The Magnificent DO

Paul M. Dorfman
Independent Consultant
Jacksonville, FL

ABSTRACT

It is well known that any program can be written without simple branching (i.e. GO TO) by using only three basic constructs: Sequence, Selection, and Repetition. In a high-level computer language, repetition is the most important of the three, for it constitutes the foundation, on which the program automation rests. Indeed, even such a basic operation as processing a file would be practically unfeasible if we had to code a separate reading instruction for each record. Repetition constructs make it possible to code a group of instructions just once, yet execute it many times, and modify the execution from iteration to iteration in a controlled manner.

In the SAS ® Language, explicit repetition is implemented via different forms of the construct known as the Do-loop. The Do statement is exceedingly powerful and laden with a myriad of features. Many of them, though, remain quite vague or are missed in the "standard" SAS learning curve, where the scope of the DO-loop is limited solely to array processing.

However, the real scope of the Do-loop in the language is much wider. In this talk, we will try to fill out some gaps, with the particular emphasis on using the Do-loop for:

- Simplifying control flow in single-step file processing.
- Organizing natural control flow for control-break processing (the "DOW-loop").
- Miscellaneous programming effects.

I. PROPAEDEUTICS

1. Anatomy

We need this brief section just to establish some anatomical Do-loop terminology that we will use liberally thereafter throughout the paper. Let us look at the Do-loop at large:

```
-------------------------------------------------------------------------
Do <Index> = <From, By, To specs>
     While | Until ( <expression> ) ;
-------------------------------------------------------------------------
 [TOP of the loop]:
 ----------------
  Evaluate Index. If Index > To then go to exit
  Evaluate While expression. If True go to exit
-------------------------------------------------------------------------
 [BODY of the loop]:
  < SAS instructions>
 If Leave  active go to Exit
 If Continue active go to Bottom
 <SAS instructions>
 -------------------------------------------------------------------------
 [BOTTOM of the loop]
  --------------------
  Evaluate Until expression. If True go to Exit
 Add By to Index
 Go to Top
-------------------------------------------------------------------------
End ;
-------------------------------------------------------------------------
 [EXIT of the loop]
-------------------------------------------------------------------------
```

There are three clearly discernable physical parts:

1. TOP. Located immediately after the Do statement, before the next explicit instruction. It contains no code, but that is where the WHILE and TO expressions are evaluated. If either expression if evaluated false, control is immediately transferred to the exit (see below), i.e. the loop terminates.
2. BODY. It is the very block of instructions the loop is intended to repeat. In some cases, the body may contain no instructions at all, but it does not mean that such loop is useless!
3. BOTTOM. Like at the top, there is not explicit code here. However, this is where (a) the UNTIL expression is evaluated and (b) CONTINUE statement, if the body has any, transfers control. In the case an index is specified, the last action taken at the bottom is adding to the index the amount to which the BY-expression has resolved.
4. EXIT. The exit is located after the END statement and before the very first instruction following END. If a LEAVE statement is present in the body, the exit point is where the program transfers control terminating the loop.

2. Control Flow

The actions in the Do-loop follow a well-defined sequence. Just as a reminder, let us recall it (the Do-sketch above gives a visual idea of the events):

1. Before the loop is launched, FROM, TO, and BY expressions are resolved and stay intact for the entire duration of the loop (meaning that trying to modify any of these expressions by changing their components in

the body of the loop is futile). The index is set to the value to which the FROM-expression has resolved.
2. Control is passed to the top. If a TO-expression is present, the value, to which it has resolved, is compared to the index. If the index is greater than the value, control is transferred to the exit, and the loop stops at once. In particular, that means that if the FROM-expression resolves to a value greater than the TO-expression, the loop will never iterate (the body be executed) once.
3. Control is still at the top. If the Do statement incorporates a WHILE expression, it is evaluated next. If it evaluates true, control is passed directly to the exit, and the loop terminates.
4. Control is passed to the body, and it is executed in the same manner as any block of SAS instructions. If the body contains an executable LEAVE statement, control is transferred to the exit, ending the loop. If there is an executable CONTINUE statement, control is passed directly to the bottom of the loop.
5. Control reaches the bottom of the loop. If an UNTIL expression is present, it is evaluated. If it evaluates true, control is immediately passed to the exit, and the loop is history. Otherwise the last implicit action at the bottom of the loop executes, adding the value, to which the BY-expression has resolved, to the current value of the index – which might have been modified by some instruction(s) in the body. Control is moved to the top of the loop, and the entire sequence is repeated from step 2.

