# Control Structures

(Loop Structures)

As discussed earlier, structured programs have easy-to-understand logic, partly because the programs are modularized (that we'll cover in future lectures). In addition, these programs should attempt to use only three basic logic structures:

- sequences
- decision structures
- loop structures

So far, we have learned to write programs that use ***sequences*** and ***decision structures***. Now we'll take a look at ***loop structures***, which allows a specified group of statements to be executed as many times as needed. We'll be looking at two types of looping statements: the `DO…LOOP` (which includes the `DO WHILE…LOOP`, the `DO…LOOP WHILE`, the `DO UNTIL…LOOP`, and the `DO…LOOP UNTIL`) and the `FOR…NEXT` loop.

## Loop structures

### Advantages of Loop Structures

Often a situation arises where a task must be performed repeatedly. For example, an instructor may need a program to find the average test score for each student in a course. A program to process one student's data is simple enough.

```
CLS

DIM grade1 AS SINGLE
DIM grade2 AS SINGLE
DIM grade3 AS SINGLE
DIM average AS SINGLE

DIM sName AS STRING

INPUT "Enter the student's name:", sName
INPUT "Enter Grade1, Grade2, and Grade3:", grade1, grade2, grade3

average = (grade1 + grade2 + grade3) / 3
PRINT sName; "'s average is: ";
PRINT USING "##.#"; average;
PRINT "%"

END
```

However, consider the problem of repeating this for 200 students. The same statements used to process one student's data would have to be written 200 times – a tedious and error-prone task. To simplify the job the statements could be written once, then executed as many times as necessary. This process is called looping (or iteration).

A variety of loop structures exist among the many programming languages used today, but all share some basic components. They all use a *loop-control-condition*, whose value determines whether a loop will be repeated. In the example of the student test scores, the *loop-control-condition* could contain the current number of students processed. All loops contain some action to be performed repeatedly; the statements that perform such an action make up the loop *body* **[***statement block***]**. In the student test score example, the loop *body* **[***statement block***]** would consist of three steps:

- Prompt the user to enter the name and scores.
- Calculate the average.
- Print the results.

Execution of basic loop structures consists of five steps:

1. The *loop-control-condition* is initialized to a particular value before loop execution begins.
2. The program tests the *loop-control-condition* to determine whether to execute the loop *body* **[***statement block***]** or exit from the loop.
3. The loop *body* **[***statement block***]**, consisting of any number of statements, is executed.
4. At some point during loop execution, the value of the *loop-control-condition* is modified to allow an exit from the loop.
5. The loop is exited when the decision of Step 2 determines that the right number of loop iterations has been made. Execution continues with the next statement following the `LOOP`.

## DO…LOOPS

The `DO…LOOP` is controlled by a ***Boolean*** loop-control-condition (remember that ***Boolean*** loop-control-conditions always evaluate as either `TRUE` or `FALSE`). Here we are going to discuss two variations of this loop: the `DO WHILE…LOOP` and the `DO UNTIL…LOOP`.

**The `DO WHILE…LOOP`:** executes while a *loop-control-condition* is `TRUE`

**`DO WHILE…LOOP` Statement Format:**

**`DO WHILE`** *loop-control-condition*
**[***statement block***]**
**`LOOP`**

A `DO WHILE…LOOP` is executed as long as the *loop-control-condition* at the top of the loop is `TRUE`. If it is `FALSE`, control transfers to the first statement after `LOOP`, which marks the loop's end.

Following is a short program that demonstrates the use of a simple `DO WHILE…LOOP`.

```
CLS

DIM sum AS INTEGER
DIM number AS INTEGER

sum = 0

INPUT "Enter the first number (–1 to quit): ", number

DO WHILE number <> –1
    sum = sum + number
    INPUT "Enter the next number (–1 to quit): ", number
LOOP
PRINT "The sum is: "; sum
END
```

Execution of the above DO WHILE…LOOP proceeds as follows:

1. The *loop-control-condition* in the DO WHILE is evaluated as TRUE or FALSE. In the preceding program, as long as the *loop-control-condition* (number <> –1) is TRUE, the *body* **[***statement block***]** of the loop is executed.
2. If the *loop-control-condition* is TRUE, the statements in the loop's *body* **[***statement block***]** are executed until LOOP is encountered; if the *loop-control-condition* is FALSE, control passes to the first statement after LOOP. As with the block IF and the SELECT CASE, the *body* **[***statement block***]** of a DO…LOOP consists of a statement block. Its boundaries are clearly marked by the keywords DO and LOOP. The *body* **[***statement block***]** of the loop in the preceding code snippet contains two statements:

```
sum = sum + number
INPUT "Enter the next number (–1 to quit): ", number
```

3. When LOOP is encountered, control passes back to the DO WHILE, and the *loop-control-condition* is checked again.
4. If the *loop-control-condition* is still TRUE, the loop *body* **[***statement block***]** is executed again; if FALSE, the loop is exited.

