Count-Controlled (DO) Loops Outline

1. Count-Controlled (DO) Loops Outline
2. A DO WHILE Loop That Counts
3. Count-Controlled Loops
4. Count-Controlled Loop Flowchart
5. Explicitly Count-Controlled DO Loops
6. Explicitly Count-Controlled DO Loop Flowchart
7. Three Programs That Behave Identically
8. Identical Behavior: Proof
9. Identical Behavior: Proof (continued)
10. Explicitly Count-Controlled DO Loop
11. DO Loop Details
12. DO Loop Application
13. DO Loop Application (continued)
14. DO Loop with an Explicit Increment
15. DO Loop w/Explicit Increment (continued)
16. DO Loop with a Negative Increment
17. DO Loop with Named Constants
18. DO Loop with Variables
19. DO Loop with Expressions
20. DO Loop with a REAL Counter: BAD BAD BAD
21. Why REAL Counters Are BAD BAD BAD
22. Replacing a REAL Counter with an INTEGER Counter
23. Debugging a DO Loop
24. Debugging a DO Loop: PRINT Statements in the Loop Body
25. Debugging a DO Loop: PRINT Statements (continued)
26. Debugging a DO Loop: Removing PRINT Statements
27. Nesting DO Loops Inside IF-THEN Blocks and Vice Versa
28. Nesting DO Loop Inside IF-THEN Block Example Run
29. Nested DO Loops
30. Output of Nested DO Loop Example
31. Changing the Loop Bounds Inside the Loop: BAD BAD BAD!
32. Changing the Loop Index Inside the Loop: ILLEGAL!

See *Programming in Fortran 90/95*, 1st or 2nd edition, Chapter 13, section 13.1.

A `DO WHILE` Loop That Counts

```
% cat dowhilecount.f90
PROGRAM do_while_count
    IMPLICIT NONE
    INTEGER :: initial_value, final_value
    INTEGER :: current_value
    INTEGER :: sum
    PRINT *, "What value would you like to ", &
 &      "start counting at?"
    READ *, initial_value
    PRINT *, "What value would you like to ", &
 &      "stop counting at,"
    PRINT *, "  which must be greater than ", &
 &      "or equal to ", initial_value, "."
    READ *, final_value
    IF (final_value < initial_value) THEN
        PRINT *, "ERROR: the final value ", &
 &          final_value
        PRINT *, "  is less than the ", &
 &          "initial value ", initial_value, "."
        STOP
    END IF !! (final_value < initial_value)
    sum = 0
    current_value = initial_value
    DO WHILE (current_value <= final_value)
        sum = sum + current_value
        current_value = current_value + 1
    END DO !! WHILE (current_value <= final_value)
    PRINT *, "The sum of the integers from ",    &
 &      initial_value, " through ", final_value, &
 &      " is ", sum, "."
END PROGRAM do_while_count
% f95 -o dowhilecount dowhilecount.f90
% dowhilecount
 What value would you like to start counting at?
 1
 What value would you like to stop counting at,
   which must be greater than or equal to  1 .
 0
 ERROR: the final value  0
   is less than the initial value  1 .
% dowhilecount
 What value would you like to start counting at?
 1
 What value would you like to stop counting at,
   which must be greater than or equal to  1 .
 5
 The sum of the integers from  1  through  5  is  15 .
```

Count-Controlled Loops

On the previous slide, we saw a case of a loop that executes a specific number of *iterations*, by using a counter variable that is initialized to a particular initial value and is incremented at the end of each iteration of the loop, until it passes a particular final value:

```
sum = 0
current_value = initial_value
DO WHILE (current_value <= final_value)
    sum = sum + current_value
    current_value = current_value + 1
END DO !! WHILE (current_value <= final_value)
```

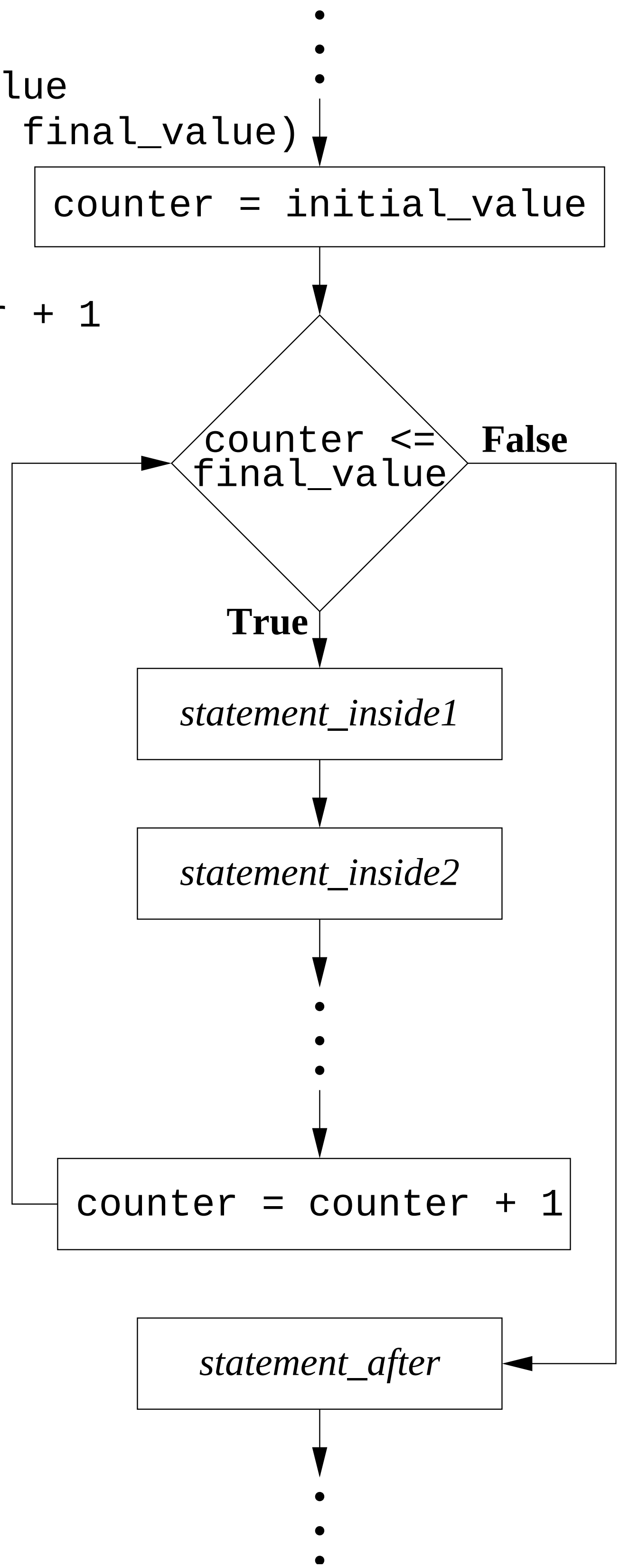
We call this kind of loop a *count-controlled loop.* If we express a count-controlled loop as a `DO WHILE` loop, then the general form is:

```
counter = initial_value
DO WHILE (counter <= final_value)
    statement1
    statement2
    ...
    counter = counter + 1
END DO !! WHILE (counter <= final_value)
```

