**DO WHILE Loop Outline**

1. DO WHILE Loop Outline
2. Repetition/
   Looping
3. The DO WHILE Loop
4. DO WHILE Loop Flowchart
5. DO WHILE Loop Example
6. DO WHILE Loop Example’s Flowchart
7. How Many Times Will the Body of a DO WHILE Loop Be Executed?
8. An Infinite Loop
9. What Kind of Statements Can Go Inside a DO WHILE Loop?
10. Jargon: Compound Statement
11. Another DO WHILE Loop Example
12. Another DO WHILE Loop Example (continued)
13. Yet Another DO WHILE Loop Example
14. States and Traces
15. Tracing the Loop

See *Programming in Fortran 90/95*, 1st or 2nd edition, Chapter 13, section 13.4.
Repetition

*Repetition* means performing the same set of statements over and over. The most common way to perform repetition is via *looping*.

Looping

A *loop* is a sequence of statements to be executed, in order, over and over, until some condition is reached.

Fortran 90 has a loop construct known as the `DO WHILE` loop:

```fortran
DO WHILE ( condition )
    statement1
    statement2
    
    END DO
```

Here, the condition is a `LOGICAL` expression completely enclosed in parentheses, just like in an `IF` block.

The sequence of statements between the `DO WHILE` statement and the `END DO` statement is known as the *loop body*. 
The **DO WHILE** Loop

DO WHILE ( condition )
    statement1
    statement2
    ...
END DO

When a **DO WHILE** statement is encountered, the following sequence of events occurs, in order:

1. The condition (a **LOGICAL** expression completely enclosed in parentheses) is evaluated, resulting in a value of either .TRUE. or .FALSE.
2. If the condition evaluates to .FALSE., then the statements inside the loop body are skipped, and control is passed to the statement immediately after the **END DO** statement.
3. If the condition evaluates to .TRUE., then the statements inside the loop body are executed in sequence.
4. When the **END DO** statement is encountered, the program jumps back up to the associated **DO WHILE** statement and starts over with Step 1.

A **DO WHILE** loop is similar to an **IF** block, except that:

- unlike an **IF** block, when a **DO WHILE** loop encounters its **END DO** statement, it jumps back up to the associated **DO WHILE** statement;
- unlike an **IF** block, a **DO WHILE** loop can have only one clause, which is analogous to the **IF** clause; a **DO WHILE** loop cannot have anything analogous to an **ELSE IF** clause or an **ELSE** clause.
DO WHILE Loop Flowchart

statement_before
DO WHILE (condition)
    statement_inside1
    statement_inside2
    ...
    statement_inside_last
END DO
statement_after
DO WHILE Loop Example

% cat chickenidiot_dowhile.f90
PROGRAM chicken_idiotproof_do_while
   IMPLICIT NONE
   INTEGER,PARAMETER :: no_chickens = 0
   INTEGER :: number_of_chickens
   PRINT *, "How many chickens did you think of?"
   READ *, number_of_chickens
   DO WHILE (number_of_chickens < no_chickens)
      PRINT *, "ERROR: you can’t think of ", &
           "negative chickens!"
      PRINT *, "So really, how many chickens ", &
           "did you think of?"
      READ *, number_of_chickens
   END DO
   PRINT *, "The number of chickens is valid."
END PROGRAM chicken_idiotproof_do_while
% f95 -o chickenidiot_dowhile chickenidiot_dowhile.f90
% chickenidiot_dowhile
   How many chickens did you think of?
   -1
   ERROR: you can’t think of negative chickens!
   So really, how many chickens did you think of?
   -22
   ERROR: you can’t think of negative chickens!
   So really, how many chickens did you think of?
   0
   The number of chickens is valid.
% chickenidiot_dowhile
   How many chickens did you think of?
   12
   The number of chickens is valid.
DO WHILE Loop Example’s Flowchart

READ *, number_of_chickens
DO WHILE (number_of_chickens < no_chickens)
   PRINT *, "ERROR: you can’t think of", &
   & "negative chickens!"
   PRINT *, "So really, how many chickens ", &
   & "did you think of?"
   READ *, number_of_chickens
END DO !! WHILE (number_of_chickens < no_chickens)
PRINT *, "The number of chickens is valid."

Prompt for # chickens.

Input # chickens.

chickens < 0 False

Output error.

Prompt again.

Input # chickens again.

Output valid.
How Many Times Will the Body of a **DO WHILE** Loop Be Executed?