In a nutshell, this is how the Do-loop operates. We will consider a number of practically useful, even if a bit peculiar, variants of the loop (such as the bodiless, infinite, dead-code forms of the loop) in the section purposely dedicated to things of this nature.

3. The "Golden Rule" of Repetition Coding

Suppose there is an action we want to perform before the loop begins and another action to be performed after the loop ends. Common sense tells us that such actions should never be coded inside the loop, but rather exactly where they are needed, i.e. before and after the loop. It does not really matter what kind of instructions the actions may incorporate – any instruction takes computer time, and placing it inside the loop potentially makes the computer do millions of times what can be done once. Generally speaking, for any repetition structure, such as the Do-loop, there is a general programming rule:

> Any instruction, whose action is not modified by the repetition process, must be placed outside the loop, unless repeating it is an intended behavior.

What is an unmodified instruction? It is an instruction whose components are neither intended nor expected to change as the loop iterates. For example, an instruction pre-computing an expression to yield a value, intended to stay unaltered across the iterations of a loop, is unmodified. However, an instruction containing array elements referenced by a loop index is modified. A file-reading statement, such as INPUT, is also a modified instruction because the record pointer moves as the loop cycles.

Despite common sense expressed by the "Golden Rule" – basically telling to avoid making the machine do many times what should be done once – it is broken all the time in the most trivial SAS programming.

II. DO-LOOP: FILE PROCESSING LOGIC

1. General

In a general programming language, in order to process a file a file-reading statement is enclosed in an explicit loop, which stops iterating after the end-of-file mark has been reached. For instance, in COBOL, it goes something like this:

```
PERFORM WITH TEST AFTER
  READ FILE AT END SET EOF TO TRUE
       NOT AT END PERFORM PROCESS-RECORD
END-PERFORM
```

Even for those not familiar with the language, it should be clear what is going on without delving into its gory details. Apparently, the same thing can be done in SAS – only more concisely and clearly – by using the Do-loop.

However, the first thing the standard learning curve teaches a SAS newcomer is that organizing the explicit loop is unnecessary because of the availability of the ubiquitous and almighty, implied and automatic "observation loop". (It is even touted as a sort of a SAS hallmark (although some other 3GL/4GL mixes, such as Easytrieve, feature a similar concept). The automatic loop is engraved in the SAS usage mentality to such an extent that it has attained an almost religious status. Endless papers and books have been written revealing and explaining its behind-the-scenes actions, whose intricacies a naked eye of a mere mortal fails to discern. As a result, almost every time when a file, be it an external file or SAS data set, has to be processed, an attempt is subconsciously made to use the implied loop, whatever it takes.

Such a rigid approach is practically tantamount to forcing a program into the fixed cage of an existing programming construct. But programming is not meant to be this way. It makes sense to choose the tool best fitting the task, rather than tweaking the task to fit the tool.

Of course, often times the implied loop suits the program structure just right, but many SAS programmers raised in the implied-loop faith will be surprised that it occurs not quite as often as they tend to believe. Such a match happens primarily when all records of a single file (or a group of concatenated files) are to be processed in the same manner, without the need for any programming actions to be performed before the first file, in between the files, or after the last file has been processed.

2. Implied Loop Anatomy in Do-loop Terms

Let us try simulating the behavior of the implied Data step loop by using an explicit Do repetition. This way, we may better delve into the roots of their mutual advantages and shortcomings. (Some details have been omitted from this scheme on purpose.)

```
< At compile, populate all valued retains >

Do Internal_Counter = 1 By +1 ;
  < Initialize non-retains to missing >
  _N_ = Internal_Counter ;
  _Error_ = 0 ;
  < ... SAS statements ... >
  < SET, MERGE, INPUT, UPDATE > ;
    IF < buffer-empty > Then Do ;
      If _Error_ NOT =  0 Then Put _All_ ;
      LEAVE ;
    End ;
  < ... SAS statements ... >
    If < DELETE-statement-active >  Then CONTINUE ;
    If < RETURN-statement-active > Then Do ;
         OUTPUT ;
         CONTINUE ;
    End ;
    If < STOP-active > Then LEAVE ;
    If < no-OUTPUT-statement-elsewhere > Then OUTPUT ;
    If _Error_ NOT =  0 Then Put _All_ ;
End ;
```