You may be wondering why this program contains two INPUT statements.

The first INPUT statement…

```
INPUT "Enter the first number (–1 to quit): ", number
```

gets the first number before the loop is entered:

This is referred to as a *priming value* and initializes number to a starting value before the condition in the DO WHILE is evaluated for the first time.

The second INPUT…

```
INPUT "Enter the next number (–1 to quit): ", number
```

is at the bottom of the loop and reads the subsequent numbers. It is important that this second `INPUT` be the last statement in the loop *body* **[***statement block***]** because if it occurred before `number` was added to `sum`, the first value of `number` would be lost before it was added.

Notice that `sum` is set to `0` (zero) immediately before the loop is entered. This is because we want `sum` to start at `0` (zero) and then add numbers to it.

Technically, however, this statement setting `sum` to `0` (zero) is not need. Before a program is executed, QBasic automatically sets the values of numeric variables to `0` (zero). In addition, string variables are automatically set to the `null` (empty) string. However, it is good programming practice to initialize variables yourself, rather than depending on "`default`" initializations.

## Controlling Loops

One of the most difficult aspects of using loops is making them stop executing at the proper time. There are two primary methods of controlling loop repetition:

1. the use of trailer/sentinel values
2. counting loops

### Trailer/Sentinel Values

A trailer/sentinel value is a "dummy" value that follows, or trails, the data items being processed. The trailer/sentinel value signals to the program that all data has been entered. The loop in the preceding program used a trailer/sentinel value of `-1`. When the user enters that value, the loop stops, and `sum` is displayed. The trailer/sentinel value can be either a numeric value or a character string, depending on the type of data being input, but it should always be a value outside the range of the actual data.

For example:

- If a program prompts the user to enter people's ages, a good trailer/sentinel value might be `-1`.
- If names are being input, an example of a good trailer/sentinel value might be "`Finished`".

As previously mentioned, a priming value should be used to assign a value to the *loop-control-condition* before loop repetition begins. Subsequent values are assigned to the *loop-control-condition* at the bottom of the loop. If the *loop-control-condition's* value does not equal the trailer/sentinel value, the loop is repeated. The following program shows another loop using a character string trailer/sentinel value ("`Finished`").

```
CLS

DIM rate AS SINGLE
DIM hours AS SINGLE
DIM wage AS SINGLE
DIM nme AS STRING

rate = 8
INPUT "Enter name and number of hours: ", nme, hours
DO WHILE nme <> "Finished"
    wage = rate * hours
    PRINT "Name:,"; wage; ""
    PRINT nme, wage
    PRINT
    INPUT "Enter next name and number of hours: ", nme, hours
    PRINT
LOOP

PRINT "Finished"

END
```

This program calculates the `wages` of employees of the Village People Hotel. After each employee's data is entered, the following statement tests for the trailer/sentinel value:

```
DO WHILE nme <> "Finished"
```

If the condition is `TRUE`, the loop's *body* **[***statement block***]** is executed; otherwise, execution passes to the first statement following the `LOOP`.

## Counting Loop

A second method of controlling a loop is to create a *loop-control-condition*, called a *counter*, which keeps track of the number of times the loop has been executed. In a counting loop, the *counter* is incremented or decremented after each iteration. When it reaches a specified value, loop execution stops. A counting loop is useful when the programmer knows, before the loop is first entered, how many times it should execute. The following steps are used to set up a counting loop:

1. Initialize the *counter* by setting it to a beginning value before entering the loop.
2. Increment the *counter* each time the loop is executed.
3. Test the *counter* each time the loop is executed to see if the loop has been repeated the needed number of times.

The following program is a rewrite of the previous to use a *counter* rather than a trailer/sentinel value. Data on three employees must be entered, so the loop is executed three times.

```
CLS

DIM rate AS SINGLE
DIM hours AS SINGLE
DIM wage AS SINGLE
DIM nme AS STRING
DIM counter AS INTEGER

rate = 8
counter = 1

DO WHILE counter <= 3
    INPUT "Enter name and number of hours: ", nme, hours
    wage = rate * hours
    PRINT "Name:,"; wage; ""
    PRINT nme, wage
    PRINT
    counter = counter + 1
LOOP

PRINT "Finished"

END
```

Always be careful to initialize the *loop-control-condition* to the appropriate starting value before entering the loop. In addition, make certain that at some point the *loop-control-condition* reaches the value needed to stop the loop.

