compiled code (**a library**) and some extra information for the compiler about how that code works (**a module file**).

C works in a similar way, with libraries and “header” files.

Modules

Let's write a test program to demonstrate modules.

The output from will be:

A line showing a "version number".

A nicely-formatted matrix.

A counter that increases each time we print the matrix.

```
My module version:  1.0

  1.00   1.00   1.00   1.00   1.00
  2.00   2.00   2.00   2.00   2.00
  3.00   3.00   3.00   3.00   3.00
  4.00   4.00   4.00   4.00   4.00
  5.00   5.00   5.00   5.00   5.00
Called   1 times

  1.00   1.00   1.00   1.00   1.00
  2.00   2.00   2.00   2.00   2.00
  3.00   3.00   3.00   3.00   3.00
  4.00   4.00   4.00   4.00   4.00
  5.00   5.00   5.00   5.00   5.00
Called   2 times
```

Main program using a module

Like `import` in Python.

USE statements always come first!

Call the subroutine defined in the module.

main.f90

```
program main
  use my_module
  implicit none

  real :: A(5,5)
  integer :: i,j

  ! This variable is defined in my_module
  write(*,'(a,f4.1)') 'My module version: ', my_module_version
  write(*,*)

  ! Fill the array with some numbers
  A(:,:) = reshape([(1, (j, j=2,5),  i=1, 5)], shape(A))

  ! This subroutine is defined in my_module
  call print_matrix(A)
  call print_matrix(A)

end program main
```

Module

Module-level variables

“Contains” block

Individual functions and subroutines

(n.b. this format syntax is F2008)

my_module.f90

```
module my_module
  implicit none

  private  ! All entities are now module-private by default
  public my_module_version, print_matrix  ! Explicitly export public entities

  real, parameter :: my_module_version = 1.0
  integer, save :: print_matrix_n_calls = 0

contains

  subroutine print_matrix(A)
    implicit none
    ! Print matrix A to screen

    real, intent(in) :: A(:,:)  ! An assumed-shape dummy argument
    integer :: i

    print_matrix_n_calls = print_matrix_n_calls + 1

    do i = 1, size(A,1)
      write(*,'(*(f6.2, 1x))') A(i,:)
    end do
    write(*,'(a,1x,i3,1x,a)') 'Called',print_matrix_n_calls,'times'
    write(*,*)
  end subroutine print_matrix

end module my_module
```

Module

What happens here?

What happens here?

```
module my_module
  implicit none

  private  ! All entities are now module-private by default
  public my_module_version, print_matrix  ! Explicitly export public entities

  real, parameter :: my_module_version = 1.0
  integer, save :: print_matrix_n_calls = 0

contains

  subroutine print_matrix(A)
    implicit none
    ! Print matrix A to screen

    real, intent(in) :: A(:,:)  ! An assumed-shape dummy argument
    integer :: i

    print_matrix_n_calls = print_matrix_n_calls + 1

    do i = 1, size(A,1)
      write(*,'(*(f6.2, 1x))') A(i,:)
    end do
    write(*,'(a,1x,i3,1x,a)') 'Called',print_matrix_n_calls,'times'
    write(*,*)
  end subroutine print_matrix

end module my_module
```

Compiling the module

We now have two steps to compile the full program:

Compile the module

```
gfortran -c my_module.f90
```

This produces `my_module.o` and `my_module.mod`

Compile the program and link to the module.

```
gfortran -o main main.f90 my_module.o
```

You can see this will get tedious if we have many modules. Later today we will look at Makefiles, the standard way to deal with that problem.

Automatic interfaces

We didn't have to tell the main program anything about the procedures in the module. The `.mod` file does that for us.

This is true even for the functions; there is **no need for "external" function type declarations to be repeated in the main program**.

You can try this for yourself — add a function to the module that returns the value of the private variable `print_matrix_n_calls`.

Interfaces

In Python, everything is an object. Functions don't care what type their arguments are (maybe in some cases they *should* care, but that is an advanced topic — see PEP 484).

For example, we can pass an array of any rank and check its dimensions *inside* the function, if we need that information.

In Fortran, it seems we don't have that luxury: we **have to** declare the type for every argument. If an argument is an array, we have to say what its rank is (the *size* can be flexible, but not the rank).

What if we wanted our "print_matrix" function to also print vectors?

One solution is to write two functions: "`print_matrix`" and "`print_vector`". But what if we had to deal with matrices and vectors of `integer` and `real(8)` as well as `real`?

