Introduction to Modern Fortran

Advanced Use Of Procedures

Nick Maclaren

nmm1@cam.ac.uk

March 2014

Summary

We have omitted some important concepts
They are complicated and confusing

There are a lot of features we have omitted
Mostly because they are hard to use correctly
And sometimes because they are inefficient

This lecture covers some of the most important

- Refer to this when you need to

ALLOCATABLE and POINTER

You can pass ALLOCATABLE and POINTER arrays
In the usual case, the procedure has neither
The dummy argument is associated with the data

- You can't reallocate or redirect in the procedure

To do that, declare the dummy argument as
ALLOCATABLE or POINTER, as appropriate

Warning for INTENT(OUT) and ALLOCATABLE:
These are deallocated on entry, even if not used

Association (1)

Fortran uses argument association in calls
Dummy arguments refer to the actual ones

- You don't need to know exactly how it is done

It may be aliasing or copy–in/copy–out

Expressions are stored in a hidden variable
The dummy argument is associated with that

- It obviously must not be updated in any way

Using INTENT is strongly recommended

Association (2)

```
REAL, DIMENSION(1:10, 1:20, 1:3) :: data
CALL Fred (data(:, 5:15, 2), 1.23*xyz )

SUBROUTINE Fred (array, value)
REAL, DIMENSION(:, :) :: array
REAL, INTENT(IN) :: value
```

array in fred refers to data(:, 5:15, 2)
value refers to a location containing 1.23*xyz

Updating Arguments (1)

A dummy argument must not be updated if:

- The actual argument is an expression
- It overlaps another argument in any way

```
REAL, DIMENSION(1:20, 1:3) :: data
CALL Fred (data(5:15, 2), data(17:, 2))

SUBROUTINE Fred (arr1, arr2)
REAL, DIMENSION(:) :: arr1, arr2
arr1 = 1.23 ;    arr2 = 4.56
```

- The above works as you expect

Updating Arguments (2)

```
REAL, DIMENSION(1:20, 1:3) :: data
CALL Fred (data(5:15, 2), data(1:10, 2))

SUBROUTINE Fred (arr1, arr2)
REAL, DIMENSION(:) :: arr1, arr2
arr2(1, 1) = 4.56
```

- The above is not allowed

Because arr1 and arr2 overlap

Even though arr2(1, 1) is not part of arr1

Updating Arguments (3)

```
REAL :: X
CALL Fred (X + 0.0)

SUBROUTINE Fred (Y)
Y = 4.56
```

- The above is not allowed – obviously

- That also applies to array expressions

Vector indexing behaves like an expression

Warning for C/C++ People

```
REAL, DIMENSION(1:20) :: data
CALL Fred (data(2), data)

SUBROUTINE Fred (var, array)
REAL :: var
REAL, DIMENSION(:) :: array
array = 4.56
```

- The above is not allowed, either

Even array elements are associated

Using Functions

Functions are called just like built–in ones
They may be optimised in similar ways

```
REAL :: scale, data(1000)
      . . .
READ *, scale    ! assume that this reads 0.0
Z = Variance(data)/(scale+Variance(data)}
```

Variance may be called 0, 1 or 2 times

Impure Functions

Pure functions have defined behaviour

- Whether they are declared PURE or not

Impure functions occasionally misbehave
Generally, because they are over–optimised

There are rules for safety in practice
But they are too complicated for this course

- Ask if you need help with this

FUNCTION Result Variable

The function name defines the result variable
You can change this if you prefer

```
FUNCTION Variance_of_an_array (Array) RESULT(var)
    REAL :: var
    REAL, INTENT(IN), DIMENSION(:) :: Array
    var = SUM(Array)/SIZE(Array)
    var = SUM((Array-var)**2)/SIZE(Array)
END FUNCTION Variance_of_an_array

    REAL, DIMENSION(1000) :: data
        . . .
    Z = Variance_of_an_array(data)
```

PURE Subroutines

You can declare a subroutine to be PURE

Like functions, but with one fewer restriction
INTENT(OUT) and INTENT(INOUT) are allowed

```
PURE SUBROUTINE Init (array, value)
    REAL, DIMENSION(:), INTENT(OUT) :: array
    REAL, INTENT(IN) :: value
    array = value
END SUBROUTINE Init
```

They can be declared as ELEMENTAL, too

Recursion

Fortran 90 allowed this for the first time
Recursive procedures must be declared as such

- If you don't, recursion may cause chaos

```
RECURSIVE SUBROUTINE Chop (array, value)
    ...
```

- Avoid it unless you actually need it
- Check all procedures in the recursive loop