Because there is no TO-value in the Do-loop above, it launches an infinite cycle. The internal counter counts the number of times control is found at the top of the loop – in other words, the number of iterations. Then this incremented value is moved to _N_. (Incidentally, that means, that SAS maintains program-independent control over _N_, i.e. no matter what we do with _N_ between the top and bottom, at the top of the loop _N_ will contain the correct number of iterations. In particular, it is convenient to use _N_ as an automatically dropped index with array processing.) The value of _Error_ is reset to zero at the top of each iteration. If _Error_ is not zero at the bottom of the loop, _All_ Data step variables are dumped to the log. The same happens if the loop stops before reaching its bottom even once.

3. Implied Loop vs. Explicit Do-loop: A Single File

Schemes are good and useful things; however, the advantages of using the Do-loop explicitly to organize file processing can be plainly perceived from a single-file example – actually derived from real-life.

A friend of mine, an (extremely good) COBOL programmer, needed to pull some credit card accounts for his use, but the data were in the form of a SAS data file, let us call it ACCOUNTS, containing the needed accounts as a 16-digit text variable ACCTNO. He needed to select some accounts and write them to an external text file tied to a file reference OUT in the following manner:

1. Write a header to an external file OUT with current date formatted as YYYY-MM-DD (at position 1).
2. Read a credit card account from a SAS data set ACCOUNTS and select only observations containing VISA numbers (they begin with 4). Write each selected account to OUT at position 1.
3. After ACCOUNTS has been processed, write a trailer, with the date formatted as YYYY-MM-DD (positions 1-10) and total number of records in the file, excluding the header and trailer, with leading zeroes (positions 11-20).

Quickly (a speedy response was what was needed the most at the moment), I offered something analogous to the following lines (NOT recommended!):

```
Data _Null_ ;
  Retain Date ;
  If _N_ = 1 Then Do ;
    Date = Date () ;
    Put @1 Date YYMMDD10. ;
  End ;
  If EoF Then Put @1 Date YYMMDD10. @11 N z10.  ;
  Set ACCOUNTS End = EoF ;
  If ACCTNO NE: "4" Then Delete ;
  N ++ 1 ;
  Put @1 ACCTNO $16. ;
Run ;
```

That works fine. However, my friend is a very inquisitive and logical fellow, and as has already been said, is a great programmer. Besides, he wanted to put the little piece into production and needed to understand the logic behind it, so he started asking questions, chiefly:

1. We need to compute the date only once before reading the file. If so, why is the calculation conditional? By the same token, why is writing the header conditional? And what is that _N_? What is RETAIN and why do we need it?
2. Writing the trailer is the last thing we need to do, after the very last record has been read and evaluated. Then why is the corresponding statement located before the SET?
3. What is DELETE and where does it transfer control? Is it like GO TO?

I started off going into the gory details of the workings and defaults of the implied loop. But soon he interrupted me saying: "I do not get it. What could be simpler?

1. First, compute the date and write the header.
2. Next, loop over the file and write out the records satisfying the VISA criterion.
3. Next, write the trailer.
4. Finally, stop.

What, it cannot be done this way in SAS?"

But of course we can, just like in any other 3GL! My friend was absolutely right. I realized that under such

circumstances, trying to fit the logic of the process into the existing frame of the implied loop is nonsensical. What is wrong with it?

1. No instructions can be coded to be performed outside the implied loop, even if the logic itself, like in this case, dictates such course of actions. The only time when it can be done, and only before the loop, is the situation when something can be set or computed at the compilation time. For instance, here, we could move the date calculation to the compilation phase by coding

   RETAIN Date %SysFunc(DATE()) ;

2. Instead of putting it under _N_=1 condition. However, that does not win much, since the condition is necessary to write the header, anyway.
3. By coding the instructions to be performed once before and once after the loop inside the loop, we directly violate the Golden Rule of programming
4. It almost never gets due attention, but there is a performance price to be paid for such violation. Since the _N_=1 and EoF conditions are placed inside the loop, the software has to evaluate both of them each time the loop iterates. With a large file (and nowadays, they have the propensity to get only larger), it can be noticeable, especially if the job has to be slotted in a tight schedule.
5. It is simply illogical to program that way!