Count-controlled loops are among the most commonly used kinds of loops. They're so common that we have a special construct for them, called an *explicitly count-controlled loop.*

Count-Controlled Loop Flowchart

```
counter = initial_value
DO WHILE (counter <= final_value)
    statement_inside1
    statement_inside2
    ...
    counter = counter + 1
END DO
statement_after
```



Explicitly Count-Controlled DO Loops

An *explicitly count-controlled* DO *loop* has this form:

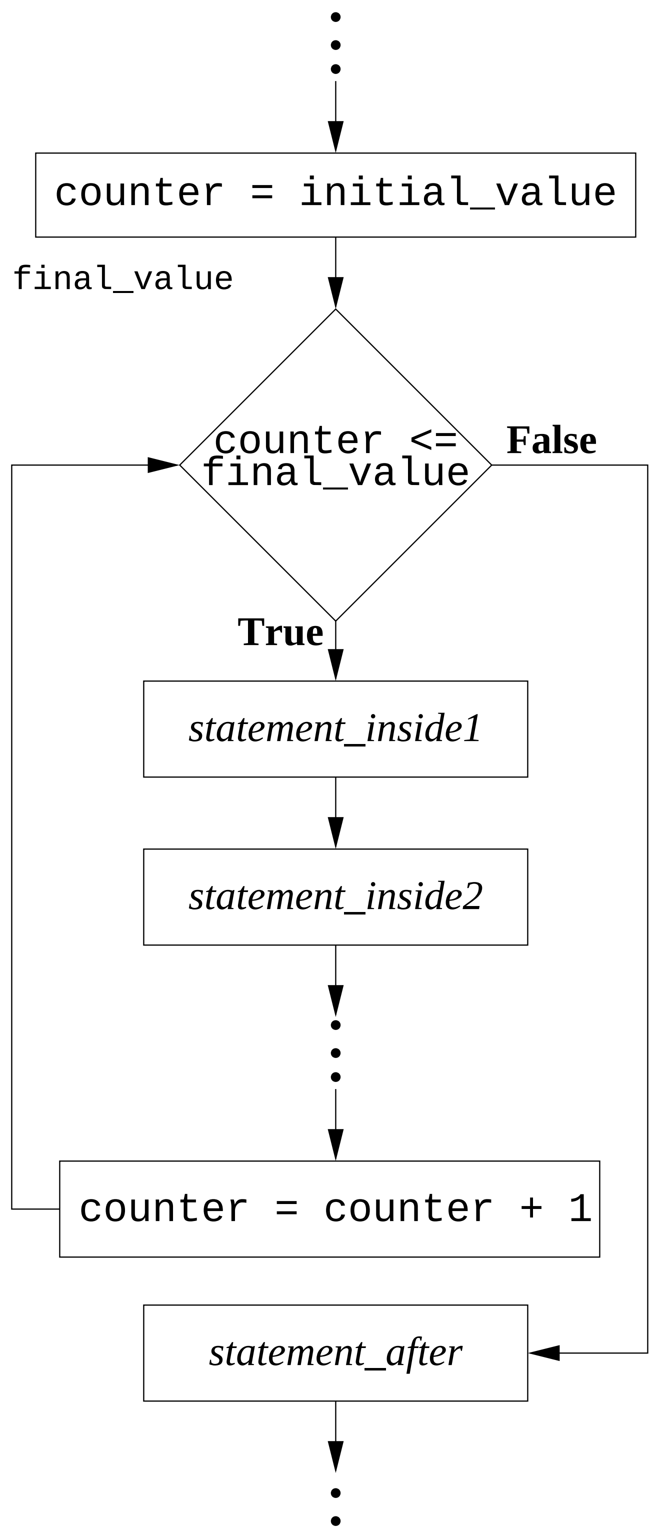
```
DO counter = initial_value, final_value
  statement1
  statement2
    ...
END DO !! counter = initial_value, final_value
```