Interfaces

An interface block defines a function name and list of procedures in the module that implement this name for different combinations of arguments and return types.

It is possible to write interfaces for each function by hand.

That might be needed if you want to do something clever, but this module procedure way is much easier for simple situations.

```
module ...

public print_matrix

INTERFACE print_matrix
  MODULE PROCEDURE print_matrix_rank_1
  MODULE PROCEDURE print_matrix_rank_2
END INTERFACE print_matrix

...

contains

...
```

The contains block

The `contains` statement divides the code part of module files from the "header" part.

In a module, the statements **before** the `contains` specify interfaces for groups of procedures. They can also declare module-level variables.

The contains block can also appear in the main program, where it does more-or-less the same job.

Fundamentally, the program block is almost the same thing as a module. Putting functions in the main program's contains block rather than having them floating around after "end program" can help to avoid lots of external function declarations.

Summary: modules

Use modules to organise large Fortran programs, “just like” python modules.

`module use` statements come first, before `implicit none`.

Use interface blocks with the `module procedure` keyword to create groups of functions doing the same job with different signatures (type, order, and number of arguments, and return values).

Modules have to be compiled into object files first, and then linked to the main program.

ASTR 660 / Class 7

Linking to external libraries

Photo: 陳承光

Black Drongo
大卷尾

Crested Serpent-Eagle
大冠鷲

Linking to third-party libraries

You will often need to link with libraries that you did not write yourself. You may not have their source files, only the compiled versions.

You need to tell the compiler where to look for those libraries ("library directories") and their associated header files ("include directories").

This is usually done by adding flags like "`-L/path/to/somelibrary`" and "`-I/path/to/somelibrary_headers`", along with linker flags like "`-lsomelibrary`".

There are many common conventions. By default, libraries are installed in `/usr/local/lib/somelibrary`, headers in `/usr/local/include/somelibrary`, and the name of the library is `liblibrary.so`.

Linker environment variables

Shell environment variables can be used to pass options to the linker.

This is convenient, especially combined with some standard conventions and a build system like GNU make (see later).

The locations of libraries might be different on different machines.

External libraries example #1

The equivalent of the `np.linalg.solve` in Fortran is the `SGESV` routine from the LAPACK library:

```
external :: sgesv
…
! call sgesv(n, nrhs, a, lda, ipiv, b, ldb, info)
! This example is for A(N_grid,N_grid), B(N_grid)
Call SGESV(N_grid, 1, A, N_grid, pivot, B, N_grid, info)
```

As well as the arrays `A` and `B`, the standard LAPACK routines need to be given their sizes (`N_grid` in this case) and the size of the output (`nrhs`). We also have to pass arrays to hold the output (`pivot` and `info`). The **actual** output array `X` overwrites `B`! All this is for backwards compatibility.

But how can we get our code to see this subroutine? Where does it come from?

External libraries example #1

```
gfortran -o finite_diff finite_diff.f90 -L/opt/homebrew/opt/lapack/lib -llapack
```

We provide a library path using `-L` and a linker command `-llapack` to include the compiled LAPACK library.

The compiler will expect to find "liblapack.a" (a static library), "liblapack.so" (a dynamic library) or similar in one of the paths given with `-L`.

On CICA, the easiest way to get LAPACK functions is via the OpenBLAS library:

```
module load gcc openblas

gfortran -o finite_diff finite_diff.f90 -L$OPENBLAS_PATH/lib -lopenblas
```

External libraries example #2: HDF5

```
> module load gcc hdf5

> module list
Currently Loaded Modulefiles:
 1) gcc/8.3.0(default)   2) flavor/hdf5/serial   3) hdf5/1.10.5(default)

> module show hdf5
-------------------------------------------------------------------
/cluster/software/modulefiles/hdf5/1.10.5:

conflict        hdf5
module          load flavor/hdf5/serial
prereq          flavor/hdf5/serial flavor/hdf5/serial_api_v18
prereq          gcc/8.3.0
setenv          HDF5_ROOT /cluster/software/hdf5/1.10.5/gcc--8.3.0/serial
prepend-path    PATH /cluster/software/hdf5/1.10.5/gcc--8.3.0/serial/bin
prepend-path    LD_RUN_PATH /cluster/software/hdf5/1.10.5/gcc--8.3.0/serial/lib/
prepend-path    --delim { } LDFLAGS -Wl,-rpath=/cluster/software/hdf5/1.10.5/gcc--8.3.0/serial/lib/
prepend-path    --delim { } LDFLAGS -L/cluster/software/hdf5/1.10.5/gcc--8.3.0/serial/lib/
prepend-path    LIBRARY_PATH /cluster/software/hdf5/1.10.5/gcc--8.3.0/serial/lib/
prepend-path    --delim { } F90_MODULE_FLAGS -I/cluster/software/hdf5/1.10.5/gcc--8.3.0/serial/include
```

