Introduction to Modern Fortran

Procedures

Nick Maclaren

Computing Service

nmm1@cam.ac.uk, ext. 34761

November 2007
Sub-Dividing The Problem

• Most programs are thousands of lines
  Few people can grasp all the details

• You often use similar code in several places

• You often want to test parts of the code

• Designs often break up naturally into steps

Hence, all sane programmers use procedures
What Fortran Provides

There must be a single main program
There are subroutines and functions
All are collectively called procedures

A subroutine is some out-of-line code
There are very few restrictions on what it can do
It is always called exactly where it is coded

A function’s purpose is to return a result
There are some restrictions on what it can do
It is called only when its result is needed
Example – Cholesky (1)

We saw this when considering arrays. It is a very typical, simple subroutine.

```
SUBROUTINE CHOLESKY (A)
    IMPLICIT NONE
    INTEGER :: J, N
    REAL :: A(:, :), X
    N = UBOUND(A, 1)
    DO J = 1, N
        . . .
    END DO
END SUBROUTINE CHOLESKY
```
Example – Cholesky (2)

And this is how it is **called**

```fortran
PROGRAM MAIN
    IMPLICIT NONE
    REAL, DIMENSION(5, 5) :: A = 0.0
    REAL, DIMENSION(5) :: Z
    
    CALL CHOLESKY (A)

END PROGRAM MAIN
```

We shall see how to declare it later
Example – Variance

FUNCTION Variance (Array)
    IMPLICIT NONE
    REAL :: Variance, X
    REAL, INTENT(IN), DIMENSION(:) :: Array
    X = SUM(Array)/SIZE(Array)
    Variance = SUM((Array–X)**2)/SIZE(Array)
END FUNCTION Variance

REAL, DIMENSION(1000) :: data
    
    Z = Variance(data)

We shall see how to declare it later
Example – Sorting (1)

This was the harness of the selection sort
Replace the actual sorting code by a call

PROGRAM sort10
  IMPLICIT NONE
  INTEGER, DIMENSION(1:10) :: nums
  
  ! --- Sort the numbers into ascending order of magnitude
  CALL SORTIT (nums)
  ! --- Write out the sorted list

END PROGRAM sort10
Example – Sorting (2)

SUBROUTINE SORTIT (array)
  IMPLICIT NONE
  INTEGER :: temp, array(:), J, K
L1:   DO J = 1, UBOUND(array,1)-1
L2:    DO K = J+1, UBOUND(array,1)
      IF(array(J) > array(K)) THEN
        temp = array(K)
        array(K) = array(J)
        array(J) = temp
      END IF
    END DO L2
  END DO L1
END SUBROUTINE SORTIT
CALL Statement

The CALL statement evaluates its arguments. The following is an over-simplified description:

- Variables and array sections define memory
- Expressions are stored in a hidden variable

It then transfers control to the subroutine. Passing the locations of the actual arguments.

Upon return, the next statement is executed.
SUBROUTINE Statement

Declares the **procedure** and its **arguments**
These are called **dummy arguments** in Fortran

The subroutine’s **interface** is defined by:
• The **SUBROUTINE** statement itself
• The **declarations** of its **dummy arguments**
• And anything that those use (see later)

**SUBROUTINE SORTIT** (array)
**INTEGER** :: [ temp, ] array(:) [ , J, K ]
Statement Order

A **SUBROUTINE** statement starts a subroutine
Any **USE** statements must come next
Then **IMPLICIT NONE**
Then the rest of the **declarations**
Then the **executable statements**
It ends at an **END SUBROUTINE** statement

**PROGRAM** and **FUNCTION** are similar

There are other rules, but you may ignore them
Dummy Arguments

• Their *names* exist only in the *procedure*
  They are declared much like *local variables*  

  Any *actual argument* names are irrelevant
  Or any other names outside the *procedure*

• The *dummy arguments* are *associated*
  with the *actual arguments*  

  Think of *association* as a bit like *aliasing*
Argument Matching

**Dummy** and **actual argument** lists must match
The **number** of arguments must be the same
Each argument must match in **type** and **rank**

That can be relaxed in several ways
See under **advanced** use of procedures

We shall come back to **array arguments** shortly
Most of the complexities involve them
This is for compatibility with old standards
Functions

Often the required result is a single value
It is easier to write a **FUNCTION** subprogram

E.g. to find the largest of three values:
- Find the largest of the first and second
- Find the largest of that and the third

Yes, I know that the **MAX** function does this!