An explicitly count-controlled DO loop behaves exactly the same as a count-controlled DO WHILE loop:

```
counter = initial_value
DO WHILE (counter <= final_value)
    statement1
    statement2
    ...
    counter = counter + 1
END DO !! WHILE (counter <= final_value)
```

Explicitly Count-Controlled Loop Flowchart

```
DO counter = initial_value, final_value
    statement_inside1
    statement_inside2
    ...
END DO
statement_after
```



Three Programs That Behave Identically

```
PROGRAM exam1outputa3
    IMPLICIT NONE
    INTEGER :: count
    INTEGER :: sum = 0
    count = 1
    sum = sum + count
    count = count + 1
    sum = sum + count
    count = count + 1
    sum = sum + count
    count = count + 1
    sum = sum + count
    count = count + 1
    sum = sum + count
    count = count + 1
    PRINT *, "count = ", count
    PRINT *, "sum = ", sum
END PROGRAM exam1outputa3
```

```
PROGRAM exam1outputa3_dowhile_loop
    IMPLICIT NONE
    INTEGER :: count
    INTEGER :: sum = 0
    count = 1
    DO WHILE (count <= 5)
        sum = sum + count
        count = count + 1
    END DO !! WHILE (count <= 5)
    PRINT *, "count = ", count
    PRINT *, "sum = ", sum
END PROGRAM exam1outputa3_dowhile_loop
```

```
PROGRAM exam1outputa3_count_loop
    IMPLICIT NONE
    INTEGER :: count
    INTEGER :: sum = 0
    DO count = 1, 5
        sum = sum + count
    END DO !! count = 1, 5
    PRINT *, "count = ", count
    PRINT *, "sum = ", sum
END PROGRAM exam1outputa3_count_loop
```

Identical Behavior: Proof

```
% cat exam1outputa3.f90
PROGRAM exam1outputa3
    IMPLICIT NONE
    INTEGER :: count
    INTEGER :: sum = 0
    count = 1
    sum = sum + count
    count = count + 1
    sum = sum + count
    count = count + 1
    sum = sum + count
    count = count + 1
    sum = sum + count
    count = count + 1
    sum = sum + count
    count = count + 1
    PRINT *, "count = ", count
    PRINT *, "sum = ", sum
END PROGRAM exam1outputa3
% f95 -o exam1outputa3 exam1outputa3.f90
% exam1outputa3
 count =  6
 sum =  15
```

```
% cat exam1outputa3_dowhile_loop.f90
PROGRAM exam1outputa3_dowhile_loop
    IMPLICIT NONE
    INTEGER :: count
    INTEGER :: sum = 0
    count = 1
    DO WHILE (count <= 5)
        sum = sum + count
        count = count + 1
    END DO !! WHILE (count <= 5)
    PRINT *, "count = ", count
    PRINT *, "sum = ", sum
END PROGRAM exam1outputa3_dowhile_loop
% f95 -o exam1outputa3_dowhile_loop \
         exam1outputa3_dowhile_loop.f90
% exam1outputa3_dowhile_loop
 count =  6
 sum =  15
```

Identical Behavior: Proof (continued)

```
% cat exam1outputa3_dowhile_loop.f90
PROGRAM exam1outputa3_dowhile_loop
    IMPLICIT NONE
    INTEGER :: count
    INTEGER :: sum = 0
    count = 1
    DO WHILE (count <= 5)
        sum = sum + count
        count = count + 1
    END DO !! WHILE (count <= 5)
    PRINT *, "count = ", count
    PRINT *, "sum = ", sum
END PROGRAM exam1outputa3_dowhile_loop
% f95 -o exam1outputa3_dowhile_loop \
         exam1outputa3_dowhile_loop.f90
% exam1outputa3_dowhile_loop
 count =  6
 sum =  15
```

```
% cat exam1outputa3_count_loop.f90
PROGRAM exam1outputa3_count_loop
    IMPLICIT NONE
    INTEGER :: count
    INTEGER :: sum = 0
    DO count = 1, 5
        sum = sum + count
    END DO !! count = 1, 5
    PRINT *, "count = ", count
    PRINT *, "sum = ", sum
END PROGRAM exam1outputa3_count_loop
% f95 -o exam1outputa3_count_loop \
         exam1outputa3_count_loop.f90
% exam1outputa3_count_loop
 count =  6
 sum =  15
```

Explicitly Count-Controlled DO Loop

```
% cat product_loop.f90
PROGRAM product_loop
    IMPLICIT NONE
    INTEGER :: product = 1
    INTEGER :: count
    DO count = 1, 5
       product = product * count
    END DO !! count = 1, 5
    PRINT *, "After the loop: count = ", &
 &           count, ", product = ", product
END PROGRAM product_loop
% f95 -o product_loop product_loop.f90
% product_loop
 After the loop: count =  6 , product =  120
```

When the `DO` statement is encountered:

1. The *loop counter variable* (sometimes called the *loop index*) is assigned the *initial value* (sometimes called the *lower bound*).
2. The loop counter is compared to the *final value* (sometimes called the *upper bound*), and if the loop counter is greater than the final value, then the loop is exited.
3. Each statement inside the *loop body* is executed in sequence.
4. When the end of the loop body is reached (indicated by the `END DO` statement), the loop counter is incremented by the *loop increment* value, sometimes called the *stride*. By default, the loop increment value is 1 (though it can be explicitly set to any **integer** value).
5. The program jumps back up to step 2.

We refer to each trip through the body of the loop as an *iteration* or a *pass*.