Using an explicit Do-loop instead allows aligning the task logic with the Data step logic:

```
Data _Null_ ;
  Date = Date () ;
  Put @1 Date YYMMDD10. ;
  Do Until ( EoF ) ;
    Set ACCOUNTS End = EoF ;
    If ACCTNO NE: '4' Then Continue ;
    N ++ 1 ;
    Put @1 ACCTNO $16. ;
  End ;
  Put @1 Date YYMMDD10. @11 N Z10.  ;
  Stop ;
Run ;
```

When explicit DO-loops are used to process files, it is generally a good idea to code the STOP statement regardless of whether the step will or will not end by itself because of no more
input. In this particular case, having STOP is imperative: It prevents the implied loop control from reaching the bottom of the step, so it does not iterate even once. Without STOP, implied loop control would reach the top of the step; another, unneeded, header would be written after the trailer and the implied loop would quit at the attempt of SET to read from the empty buffer. Because the implied loop never iterates, it effectively makes the Data and Run statements serve merely as a shell housing programming instructions. In essence, the implied loop is turned off.

Replacing the implied loop with an explicit Do-loop thus affords a number of advantages:

1. It allows for the implementation of straightforward, stream-of-the-consciousness logic.
2. It makes it unnecessary to retain non-descriptor variables. Since control never returns to the top of the implied loop, the variables are never reset to missing and hold their values until the program (step) ends, just as you would expect in any language.
3. It satisfies the Golden Rule of sound programming of repetition by making it possible to place the necessary instructions outside the loop.
4. Because of 3, it eschews the extraneous _N_ and EoF conditionals, thus relieving the computer from asking idle questions, possibly millions of times. This means that we get a more resource-efficient program without exacting any extra price for it.
5. It provides for a more stable program. With the implied loop active, the reason the end-of-file condition has to be placed before the file-reading verb loop is known as an infamous SAS Data step trap: If a sub-setting IF, DELETE, or RETURN happen to become active on the last record in the file, control will circumvent any statement placed after them and the bottom of the loop. If that happened in our example, the output file would miss the trailer. Explicit Do-looping makes such turn of the events principally impossible because the trailer is written unconditionally, after the loop has completely processed the input file and terminated.
6. If renders the program less SAS-centric, making it clearer and easier to understand for programmers familiar with tongues other than SAS.

4. What About That _N_ ?

Of course, if an explicit Do-loop is used instead of the implied one, the value of _N_, if not modified, stays the same during the entire Data step execution, namely:

_N_ = 1 ;

This is because _N_ can climb higher only if control is transferred back to the top of the implied loop, but with the explicit Do-loop and STOP in place, control gets grounded before even reaching the bottom of the implied loop. However, some folks who would in principle consider using the explicit loop have the reservation of having gotten too accustomed to using the automatically incremented _N_ for debugging purposes. Indeed, if an error occurs in a plain vanilla step like

```
Data A ;
  Set  B  ;
  < ... SAS stuff ... >
Run ;
```

SAS dumps _ALL_ variables into the log, with _N_ indicating the record number in B, at which the error occurred. Well, the desire to be able to identify the record from the log is quite legitimate. Additionally, the automatic variable

ERROR is reset to zero only at the top of the implied loop. However, if these considerations are paramount, nothing prevents the explicit loop from incrementing _N_ on its own, and/or setting _ERROR_ to 0 explicitly:

```
Data ... ;
  Do _N_ = 1 By 1 Until ( EoF ) ;
    _Error_ = 0 ;
    < ... SAS code ... >
    Set A End = EoF ;
    < ... SAS code ... >
    If _Error_ Then Put _All_ ;
  End ;
  Stop ;
Run ;
```

5. Implied Loop vs. Explicit DO-Loop: Multiple Files

Above, we have seen that even in the simplest case of a stand-alone file with pre- and after-processing, the explicit Do-loop results in a program better in all respects and not in the least longer than the traditional way of using the implied Data step looping. But if with a single file, forcing the programming into the rigid frame of the implied loop is, after all, relatively simple and painless, the situation changes dramatically if we have more than once file to process sequentially in the same Data step.