In the following loop, the *loop-control-condition* is never modified:

```
count = 1
DO WHILE count < 50
      PRINT count
LOOP
```

This is an example of an **infinite loop**. The loop will be executed infinitely (or at least until the computer's resources are used up) because the *loop-control-condition* is not incremented within the body of the loop.

This loop can be correctly written as follows:

```
count = 1
DO WHILE count < 50
      PRINT count
      count = count + 1
LOOP
```

If your program becomes caught in an infinite loop, stop the execution by pressing ***Ctrl+Scroll Lock*** on the keyboard. You can then correct the problem and choose the ***Continue*** option from the ***Run*** menu to continue execution.

**The DO UNTIL…LOOP:** executes until a *loop-control-condition* is TRUE (these may be used as a counting loop)

**DO UNTIL…LOOP Statement Format:**
**DO UNTIL** *loop-control-condition*
**[***statement block***]**
**LOOP**

The second type of DO…LOOP, the DO UNTIL…LOOP, is very similar to the DO WHILE…LOOP. The only difference is that the loop is executed until the ***Boolean*** *loop-control-condition* becomes TRUE rather than while it is TRUE.

The controlling condition at the top of the DO WHILE…LOOP in the previous program

```
DO WHILE number <> -1
```

would be rewritten as follows:

```
DO UNTIL number = -1
```

Other than this change, the program would remain exactly the same. Rather than executing while number is <> (not equal to) −1, as the DO WHILE…LOOP did the DO UNTIL…LOOP executes until number equals −1.

Because the end result is the same, any DO WHILE…LOOP can be rewritten as a DO UNTIL…LOOP.

The following program shows how the previous DO WHILE…LOOP program could be rewritten using the DO UNTIL…LOOP.

```
CLS

DIM rate AS SINGLE
DIM hours AS SINGLE
DIM wage AS SINGLE
DIM nme AS STRING

rate = 8
INPUT "Enter name and number of hours: ", nme, hours
DO UNTIL nme = "Finished"
    wage = rate * hours
    PRINT "Name:,"; wage; ""
    PRINT nme, wage
    PRINT
    INPUT "Enter next name and number of hours: ", nme, hours
    PRINT
LOOP

PRINT "Finished"

END
```

In addition to placing the *loop-control-condition* at the beginning of a `DO UNTIL…LOOP` or a `DO WHILE…LOOP` we can place the *loop-control-condition* at the end of the loop using one of the following formats:

**DO…LOOP WHILE Statement Format:**

```
DO
        [statement block]
LOOP WHILE loop-control-condition
```

**or**

**DO LOOP…UNTIL Statement Format:**

```
DO
        [statement block]
LOOP UNTIL loop-control-condition
```

In this new format we have moved the *loop-control-condition* to the end of the loop and have fundamentally changed the structure of the loop, thereby forcing it to execute the *statement block* one time before deciding if it should iterate more times.

### The `EXIT` Statement

The `EXIT` statement can lead to program errors. It also makes program logic more difficult to follow. Therefore, it is best to avoid its use.

Occasionally, it may be necessary to exit a loop prematurely. Most commonly, this occurs if an error condition is encountered, for example, if invalid data has been entered. The `EXIT` statement can be placed anywhere in a `DO…LOOP` and causes execution to be immediately transferred to the first statement after `LOOP`. For example, the following program is supposed to prompt the user to enter the names of 12 voters. However, if a value of less than `18` is read for `age` (as it is in this example), an error message is displayed and the loop terminates.

```
CLS

DIM count AS INTEGER
DIM age AS INTEGER
DIM voter AS STRING

count = 1
INPUT "Enter person's name: ", voter
DO WHILE count <= 12
    IF age < 18 THEN
        PRINT
        PRINT voter; " is not eligible to vote."
        EXIT DO
    END IF
    count = count + 1
    INPUT "Enter person's name: ", voter
    INPUT "Enter person's age: ", age
LOOP
```

```
END
```

When the `EXIT DO` statement is executed, control transfers to the first statement after `LOOP`. However, in the preceding example the `EXIT DO` statement is executed only if a value less than `18` is entered for `age` (note: because QBasic is left to initialize the variable `age`, it's starting value is `0` and the loop is exited). The `EXIT` statement can be used with other structures, such as `SUB` procedures. The statement `EXIT SUB` allows control to be transferred immediately back to the calling program.