DO Loop Details

Suppose you have an explicitly count-controlled `DO` loop that looks like this:

```
INTEGER :: product = 1
INTEGER :: count
DO count = 1, 5
    product = product * count
END DO !! count = 1, 5
```

The above program fragment behaves **identically** the same as:

```
                          ! Program Trace

INTEGER :: product = 1    !            product =   1

INTEGER :: count          ! count is undefined

count = 1                 ! count = 1, product =   1

product = product * count ! count = 1, product =   1

count     = count + 1     ! count = 2, product =   1

product = product * count ! count = 2, product =   2

count     = count + 1     ! count = 3, product =   3

product = product * count ! count = 3, product =   6

count     = count + 1     ! count = 4, product =   6

product = product * count ! count = 4, product =  24

count     = count + 1     ! count = 5, product =  24

product = product * count ! count = 5, product = 120

count     = count + 1     ! count = 6, product = 120
```

DO Loop Application

Suppose that there's a line of a dozen students waiting for tickets for the next OU-Texas football game.

How many different orders can they have in line?

- The head of the line could be any student.
- The 2nd position in line could be any student except the student at the head of the line.
- The 3rd position in line could be any student except the student at the head of the line or the student in the 2nd position.

And so on.

Generalizing, we have that the number of different orders of the students is:

$$(12)\ (11)\ (10)\ \ldots\ (2)\ (1)$$

We can also express this in the other direction:

$$(1)\ (2)\ (3)\ \ldots\ (12)$$

In fact, for any number of students n, we have that the number of orders is:

$$(1)\ (2)\ (3)\ \ldots\ (n)$$

This arithmetic expression is called *"n factorial"*, denoted *n!*

We say that there are *n! permutations*, or orderings, of the *n* students.

DO Loop Application (continued)

The number of permutations of *n* objects is:

$$P(n) = n! = (1)\ (2)\ (3)\ \ldots\ (n)$$

Here's a program that calculates permutations:

```
% cat permute.f90
PROGRAM permute
    IMPLICIT NONE
    INTEGER :: number_of_students
    INTEGER :: permutations
    INTEGER :: count
    PRINT *, "How many students are ", &
 &           "in line for tickets?"
    READ *, number_of_students
    permutations = 1
    DO count = 1, number_of_students
        permutations = permutations * count
    END DO !! count = 1, number_of_students
    PRINT *, "There are ", permutations, &
 &           " different orders"
    PRINT *, "  in which the ",  &
 &           number_of_students, &
 &           " students can stand"
    PRINT *, "  in line."
END PROGRAM permute
% f95 -o permute permute.f90
% permute
 How many students are in line for tickets?
 12
 There are  479001600  different orders
   in which the  12  students can stand
   in line.
```

`DO` Loop with an Explicit Increment

The most common increment for a `DO` loop is 1. For convenience, therefore, we allow a loop increment of 1 to be *implied*: if a `DO` loop has an increment of 1, then the `DO` statement doesn't require the increment to be stated explicitly. For example:

```
INTEGER :: product = 1
INTEGER :: count
DO count = 1, 5
    product = product * count
END DO !! count = 1, 5
```

On the other hand, we could state the loop increment explicitly in the `DO` statement, by putting a comma after the final value, and then the increment:

```
INTEGER :: product = 1
INTEGER :: count
DO count = 1, 5, 1
    product = product * count
END DO !! count = 1, 5, 1
```

The above two program fragments behave **identically**. Notice that both of the above loops have 5 iterations:

- `count = 1`
- `count = 2`
- `count = 3`
- `count = 4`
- `count = 5`

DO Loop w/Explicit Increment (continued)

On the other hand, if the loop increment is not 1, then it **must** be explicitly stated:

```
INTEGER :: product = 1
INTEGER :: count
DO count = 1, 5, 2
    product = product * count
END DO !! count = 1, 5, 2
```

Notice that the above loop has only 3 iterations:

- `count = 1`
- `count = 3`
- `count = 5`

The above program fragment behaves **identically** to:

```
INTEGER :: product = 1
INTEGER :: count
count = 1                    ! count = 1, product = 1
product = product * count    ! count = 1, product = 1
count = count + 2            ! count = 3, product = 1
product = product * count    ! count = 3, product = 3
count = count + 2            ! count = 5, product = 3
product = product * count    ! count = 5, product = 15
count = count + 2            ! count = 7, product = 15
```

DO Loop with a Negative Increment

Sometimes, we want to loop backwards, from a high initial value to a low final value. To do this, we use a negative loop increment:

```
% cat decimaldigits.f90
PROGRAM decimal_digits
    IMPLICIT NONE
    INTEGER,PARAMETER :: input_digits = 4
    INTEGER,PARAMETER :: base = 10
    INTEGER :: base_power, input_value
    INTEGER :: base_digit_value, output_digit
    PRINT *, "Input an integer of no more than ", &
 &           input_digits, " digits:"
    READ *, input_value
    DO base_power = input_digits - 1, 0, -1
        base_digit_value = base ** base_power
        IF (input_value >= base_digit_value) THEN
            output_digit = &
 &              input_value / base_digit_value
            PRINT "(I2,A,I2,A,I1)", base, " ** ", &
 &                   base_power, ":  ", output_digit
            input_value =     &
 &              input_value - &
 &              output_digit * base_digit_value
        END IF !! (input_value >= base_digit_value)
    END DO !! base_power = input_digits - 1, 0, -1
END PROGRAM decimal_digits
% f95 -o decimaldigits decimaldigits.f90
% decimaldigits
 Input an integer of no more than  4  digits:
 2345
10 **  3:  2
10 **  2:  3
10 **  1:  4
10 **  0:  5
% decimaldigits
 Input an integer of no more than  4  digits:
 8765
10 **  3:  8
10 **  2:  7
10 **  1:  6
10 **  0:  5
```