Suppose we need to do something similar to the following:

1. Print some initial message, say, LET US BEGIN.
2. Across all observations from file X, compute some variables X1-X3.
3. Display X1-X3 in the log.
4. Use X1-X3 in some observations from file Y to do some calculations, produce some new variables Y1-Y5, and write some records to a file YY.
5. Divide the last values of Y1-Y5 by the number of records YOBS read from Y.
6. Print YOBS, and Y1-Y5 to the log.
7. Take the last computed values Y1-Y5 and use them as weights for variables Z1-Z5 coming from file Z. Multiply Z1-Z5 by their weights and output each observation to file ZZ. Compute totals for Z1-Z5 across all Z records yielding ZS1-ZS5.
8. Print the total number of records in Z, ZOBS, and ZS1-ZS5 totals.
9. Print END PROGRAM.
10. Terminate.

Using explicit, file-reading Do-loops to achieve the goal is straightforward:

```
Data YY ( Keep=Y: ) ZZ ( Keep=ZS: ) ;
  Put "LET US BEGIN" ;

  Do Until ( XEoF) ;
    Set X End = XEoF ;
    < compute X1-X3 over X-records >
  End ;

  Put ( X1-X3 ) (=) ;

  Do YOBS = 1 By 1 Until ( YEoF) ;
    Set Y End = YEoF ;
    < compute Y1-Y3 over Y-records >
    If < condition > Then Output YY ;
  End ;

  Put YOBS= ( Y1-Y5 ) (=) ;

  Do ZOBS = 1 By 1 Until ( ZEoF) ;
    Set Z End = ZEoF ;
    < multiply all Y * Z >
    < add Z1-Z5 to ZS1-ZS5 >
  End ;

  Put ZOBS= ( Z1-Z5 ) (=) / "END PROGRAM" ;
  Stop ;
Run ;
```

We never stopped and never quit the Data step. Control never reached the bottom of the implied loop (thus it is off), so no RETAINs were needed in the calculations. The number of records was counted on the fly. In this code, nothing principally would change if we had external text files instead of the SAS data files, only SETs would be replaced by INPUTs with the corresponding FILE statements.

Now how would we proceed to replicate it using the default implied loop? We would have to concatenate the files as in (NOT recommended!)

```
Data YY ( Keep=Y: ) ZZ ( Keep=ZS: ) ;
  If _N_ Then Put "LET US BEGIN" ;
  If EoF Then Put ZOBS= ( Z1-Z5 ) (=) /
    "END PROGRAM" ;
  Set X (In=InX)  Y (In=InY) Z (In=InZ) End = Eof ;
  YOBS ++ InY ;
  ZOBS ++ InZ ;
  If InY and YOBS = 1 Then Do ... ;
  If InZ and ZOBS = 1 Then Do ... ;
  < ... SAS statements ... >
Run ;
```

Using _N_ and EoF to write the pre- and final post-processing messages is simple enough, but what about the stuff required to be done in between the files? That is the rub. We would know what file we are at from the IN-variables, but there would be no clearly-defined location in the code where one file would have already ended while the next not yet begun. Indeed, since the files are concatenated, the EoF option above marks the end of file Z only, so to recognize the beginning of each file, we would have to use conditions like

```
If InY and YOBS = 1 Then Do ... ;
If InZ and ZOBS = 1 Then Do ... ;
```

In other words, by trying to fit the program in the Procrustean bed of the implied loop, we end up with a mess, where the main programming attention is principally

directed at the details of implementation, rather than the underlying task logic. If this is not bad enough, performance issues mentioned above in the single-file section are exacerbated here by swarms of IF statements executed for every record of each file just for the sake of printing stuff when needed.

6. Something Is Still ON!

That the explicit Do-loop, for all intents and purposes, turns the implied loop off, does not mean that all automatic actions in the Data step stop their action.
Clearly, with the implied loop turned off, all non-descriptor-identified variables are effectively retained (even though they do not reside in the retained-for area of memory). To put it more accurately, since the top of the implied loop is never reached again after the loop has been launched, the default cleanup action of the Data step never materializes except before the step has begun, and the variables simply hold on to their values, until and unless they are explicitly overwritten.