The **function name** defines a **local variable**
- Its value on **return** is the result returned
The **RETURN** statement does not take a value
Example (1)

FUNCTION largest_of (first, second, third)
  IMPLICIT NONE
  INTEGER :: largest_of
  INTEGER :: first, second, third
  IF (first > second) THEN
    largest_of = first
  ELSE
    largest_of = second
  END IF
  IF (third > largest_of) largest_of = third
END FUNCTION largest_of
Example (2)

INTEGER :: trial1, trial2, trial3, total, count

total = 0 ;  count = 0

DO

    PRINT *, ’Type three trial values:’
    READ *, trial1, trial2, trial3
    IF (MIN(trial1, trial2, trial3) < 0) EXIT
        count = count + 1
        total = total + &
        largest_of(trial1, trial2, trial3)

END DO

PRINT *, ’Number of trial sets = ’, count, &
    ’ Total of best of 3 = ’, total
Warning: Time Warp

Unfortunately, we need to define a module. We shall cover those quite a lot later.

The one we shall define is trivial. Just use it, and don’t worry about the details.

Everything you need to know will be explained.
Using Modules (1)

This is how to compile procedures separately
First create a file (e.g. mymod.f90) like:

```fortran
MODULE mymod
CONTAINS
    FUNCTION Variance (Array)
    REAL :: Variance, X
    REAL, INTENT(IN), DIMENSION(:) :: Array
    X = SUM(Array)/SIZE(Array)
    Variance = SUM((Array-X)**2)/SIZE(Array)
    END FUNCTION Variance
END MODULE mymod
```
The module name need not be the file name.
Doing that is strongly recommended, though.

- You can include any number of procedures.

You now **compile** it, but don’t **link** it:

```
nagfor -C=all -c mymod.f90
```

It will create files like `mymod.mod` and `mymod.o`.
They contain the **interface** and the **code**.
Using Modules (3)

You use it in the following way

- You can use any number of modules

```fortran
PROGRAM main
  USE mymod
  REAL, DIMENSION(10) :: array
  PRINT *, 'Type 10 values'
  READ *, array
  PRINT *, 'Variance = ', Variance(array)
END PROGRAM main
```

```bash
nagfor -C=all -o main main.f90 mymod.o
```
Internal Procedures (1)

PROGRAM, SUBROUTINE or FUNCTION
Can use CONTAINS much like a module

Included procedures are internal subprograms
Most useful for small, private auxiliary ones
• You can include any number of procedures

Visible in the outer procedure only
Internal subprograms may not contain their own internal subprograms
PROGRAM main
   REAL, DIMENSION(10) :: vector
   PRINT *, 'Type 10 values'
   READ *, vector
   PRINT *, 'Variance = ', Variance(vector)
END PROGRAM main

CONTAINS
FUNCTION Variance (Array)
   REAL :: Variance, X
   REAL, INTENT(IN), DIMENSION(:) :: Array
   X = SUM(Array)/SIZE(Array)
   Variance = SUM((Array-X)**2)/SIZE(Array)
END FUNCTION Variance

END PROGRAM main
Internal Procedures (3)

Everything accessible in the *enclosing procedure* can also be used in the *internal procedure*

This includes *all* of the local declarations
And anything imported by *USE* (covered later)

Internal procedures need only a few arguments
Just the things that vary between calls
Everything else can be used directly
Internal Procedures (4)

A local name takes precedence

PROGRAM main
  REAL :: temp = 1.23
  CALL pete (4.56)
END PROGRAM main

CONTAINS
  SUBROUTINE pete (temp)
    PRINT *, temp
  END SUBROUTINE pete
END PROGRAM main

Will print 4.56, not 1.23
Avoid doing this – it’s very confusing
Using Procedures