DO Loop with Named Constants

For the loop lower bound and upper bound, and the stride if there is one, we can use **INTEGER** named constants:

```
% cat loopbndconsts.f90
PROGRAM loop_bounds_named_constants
    IMPLICIT NONE
    INTEGER,PARAMETER :: initial_value =  1
    INTEGER,PARAMETER :: final_value   = 20
    INTEGER,PARAMETER :: stride        =  3
    INTEGER :: count, sum = 0
    DO count = initial_value, final_value, stride
        sum = sum + count
        PRINT *, "count = ", count, ", sum = ", sum
    END DO !! count = initial_value, final_value, stride
    PRINT *, "After loop, count = ", count, &
 &           ", sum = ", sum, "."
END PROGRAM loop_bounds_named_constants
% f95 -o loopbndconsts loopbndconsts.f90
% loopbndconsts
 count =  1 , sum =  1
 count =  4 , sum =  5
 count =  7 , sum =  12
 count =  10 , sum =  22
 count =  13 , sum =  35
 count =  16 , sum =  51
 count =  19 , sum =  70
 After loop, count =  22 , sum =  70 .
```

In fact, we **should** use INTEGER **named** constants rather than INTEGER **literal** constants: it's much better programming practice, because it makes it much easier to change the loop bounds (and the stride, if there is one).

DO Loop with Variables

For the loop lower bound and upper bound, and the stride if there is one, we can use **INTEGER** variables:

```
% cat loopbndvars.f90
PROGRAM loop_bounds_variables
    IMPLICIT NONE
    INTEGER :: initial_value, final_value, stride
    INTEGER :: count, sum = 0
    PRINT *, "What are the initial, final and ", &
 &           "stride values?"
    READ *, initial_value, final_value, stride
    DO count = initial_value, final_value, stride
        sum = sum + count
        PRINT *, "count = ", count, ", sum = ", sum
    END DO !! count = initial_value, final_value, stride
    PRINT *, "After the loop, count = ", count, &
 &           ", sum = ", sum, "."
END PROGRAM loop_bounds_variables
% f95 -o loopbndvars loopbndvars.f90
% loopbndvars
 What are the initial, final and stride values?
 1,  20,  4
 count =  1 , sum =  1
 count =  5 , sum =  6
 count =  9 , sum =  15
 count =  13 , sum =  28
 count =  17 , sum =  45
 After the loop, count =  21 , sum =  45 .
% loopbndvars
 What are the initial, final and stride values?
 5  25  5
 count =  5 , sum =  5
 count =  10 , sum =  15
 count =  15 , sum =  30
 count =  20 , sum =  50
 count =  25 , sum =  75
 After the loop, count =  30 , sum =  75 .
```

DO Loop with Expressions

If we don’t happen to have a variable handy that represents one of the loop bounds or the loop increment, then we can use an expression:

```
% cat loopbndexprs.f90
PROGRAM loop_bounds_expressions
    IMPLICIT NONE
    INTEGER :: initial_value, final_value, multiplier
    INTEGER :: count, sum = 0
    PRINT *, "What are the initial, final and ", &
 &           "multiplier values?"
    READ *, initial_value, final_value, multiplier
    DO count = initial_value * multiplier, &
 &             final_value * multiplier,   &
 &             multiplier - 1
        sum = sum + count
        PRINT *, "count = ", count, ", sum = ", sum
    END DO !! count = ...
    PRINT *, "After the loop, count = ", count, &
 &           ", sum = ", sum, "."
END PROGRAM loop_bounds_expressions
% f95 -o loopbndexprs loopbndexprs.f90
% loopbndexprs
 What are the initial, final and multiplier values?
 1,  9,  4
 count =  4 , sum =  4
 count =  7 , sum =  11
 count =  10 , sum =  21
 count =  13 , sum =  34
 count =  16 , sum =  50
 count =  19 , sum =  69
 count =  22 , sum =  91
 count =  25 , sum =  116
 count =  28 , sum =  144
 count =  31 , sum =  175
 count =  34 , sum =  209
 After the loop, count =  37 , sum =  209 .
```

DO Loop with a REAL Counter: BAD BAD BAD

All of the examples of DO loops that we've seen so far have used INTEGER counters. In principle, Fortran 90 also supports REAL counters:

```
% cat doreal.f90
PROGRAM do_real_counter
    IMPLICIT NONE
    REAL :: real_count
    REAL :: sum = 0.0
    DO real_count = 1.0, 10.0
        sum = sum + real_count
    END DO !! real_count = 1.0, 10.0
    PRINT *, "After the loop:"
    PRINT *, "  real_count = ", real_count, &
 &           ", sum = ", sum, "."
END PROGRAM do_real_counter
% f95 -o doreal doreal.f90
Deleted feature used: doreal.f90, line 5:
  Non-integer DO control variable
Deleted feature used: doreal.f90, line 5:
  Non-integer DO limit expression
Deleted feature used: doreal.f90, line 5:
  Non-integer DO limit expression
% doreal
 After the loop:
   real_count =   11.0000000 , sum =   55.0000000 .
```

Notice that the compiler objects very strongly to the use of a REAL counter in the DO loop. **Why?**