Recall:

```plaintext
DO WHILE ( condition )
    statement1
    statement2
    ...
END DO
```

1. The condition (a **LOGICAL** expression completely enclosed in parentheses) is evaluated, resulting in a value of either **.TRUE.** or **.FALSE.**
2. If the condition evaluates to **.FALSE.**, then the statements inside the loop body are skipped, and control is passed to the statement immediately after the **END DO** statement.
3. If the condition evaluates to **.TRUE.**, then the statements inside the loop body are executed in sequence.
4. When the **END DO** statement is encountered, the program jumps back up to the associated **DO WHILE** statement and starts over with Step 1.

Suppose that, the first time that the **DO WHILE** statement is encountered, its condition evaluates to **.FALSE.**

In that case, the loop body will be skipped completely.

So, it may be the case that the loop body is not executed at all, or it may be executed once, or twice, or many times.
An Infinite Loop

An *infinite loop* is a loop that never reaches its termination condition. For example:

```
% cat infiniteloop.f90
PROGRAM infinite_loop
  IMPLICIT NONE
  INTEGER,PARAMETER :: my_number = 5
  INTEGER :: your_number
  PRINT *, "Enter an integer:" 
  READ *, your_number 
  PRINT *, "I had ", my_number, "."
  DO WHILE (your_number < my_number)
    PRINT *, "Your number is less than mine!"
  END DO !! WHILE (your_number < my_number)
END PROGRAM infinite_loop
% f95 -o infiniteloop infiniteloop.f90
% infiniteloop
Enter an integer:
  8
I had 5 .
% infiniteloop
Enter an integer:
  3
I had 5 .
Your number is less than mine!
Your number is less than mine!
Your number is less than mine!
Your number is less than mine!
Your number is less than mine!
Your number is less than mine!
Your number is less than mine!
...

Ctrl-C
```

On most Unix systems, including *roosevelt*, *lincoln* and *kennedy*, you can quit out of a program that is currently executing by typing

```
Ctrl-C
```
What Kind of Statements Can Go Inside a DO WHILE Loop?

Between the DO WHILE statement and the END DO statement, there can be any kind of executable statements, and any number of them. For example:

- PRINT statements;
- READ statements;
- assignment statements;
- STOP statements;
- IF blocks;
- DO WHILE loops.

There are other kinds of executable statements that can occur inside a DO WHILE loop, some of which we’ll learn later in the semester.

In the event that the DO WHILE condition evaluates to .TRUE., these statements will be executed one by one, in the order in which they appear in the DO WHILE loop.

Notice that a DO WHILE loop CANNOT contain declaration statements, because the DO WHILE statement is an executable statement, and ALL declarations MUST come before ANY executable statements.
Jargon: Compound Statement

A compound statement is a sequence of statements with a well-defined beginning and a well-defined end that are executed, in order, under certain circumstances.

Examples of compound statements include IF blocks and DO WHILE loops.

Although an IF block is actually a sequence of statements, we can think of it as a single “super” statement in some contexts. The same is true of a DO WHILE loop.

Compound statements are sometimes called blocks. Thus, we speak of an IF block. A DO WHILE loop is also a kind of block.
Another DO WHILE Loop Example

PROGRAM warmer_colder
IMPLICIT NONE
INTEGER,PARAMETER :: minimum_number = 1
INTEGER,PARAMETER :: maximum_number = 100
INTEGER,PARAMETER :: my_number = 32
INTEGER,PARAMETER :: close_distance = 5
INTEGER,PARAMETER :: very_close_distance = 1
INTEGER,PARAMETER :: negative_distance = -1
INTEGER,PARAMETER :: no_distance = 0
INTEGER :: your_number, your_distance
INTEGER :: your_last_distance = negative_distance
LOGICAL :: correct_number_hasnt_been_input = .TRUE.

PRINT *, "I’m thinking of a number between ", &
  minimum_number, " and ", maximum_number, "."
DO WHILE (correct_number_hasnt_been_input)
  PRINT *, "What number am I thinking of?"
  READ *, your_number
  IF ((your_number < minimum_number) .OR. &
    (your_number > maximum_number)) THEN
    PRINT *, "Hey! That’s not between ", &
    minimum_number, " and ", maximum_number, "."
  ELSE IF (your_number == my_number) THEN
    PRINT *, "That’s amazing!"
    correct_number_hasnt_been_input = .FALSE.
  ELSE !! (your_number == my_number)
    your_distance = ABS(your_number - my_number)
    IF (your_distance == very_close_distance) THEN
      PRINT *, "You’re incredibly hot!"
    ELSE IF (your_last_distance < no_distance) THEN
      PRINT *, "Not bad for your first try."
    ELSE IF (your_distance < your_last_distance) THEN
      PRINT *, "You’re getting warmer ...."
    ELSE IF (your_distance > your_last_distance) THEN
      PRINT *, "Ouch! You’re getting colder."
    ELSE !! (your_distance > your_last_distance)
      PRINT *, "Uh oh. You made no progress."
    END IF !! (your_distance > your_last_distance)...ELSE
    your_last_distance = your_distance
  END IF !! (your_number == my_number)...ELSE
  END DO !! WHILE (correct_number_hasnt_been_input)
PRINT *, "Good for you!"
END PROGRAM warmer_colder
Another DO WHILE Loop Example (continued)