Use either technique for solving test problems

- They are the best techniques for real code
  Simplest, and give full access to functionality
  We will cover some other ones later

- Note that, if a procedure is in a module
  it may still have internal subprograms
Example

MODULE mymod
CONTAINS
   SUBROUTINE Sorter (array, opts)
      ...
   CONTAINS
      FUNCTION Compare (value1, value2, flags)
      ...
      END FUNCTION Compare
   SUBROUTINE Swap (loc1, loc2)
      ...
      END FUNCTION Swap
   END SUBROUTINE Sorter
END MODULE mymod
INTENT (1)

You can make arguments *read–only*

```
SUBROUTINE Summarise (array, size)
    INTEGER, INTENT(IN) :: size
    REAL, DIMENSION(size) :: array
```

That will prevent you writing to it by accident
Or calling another procedure that does that
It may also help the compiler to optimise

- **Strongly recommended** for *read–only* args
INTENT (2)

You can also make them **write-only**
Less useful, but still very worthwhile

```fortran
SUBROUTINE Init (array, value)
IMPLICIT NONE
REAL, DIMENSION(:), INTENT(OUT) :: array
REAL, INTENT(IN) :: value
array = value
END SUBROUTINE Init
```

As useful for optimisation as **INTENT(IN)**
The default is effectively `INTENT(INOUT)`

- But specifying `INTENT(INOUT)` is useful
  It will trap the following nasty error

```fortran
SUBROUTINE Munge (value)
  REAL, INTENT(INOUT) :: value
  value = 100.0*value
  PRINT *, value
END SUBROUTINE Munge

CALL Munge(1.23)
```
Example

SUBROUTINE expsum(n, k, x, sum)

IMPLICIT NONE

INTEGER, INTENT(IN) :: n
REAL, INTENT(IN) :: k, x
REAL, INTENT(OUT) :: sum

INTEGER :: i

sum = 0.0

DO i = 1, n
    sum = sum + exp(-i*k*x)

END DO

END SUBROUTINE expsum
Aliasing

Two arguments may overlap only if read-only
Also applies to arguments and global data
• If either is updated, weird things happen

Fortran doesn’t have any way to trap that
Nor do any other current languages – sorry

Use of INTENT(IN) will stop it in many cases

• Be careful when using array arguments
Including using array elements as arguments
PURE Functions

You can declare a function to be PURE

All data arguments must specify INTENT(IN)
It must not modify any global data
It must not do I/O (except with internal files)
It must call only PURE procedures
Some restrictions on more advanced features

Generally overkill – but good practice
Most built-in procedures are PURE
Example

This is the cleanest way to define a function

```
PURE FUNCTION Variance (Array)  
  IMPLICIT NONE  
  REAL :: Variance, X  
  REAL, INTENT(IN), DIMENSION(:) :: Array  
  X = SUM(Array)/SIZE(Array)  
  Variance = SUM((Array-X)**2)/SIZE(Array)  
END FUNCTION Variance
```

Most safety, and best possible optimisation
ELEMENTAL Functions

Functions can be declared as ELEMENTAL
Like PURE, but arguments must be scalar

You can use them on arrays and in WHERE
They apply to each element, like built-in SIN

```
ELEMENTAL FUNCTION Scale (arg1, arg2)
  REAL, INTENT(IN) :: arg1, arg2
  Scale = arg1/sqrt(arg1**2+arg2**2)
END FUNCTION Scale

REAL, DIMENSION(100) :: arr1, arr2, array
array = Scale(arr1, arr2)
```
Keyword Arguments (1)

SUBROUTINE AXIS (X0, Y0, Length, Min, Max, Intervals)
   REAL, INTENT(IN) :: X0, Y0, Length, Min, Max
   INTEGER, INTENT(IN) :: Intervals
END SUBROUTINE AXIS

CALL AXIS(0.0, 0.0, 100.0, 0.1, 1.0, 10)

- Error prone to write and unclear to read

And it can be a lot worse than that!
Keyword Arguments (2)

Dummy arg. names can be used as keywords
You don’t have to remember their order

SUBROUTINE AXIS (X0, Y0, Length, Min, Max, Intervals)