Why REAL Counters Are BAD BAD BAD

REAL counters are generally considered to be very poor programming practice, because a REAL counter is an approximation, and therefore a loop with lots of iterations will accumulate a lot of error in the counter, as the error from each approximation adds up:

```
% cat doreal2.f90
PROGRAM do_real_counter2
    IMPLICIT NONE
    REAL,PARAMETER :: pi = 3.14
    REAL :: radians
    DO radians = 0, 100.0 * pi, pi / 5.0
        PRINT '(A,F19.15)', "radians = ", radians
    END DO !! radians = 0, 100.0 * pi, pi / 5.0
    PRINT *, "After the loop:"
    PRINT '(A,F19.15)', "  100.0 * pi = ", 100.0 * pi
    PRINT '(A,F19.15)', "  radians    = ", radians
END PROGRAM do_real_counter2
% f95 -o doreal2 doreal2.f90
Deleted feature used: doreal2.f90, line 5:
  Non-integer DO control variable
Deleted feature used: doreal2.f90, line 5:
  Non-integer DO limit expression
Deleted feature used: doreal2.f90, line 5:
  Non-integer DO limit expression
% doreal2
radians =   0.000000000000000
radians =   0.628000020980835
radians =   1.256000041961670
radians =   1.884000062942505
radians =   2.512000083923340
radians =   3.140000104904175
radians =   3.768000125885010
...
radians = 308.976196289062500
radians = 309.604187011718750
radians = 310.232177734375000
radians = 310.860168457031250
radians = 311.488159179687500
radians = 312.116149902343750
radians = 312.744140625000000
radians = 313.372131347656250
 After the loop:
  100.0 * pi = 314.000000000000000
  radians    = 314.000122070312500
```

Replacing a REAL Counter with an INTEGER Counter

Happily, we rarely need a REAL counter, because we can use an INTEGER counter and calculate the REAL value in the loop body:

```
% cat doreal2int.f90
PROGRAM do_real_counter2_integer
    IMPLICIT NONE
    REAL,PARAMETER :: pi = 3.14
    REAL :: radians
    INTEGER :: radians_counter
    DO radians_counter = 0, 500
        radians = radians_counter * pi / 5.0
        PRINT '(A,F19.15)', "radians = ", radians
    END DO !! radians_counter = 0, 500
    PRINT *, "After the loop:"
    PRINT '(A,F19.15)', "  100.0 * pi      = ", &
 &                      100.0 * pi
    PRINT '(A,F19.15)', "  radians         = ", &
 &                      radians
    PRINT '(A,I3)',     "  radians_counter = ", &
 &                      radians_counter
END PROGRAM do_real_counter2_integer
% f95 -o doreal2int doreal2int.f90
% doreal2int
radians =   0.000000000000000
radians =   0.628000020980835
radians =   1.256000041961670
radians =   1.884000062942505
radians =   2.512000083923340
radians =   3.140000104904175
radians =   3.768000125885010
...
radians = 308.976013183593750
radians = 309.604003906250000
radians = 310.232025146484375
radians = 310.860015869140625
radians = 311.488006591796875
radians = 312.115997314453125
radians = 312.744018554687500
radians = 313.372009277343750
radians = 314.000000000000000
 After the loop:
  100.0 * pi      = 314.000000000000000
  radians         = 314.000000000000000
  radians_counter = 501
```

Notice that there's no **accumulated** error from approximating REAL quantities, because each approximation is independent of the others.

Debugging a DO Loop

Suppose you have a program that has a DO loop, and it looks like the DO loop has a bug in it:

```
% cat sumbad.f90
PROGRAM summer
    IMPLICIT NONE
    INTEGER :: initial_value, final_value, count
    INTEGER :: sum = 0
    PRINT *, "What are the summation limits?"
    READ *, initial_value, final_value
    DO count = initial_value, final_value
        sum = sum * count
    END DO !! count = initial_value, final_value
    PRINT *, "The sum from ", initial_value, " to ", &
 &      final_value, " is ", sum, "."
END PROGRAM summer
% f95 -o sumbad sumbad.f90
% sumbad
 What are the summation limits?
 1, 5
 The sum from  1  to  5  is  0 .
```

Assuming that the bug isn't obvious just from looking, how do we figure out where the bug is?

Debugging a DO Loop: PRINT Statements in the Loop Body

One thing we can try is to put some PRINT statements inside the loop body:

```
% cat sumbaddebug.f90
PROGRAM summer
    IMPLICIT NONE
    INTEGER :: initial_value, final_value, count
    INTEGER :: sum = 0
    PRINT *, "What are the summation limits?"
    READ *, initial_value, final_value
    DO count = initial_value, final_value
        sum = sum * count
        PRINT *, "count = ", count, ", sum = ", sum
    END DO !! count = initial_value, final_value
    PRINT *, "The sum from ", initial_value, " to ", &
 &      final_value, " is ", sum, "."
END PROGRAM summer
% f95 -o sumbaddebug sumbaddebug.f90
% sumbaddebug
 What are the summation limits?
 1, 5
 count =  1 , sum =  0
 count =  2 , sum =  0
 count =  3 , sum =  0
 count =  4 , sum =  0
 count =  5 , sum =  0
 The sum from  1  to  5  is  0 .
```

Often, the output of the loop body PRINT statements will tell us where to find the bug.