OPTIONAL Arguments

- Use OPTIONAL for setting defaults only
On entry, check and copy ALL args
Use ONLY local copies thereafter
Now, all variables are well defined when used

- Can do the converse for optional results
Just before returning, check and copy back

- Beyond this should be done only by experts

OPTIONAL Example (1)

```
FUNCTION fred (alf, bert)
REAL :: fred, alf, mybert
REAL, OPTIONAL, INTENT(IN) :: bert
IF (PRESENT(bert)) THEN
    mybert = bert
ELSE
    mybert = 0.0
END IF
```

Now use mybert in rest of procedure

OPTIONAL Example (2)

```
SUBROUTINE fred (alf, bert)
REAL :: alf
REAL, OPTIONAL, INTENT(OUT) :: bert
...
IF (PRESENT(bert)) bert = ...
END SUBROUTINE fred
```

Fortran 2003

Adds potentially useful VALUE attribute
See OldFortran course for information
Seriously. It's also useful for conversion

And the PROCEDURE declaration statement
A cleaner and more modern form of EXTERNAL
Its usage is not what you would expect, though

And probably more ...

Arrays and CHARACTER

We have over–simplified these so far
No problem, if you use only recommended style

- You need to know more if you go beyond that

- We start by describing what you can do
Including some warnings about efficient use

And then continue with how it actually works

Array Valued Functions

Arrays are first–class objects in Fortran
Functions can return array results

- In practice, doing so always needs a copy

However, don't worry too much about this

Declare the function just as for an argument
The constraints on the shape are similar

- If it is too slow, ask for advice

Example

This is a bit futile, but shows what can be done

```
FUNCTION operate (mat1, mat2, mat3)
    IMPLICIT NONE
    REAL, DIMENSION(:, :), INTENT(IN) :: &
        mat1, mat2, mat3
    REAL, DIMENSION(UBOUND(mat1, 1), &
        UBOUND(mat2, 2)) :: operate
! Checking omitted, again
    operate = MATMUL(mat1, mat2) + mat3
END FUNCTION operate
```

Array Functions and Copying

The result need not be copied on return
The interface provides enough information
In practice, don't bet on it ...

Array functions can also fragment memory
Ask if you want to know how and why

- Generally a problem only for HPC

I.e. when either time or memory are bottlenecks

What Can Be Done

- Just use array functions regardless
If you don't have a problem, why worry?

- Time and profile your program
Tune only code that is a bottleneck

- Rewrite array functions as subroutines
I.e. turn the result into an argument

- Use ALLOCATABLE results (sic)

- Ask for further advice with tuning

CHARACTER And Copying

In this respect, CHARACTER $\equiv$ array
Most remarks about arrays apply, unchanged

- But it is only rarely important

Fortran is rarely used for heavy character work
It works fairly well, but it isn't ideally suited
Most people find it very tedious for that

- If you need to, ask for advice

Character Valued Functions (1)

Earlier, we considered just one form
Almost anything more needs a copy
Some compilers will copy even those

- Often, the cost of that does not matter

You are not restricted to just that form
Declare the function just as for an argument
The constraints on the shape are similar

- If it is too slow, ask for advice

Character Valued Functions (2)

The result length can be taken from an argument

```
FUNCTION reverse_word (word)
    IMPLICIT NONE
    CHARACTER(LEN=*), INTENT(IN) :: word
    CHARACTER(LEN=LEN(word)) :: reverse_word
    INTEGER :: I, N
    N = LEN(word)
    DO I = 1, N
        reverse_word(I:I) = word(N+1-I:N+1-I)
    END DO
END FUNCTION reverse_word
```

Character Valued Functions (3)

This is a bit futile, but shows what can be done
The result length is a non–trivial expression

```
FUNCTION interleave (text1, count, text2)
    IMPLICIT NONE
   CHARACTER(LEN=*), INTENT(IN) :: text1, text2
    INTEGER, INTENT(IN) :: count
    CHARACTER(LEN=LEN(text1)+count+ &
        LEN(text2)) :: interleave
    interleave = text1 // REPEAT(' ', count) // text2
END FUNCTION interleave
```

Explicit/Assumed Size/Shape (1)

- The good news is that everything works

Can mix assumed and explicit *ad lib.*

There are some potential performance problems

- Passing assumed to explicit forces a copy

- It can be a problem calling some libraries

Especially ones written in old Fortran

- Write clean code, and see if it is fast enough

If you find that it isn't, ask for advice

Explicit/Assumed Size/Shape (2)

This code is not a problem:

```
SUBROUTINE Weeble (matrix)
    REAL, DIMENSION(:, :) :: matrix
END SUBROUTINE Weeble

SUBROUTINE Burble (space, M, N)
    REAL, DIMENSION(M, N) :: space
    CALL Weeble(space)
END SUBROUTINE Burble

REAL, DIMENSION(100,200) :: work
CALL Burble(work, 100, 200)
```

