Subroutines, Functions and Modules

Subdividing the Problem

- Most problems are thousands of lines of code. Few people can grasp all of the details.
- You often use similar code in several places.
- You often want to test only parts of the code.
- Designs often break up naturally into steps.

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

function

- Purpose is to return a single result
- Invoked by inserting the function name
- It is called only when its result is needed

subroutine

- May or may not return result(s)
- Invoked with the CALL statement

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 declaration of its dummy arguments
- And anything that use those (see later)

```
SUBROUTINE Sortit(array)
INTEGER :: [temp, ] array(:) [, J, K]
```

Structure and Syntax

Subroutine syntax:

```
SUBROUTINE subroutine-name(arg1, arg2,...,argn)
  IMPLICIT NONE
   [specification part]
   [execution part]
END SUBROUTINE subroutine-name
```

If the subroutine does not require any arguments, the (arg1, arg2,...,argn) can be omitted.

Similar syntax is used for functions.

Subroutines with No Arguments

You aren't required to have any arguments

You can omit the parentheses if you prefer

Probably either do or don't, but you can mix uses

```
SUBROUTINE Joe ()
SUBROUTINE Joe

CALL Joe ()
CALL Joe
```

Example: sort3[a,b].f90

Dummy Arguments

also known as formal arguments

Their names exists 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

In general, dummy and actual argument lists must match

- The number of arguments must be the same
- Each argument must match in type and rank

These can be relaxed in some cases.

Most of the complexities involve array arguments

Functions (1)

Often the required result is a single value (or array)
In that case it makes more sense to write a function

Function syntax:

```
type FUNCTION funct-name(arg1,...,argn) [result
   return-value-name]
  IMPLICIT NONE
  [specification part]
  [execution part]
END FUNCTION funct-name
```

Functions (2)

- If a result variable is not specifically defined then the result is returned through the function name.
- The result variable must be declared in the function's specification area.
- You can optionally specify the type of the function:

```
REAL FUNCTION VARIANCE(array)
```

  - If this is done, no local declaration is needed.

Functions with No Arguments

You aren't required to have any arguments

You must **NOT** omit the parentheses

```
FUNCTION Fred ()
  INTEGER :: Fred

X = 1.23 * Fred()
CALL Alf ( Fred() )
```

Examples: variance.f90, series.f90

Usage

How do we incorporate subroutines and functions into our code?

1. Attach them to a main program as internal procedures using the CONTAINS statement
2. Include them in a MODULE (also with CONTAINS)

Legacy Fortran had to use external procedures. I will show you why these are inferior to internal procedures

Examples: variance.f90, series.f90

Internal Procedures (1)

For relatively small programs you can include procedures in the main program using CONTAINS

- You can include any number of procedures
- Visible to the outer program only
- These internal subprograms may **not** contain their own internal subprograms

Internal Procedures (2)

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

- All of the local declarations
- 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 (3)

A local name takes precedence

```
PROGRAM main
    REAL :: temp = 1.23
    CALL myval(4.56)
CONTAINS
    SUBROUTINE myval(temp)
        PRINT *, temp
    END SUBROUTINE myval
END PROGRAM main
```

This will print 4.56, not 1.23

Avoid doing this as it's very confusing

Internal vs External

Most compilers **cannot** check for argument list mismatches with external procedures, but they CAN perform this check for internal procedures.

- Used to be that all compilers did not have this capability, but now gfortran seems to be an exception.

Example: checkarg_int.f90, checkarg_ext.f90

Module Procedures

You can also place procedures in a module using a CONTAINS statement

- Module internal subprograms may contain their own internal subprograms
- Module name need not be the same as the file name but for large programs that is highly recommended
- Include the module with the USE statement

Example: checkarg_mod.f90

Intent (1)

You can make arguments read-only

```
SUBROUTINE Summarize(array, size)
  IMPLICIT NONE
  INTEGER, INTENT(IN) :: size
  REAL, DIMENSION(size) :: array
```

Will prevent you from writing to a variable by accident
Or calling another procedure that does that
May also help the compiler to optimize