CALL AXIS(Interval=10, Length=100.0, &
Min=0.1, Max=1.0, X0=0.0, Y0=0.0)

• The argument order now doesn’t matter
The keywords identify the dummy arguments
Keyword Arguments (3)

Keywords arguments can follow positional
The following is allowed

SUBROUTINE AXIS (X0, Y0, Length, Min, Max, Intervals)
    
CALL AXIS(0.0, 0.0, Intervals=10, Length=100.0, &
    Min=0.1, Max=1.0)

• Remember that the best code is the clearest
Use whichever convention feels most natural
Keyword Reminder

Keywords are not names in the calling procedure. They are used only to map to dummy arguments. The following works, but is somewhat confusing.

```fortran
SUBROUTINE Nuts (X, Y, Z)
    REAL, DIMENSION(:) :: X
    INTEGER :: Y, Z
END SUBROUTINE Nuts

INTEGER :: X
REAL, DIMENSION(100) :: Y, Z
CALL Nuts (Y=X, Z=1, X=Y)
```
Hiatus

That is most of the basics of procedures
Except for arrays and CHARACTER

Now might be a good time to do some examples
The first few questions cover the material so far
Assumed Shape Arrays (1)

- The best way to declare array arguments
  You must declare procedures as above

- Specify all bounds as simply a colon (‘:’) 
  The rank must match the actual argument 
  The lower bounds default to one (1) 
  The upper bounds are taken from the extents

  REAL, DIMENSION(:) :: vector  
  REAL, DIMENSION(:, :) :: matrix  
  REAL, DIMENSION(:, :, :) :: tensor
Example

SUBROUTINE Peculiar (vector, matrix)
  REAL, DIMENSION(:), INTENT(INOUT) :: vector
  REAL, DIMENSION(:, :) , INTENT(IN) :: matrix
  
END SUBROUTINE Peculiar

REAL, DIMENSION(1000), :: one
REAL, DIMENSION(100, 100) :: two
CALL Peculiar (one(101:160), two(21:, 26:75) )

vector will be DIMENSION(1:60)
matrix will be DIMENSION(1:80, 1:50)
Query functions were described earlier

**SIZE, SHAPE, LBOUND and UBOUND**

So you can write completely *generic* procedures

```fortran
SUBROUTINE Init (matrix, scale)
    REAL, DIMENSION(:, :), INTENT(OUT) :: matrix
    INTEGER, INTENT(IN) :: scale
    DO N = 1, UBOUND(matrix,2)
        DO M = 1, UBOUND(matrix,1)
            matrix(M, N) = scale*M + N
        END DO
    END DO
END DO
END SUBROUTINE Init
```

Cholesky Decomposition

SUBROUTINE CHOLESKY(A)
  IMPLICIT NONE
  INTEGER :: J, N
  REAL, INTENT(INOUT) :: A(:, :), X
  N = UBOUND(A, 1)
  IF (N < 1 .OR. UBOUND(A, 2) /= N)
    CALL Error("Invalid array passed to CHOLESKY")
  DO J = 1, N
    ...
  END DO
END SUBROUTINE CHOLESKY

Now I have added appropriate checking
Setting Lower Bounds

Even when using assumed shape arrays you can set any lower bounds you want

- You do that in the called procedure

```
SUBROUTINE Orrible (vector, matrix, n)
  REAL, DIMENSION(2*n+1:) :: vector
  REAL, DIMENSION(0:, 0:) :: matrix
  
END SUBROUTINE Orrible
```
Warning

Argument overlap will not be detected
Not even for assumed shape arrays

- A common cause of obscure errors

No other language does much better
Explicit Array Bounds

In **procedures**, they are more flexible
Any reasonable **integer expression** is allowed

Essentially, you can use any ordinary formula
Using only **constants** and **integer variables**
Few programmers will ever hit the restrictions

The most common use is for **workspace**
But it applies to all **array declarations**
Automatic Arrays (1)

Local arrays with run-time bounds are called automatic arrays

Bounds may be taken from an argument
Or a constant or variable in a module

SUBROUTINE aardvark (size)
USE sizemod  ! This defines worksize
INTEGER, INTENT(IN) :: size

REAL, DIMENSION(1:worksize) :: array_1
REAL, DIMENSION(1:size*(size+1)) :: array_2
Another very common use is a ‘shadow’ array i.e. one the same shape as an argument

SUBROUTINE pard (matrix)
REAL, DIMENSION(:, :) :: matrix

REAL, DIMENSION(UBOUND(matrix, 1), &
    UBOUND(matrix, 2)) :: &
    matrix_2, matrix_3

And so on – automatic arrays are very flexible
Explicit Shape Array Args (1)

We cover these because of their importance.
They were the only mechanism in Fortran 77.