Debugging a DO Loop: PRINT Statements (Continued)

When we've made a change, we can check to make sure things are going well using the same PRINT statements inside the loop body:

```
% cat sumgooddebug.f90
PROGRAM summer
    IMPLICIT NONE
    INTEGER :: initial_value, final_value, count
    INTEGER :: sum = 0
    PRINT *, "What are the summation limits?"
    READ *, initial_value, final_value
    DO count = initial_value, final_value
        sum = sum + count
        PRINT *, "count = ", count, ", sum = ", sum
    END DO !! count = initial_value, final_value
    PRINT *, "The sum from ", initial_value, " to ", &
 &      final_value, " is ", sum, "."
END PROGRAM summer
% f95 -o sumgooddebug sumgooddebug.f90
% sumgooddebug
 What are the summation limits?
 1, 5
 count =  1 , sum =  1
 count =  2 , sum =  3
 count =  3 , sum =  6
 count =  4 , sum =  10
 count =  5 , sum =  15
 The sum from  1  to  5  is  15 .
```

Debugging a `DO` Loop: Removing `PRINT` Statements

Once we know that the loop is debugged, we can delete the `PRINT` statements inside the loop body:

```
% cat sumgood.f90
PROGRAM summer
    IMPLICIT NONE
    INTEGER :: initial_value, final_value, count
    INTEGER :: sum = 0
    PRINT *, "What are the summation limits?"
    READ *, initial_value, final_value
    DO count = initial_value, final_value
        sum = sum + count
    END DO !! count = initial_value, final_value
    PRINT *, "The sum from ", initial_value, " to ", &
 &      final_value, " is ", sum, "."
END PROGRAM summer
% f95 -o sumgood sumgood.f90
% sumgood
 What are the summation limits?
 1, 5
 The sum from  1  to  5  is  15 .
```

Aside: why can’t the name of this program be `sum`?

Nesting DO Loops Inside IF-THEN Blocks and Vice Versa

We can nest DO loops inside IF-THEN blocks and IF-THEN blocks inside DO loops:

```
PROGRAM it_is_prime
    IMPLICIT NONE
    INTEGER,PARAMETER :: first_prime = 2
    INTEGER,PARAMETER :: no_remainder = 0, increment = 1
    INTEGER :: input_value, factor, remainder
    LOGICAL :: is_prime
    PRINT *, "What integer greater than or equal to ", &
 &      first_prime, " would you"
    PRINT *, "  like to check to see whether it’s prime?"
    READ *, input_value
    IF (input_value < first_prime) THEN
        PRINT *, "Sorry, I can’t determine whether ", &
 &          input_value, " is a"
        PRINT *, "  prime, because it isn’t at least ", &
 &          first_prime, "."
    ELSE IF (input_value == first_prime) THEN
        PRINT *, "Duh!  Of course ", first_prime, &
 &          " is a prime!"
    ELSE   !! (input_value == first_prime)
        is_prime = .TRUE.
        factor = first_prime
        DO WHILE (is_prime .AND. (factor < input_value))
            remainder = &
 &              input_value - ((input_value / factor) * factor)
            IF (remainder == no_remainder) THEN
                is_prime = .FALSE.
            ELSE   !! (remainder == no_remainder)
                factor = factor + increment
            END IF !! (remainder == no_remainder)...ELSE
        END DO !! WHILE (is_prime .AND. (factor < input_value))
        IF (is_prime) THEN
            PRINT *, "Yes! ", input_value, " is a prime!"
        ELSE   !! (is_prime)
            PRINT *, "Hey! ", input_value, " isn’t a prime!"
            PRINT *, "One of its factors is ", factor, "."
        END IF !! (is_prime)...ELSE
    END IF !! (input_value == first_prime)...ELSE
END PROGRAM it_is_prime
```

We can also nest IF-THEN blocks inside IF-THEN blocks inside DO loops, and DO loops inside IF-THEN blocks inside IF-THEN blocks, and so on, and so on, and so on ...

Nested DO Loop Inside IF-THEN Block Example Run

```
% f95 -o itisprime itisprime.f90
% itisprime
 What integer greater than or equal to  2  would you
   like to check to see whether it’s prime?
 1
 Sorry, I can’t determine whether  1  is a
   prime, because it isn’t at least  2 .
% itisprime
 What integer greater than or equal to  2  would you
   like to check to see whether it’s prime?
 2
 Duh!  Of course  2  is a prime!
% itisprime
 What integer greater than or equal to  2  would you
   like to check to see whether it’s prime?
 3
 Yes!  3  is a prime!
% itisprime
 What integer greater than or equal to  2  would you
   like to check to see whether it’s prime?
 4
 Hey!  4  isn’t a prime!
 One of its factors is  2 .
% itisprime
 What integer greater than or equal to  2  would you
   like to check to see whether it’s prime?
 12345
 Hey!  12345  isn’t a prime!
 One of its factors is  3 .
% itisprime
 What integer greater than or equal to  2  would you
   like to check to see whether it’s prime?
 97
 Yes!  97  is a prime!
```