However, in certain situations, descriptor-identified variables, although retained by default, are automatically set to missing regardless of whether the top of the implied loop has been reached again or not. Irrespective of the iterative mode (implied or explicit), descriptor-identified input variables are set to missing values in two notorious cases:

1. When there is a BY statement, and a new BY-group is loaded in memory.
2. At the execution of a file-reading instruction (SET, MERGE, UPDATE), when the Data step switches from one buffer to another for this instruction. In particular, this occurs when files are concatenated.

That is, in a step like

```
Data ... ;
  Do Until (EoF) ;
    Set A B C End = EoF ;
    ...
  End ;
  ....
  Stop ;
Run ;
```

the variables coming from A will be set to missing at the very first record read from B, then those coming from B will become missing at the first record read from C.

III. DOW-LOOP: NATURAL CONTROL-BREAK PROCESSING

1. The Big Lesson

What is the DOW-loop? It is not a "standard industry term". Rather, it is an abbreviation that has become quite common on SAS-L in the last year or so. Here is the short story.

Once upon a time on SAS-L, the renowned Master of the SAS Universe Ian Whitlock, responding to an inquiry, casually used a DO-loop in a certain peculiar manner, which I (apparently being not exceedingly bright at the moment) failed to fully comprehend at once. However, the more I thought about it, the more Irealized the potential and programmatic elegance hidden in such a structure, and eventually started propagating it on SAS-L and using in my own daily work.

Extensive usage helped me understand the DOW-loop even better and gave birth to certain tricks I have added myself thereafter. First, I called the structure plainly, the Whitlock DO-loop, but it seemed to be in need of an easily digested abbreviation. But enough for the with the semantics and history.

2. DOW-Loop: The Scheme

Let us consider the following Data step structure:

```
Data ... ;
  <Stuff done before break-event> ;
  Do <Index Specs> Until ( Break-Event ) ;
    Set A ;
    <Stuff done for each record> ;
  End ;
  <Stuff done after break-event... > ;
Run ;
```

The code between the angle brackets is, generally speaking, optional. We call the structure the DOW-loop.

The intent of organizing such a structure is to achieve a logical isolation of instructions executed between two successive break-events from actions performed before and after a break-event, and to do it in the most natural (programmatically-wise) manner. In most (although not all) situations where the DOW-loop is applicable, the input data set is grouped and/or sorted, and the break-event occurs when the last observation in the by-group has been processed. In such a case, the DOW-loop logically separates actions that are performed:

1. Between the top of the implied loop and before the first record in a by-group is read.
2. For each record in the by-group.
3. After the last record in the by-group has been processed and before the bottom of the implied loop.

3. DOW-Loop: A Simple Example

Let us assume that an input SAS data file A is sorted by ID. The step below:

1. Multiplies and summarizes all VAR values within each ID-group.
2. Counts the number of all and non-missing records in each group.
3. Finds the group average.

4. Writes a single record (with COUNT, SUM, MEAN, PROD) per group to file B.

```
Data B ( Keep = Id Prod Sum Count Mean) ;
  Prod = 1 ;
  Do Count = 1 By 1 Until ( Last.Id ) ;
    Set A ;
    By  Id ;
    If Missing (Var) Then Continue ;
    Mcount = Sum (Mcount, 1) ;
    Prod = Prod * Var ;
    Sum = Sum (Sum, Var) ;
  End ;
  Mean = Sum / Mcount ;
Run ;
```

Now let us see how it all works in concert:

1. Between the top of the implied loop and the beginning of an ID-group: PROD and COUNT are set to 1, and the non-retained SUM, MEAN, and MCOUNT are set to missing by default (control at the top of the implied loop).
2. Between the first iteration and break-point: DOW-loop starts to iterate, reading the next record from A at the top of each iteration. While it iterates, control never leaves the Do-End boundaries. If VAR is missing, CONTINUE passes control straight to the bottom of the loop. Otherwise MCOUNT, PROD and SUM are computed. After the last record in the group is processed, the loop terminates.
3. Between the break-point and the bottom of the step: Control is passed to the statement following the DOW-loop. At this point, PROD, COUNT, SUM, and MEAN contain the group-aggregate values. MEAN is computed, and control is passed to the bottom of the implied loop, where the implied OUTPUT statement writes the record to file B. Control is passed to the top of the implied loop, where non-retained, non-descriptor variables are re-initialized, and the structure proceeds to process the next ID-group.