## The FOR…NEXT Loop: counting loops that execute a pre-specified number of times

**FOR…NEXT Statement Format**:

**FOR** *loop-control-variable* **=** *initial* value **TO** *terminal* value **[STEP** *step-value***]**
**[***statement-block***]**
**NEXT** *loop-control-variable*

The `FOR…NEXT` loop is used to create a counting loop.

For example, this loop:

```
FOR counter = 1 TO 8 STEP 1
      PRINT counter;
NEXT counter
```

executes eight times and outputs the following:

```
1     2     3     4     5     6     7     8
```

The *loop-control-variable* (`counter`) starts at `1` and is incremented by `1` (**`STEP 1`**) at the end of each iteration until its value is greater than `8`. The `FOR` keyword marks the beginning of the loop, and the `NEXT` keyword marks its end; the statement block (`PRINT counter ;`) makes up the loop's *body* **[***statement block***]**. The *step-value* (`1`) determines the amount by which the *loop-control-variable* is incremented.

If the *step-value* were changed to `2`…

```
FOR counter = 1 TO 8 STEP 2
      PRINT counter;
NEXT counter
```

the following would be displayed:

```
1     3     5     7
```

If no *step-value* is specified, QBasic assumes a value of `1`. Therefore, the following two statements are equivalent:

```
FOR counter = 1 TO 8 STEP 1
FOR counter = 1 TO 8
```

The *loop-control-variable* (`counter`) must be a numeric variable. The initial value (`1`) and terminal/sentinel value (`8`) and the optional *step-value* (`1`), taken together, determine the number of times the loop *body* **[***statement*

*block*] is executed. These values must be numeric. They can consist of constants, variables, or loop-control-conditions.

The following program shows how the DO WHILE…LOOP can be rewritten using a FOR…NEXT loop.

```
CLS

DIM rate AS SINGLE
DIM hours AS SINGLE
DIM wage AS SINGLE
DIM nme AS STRING
DIM counter AS INTEGER

rate = 8

FOR counter = 1 TO 3
    INPUT "Enter next name and number of hours: ", nme, hours
    wage = rate * hours
    PRINT "Name:,"; wage; ""
    PRINT nme, wage
    PRINT
NEXT counter

PRINT "Finished"

END
```

A number of actions occur when a FOR statement is first encountered:

1. The *initial*, *terminal*, and (if given) *step-value* are evaluated.
2. The *loop-control-variable* is assigned the *initial* value.
3. The value of the *loop-control-variable* is tested against the *terminal* value.
4. If the *loop-control-variable* is less than or equal to the *terminal* value, the loop *body* [*statement block*] is executed.
5. If the *loop-control-variable* is greater than the *terminal* value, the loop *body* [*statement block*] is skipped, and control passes to the first statement following the NEXT. This means that the loop will not be executed.

Here is what happens when the NEXT is encountered:

1. The *step-value* indicated in the FOR statement is added to the *loop-control-variable*. If the *step-value* is omitted, +1 is added.
2. A check is performed to determine if the value of the *loop-control-variable* exceeds the *terminal* value.
3. If the *loop-control-variable* is less than or equal to the *terminal* value, control transfers to the *body* [*statement block*] of the FOR statement, and the loop is repeated. Otherwise, the loop is exited, and execution continues with the statement following the NEXT.

It is possible to use a negative, rather than a positive, *step-value*. In this case, the *loop-control-variable* is decremented, rather than incremented, after each loop iteration. Therefore, the following statement is valid:

```
FOR count = 8 TO 4 STEP –2
```

When a negative *step-value* is used, the loop *body* **[***statement block***]** is executed if the *loop-control-variable* is greater than or equal to the *terminal* value. When the NEXT is encountered, the *step-value* is added to the *loop-control-variable* as usual; and because this value is negative, the *loop-control-variable* is decremented.