Nested DO Loops

```
PROGRAM all_primes
    IMPLICIT NONE
    INTEGER,PARAMETER :: first_prime = 2
    INTEGER,PARAMETER :: no_remainder = 0
    INTEGER,PARAMETER :: increment = 1, decrement = -1
    INTEGER :: initial_value,final_value,loop_increment
    INTEGER :: this_value,remainder,factor
    LOGICAL :: is_prime
    PRINT *, "What are the loop bounds that you would like"
    PRINT *, "    to check to see which numbers are prime?"
    READ *, initial_value,final_value
    IF (initial_value < first_prime) THEN
        IF (final_value < first_prime) THEN
            PRINT *, "Hey!    None of the values you want are ", &
 &              first_prime
            PRINT *, "    or greater, so none of them can be primes."
            STOP
        END IF !! (final_value < first_prime)
        PRINT *, "No value less than ", first_prime, &
 &         " is prime, so I'll start at ", first_prime, "."
        initial_value = first_prime
    END IF !! (initial_value < first_prime)
    IF (final_value < first_prime) THEN
        PRINT *, "No value less than ", first_prime, &
 &         " is prime, so I'll end at ", first_prime, "."
        final_value = first_prime
    END IF !! (final_value < first_prime)
    IF (initial_value > final_value) THEN
        loop_increment = decrement
    ELSE   !! (initial_value > final_value)
        loop_increment = increment
    END IF !! (initial_value > final_value)...ELSE
    PRINT *, "Primes from ", initial_value, " to ", final_value, ":"
    DO this_value = initial_value, final_value, loop_increment
        is_prime = .TRUE.
        factor = first_prime
        DO WHILE (is_prime .AND. (factor < this_value))
            remainder = &
 &              this_value - ((this_value / factor) * factor)
            IF (remainder == no_remainder) THEN
                is_prime = .FALSE.
            ELSE   !! (remainder == no_remainder)
                factor = factor + increment
            END IF !! (remainder == no_remainder)...ELSE
        END DO !! WHILE (is_prime .AND. (factor < this_value))
        IF (is_prime) THEN
            PRINT *, this_value
        END IF !! (is_prime)
    END DO !! this_value = initial_value, final_value, loop_increment
END PROGRAM all_primes
```

Output of Nested DO Loop Example

```
% f95 -o allprimes allprimes.f90
% allprimes
 What are the loop bounds that you would like
     to check to see which numbers are prime?
 0 1
 Hey!    None of the values you want are  2
     or greater, so none of them can be primes.
% allprimes
 What are the loop bounds that you would like
     to check to see which numbers are prime?
 2 2
 Primes from  2  to  2 :
 2
% allprimes
 What are the loop bounds that you would like
     to check to see which numbers are prime?
 4 2
 Primes from  4  to  2 :
 3
 2
% allprimes
 What are the loop bounds that you would like
     to check to see which numbers are prime?
 1 100
 No value less than  2  is prime, so I'll start at  2 .
 Primes from  2  to  100 :
 2
 3
 5
 7
 11
 13
 17
 19
 23
 29
 31
 37
 41
 43
 47
 53
 59
 61
 67
 71
 73
 79
 83
 89
 97
```

Changing the Loop Bounds Inside the Loop: BAD BAD BAD!

```
% cat loopbndschg.f90
PROGRAM loop_bounds_change
    IMPLICIT NONE
    INTEGER :: initial_value, final_value, &
&                       maximum_value
    INTEGER :: count, sum = 0
    PRINT *, "What are the initial, final and ", &
&       "maximum values?"
    READ *, initial_value, final_value, maximum_value
    DO count = initial_value, final_value
        sum = sum + count
        IF (sum > maximum_value) THEN
            ! BAD BAD BAD
            ! BAD BAD BAD
            ! BAD BAD BAD

            final_value = final_value - 1 ! BAD BAD BAD

            ! BAD BAD BAD
            ! BAD BAD BAD
            ! BAD BAD BAD
        END IF !! (sum > maximum_value)
        PRINT *, "count = ", count, ", sum = ", sum, &
&          ", final_value = ", final_value
    END DO !! count = initial_value, final_value
    PRINT *, "sum = ", sum
END PROGRAM loop_bounds_change
% f95 -o loopbndschg loopbndschg.f90
% loopbndschg
 What are the initial, final and maximum values?
 1, 10, 40
 count =  1 , sum =  1 , final_value =  10
 count =  2 , sum =  3 , final_value =  10
 count =  3 , sum =  6 , final_value =  10
 count =  4 , sum =  10 , final_value =  10
 count =  5 , sum =  15 , final_value =  10
 count =  6 , sum =  21 , final_value =  10
 count =  7 , sum =  28 , final_value =  10
 count =  8 , sum =  36 , final_value =  10
 count =  9 , sum =  45 , final_value =  9
 count =  10 , sum =  55 , final_value =  8
 sum =  55
```

Changing the Loop Index Inside the Loop: ILLEGAL!

```
% cat loopidxchg.f90
PROGRAM loop_index_change
    IMPLICIT NONE
    INTEGER :: initial_value, final_value, &
 &             maximum_value
    INTEGER :: count, sum = 0
    PRINT *, "What are the initial, ", &
&       "final and maximum values?"
    READ *, initial_value, final_value, &
 &      maximum_value
    DO count = initial_value, final_value
        sum = sum + count
        IF (sum > maximum_value) THEN
            ! ILLEGAL ILLEGAL ILLEGAL
            ! ILLEGAL ILLEGAL ILLEGAL
            ! ILLEGAL ILLEGAL ILLEGAL

            count = count + 1 ! ILLEGAL ILLEGAL

            ! ILLEGAL ILLEGAL ILLEGAL
            ! ILLEGAL ILLEGAL ILLEGAL
            ! ILLEGAL ILLEGAL ILLEGAL
        END IF !! (sum > maximum_value)
        PRINT *, "count = ", count, ", sum = ", sum, &
&           ", final_value = ", final_value
    END DO !! count = initial_value, final_value
    PRINT *, "sum = ", sum
END PROGRAM loop_index_change
% f95 -o loopidxchg loopidxchg.f90
Error: loopidxchg.f90, line 17:
    Assignment to DO variable COUNT
    detected at COUNT@=
[f95 terminated - errors found by pass 1]
```