External libraries example #2: HDF5

```
program test_hdf5
  use hdf5
  implicit none

  integer :: h5_err
  integer :: majnum, minnum, relnum

  call h5open_f(h5_err)
  call h5get_libversion_f(majnum, minnum, relnum, h5_err)
  write(*,'(a,1x,i0.2,".",i0.2,".",i0.2)') 'Using HDF5', majnum, minnum, relnum

end program test_hdf5
```

External libraries example #2: HDF5

In a bash shell:

```
> gfortran $F90_MODULE_FLAGS $LDFLAGS -lhdf5_fortran -o test_hdf5 test_hdf5.f90
```

Why adding `-lhdf5_fortran`? It turns out only `-lhdf5` is added to the linker flags when you load the HDF5 module, but this is the C version of the library. I had to figure this out by looking in the HDF5 library path that the -L flag was pointing to, and noticing that there was a `libhd5_fortran.so` library.

zsh:

```
> gfortran ${=F90_MODULE_FLAGS} $LDFLAGS -lhdf5_fortran -o test_hdf5 test_hdf5.f90
```

ASTR 660 / Class 7

Makefiles

R. Foster
@TaiwanBirding

Taiwan Barbet
五色鳥

Makefiles

The “make” tool is the standard, old-fashioned **build system**: a system for organising all the steps required to make an executable out of a set of source files that depend on each other in some complicated way.

Makefiles (the input to make) are a special programming (scripting) language, designed to do one specific job (called a **domain-specific language**).

There are lots of alternatives to using make, just like there are lots of alternatives to writing programs in `vim`.

We will only look at the basic idea of make and Makefiles.

`https://fortran-lang.org/en/learn/building_programs/build_tools/`

`https://makefiletutorial.com/`

A simple makefile

The Makefile specifies tasks (targets) to make, and rules to make them.

This file has 4 targets.

The make program can be run starting from any one of these targets.

```
all: main

main: my_module.o
  gfortran -o main main.f90 my_module.o

my_module.o:
  gfortran -c -o my_module.o my_module.f90

clean:
  rm main *.o *.mod
```

A simple makefile

Targets depend on other targets. To make “all”, we first have to make “main”.

To make “main”, we can run gfortran. But first, we have to make `my_module.o`.

The `my_module.o` target has no dependencies. Make it by running `gfortran`.

The “clean” target doesn’t depend on anything. It just deletes everything made by the other steps.

```
all: main

main: my_module.o
  gfortran -o main main.f90 my_module.o

my_module.o:
  gfortran -c -o my_module.o my_module.f90

clean:
  rm main *.o *.mod
```

Indents in Makefiles must be tab characters

Indents in Makefiles are important (like Python) and they **must be made with tabs**, not multiple spaces (*unlike* Python).

INDENTS IN MAKEFILES MUST BE TABS!

INDENTS IN MAKEFILES MUST BE TABS!

INDENTS IN MAKEFILES MUST BE TABS!

```
all: main

main: my_module.o
	gfortran -o main main.f90 my_module.o

my_module.o:
	gfortran -c -o my_module.o my_module.f90

clean:
	rm main *.o *.mod
```

Running make

If we put targets in a file called “Makefile”, they will be found automatically by make. (Otherwise, `make -f makefile_name`).

```
> make all

gfortran -c -o my_module.o my_module.f90

gfortran -o main main.f90 my_module.o
```

Make prints each command it runs.

```
> make clean

rm main *.o *.mod
```

Makefile variables and patterns

The simple Makefile on the previous slides specifies the files involved in each task by hand.

The real point the Makefile language is to **avoid repetition** by using variables and pattern-matching.

Variables are defined with `NAME := something` and referenced with `$(NAME)`.

The pattern-matching syntax is rather obscure. $@ means the file name of the target, $< means the first file required to build the target, and so on.

```
OBJS := my_module.o
PROG := main

all: $(PROG)

$(PROG): $(OBJS)
  gfortran -o $@ $^

$(OBJS): %.o: %.f90
  gfortran -c -o $@ $<
```

https://www.gnu.org/software/make/manual/html_node/Quick-Reference.html

Makefile patterns, wildcards, substitution

The Makefile language makes several questionable design choices, especially around all the substitution and pattern matching stuff.

