# Introduction to Modern Fortran

## *Control Constructs*

Nick Maclaren

Computing Service

**nmm1@cam.ac.uk, ext. 34761**

November 2007

# Control Constructs

These change the sequential execution order
We cover the main constructs in some detail
We shall cover procedure call later

The main ones are:

- Conditionals (IF etc.)
- Loops (DO etc.)
- Switches (SELECT/CASE etc.)
- Branches (GOTO etc.)

Loops are by far the most complicated

# Single Statement IF

Oldest and simplest is the single statement IF

IF (logical expression) simple statement

If the LHS is .True., the RHS is executed

If not, the whole statement has no effect

```
IF (MOD(count,1000) == 0)    &
      PRINT *, 'Reached', count
IF (X < A) X = A
```

Unsuitable for anything complicated

- Only action statements can be on the RHS

No IFs or statements containing blocks

# Block IF Statement

A block IF statement is more flexible
The following is the most traditional form of it

```
IF (logical expression) THEN
      then block of statements
ELSE
      else block of statements
END IF
```

If the expr. is .True., the first block is executed
If not, the second one is executed

END IF can be spelled ENDIF

# Example

```
LOGICAL :: flip

IF (flip .AND. X /= 0.0) THEN
      PRINT *, 'Using the inverted form'
      X = 1.0/A
      Y = EXP(–A)
ELSE
      X = A
      Y = EXP(A)
END IF
```

# Omitting the ELSE

The ELSE and its block can be omitted

```
IF (X > Maximum) THEN
      X = Maximum
END IF

IF (name(1:4) == "Miss" .OR.       &
            name(1:4) == "Mrs.") THEN
      name(1:3) = "Ms."
      name(4:) = name(5:)
END IF
```

# Including ELSE IF Blocks (1)

ELSE IF functions much like ELSE and IF

```
IF (X < 0.0) THEN        ! This is tried first
      X = A
ELSE IF (X < 2.0) THEN        ! This second
      X = A + (B–A)*(X–1.0)
ELSE IF (X < 3.0) THEN        ! And this third
      X = B + (C–B)*(X–2.0)
ELSE        ! This is used if none succeed
      X = C
END IF
```

# Including ELSE IF Blocks (2)

You can have as many ELSE IFs as you like
There is only one END IF for the whole block

All ELSE IFs must come before any ELSE
Checked in order, and the first success is taken

You can omit the ELSE in such constructs

ELSE IF can be spelled ELSE IF

# Named IF Statements (1)

The IF can be preceded by <name> :
And the END IF followed by <name> – note!
And any ELSE IF/THENand ELSE may be

```
gnole : IF (X < 0.0) THEN
        X = A
ELSE IF (X < 2.0) THEN gnole
        X = A + (B–A)*(X–1.0)
ELSE gnole
        X = C
END IF gnole
```

# Named IF Statements (2)

The IF construct name must match and be distinct
A great help for checking and clarity

- You should name at least all long IFs

If you don't nest IFs much, this style is fine

```
gnole : IF (X < 0.0) THEN
        X = A
ELSE IF (X < 2.0) THEN
        X = A + (B–A)*(X–1.0)
ELSE
        X = C
END IF gnole
```

# Block Contents

- Almost any executable statements are OK

Both kinds of IF, complete loops etc.
You may never notice the few restrictions

That applies to all of the block statements
IF, DO, SELECT etc.
And all of the blocks within an IF statement

- Avoid deep levels and very long blocks

Purely because they will confuse human readers

# Example

```
phasetest: IF (state == 1) THEN
      IF (phase < pi_by_2) THEN
            . . .
      ELSE
            . . .
      END IF
ELSE IF (state == 2) THEN phasetest
      IF (phase > pi) PRINT *, 'A bit odd here'
ELSE phasetest
      IF (phase < pi) THEN
            . . .
      END IF
END IF phasetest
```

# Basic Loops (1)

- A single loop construct, with variations

The basic syntax is:

```
[ loop name : ] DO [ [ , ] loop control ]
        block
END DO [ loop name ]
```

loop name and loop control are optional
With no loop control, it loops indefinitely

END DO can be spelled ENDDO
The comma after DO is entirely a matter of taste

# Basic Loops (2)

```
DO      ! Implement the Unix 'yes' command
        PRINT *, 'y'
END DO

yes: DO
        PRINT *, 'y'
END DO yes
```

The loop name must match and be distinct

- You should name at least all long loops

A great help for checking and clarity

Other of it uses are described later

# Indexed Loop Control

The loop control has the following form

<integer variable> = <LWB> , <UPB>

The bounds can be any integer expressions

The variable starts at the lower bound

A: If it exceeds the upper bound, the loop exits

The loop body is executed †

The variable is incremented by one

The loop starts again from A

† See later about EXIT and CYCLE

# Examples

```
DO I = 1 , 3
      PRINT *, 7*I-3
END DO
```

Prints 3 lines containing 4, 10 and 17

```
DO I = 3 , 1
      PRINT *, 7*I-3
END DO
```

Does nothing

# Using an increment

The general form is

<var> = <start> , <finish> , <step>

<var> is set to <start>, as before
<var> is incremented by <step>, not one
Until it exceeds <finish> (if <step> is positive)
Or is smaller than <finish> (if <step> is negative)

- The direction depends on the sign of <step>

The loop is invalid if <step> is zero, of course

# Examples

```
DO I = 1 , 20 , 7
      PRINT *, I
END DO
```

Prints 3 lines containing 1, 8 and 15

```
DO I = 20 , 1 , 7
      PRINT *, I
END DO
```

