Control structures

Arjen Markus

Deltares

June 6, 2018

Control structures

Control structures take one of several forms:

- Looping constructs
- Branching based on one or more conditions
- Constructs defining a new scope

Looping constructs

The (structured) looping constructs are various forms of

```
do "condition"
    ...
enddo
```

The statements `cycle` and `exit` cause control to jump to the next iteration or to skip the rest of the loop

Forms of DO-loops

DO-loops come in various flavours:

- The classic loop – iterate over a finite (fixed) range

```
do i = 1,10
    ...
enddo
```

- The endless loop – you will need to explicitly terminate it

```
do
    ...
enddo
```

- The conditional loop – loop as long as the condition is true

```
do while ( x < 1.0 )
    ...
enddo
```

Nested loops

Loops can be nested and you can use "named" loops to jump out of any nesting:

```
outer_loop: &
    do k = 1,100
        do j = 1,100
            do i = 1,100
                ...
                if ( x(i,j,k) > 100.0 ) exit outer_loop
            enddo
        enddo
    enddo &
outer_loop
```

Parallel looping constructs

Fortran has several other looping constructs: `forall` and `do concurrent`.
There is also the `where` construct that in principle can be parallellised.
The `forall` loop has proven to be problematic for optimisation and has been largely superseded by the `do concurrent` loop:

```
do concurrent (i = 1:m, j = 1,n)
    x(i,j) = i ** 2 + j ** 2
enddo
```

The `do concurrent` loop is automatically parallellised if the compiler can prove that the iterations are independent. (Note: A `write` statement will hamper such parallellisation)

Branching constructs

Two constructs allow you to branch on the basis of one or more conditions:

- The `if (condition) then` construct
- The `select case (variable)` construct

The latter selects on the *value* or *range of values* of an integer or character variable:

```
integer :: type
...
select case (type)
    case (1)
        write(*,*) 'Type A calculation'
    case (2, 5:10)
        write(*,*) 'Type B calculation'
    case (:1)
        write(*,*) 'Type C calculation'
    case default
        write(*,*) 'Unknown/unsupported type: ', type
end select
```

Named constructs

Constructs like do-loops but also if-blocks and select-case blocks can have *names*. These names can be used to jump to the end. This makes the use of GOTO statements superfluous.
For instance:

```
integer :: x

named_if: &
    if ( x > 0 ) then
        write(*,*) 'x positive'
        if ( x < 10 ) exit named_if
        write(*,*) 'x big enough'
    endif named_if
```

Scoping constructs

With the introduction of modules, recursive and internal routines in Fortran 90 the scope of variables has become more varied. New constructs give even more variation, but also more control:

- Recursive routines – implicit looping
- Internal routines – access to variables in the containing routine
- The `associate` construct – sort of aliases
- The `block` construct – localised variables

Recursive routines

A classic example of a recursive routine:

```
recursive function factorial( n ) result ( f ) ! "result" is required h
    integer, intent(in) :: n

    if ( n < 2 ) then
        f = 1
    else
        f = n * factorial( n - 1 )
    endif
end function factorial
```

Internal routines

Internal routines can be added to programs and subroutines, but not to other internal routines. They replace *statement functions*:

```
function eval_polynomial( coeff, x )
    real, dimension(:), intent(in) :: coeff
    real, intent(in)               :: x

    eval_polynomial = 0.0
    if ( abs(x) < 100.0 ) then
        eval_polynomial = horner( coeff, x )
    endif
contains
real function horner( coeff, x )
    real, dimension(:), intent(in) :: coeff
    real, intent(in)               :: x

    ...

end function horner
end function eval_polynomial
```

Block constructs

Block constructs introduce a local variable scope:

```
subroutine print_all( array )
    implicit none

    real, dimension(:), intent(in) :: array

    block
        integer :: n
        do n = 1,size(array)
            write(*,*) n, array(n)
        enddo
    end block

    ! n not defined outside the block!
end subroutine print_all
```

Associate constructs (1)

Associate constructs introduce local *aliases* and these are surprisingly versatile:

```
type vector_3d
    real, dimension(3) :: coords
end type vector_3d

type(vector_3d), dimension(10) :: v

associate( x => v%coords(1), y => v%coords(2), z =>v%coords(3) )
    do i = 1,size(v)
        write(*,'(3f10.4)') x(i), y(i), z(i)
    enddo
end associate
```

Associate constructs (2)

Associate constructs can even be used in the following way:

```
function vec( x, y, z )
    real, intent(in) :: x, y, z
    type(vector_3d)  :: vec

    vec%coords(1) = x
    vec%coords(2) = x + y
    vec%coords(3) = x + y + z
end function vec

associate( w => vec(1.0, 2.0, 3.0) )
    write(*,*) w%coords(1)
end associate
```