The various ways * and % can work are very hard to explain, in my opinion.

https://makefiletutorial.com/

Makefile example

A common way to proceed is to copy an example Makefile and adapt it for your project.

Here is the example from the tutorial on `fortran-lang.org`.

Of course, some parts would need to be modified to use this: for example, the source files and program name, and the compiler options.

Copy and pasting Makefiles?
Watch out for the tabs!

```
# Disable all of make's built-in rules (similar to Fortran's implicit none)
MAKEFLAGS += --no-builtin-rules --no-builtin-variables
# configuration
FC := gfortran
LD := $(FC)
RM := rm -f
# list of all source files
SRCS := tabulate.f90 functions.f90
PROG := my_prog

OBJS := $(addsuffix .o, $(SRCS))

.PHONY: all clean
all: $(PROG)

$(PROG): $(OBJS)
   $(LD) -o $@ $^

$(OBJS): %.o: %
   $(FC) -c -o $@ $<

# define dependencies between object files
tabulate.f90.o: functions.f90.o user_functions.mod

# rebuild all object files in case this Makefile changes
$(OBJS): $(MAKEFILE_LIST)

clean:
   $(RM) $(filter %.o, $(OBJS)) $(wildcard *.mod) $(PROG)
```

Makefile example

Here I've modified the previous example:

I add a new target (which prints the relevant environment variables).

I add the environment variables to the command lines.

I add the "_EXTRA" makefile variables to add to the environment variables.

This is all a bit fragile — if in doubt, just eliminate all the shell environment variables by hardcoding the paths into the makefile, and worry about it later if you ever need to compile your code on a different machine..

```
# Disable all of make's built-in rules (similar to Fortran's implicit none)
MAKEFLAGS += --no-builtin-rules --no-builtin-variables
# configuration
FC := gfortran
FCFLAGS_EXTRA := -g
LD := $(FC)
LDFLAGS_EXTRA := -lhdf5_fortran
RM := rm -f
# list of all source files
SRCS := test_hdf5.f90
PROG := test_hdf5

OBJS := $(addsuffix .o, $(SRCS))

.PHONY: all clean showenv
all: $(PROG)

# The @ stops the echo commands themselves being printed
showenv:
	@echo
	@echo FCFLAGS=${FCFLAGS}
	@echo LDFLAGS=${LDFLAGS}
	@echo F90_MODULE_DIR=${F90_MODULE_FLAGS}
	@echo

$(PROG): $(OBJS)
	$(LD) ${LDFLAGS} $(LDFLAGS_EXTRA) -o $@ $^

$(OBJS): %.o: %
	$(FC) ${FCFLAGS} ${FCFLAGS_EXTRA} ${F90_MODULE_FLAGS} -c -o $@ $<

# rebuild all object files in case this Makefile changes
$(OBJS): $(MAKEFILE_LIST)

clean:
	$(RM) $(filter %.o, $(OBJS)) $(wildcard *.mod) $(PROG)
```

Other build systems

Higher-level build systems exist, partly to write makefiles automatically (!) for large projects. These systems are usually overkill for small projects.

A widely-used example is CMake (`module load cmake` on CICA).

A closely related problem when distributing software to other people is how to set options that change deepening on the environment in which the software is installed.

> *For example, options that depend on if and where certain other libraries are installed, and how those libraries were built; or on the type of processor.*

In the end, the output of most of these tools is some sort of automatically generated Makefile. The idea is that you change options in the Makefile via the build system not by hand.

Compiling other people's code

The industry standard for this configuration process is the GNU **autoconf** tool. The SOP for building well-constructed third party software is:

```
./configure –-prefix=/path/for/your/installation # default: /usr/local

make # compiles the code

make install # copies the compiled binaries to ${PREFIX}
```

Always look for a README or INSTALL file first.

Summary: makefiles

Make is a tool for building (compiling and linking) complex programs.

Makefiles are programs written in a “domain-specific” language that are interpreted by the “make” command.

The Makefile language can be a pain, but so far none of the alternatives have replaced it.

Getting most astrophysical software running will require fighting with a Makefile to configure the software for your system, find external libraries etc.

Who makes the makefile? If the software you want to build has been written by scientists, there is a good chance the process involves editing a Makefile by hand.