Program Flow Control in Looping Code Structures
John R Whittington, Mediscience Services Ltd., Buckingham, UK

ABSTRACT

In common with all programming languages, looping code constructs are a major part of the SAS Language. The type of looping constructs found in the SAS Language are reviewed and discussed, with particular reference to the facilities available for program flow control within these loops.

INTRODUCTION

Looping code constructs, which are frequently nested, are a major feature of virtually all programming languages, not the least because computer processing of data is probably most valuable in relation to repetitive or iterative processes. The SAS Language is fairly unique in that one level of looping, through records in a set of data, occurs implicitly within the SAS DATA step - the primary code structure of the language. As we all know, beginners almost invariably make some mistakes by not remembering that most statements occurring within a DATA step will be executed repeatedly, once during each iteration of that step.

In terms of SAS Language programming, the DATA step invariably represents the outermost (implicit) loop, any other looping structures being within a DATA step. Somewhat to the frustration of the author, there is no direct facility in the SAS Language for a DATA step or any SAS procedures (PROCs) to be included within any loop. When programming calls for this, it can only be achieved by use of the SAS macro facility, - which functionally enables a DATA step or PROC to be placed within a (%DO ... %END) loop. Whilist such macro loops are broadly comparable with those used within DATA steps, they will not be considered here.

TYPES OF LOOP

By far the majority of 'looping' within a DATA step occurs in the context of one of the types of DO ... END loop (DO ..., END, DO WHILE ..., END and DO UNTIL ... END). However, it is also possible to re-use blocks of program code by means of LINK statements - which can occur alone or in association with DO loops.

EXPLICIT UNCONDITIONAL LOOP CONTROL

The simplest form of the DO ..., END construct, with which we are all familiar, involves explicit unconditional specification of the looping behaviour, for example:

```
data test;
do i = 0 to 10 by 2;
x = i * 5;
  output;
end;
run;
```

Such a loop executes a fixed number of times (6 in the above example), as determined by the range and increment figures specified in the 'iterative' DO statement.

It is not universally realised that the explicit range of index variable values can take more complex forms than the above, such as the following examples:

```
do i = 0 to 10 by 2, 20 to 50 by 5, 60 to 100 by 20;
do i = 1, 3, 7, 10 to 20 by 2, 50, 80 to 85;
do i = 1, 3, 7, 6 to 18 by 2, 12, 80 to 85;
```

In other words, any number of individual values or ranges (with or without specified non-unity increments) can be combined, separated by commas. As the last example above illustrates, there is no requirement for the values to occur in numerical order; they can go 'backwards and forwards' and particular values can, if required, occur more than once.

Index variables used in iterative DO loops (e.g. 1 above) are not automatically DROPPed, and therefore should be explicitly DROPPed if not required.

IMPLICIT UNCONDITIONAL LOOP CONTROL

One very specific type of looping is that which occurs through all the elements of an implicitly-dimensioned array by utilising the DO OVER statement. For example:

```
data one;
  array mine v1-v4;
  set test;
do over mine;
  if mine<20 then mine=mine*-1;
  end;
run;
```

Again, the number of iterations of the loop is fixed, being equal to the number of elements in the array; in the code within the loop (the 'DO group'), the array name ('mine' in this case) is used to represent the current element of the array being processed.

CONDITIONAL LOOP CONTROL IN DO STATEMENT: DO WHILE & DO UNTIL

Control of looping can be achieved by use of DO WHILE and DO UNTIL statements, as in the following examples:

```
data one;
do while (v2 < 50);
v2 = v1 * 10;
v1 + 1;
output;
end;
run;
```

```
data one;
do until (v2 > 40);
v2 = v1 * 10;
v1 + 1;
output;
end;
run;
```
In simple situations, as above, any desired loop terminating condition can be specified by a DO WHILE statement or by the corresponding DO UNTIL one. However, there is one very important difference between DO WHILE and DO UNTIL which becomes particularly important if variables affecting the conditional test can be altered outside of (and prior to) the DO loop - a situation particularly common when the loop in question is nested inside one or more others.