Note that contrary to the common practice of doing this kind of processing, the accumulation variables need NOT be retained. Because the DOW-loop passes control to the top of the Data step ONLY before the first record in a by-group is to be read, this is the only point where non-retained variables are reset to missing values. But this is exactly where we need it!

4. So What's the Big Deal?

What does make the DOW-loop so special? It is all in the logic. The DOW-loop programmatically separates the before-, during-, and after-group actions in the same manner and sequence as does the stream-of-the-consciousness logic:

1. If an action is to be done before the group is processed, simply code it before the DOW-loop. Note that is unnecessary to predicate this action by the <IF FIRST.ID> condition.
2. If it is to be done with each record, code it inside the loop.
3. If is has to be done after the group, like computing an average and outputting summary values, code it after the DOW-loop.

5. Implied and Explicit Again

Discussing the advantages and shortcomings of the implied loop versus explicit Do-loop for file processing, we looked at them, in a way, as antipodes. The DOW-loop is a construct that marries the two, and makes them work together perfectly in concert.

As seen from the example above, an iterative Do-loop is nested within the implied Data step loop and fits there like in a glove. As a result, there is no need to retain summary variables across observations. While the DOW-loop iterates, they hold their values, but when a by-group is over and control is passed back to the top of the implied loop, they are reset to missing. The latter occurs before the new by-group begins, i.e. just where it is needed.

6. DOW-Loop: Control-Breaks

The scope of the DOW-loop, as can be seen from its general scheme above, is not limited to by-processing. Rather, it is a natural tool for control-break processing. By control-break processing, programmers usually mean situations where a group of data elements (records, observations, array items, etc.) has a boundary expressed by some condition.

In the example above, the condition was in the form of <LAST.ID>, but it does not have to. For instance, imagine that a data set has a variable CB occasionally taking on a missing value, for example,

```
Data CB ;
  Do CB = 9, 2, .A, 5, 7, .Z, 2 ;
    Output ;
  End ;
Run ;
```

We need to summarize it until CB hits a missing value, and print the sum rounded mean across consecutive non-missing records each time it happens. Here is an excerpt from the log:

```
3  Data CBSum ;
4    Do _N_ = 1 By 1 Until (NMiss(CB) | Z) ;
5      Set CB End = Z ;
6      Sum = Sum (Sum, CB) ;
7    End ;
8    Mean = Round (Sum / (_N_-1+Z), .01) ;
9    Put (Sum Mean) (= 4.2) ;
0  Run ;

Sum=11.0 Mean=5.50
```

```
Sum=12.0 Mean=6.00
Sum=2.00 Mean=2.00
```

Above, the control-break event has nothing to do with by-processing, but just a certain pre-imposed criterion.

IV. DO-LOOP: PROGRAMMING EFFECTS

Effects associated with Do-loop programming are practically as diverse as programming itself. In this talk, we will be able to discuss only an infinitesimal fraction of them, more or less randomly selected according to my likings.

First, we will stop on effects related to the absence of certain Do-loop elements. Then, we will just dive in a bunch of curious and/or useful assorted Do-loop mix.

1. No Body (NOT Nobody)

What if a Do-loop has no body at all, that is, the space between the DO and END statements is empty? Is it then completely useless? Not quite!

Suppose we have ten variables AA1-AA10 coming from an observation in a data set. Fro each observation processed, we have to devise some fancy repetitive process, whose every new repetition begins from the next missing value of AA:

```
Do  < ... > ;
  <Code finding index of next missing AA>
  <Start with the next missing AA>
  <Do something fancy>
End ;
```

What is the most efficient way of writing the code to find the next missing AA? How about this:

```
Array AAS (*) AA1-AA10 AStop ;
Do < ... > ;
  Do X = X + 1 By 1 Until ( NMiss(AAS(X)) ) ;
  End ;
  If X = HBound (AAS) Then Leave ;
  ...
End ;
```

This inner loop will always stop exactly at the next missing value. The next time around, it will start looking beginning from the next array element because of X = X +1. And because of the sentinel placed at the right of the array, the bodiless loop makes exactly one and only one comparison (at the bottom of the loop) per iteration. If ASTOP is hit, the outer process if finished, and LEAVE terminates it.