% f95 -o warmercolder warmercolder.f90
% warmercolder
I’m thinking of a number between 1 and 100. What number am I thinking of?
-22
Hey! That’s not between 1 and 100! I’ll pretend you didn’t say that.
What number am I thinking of?
50
Not bad for your first try.
What number am I thinking of?
44
You’re getting warmer....
What number am I thinking of?
22
You’re getting warmer....
What number am I thinking of?
11
Ouch! You’re getting colder.
What number am I thinking of?
35
You’re getting warmer....
What number am I thinking of?
37
Ouch! You’re getting colder.
What number am I thinking of?
33
You’re incredibly hot!
What number am I thinking of?
32
That’s amazing!
Good for you!
Yet Another **DO WHILE** Loop Example

```fortran
% 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
  sum = 0
  current_value = initial_value
  DO WHILE (current_value <= final_value)
    sum = sum + current_value
    current_value = current_value + 1
  END DO
  PRINT *, "The sum of the integers from ", &
   " 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 .
```
States and Traces

The state of a program is the set of values of all of its variables at a given moment during execution; that is, it’s a snapshot of the memory that’s being used.

A trace of a program is a listing of the state of the program after each statement is executed.

Suppose that, in the previous example program, the user input 1 for initial_value and 5 for final_value.

Let’s examine the program fragment around the loop.

```plaintext
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)
```

If we number these statements, we have:

1 sum = 0
2 current_value = initial_value
3 DO WHILE (current_value <= final_value)
4    sum = sum + current_value
5    current_value = current_value + 1
6 END DO ! ! WHILE (current_value <= final_value)

Tracing helps us to examine the behavior of a piece of code, and so it sometimes can be useful in debugging.
Tracing the Loop

```plaintext
1 sum = 0
2 current_value = initial_value
3 DO WHILE (current_value <= final_value)
   4 sum = sum + current_value
   5 current_value = current_value + 1
4 END DO !! WHILE (current_value <= final_value)
```

<table>
<thead>
<tr>
<th>Snapshot of Iteration #</th>
<th>After Stmt #</th>
<th>Trace Value of sum</th>
<th>Trace Value of count</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>N/A</td>
<td>1</td>
<td>0</td>
<td>garbage</td>
<td>Haven’t entered loop yet</td>
</tr>
<tr>
<td>N/A</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>Haven’t entered loop yet</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>Condition evaluates to .TRUE.</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>new sum = old sum + count = 0 + 1 = 1</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>1</td>
<td>2</td>
<td>new count = old count + 1 = 1 + 1 = 2</td>
</tr>
<tr>
<td>1</td>
<td>6</td>
<td>1</td>
<td>2</td>
<td>jump back up to stmt #3 to start iteration #2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>Condition evaluates to .TRUE.</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>new sum = old sum + count = 1 + 2 = 3</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>3</td>
<td>3</td>
<td>new count = old count + 1 = 2 + 1 = 3</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>3</td>
<td>3</td>
<td>jump back up to stmt #3 to start iteration #2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>Condition evaluates to .TRUE.</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>6</td>
<td>3</td>
<td>new sum = old sum + count = 3 + 3 = 6</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>6</td>
<td>4</td>
<td>new count = old count + 1 = 3 + 1 + 4</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>6</td>
<td>4</td>
<td>jump back up to stmt #3 to start iteration #2</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>6</td>
<td>4</td>
<td>Condition evaluates to .TRUE.</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>10</td>
<td>4</td>
<td>new sum = old sum + count = 6 + 4 = 10</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>10</td>
<td>5</td>
<td>new count = old count + 1 = 4 + 1 = 5</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>10</td>
<td>5</td>
<td>jump back up to stmt #3 to start iteration #2</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>10</td>
<td>5</td>
<td>Condition evaluates to .TRUE.</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>15</td>
<td>5</td>
<td>new sum = old sum + count = 10 + 5 = 15</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>15</td>
<td>6</td>
<td>new count = old count + 1 = 5 + 1 = 6</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>15</td>
<td>6</td>
<td>jump back up to stmt #3 to start iteration #2</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>15</td>
<td>6</td>
<td>Condition evaluates to .FALSE., loop exited</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>15</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>