The difference is that the test of the DO WHILE expression is conducted 'at the top of the loop' (prior to an iteration), whilst the test for a DO UNTIL expression is conducted 'at the bottom' of the loop (after an iteration has been completed). The most important consequence of this is that any pre-existing value (e.g. from a higher level loop, or an assignment statement) for a variable which occurs in a WHILE expression will be utilised in a test of that expression prior to the first iteration of the DO loop. If that test produces a 'false' result, then not even one pass through the DO loop will be executed. In contrast, a DO UNTIL loop will always execute at least one iteration, since the test is not performed until the end of that first execution. In the first of the following examples, using DO WHILE, no execution of the loop occurs, since the pre-existing value of v2 causes a false test result at the top of the first loop. In the second example, however, the DO UNTIL loop does execute the first loop, even though the pre-existing value of v2 results in a terminating result for the test; subsequent iterations then also occur, since the value of v2 is modified by the DO group code:

```plaintext
data one;
  v2 = 51;
do while (v2 < 50);
  v2 = v1 * 10;
  v1 + 1;
  output;
end;
run;

data one;
  v2 = 41;
do until (v2 > 40);
  v2 = v1 * 10;
  v1 + 1;
output;
end;
run;
```

It should also be remembered that, in some coding situations (such as some examples above), there may be few or no variables with non-missing values prior to the first pass through a DO WHILE loop. In such situations, it is very probable that the conditional test at the top of that loop would fail and that the loop would never be executed - so that a DO UNTIL construct would be necessary. Some DO WHILE examples above 'get away with it' because the condition was of the form WHILE (var < X), a condition which is satisfied by a missing value.

**CONDITIONAL TERMINATION OF AN EXPLICITLY SPECIFIED LOOP**

Loops conditionally controlled by DO WHILE and DO UNTIL statements clearly depend upon tests of variable(s) which alter during iterations of the loop - which would otherwise loop endlessly. In the simple examples given above (which could have been coded explicitly), incrementing of a variable during each loop iteration ensures that the exit condition will be met after a certain finite number of iterations.

However, in other situations, the conditional test may not be satisfied for a very large number of iterations or, in some situations, may never be satisfied. For example:

```plaintext
data one;
do until (v = 0.12345);
  v = ranuni(1453678);
  output;
  end;
run;
```

In order to limit the maximum number of possible iterations, one can combine an explicit specification of the range of values for an index variable with a WHILE or UNTIL clause, the latter usually being safer because it ensures that the DO loop does indeed start. For example:

```plaintext
data one;
do i = 0 to 100 until (v = 0.12345);
  v = ranuni(1453678);
  output;
end;
run;
```

This DO loop will clearly iterate a maximum of 100 times, terminating prior to this if the UNTIL condition is met. A similar technique can be used if one wants only to utilise certain values of a variable, for example:

```plaintext
data one;
do v1 = 0 to 80 by 4 until (v3 = 12340);
  v2 = v1 * v2;
  output;
end;
run;
```

This will iterate through values of v1 of 0, 4, 8 ...... 80 or until the UNTIL condition is met.

**CONDITIONAL TERMINATION OF A DO LOOP FROM WITHIN THAT LOOP**

Situations so far considered have been those in which conditional decisions regarding loop termination are made on the basis of tests performed either at 'the top' or 'the bottom' of a DO loop. There are however, situations in which one wishes to be able to terminate a loop during the course of one of it's executions. This typically occurs if one not only wants to avoid any further iterations of the loop, but also wants to avoid some of the code within the DO group (e.g. INPUT or OUTPUT statements).