Does nothing

# Examples

```
DO I = 20 , 1 , -7
      PRINT *, I
END DO
```

Prints 3 lines containing 20, 13 and 6

```
DO I = 1 , 20 , -7
      PRINT *, I
END DO
```

Does nothing

# Mainly for C Programmers

The control expressions are calculated on entry

- Changing their variables has no effect

- It is illegal to assign to the loop variable

```
DO index = i*j, n**21, k
      n = 0; k = –1        ! Does not affect the loop
      index = index+1      ! Is forbidden
END DO
```

# Loop Control Statements

EXIT leaves the innermost loop
CYCLE skips to the next iteration
EXIT/CYCLE name is for the loop named name
These are usually used in single–statement IFs

```
DO
    x = read_number()
    IF (x < 0.0) EXIT
    count = count+1; total = total+x
    IF (x == 0.0) CYCLE
    . . .
END DO
```

# Example

```
INTEGER ::  state(right), table(left , right)
FirstMatch = 0
outer: DO i = 1 , right
       IF (state(right) /= OK) CYCLE
       DO j = 1 , left
              IF (found(table(j,i)) THEN
                     FirstMatch = i
                     EXIT outer
              END IF
       END DO
END DO outer
```

# Warning

What is the control variable's value after loop exit?

- It is undefined after normal exit

Web pages and ignoramuses often say otherwise

It IS defined if you leave by EXIT
Generally, it is better not to rely on that fact

# WHILE Loop Control

The loop control has the following form

```
WHILE ( <logical expression> )
```

The expression is reevaluated for each cycle
The loop exits as soon as it becomes .FALSE.
The following are equivalent:

```
DO WHILE ( <logical expression> )
```

```
DO
    IF (.NOT. ( <logical expression> )) EXIT
```

# CONTINUE

CONTINUE is a statement that does nothing
It used to be fairly common, but is now rare

Its main use is in blocks that do nothing
Empty blocks aren't allowed in Fortran

Otherwise mainly a placeholder for labels
This is purely to make the code clearer

But it can be used anywhere a statement can

# RETURN and STOP

RETURN returns from a procedure

- It does not return a result

How to do that is covered under procedures

STOP halts the program cleanly

- Do not spread it throughout your code

Call a procedure to tidy up and finish off

# Multi-way IFs

```
IF (expr == val1) THEN
      x = 1.23
ELSE IF (expr >= val2 .AND. expr <= val3) THEN
      CONTINUE
ELSE IF (expr == val4) THEN
      x = x + 4.56
ELSE
      x = 7.89 – x
END IF
```

Very commonly, expr is always the same
And all of the vals are constant expressions
Then there is another way of coding it

# SELECT CASE (1)

```
PRINT *, 'Happy Birthday'
SELECT CASE (age)
CASE(18)
      PRINT *, 'You can now vote'
CASE(40)
      PRINT *, 'And life begins again'
CASE(60)
      PRINT *, 'And free prescriptions'
CASE(100)
      PRINT *, 'And greetings from the Queen'
CASE DEFAULT
      PRINT *, 'It''s just another birthday'
END SELECT
```

# SELECT CASE (2)

- The CASE clauses are statements

To put on one line, use ‘CASE(18) ; <statement>’

The values must be initialisation expressions
INTEGER, CHARACTER or LOGICAL
You can specify ranges for the first two

```
CASE (–42:42)       ! –42 to 42 inclusive
CASE (42:)       ! 42 or above
CASE (:42)       ! Up to and including 42
```

Be careful with CHARACTER ranges

# SELECT CASE (3)

SELECT CASE can be spelled SELECTCASE
END SELECT can be spelled ENDSELECT

- CASE DEFAULT but NOT CASEDEFAULT

SELECT and CASE can be named, like IF

- It is an error for the ranges to overlap

It is not an error for ranges to be empty
Empty ranges don't overlap with anything
It is not an error for the default to be unreachable

# Labels and GOTO

Warning: this area gets seriously religious!

Most executable statements can be labelled
GOTO <label> branches directly to the label

In old Fortran, you needed to use a lot of these

- Now, you should almost never use them

If you think you need to, consider redesigning

- Named loops, EXIT and CYCLE are better

# Remaining uses of GOTO

- Useful for branching to clean–up code

E.g. diagnostics, undoing partial updates etc.
This is by FAR the main remaining use

Fortran does not have any cleaner mechanisms
E.g. it has no exception handling constructs

- They make a few esoteric algorithms clearer

E.g. certain finite–state machine models
I have seen such code 3–4 times in 40+ years

# Clean-up Code (1)

```
SUBROUTINE Fred
DO . . .
      CALL SUBR (arg1 , arg2 , . . . , argn , ifail)
      IF (ifail /= 0) GOTO 999
END DO
. . . lots more similar code . . .
RETURN

999 SELECT CASE (ifail)
CASE(1) ! Code for ifail = 1
      . . .
CASE(2) ! Code for ifail = 2
      . . .
END SUBROUTINE Fred
```

# Clean-up Code (2)

Many people regard this as better style:

```
SUBROUTINE Fred
DO . . .
      CALL SUBR (arg1 , arg2 , . . . , argn , ifail)
      IF (ifail /= 0) GOTO 999
END DO

999 CONTINUE
SELECT CASE (ifail)
CASE(1) ! Code for ifail = 1
      . . .
END SUBROUTINE Fred
```