- But, generally, they should be avoided.

In this form, all bounds are explicit.
They are declared just like automatic arrays.
The dummy should match the actual argument.
Making an error will usually cause chaos.

- Only the very simplest uses are covered.
There are more details in the extra slides.
Explicit Shape Array Args (2)

You can use constants

SUBROUTINE Orace (matrix, array)
    INTEGER, PARAMETER :: M = 5, N = 10
    REAL, DIMENSION(1:M, 1:N) :: matrix
    REAL, DIMENSION(1000) :: array
    ...
END SUBROUTINE Orace

INTEGER, PARAMETER :: M = 5, N = 10
REAL, DIMENSION(1:M, 1:N) :: table
REAL, DIMENSION(1000) :: workspace
CALL Orace(table, workspace)
Explicit Shape Array Args (3)

It is common to pass the **bounds** as **arguments**

```fortran
SUBROUTINE Weeble (matrix, m, n)  
  INTEGER, INTENT(IN) :: m, n  
  REAL, DIMENSION(1:m, 1:n) :: matrix  
  . . .
END SUBROUTINE Weeble
```

You can use **expressions**, of course

- But it is not really recommended

Purely on the grounds of human confusion
Explicit Shape Array Args (4)

You can define the **bounds** in a **module**
Either as a **constant** or in a **variable**

```fortran
SUBROUTINE Wobble (matrix)
    USE sizemod ! This defines m and n
    REAL, DIMENSION(1:m, 1:n) :: matrix
    ...
END SUBROUTINE Weeble
```

- The same remarks about **expressions** apply
Assumed Size Array Args

The last upper bound can be *
I.e. unknown, but assumed to be large enough

SUBROUTINE Weeble (matrix, n)
  REAL, DIMENSION(n, *) :: matrix
  
END SUBROUTINE Weeble

• You will see this, but generally avoid it
  It makes it very hard to locate bounds errors
  It also implies several restrictions
Warnings

The size of the dummy array must not exceed the size of the actual array argument

- Compilers will rarely detect this error

There are also some performance problems when passing assumed shape and array sections to explicit shape or assumed size dummies

That is in the advanced slides on procedures
Sorry – but it’s complicated to explain
Example (1)

We have a subroutine with an interface like:

```fortran
SUBROUTINE Normalise (array, size)
INTEGER, INTENT(IN) :: size
REAL, DIMENSION(size) :: array
```

The following calls are correct:

```fortran
REAL, DIMENSION(1:10) :: data
CALL Normalise (data, 10)
CALL Normalise (data(2:5), SIZE(data(2:5)))
CALL Normalise (data, 7)
```
Example (2)

SUBROUTINE Normalise (array, size)
INTEGER, INTENT(IN) :: size
REAL, DIMENSION(size) :: array

The following calls are not correct:

INTEGER, DIMENSION(1:10) :: indices
REAL :: var, data(10)

CALL Normalise (indices, 10)  ! wrong base type
CALL Normalise (var, 1)       ! not an array
CALL Normalise (data, 10.0)   ! wrong type
CALL Normalise (data, 20)     ! dummy array too big
Character Arguments

Few scientists do anything very fancy with these
See the advanced foils for anything like that

People often use a constant length
You can specify this as a digit string

Or define it using PARAMETER
That is best done in a module

Or define it as an assumed length argument
Explicit Length Character (1)

The **dummy** should match the **actual argument**
You are likely to get confused if it doesn’t

```
SUBROUTINE sorter (list)
    CHARACTER(LEN=8), DIMENSION(:) :: list
    ...
END FUNCTION sorter

CHARACTER(LEN=8) :: data(1000)
    ...
CALL sorter(data)
```
Explicit Length Character (2)

MODULE Constants
    INTEGER, PARAMETER :: charlen = 8
END MODULE Constants

SUBROUTINE sorter (list)
    USE Constants
    CHARACTER(LEN=charlen), DIMENSION(:, ::) :: list
    ...
END FUNCTION sorter

USE Constants
CHARACTER(LEN=charlen) :: data(1000)
CALL sorter(data)
Assumed Length CHARACTER

A CHARACTER length can be assumed. The length is taken from the actual argument.