For example, this program segment…

```
FOR count =10 to 1 step –1
     Print count;
NEXT count
PRINT "***  IGNITION!  ***"
```

displays the following:

```
10    9     8     7     6     5     4     3     2     1     ***  IGNITION!  ***
```

### Rules for Using FOR…NEXT Loops

The following rules apply to FOR…NEXT loops:

- The *initial*, *terminal*, and *step-values* cannot be modified in the loop *body* **[***statement block***]**.
- It is possible to modify the *loop-control-variable* in the loop *body* **[***statement block***]**, but this should never be done. Note how unpredictable the execution of the following loop would be. The value of the *loop-control-variable* (count) is dependent on the integer (X) entered by the user.

```
FOR count = 1 TO 10
     INPUT "Enter an integer"; X
     count = X
NEXT count
```

- If the *step-value* is 0 (zero), an infinite loop is created, as in the following example:

```
FOR X = 10 TO 20 STEP 0
```

  This loop could be written correctly so that it would execute 11 times as follows:

```
FOR X = 10 TO 20 STEP 1
```

- Each FOR statement must be associated with a corresponding NEXT statement.

## Nested Loops

- We've seen how block `IF` statements can be nested in one another. Similar nesting can be done with all types of loops. A pair of nested `FOR…NEXT` loops might look like this snippet:

```
FOR outer = 1 TO 4
      FOR inner = 1 TO 2
            .
            .
            .
      NEXT inner
NEXT outer
```

Nested loops should always be indented as shown to improve readability.

In the following example, each time the outer loop is executed once, the inner loop is executed twice, since the *loop-control-variable* (`inner`) varies from `1` to `2`. When the inner loop has terminated, control passes to the first statement after the `NEXT` for the inner loop, which in this case is the `NEXT` for the outer loop. This statement causes the *loop-control-variable* (`outer`) to be incremented by `1` and tested against the outer loop's *terminal* value of `4`. If the *loop-control-variable* (`outer`) is still less than or equal to the outer loop's *terminal* value (`4`), the outer loop *body* **[***statement block***]** is executed again, causing the inner loop to once again be initiated which resets the value of the *loop-control-variable* (`inner`) to `1`. The inner loop is executed repeatedly (loops) until the *loop-control-variable* (`inner`) is greater than `2`. Altogether, the outer loop is executed four times and the inner loop is executed eight (4 * 2) times.

Nesting loops can easily lead to errors so be sure to follow these rules:

- Each loop must have a unique *loop-control-variable*.
- The `NEXT` statement for an inner loop must appear within the *body* **[***statement block***]** of the outer loop, so that one loop is entirely contained within another.

| Incorrect | Correct |
|---|---|
| `FOR i = 1 TO 5`<br>`  FOR j = 1 TO 10`<br>`  .`<br>`  .`<br>`  .`<br>`  NEXT i`<br>`NEXT j` | `FOR i = 1 TO 5`<br>`  FOR j = 1 TO 10`<br>`  .`<br>`  .`<br>`  .`<br>`  NEXT j`<br>`NEXT i` |

In the incorrect example above, the outer `j` loop is not entirely inside the inner `i` loop. We can see that it extends beyond the inner `i` loop's `NEXT` statement.

The following program uses nested loops to print the multiplication tables for the numbers `1`, `2`, and `3`, with each table in a separate column. The `inner` loop controls the printing in each of the three columns, while the `outer` loop controls the printing of the rows.

Check the output of the program by following the execution from beginning to end, performing each statement by hand. (This is referred to as *desk checking* the program.)

```
CLS

FOR outer = 1 TO 10
    FOR inner = 1 TO 3
        PRINT inner; "*"; outer; "="; inner * outer,
    NEXT inner
    PRINT
NEXT outer

END
```

## Comparing Loop Structures

Either of the DO…LOOPs can be used in place of a FOR…NEXT loop, but the reverse is not true.

FOR…NEXT loops are the best choice for creating counting loops, that is loops that execute a pre-specified number of times (the number of iterations being determined by the FOR…NEXT loop's *initial* value, *terminal* value and its *loop-control-variable*). And they're less error prone when used for counting loops.

FOR…NEXT loops by design:

- Initialize and increment their *loop-control-variable*.
- Compare their *loop-control-variable* to their *terminal* value.

Because DO…LOOPs can be programmed to execute a given number of times they could be used to create counting loops, but using them requires the programmer to:

- Write a statement to initialize the *loop-control-variable* before the loop begins.
- Write a statement at the top of the loop to compare the *loop-control-variable* to a given *terminal* value.

On the other hand, because a FOR…NEXT loop can't use trailer/sentinel values to exit the loop, if the specific number of loop executions (iterations) is not known beforehand only a DO…LOOP that uses a trailer/sentinel value will work.

## In Closing

It is our job as programmers to know what tools are available to us in any given programming language and to understand how each of them works. It is then our responsibility to select the most appropriate of these tools for the job at hand.