Strongly recommended for read-only arguments

Intent (2)

You can also make arguments write-only
Less useful but still worthwhile

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

As useful for optimization as INTENT(IN)

Intent (3)

The default is effectively INTENT(INOUT)
Specifying it can be useful as it can catch certain errors

```
SUBROUTINE Mult100(value)
    REAL, INTENT(INOUT) :: value
    value = 100.0 * value
END SUBROUTINE Mult100

CALL Mult100(1.23)
```

This would be okay:

```
x = 1.23
CALL Mult100(x)
```

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
```

Keyword Arguments

Dummy argument names can be used as keywords
You don't have to remember their order

Keywords are NOT names in the calling procedure
They are only used to map dummy arguments

Example: series2.f90

Optional Arguments

Use OPTIONAL for setting defaults only

Check for existence using PRESENT function

Use only local copies thereafter

That way all variables will be well-defined when used

Example: series3.f90

Assumed Shape Arrays (1)

The best way to declare array arguments
Simply specify all bounds with 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
    …

PROGRAM main

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

In the second call vector will be dimensioned (1:60) and matrix will be dimensioned (1:80, 1:50)

Assumed Shape Arrays (2)

Array query functions were described earlier

SIZE, SHAPE, LBOUND, UBOUND

Gives the ability to write completely generic procedures

```
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
    ENDDO
END SUBROUTINE Init
```

Assumed Shape Arrays (3)

Assumed shape arrays work splendidly with internal procedures, but they will **NOT** work with external procedures without an INTERFACE block.

Examples: badpass.f90, goodpass1.f90

Setting Lower Bounds

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

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

Automatic Arrays (1)

Local arrays with bounds specified at run-time are called automatic arrays

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

```
SUBROUTINE aardvark (arrsize)
  USE sizemod    ! this defines the var "worksize"
  INTEGER, INTENT(IN) :: arrsize
  REAL, DIMENSION(1:worksize) :: array_1
  REAL, DIMENSION(1:arrsize*(arrsize+1)) :: array_2
```

Automatic Arrays (2)

Another very common use is a "shadow" array
i.e., one that is the same shape as an argument

```
SUBROUTINE swap_arrays (A, B)
  REAL, DIMENSION(:) :: A, B
  REAL, DIMENSION(SIZE(A)) :: temp

  temp = A ; A = B ; B = temp

END SUBROUTINE swap_arrays
```

Automatic Arrays (3)

Multi-dimensional example of the same concept:

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

Automatic arrays are very flexible.

Explicit Shape Array Args (1)

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

Explicit Shape Array Args (2)

You can use constants

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

INTEGER, PARAMETER :: M = 5, N = 10
REAL, DIMENSION(1:M,1:N) :: table
REAL, DIMENSION(1000) :: workspace

CALL expl_shape(table, workspace)
```

Explicit Shape Array Args (3)

It is common to pass the bounds as arguments

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

You can use expressions but it's not generally recommended

Assumed Size Array Arguments

The last upper bound can be *

```
SUBROUTINE oldschool (matrix, m)
  INTEGER, INTENT(IN) :: m
  REAL, DIMENSION(m,*) :: matrix
      ...
```

You may come across this but generally avoid it

It makes it very hard to locate bounds errors

WARNING

Argument overlap will **NOT** be detected
Not even if you turn on array-bounds checking
This is a common cause of obscure errors

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

Example: overlap.f90

Character Arguments

Few scientists do anything fancy with these

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

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 SUBROUTINE sorter

CHARACTER(LEN=8) :: 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)
     do i = 1,(n-1)/2
        if (word(i:i) /= word(n+1-i:n+1-i)) then
           RETURN
        endif
      enddo
      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
   ENDDO
END SUBROUTINE reverse_word
```

Static Data

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

You can do this with the SAVE attribute
Initialized variables get this automatically

The best style avoids this use.

Warning for C/C++ Users

Initialization in a declaration without SAVE initializes **only once!**

It does **NOT** reinitialize each time it is called

Do it with an explicit assignment statement

Example: localsave.f90, test_saves.f90