Lecture 5:

Procedures

Program Units

Fortran 90 has two main program units

- main `PROGRAM`,

  the place where execution begins and where control should eventually return before the program terminates. May contain procedures.

- `MODULE`.

  a program unit which can contain procedures and declarations. It is intended to be attached to any other program unit where the entities defined within it become accessible.

There are two types of procedures:

- `SUBROUTINE`,

  a parameterised named sequence of code which performs a specific task and can be invoked from within other program units.

- `FUNCTION`,

  as a `SUBROUTINE` but returns a result in the function name (of any specified type and kind).

Main Program Syntax

```
PROGRAM Main
 ! ...
CONTAINS ! Internal Procs
   SUBROUTINE Sub1(..)
    ! Executable stmts
   END SUBROUTINE Sub1
    ! etc.
   FUNCTION Funkyn(...)
    ! Executable stmts
   END FUNCTION Funkyn
 END PROGRAM Main
```

[`PROGRAM` [< *main program name* >]]
        < *declaration of local objects* >
                ...
        < *executable stmts* >
                ...
[`CONTAINS`
        < *internal procedure definitions* >]
`END` [`PROGRAM` [< *main program name* >]]

Program Example

```
PROGRAM Main
 IMPLICIT NONE
 REAL :: x
  READ*, x
  PRINT*, FLOOR(x) ! Intrinsic
  PRINT*, Negative(x)
CONTAINS
 REAL FUNCTION Negative(a)
  REAL, INTENT(IN) :: a
   Negative = -a
 END FUNCTION Negative
END PROGRAM Main
```

Subroutines

Consider the following example,

```
PROGRAM Thingy
 IMPLICIT NONE
 .....
  CALL OutputFigures(NumberSet)
 .....
CONTAINS
 SUBROUTINE OutputFigures(Numbers)
  REAL, DIMENSION(:), INTENT(IN) :: Numbers
   PRINT*, "Here are the figures", Numbers
 END SUBROUTINE OutputFigures
END PROGRAM Thingy
```

Internal subroutines lie between `CONTAINS` and `END PROGRAM` statements and have the following syntax

```
SUBROUTINE < procname >[ (< dummy args >) ]
       < declaration of dummy args >
       < declaration of local objects >
             ...
       < executable stmts >
END [ SUBROUTINE [< procname > ] ]
```

Note that, in the example, the `IMPLICIT NONE` statement applies to the whole program including the `SUBROUTINE`.

Functions

Consider the following example,

```
PROGRAM Thingy
 IMPLICIT NONE
 .....
  PRINT*, F(a,b)
 .....
CONTAINS
 REAL FUNCTION F(x,y)
  REAL, INTENT(IN) :: x,y
   F = SQRT(x*x + y*y)
 END FUNCTION F
END PROGRAM Thingy
```

Functions also lie between `CONTAINS` and `END PROGRAM` statements. They have the following syntax:

[< *prefix* >] `FUNCTION` < *procname* >([< *dummyargs* >])
        < *declaration of dummy args* >
        < *declaration of local objects* >
                ...
        < *executable stmts, assignment of result* >
`END` [`FUNCTION` [< *procname* >]]

It is also possible to declare the function type in the declarations area instead of in the header.

Argument Association

Recall, on the `SUBROUTINE` slide we had an invocation:

```
CALL OutputFigures(NumberSet)
```

and a declaration,

```
SUBROUTINE OutputFigures(Numbers)
```

`NumberSet` is an *actual argument* and is *argument associated* with the *dummy argument* `Numbers`.

For the above call, in `OutputFigures`, the name `Numbers` is an **alias** for `NumberSet`. Likewise, consider,

```
PRINT*, F(a,b)
```

and

```
REAL FUNCTION F(x,y)
```

The actual arguments `a` and `b` are associated with the dummy arguments `x` and `y`.

If the value of a dummy argument changes then so does the value of the actual argument.

Local Objects

In the following procedure

```
SUBROUTINE Madras(i,j)
 INTEGER, INTENT(IN) :: i, j
 REAL                :: a
 REAL, DIMENSION(i,j):: x
```

`a`, and `x` are know as *local objects*. They:

- are created each time a procedure is invoked,
- are destroyed when the procedure completes,
- *do not* retain their values between calls,
- do not exist in the programs memory between calls.

`x` will probably have a different size and shape on each call.

The space usually comes from the programs stack.

Argument Intent

Hints to the compiler can be given as to whether a dummy argument will:

- only be referenced — `INTENT(IN)`;
- be assigned to before use — `INTENT(OUT)`;
- be referenced and assigned to — `INTENT(INOUT)`;

```
SUBROUTINE example(arg1,arg2,arg3)
 REAL, INTENT(IN) :: arg1
 INTEGER, INTENT(OUT) :: arg2
 CHARACTER, INTENT(INOUT) :: arg3
 REAL :: r
  r = arg1*ICHAR(arg3)
  arg2 = ANINT(r)
  arg3 = CHAR(MOD(127,arg2))
END SUBROUTINE example
```

The use of `INTENT` attributes is recommended as it:

- allows good compilers to check for coding errors,
- facilitates efficient compilation and optimisation.

Note: if an actual argument is ever a literal, then the corresponding dummy must be `INTENT(IN)`.

Scoping Rules

Fortran 90 is *not* a traditional block-structured language:

- the *scope* of an entity is the range of program unit where it is visible and accessible;

- internal procedures can inherit entities by *host association*.