This can be achieved with a conditional LEAVE statement, which results in execution of the present iteration to cease immediately and for no more iterations of the loop to occur (regardless of any other exit conditions specified by DO, DO WHILE or DO UNTIL). When the loop in nested within others, only the current loop (and any nested within it) is affected; outer loops continue as they would when the present loop terminated for any reason. For example:

```plaintext
data test;
do i = 1 to 20;
  if i * 10 > 150 then leave;
  output; /* or more complex code segment */
end;
run;
```
In a simple example like this, the same effect could obviously be achieved by making the OUTPUT statement, or whatever, conditional (or jumping over it conditionally, with a GOTO), and combining this with conditions in the DO statement - but use of LEAVE will normally be the most convenient approach in such situations.

CONDITIONAL PREMATURE TERMINATION OF ONE ITERATION OF A DO LOOP

A related situation arises when one wishes to abort the rest of the code in a BY group, but then resume the next iteration as if the present one had been completed normally. This can be achieved with a CONTINUE statement, for example:

```
data test;
do i = 1 to 20;
   if i * 10 = 120 then continue;
      /* other code in here */
end;
run;
```

Again, in very simple situations like this, the same effect could be achieved by making one or more of the DO group statements conditional (e.g. an INPUT or OUTPUT statement). However, CONTINUE can be used when there are appreciable blocks of DO group code (e.g. INPUT statements) that one wishes to avoid when the condition is met. Much the same effect can be achieved by jumping over unwanted code using GOTO statement and a label:

```
data test;
do i = 1 to 20;
   if i * 10 = 120 then goto jump;
      /* other code in here */
jump: ;
end;
run;
```

LOOPING THROUGH CODE WITH LINK AND GOTO CONSTRUCTS

LINK and GOTO statements both result in program execution jumping to a point, specified by a label, within the same DATA step. Execution then continues until a RETURN statement is encountered or the bottom of the DATA step is reached. If an explicit RETURN statement is encountered, execution then returns to the top of the DATA step (following GOTO) or to the statement following the LINK statement.

The use of GOTO to jump over an unwanted segment of code has already been discussed. LINK may be used to repeatedly execute the same segment of code, as a 'subroutine'. Where the situation permits, the built-in termination condition tests of a DO loop normally make this the more convenient approach for repetitive execution of a code segment. However, there are some situations in which LINK ... RETURN can achieve things which are impossible with a DO loop - for example when a DO loop would require control in terms of more than one index variable. The DO loop approach would then require repetition of the loop, for example:

```
do i = 0 to 10 by 2;
   /* code here */
end;
do j = 2 to 8;
   /* code here */
end;
do k = 4 to 20 by 4;
   /* code here */
end;
```

If the block of DO group code was lengthy, and identical in all three loops, this would be a tedious approach (although there are macro methods which could be employed. However, an alternative approach, utilising both DO loops and LINK / RETURN would be:

```
do i = 0 to 10 by 2;
   link mycode;
end;
do j = 2 to 8;
   link mycode;
end;
do k = 4 to 20 by 4;
   link mycode;
end;
return; /* prevents mysub code being executed after third DO loop */
mycode: /* code here */
return;
```

LINK / RETURN can also be used in situations where a DO loop would be of no value at all, for example:

```
l = 2; j=5; link mycode;
l=7; j=23; link mycode;
l=13; j=47; k=21; link mycode;
return;
mycode: /* code here */;
return;
```

Such situations are uncommon, but it is worth bearing in mind that LINK can occasionally be of value for repetitive execution of code without resort to a macro. Using LINK to share code segments within different DO groups (even nested ones) is theoretically possible, but any attempt to use GOTO statements to jump between different DO loops is so potentially hazardous as to be very ill-advised.

CONCLUSION

Various methods of controlling behaviour of DO ... END loops have been reviewed and discussed, together with a brief review of the use of LINK / RETURN for repetitive code execution.

SAS is a registered trademark of SAS Institute Inc. in the USA and other countries. © indicates USA registration

John R Whittington
Mediscience Services Ltd.,
Twyford Manor,
Twyford, Buckingham MK18 4EI, UK
Voice: +44 1296 730225
Fax: +44 1296 738893
e-mail: johnw@mag-net.co.uk
100517.3677@compuserve.com