Explicit/Assumed Size/Shape (3)

Nor even something as extreme as this:

```
SUBROUTINE Weeble (matrix)
    REAL, DIMENSION(:, :) :: matrix
END SUBROUTINE Weeble

SUBROUTINE Burble (space, N, J1, K1, J2, K2)
    REAL, DIMENSION(N, *) :: space
    CALL Weeble(space(J1:K1, J2:K2))
END SUBROUTINE Burble

REAL, DIMENSION(100, 200) :: work
CALL Burble(work, 100, 20, 80, 30, 70)
```

Explicit/Assumed Size/Shape (4)

But this code forces a copy:

```
SUBROUTINE Bubble (matrix, M, N)
    REAL, DIMENSION(M, N) :: matrix
END SUBROUTINE Bubble

SUBROUTINE Womble (space)
    REAL, DIMENSION(:, :) :: space
    CALL Bubble(space, UBOUND(space, 1), &
        UBOUND(space, 2))
END SUBROUTINE Womble

REAL, DIMENSION(100,200) :: work
CALL Womble(work)
```

Example – Calling LAPACK

LAPACK is written in Fortran 77
It cannot handle assumed shape arrays
So here is how to call SPOTRF (Cholesky)

```
SUBROUTINE Chol (matrix, info)
    REAL, DIMENSION(:, :), INTENT(INOUT) :: matrix
        INTEGER, INTENT(INOUT) :: info
        CALL SPOTRF('L', UBOUND(matrix, 1), &
                matrix, UBOUND(matrix, 1), info)
END SUBROUTINE Chol
```

matrix will be copied on call and return

Sequence Association (1)

Have covered assumed shape and char. length
And explicit shape and char. length
but only when the dummy and actual match

- That constraint is not required (nor checked)

You need to know an extra concept to go further
That is called sequence association

- You are recommended to go cautiously here

Don't do it until you are confident with Fortran

Sequence Association (2)

Explicit shape and assumed size arrays only
If the dummy and actual bounds do not match

Argument is flattened in array element order
And is given a shape by the dummy bounds
Exactly the way the RESHAPE intrinsic works

There are important uses of this technique

- Or you can shoot yourself in the foot

Example

```
SUBROUTINE operate_1 (vector, N)
    REAL, DIMENSION(N) :: vector
    . . .
SUBROUTINE operate_2 (matrix, M, N)
    REAL, DIMENSION(M, N) :: matrix
    . . .

REAL, DIMENSION(1000000) :: workspace
. . .
IF (cols = 0) THEN
  CALL operate_1(workspace, rows)
ELSE
  CALL operate_2(workspace, rows, cols)
END IF
```

Sequence Association (3)

The same holds for explicit length CHARACTER
Everything is concatenated and then reshaped

Character lengths are like an extra dimension
Naturally, it varies faster than the first index

One restriction needed to make this work
Assumed shape arrays of CHARACTER
need assumed length or matching lengths

Example

```
SUBROUTINE operate (fields, N)
    CHARACTER(LEN=8), DIMENSION(10, N) :: fields
END SUBROUTINE operate

CHARACTER(LEN=80), DIMENSION(1000) :: lines
. . .
! Read in N lines
CALL operate(lines, N)
```

Implicit Interfaces (1)

Calling an undeclared procedure is allowed
The actual arguments define the interface

- I strongly recommend not doing this

Mistyped array names often show up as link errors

```
REAL, DIMENSION(1000) :: lines
. . .
lines(5) = lones(7)
```

Undefined symbol lones_ in file test.o

Implicit Interfaces (2)

Only Fortran 77 interface features can be used
The args and result must be exactly right
Must declare the result type of functions

```
REAL, DIMENSION(KIND=dp) :: DDOT
. . .
X = DDOT(array)
```

- This is commonly done for external libraries

I.e. ones that are written in Fortran 77, C etc.

- Interface modules are a better way

EXTERNAL

This declares an external procedure name

It's essential only when passing as argument
I.e. if the procedure name is used but not called

- I recommend it for all undeclared procedures

More as a form of documentation than anything else

- But explicit interfaces are always better

Example

Here is the LAPACK example again

```
SUBROUTINE Chol (matrix, info)
  REAL, DIMENSION(:, :), INTENT(INOUT) :: matrix
    INTEGER, INTENT(INOUT) :: info
    EXTERNAL :: SPOTRF
    CALL SPOTRF('L', UBOUND(matrix, 1), &
        matrix, UBOUND(matrix, 1), info)
END SUBROUTINE Chol
```