- objects declared in modules can be made visible by *use-association* (the `USE` statement) — useful for global data;

Host Association — Global Data

Consider,

```
PROGRAM CalculatePay
 IMPLICIT NONE
 REAL :: Pay, Tax, Delta
 INTEGER :: NumberCalcsDone = 0
 Pay = ...;  Tax = ... ; Delta = ...
 CALL PrintPay(Pay,Tax)
 Tax = NewTax(Tax,Delta)
  ....
CONTAINS
 SUBROUTINE PrintPay(Pay,Tax)
  REAL, INTENT(IN) :: Pay, Tax
  REAL :: TaxPaid
   TaxPaid = Pay * Tax
   PRINT*, TaxPaid
   NumberCalcsDone = NumberCalcsDone + 1
 END SUBROUTINE PrintPay
 REAL FUNCTION NewTax(Tax,Delta)
  REAL, INTENT(IN) :: Tax, Delta
   NewTax =  Tax + Delta*Tax
   NumberCalcsDone = NumberCalcsDone + 1
 END FUNCTION NewTax
END PROGRAM CalculatePay
```

Here, `NumberCalcsDone` is a *global* variable. It is available in all procedures by *host association*.

Scope of Names

Consider the following example,

```
PROGRAM Proggie
 IMPLICIT NONE
 REAL ::  A, B, C
 CALL sub(A)
CONTAINS
 SUBROUTINE Sub(D)
  REAL :: D      ! D is dummy (alias for A)
  REAL :: C      ! local C (diff from Proggie's C)
   C = A**3      ! A cannot be changed
   D = D**3 + C  ! D can be changed
   B = C         ! B from Proggie gets new value
 END SUBROUTINE Sub
END PROGRAM Proggie
```

In `Sub`, as `A` is argument associated it may not be have its value changed but may be referenced.

`C` in `Sub` is totally separate from `C` in `Proggie`, changing its value in `Sub` does **not** alter the value of `C` in `Proggie`.

SAVE Attribute and the SAVE Statement

SAVE attribute can be:

- applied to a specified variable. NumInvocations is initialised on **first** call and retains its new value between calls,

```
SUBROUTINE Barmy(arg1,arg2)
 INTEGER, SAVE :: NumInvocations = 0
  NumInvocations = NumInvocations + 1
```

- applied to the whole procedure, and applies to *all* local objects.

```
SUBROUTINE Mad(arg1,arg2)
 REAL :: saved
 SAVE
 REAL :: saved_an_all
```

Variables with the SAVE attribute are *static* objects.

Clearly, SAVE has no meaning in the main program.

Keyword Arguments

Can supply dummy arguments in any order using *keyword arguments*, for example, given

```
SUBROUTINE axis(x0,y0,l,min,max,i)
 REAL, INTENT(IN) :: x0,y0,l,min,max
 INTEGER, INTENT(IN) :: i
  .....
END SUBROUTINE axis
```

can invoke procedure:

- using positional argument invocation:

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

- using keyword arguments:

```
CALL AXIS(0.0,0.0,Max=1.0,Min=0.1, &
                    L=100.0,I=10)
```

The names are from the *dummy arguments*.

Keyword Arguments

Keyword arguments:

- ☐ allow arguments to be specified in any order.
- ☐ makes it easy to add an extra argument — no need to modify any calls.
- ☐ helps improve the readability of the program.
- ☐ are used when a procedure has optional arguments.

Note: once a keyword is used all subsequent arguments must be keyword arguments.

Optional Arguments

Optional arguments:

- allow defaults to be used for missing arguments;
- make some procedures easier to use.

Also:

- once an argument has been omitted all subsequent arguments must be keyword arguments,
- the `PRESENT` intrinsic can be used to check for missing arguments.

Optional Arguments Example

For example, consider the internal procedure,

```
SUBROUTINE SEE(a,b)
 REAL, INTENT(IN), OPTIONAL    :: a
 INTEGER, INTENT(IN), OPTIONAL :: b
 REAL    :: ay; INTEGER :: bee
 ay = 1.0; bee = 1 ! defaults
 IF(PRESENT(a)) ay  = a
 IF(PRESENT(b)) bee = b
  ...
```

both `a` and `b` have the optional attribute so `SEE` can be called in the following ways

```
CALL SEE()
CALL SEE(1.0,1); CALL SEE(b=1,a=1.0) ! same
CALL SEE(1.0);   CALL SEE(a=1.0)     ! same
CALL SEE(b=1)
```

Recursive Procedures

In Fortran 90 recursion is supported as a feature.

- recursive procedures call themselves (either directly or indirectly),
- recursion is a neat technique
- recursion may incur certain efficiency overheads,
- recursive procedures must be explicitly declared
- recursive functions declarations must contain a `RESULT` keyword, and one type declaration refers to both the function name and the result variable.

Recursive Function Example

The following example calculates the factorial of a number and uses $n! = n(n-1)!$

```
PROGRAM Mayne
 IMPLICIT NONE
  PRINT*, fact(12) ! etc
CONTAINS
 RECURSIVE FUNCTION fact(N) RESULT(N_Fact)
  INTEGER, INTENT(IN)  :: N
  INTEGER :: N_Fact ! also defines type of fact
   IF (N > 0) THEN
     N_Fact = N * fact(N-1)
   ELSE
     N_Fact = 1
   END IF
 END function FACT
END PROGRAM Mayne
```

To calculate 4!,

1. 4! is $4 \times 3!$, so calculate 3! then multiply by 4,
2. 3! is $3 \times 2!$, need to calculate 2!,
3. 2! is $2 \times 1!$, 1! is $1 \times 0!$ and $0! = 1$
4. can now work back up the calculation and fill in the missing values.