You use an asterisk (*) for the length. It acts very like an assumed shape array.

Note that it is a property of the type. It is independent of any array dimensions.
Example (1)

FUNCTION is_palindrome (word)
   LOGICAL :: is_palindrome
   CHARACTER(LEN=*) , INTENT(IN) :: word
   INTEGER :: N, I
   is_palindrome = .False.
   N = LEN(word)
   comp: DO I = 1, (N−1)/2
      IF (word(I:I) /= word(N+1−I:N+1−I)) THEN
         RETURN
      END IF
   END DO comp
   is_palindrome = .True.
END FUNCTION is_palindrome
Example (2)

Such arguments do not have to be read-only

SUBROUTINE reverse_word (word)
  CHARACTER(LEN=*) , INTENT(INOUT) :: word
  CHARACTER(LEN=1) :: c
  N = LEN(word)
  DO I = 1, (N-1)/2
    c = word(I:I)
    word(I:I) = word(N+1-I:N+1-I)
    word(N+1-I:N+1-I) = c
  END DO
END SUBROUTINE reverse_word
Character Workspace

The rules are very similar to those for arrays. The length can be an almost arbitrary expression, but it usually just shadows an argument.

```
SUBROUTINE sort_words (words)
  CHARACTER(LEN=*) :: words(:)
  CHARACTER(LEN=LEN(words)) :: temp
  ...
END SUBROUTINE sort_words
```
Character Valued Functions

Functions can return CHARACTER values
Fixed-length ones are the simplest

FUNCTION truth (value)
  IMPLICIT NONE
  CHARACTER(LEN=8) :: truth
  LOGICAL, INTENT(IN) :: value
  IF (value) THEN
    truth = ’.True.’
  ELSE
    truth = ’.False.’
  END IF
END FUNCTION truth
Static Data

Sometimes you need to store values locally
Use a value in the next call of the procedure

- You do this with the **SAVE** attribute
  **Initialised variables** get that automatically
  It is good practice to specify it anyway

The best style avoids most such use
It can cause trouble with **parallel** programming
But it works, and lots of programs rely on it
Example

This is a futile example, but shows the feature

```
SUBROUTINE Factorial (result)
    IMPLICIT NONE
    REAL, INTENT(OUT) :: result
    REAL, SAVE :: mult = 1.0, value = 1.0
    mult = mult+1.0
    value = value*mult
    result = value
END SUBROUTINE Factorial
```
Warning

Omitting **SAVE** will usually **appear** to work
But even a new compiler **version** may break it
As will increasing the level of **optimisation**

- Decide which variables need it during **design**
- **Always** use **SAVE** if you want it
  And preferably never when you don’t!
- **Never** assume it without specifying it
Delayed Until Modules

Sometimes you need to share global data
It’s trivial, and can be done very cleanly

Procedures can be passed as arguments
This is a very important facility for some people
For historical reasons, this is a bit messy

- However, internal procedures can’t be
  Ask if you want to know why – it’s technical

We will cover both of these under modules
It just happens to be simplest that way!
Other Features

There is a lot that we haven’t covered
We will return to some of it later

• The above covers the absolute basics
  Plus some other features you need to know

• Be a bit cautious when using other features
  Some have been omitted because of “gotchas”

• And I have over-simplified a few areas
Extra Slides

Topics in the advanced slides on procedures

• Argument association and updating
• The semantics of function calls
• Optional arguments
• Array– and character–valued functions
• Mixing explicit and assumed shape arrays
• Array arguments and sequence association
• Miscellaneous other points