This simple bodiless loop is actually used in a whole variety of high-performance applications, from quick sequential search to Quicksort. For example, the inner loop of the latter algorithm looks like:

```
Do J = H + 1 By 0 ;
  Do I = I + 1 By +1 Until (A(I) => P) ;
  End ;
  Do J = J - 1 By -1 Until (A(J) <= P) ;
  End ;
  If I => J Then Leave ;
  < Swap A(I) and A(J) > ;
End;
```

2. FROM, BY, and TO Index Expressions

If a loop index is used in the Do-loop, not all of the FROM, BY, and TO index expressions must be specified. Out of these three, the only expression that cannot be left out is the FROM expression, because it is the one where the index variable is actually named. However, what happens if BY and/or TO expressions are omitted? Suppose the index is called X. Several combinations are possible:

1. BY is absent, TO is absent. The loop iterates once with the FROM value fixed.
2. BY is present, TO is present. X is incremented by the value of BY expression, and the loop stops when on the top of it, X > TO or other loop-termination condition is satisfied before.
3. BY is absent, TO is present. This is a common situation when BY is defaulted to 1.
4. BY is present, TO is absent. It will launch an infinite loop unless another loop-stopper is present.

Out of all these variants, the last one is the most misunderstood and underused. We have already seen above how this combination can be utilized with the DOW-loop:

```
Do Count = 1 By 1 Until ( Condition ) ;
  ...
End ;
```

The loop will always stop because of the condition, and having Count in this form makes it unnecessary to write superfluous code like

```
Count = 0 ;
Do Until ( Condition ) ;
  Count ++ 1 ;
  ...
End ;
```

Additionally, in cases when it is desirable to organize an infinite loop with exit from the middle (like in the Quicksort code above) and have a loop counter at the same time, the simplest thing to do is

```
Do X = <expr> By <expr> ;
  ...
  ...  Active LEAVE/GOTO ...
  ...
End ;
```

Case 4 also lends it self to the peculiar possibility of using BY = 0 just in order to initialize a numeric variable. The first line of the Quicksort code above,

```
Do J = H + 1 By 0 ;
  ...
End ;
```

does exactly that. We could not had just the FROM expression for this purpose, for then the loop would iterate just once. By providing the dummy zero increment, we make the loop iterate infinitely until LEAVE becomes active.

3. Ad Infinitum

There exist circumstances when it is logical to organize an infinite loop and break out of it using an explicit condition within the body of the loop. In the previous section, we saw how to do it using a TO-less index. If the index is not needed, then both

```
Do Until ( 0 ) ;
  ...
End ;
```

and

```
Do While ( 1 ) ;
  ...
End ;
```

will iterate forever unless an instruction in the body will branch out.

3. LEAVE and CONTINUE

Why are these statements needed? The main reason is that TO, WHILE, and UNTIL provide exits from the top and bottom of the loop, but what if one needs to get out from the body of the loop? Of course it would be the simplest thing to GO TO an appropriately placed label. But the poor GO TO has gotten such a bad rap for nothing that the good folks in SAS decided not to play devil's advocate and disguised it as LEAVE and CONTINUE. But herein lies is a trap.

4. The LEAVE Trap

LEAVE is designed to exit the Do-loop. But it is not the only thing it does! It leads to a trap. Below, is an example of correct code for binary search algorithm looking up a search key K in an ordered array A and returning its index X (or missing value if not found):

```
X  = .              ;
Lo = LBound (A) ;
Hi = HBound (A) ;
Do Until ( Lo > Hi ) ;
  M = Floor ( (Lo + Hi) * 0.5 ) ;
  If       K < A(M) Then Hi  = M – 1 ;
  Else If K > A(M)  Then Lo = M + 1 ;
  Else Do ;
    X_Found = M ;
    Leave ;
  End ;
End ;
```

Now, if one would try to "beautify" this by replacing the IF-THEN-ELSE with SELECT, i.e. coding instead (WRONG!):

```
Do Until ( Lo > Hi ) ;
  M = Floor ( (Lo + Hi) * 0.5 ) ;
  Select ;
    When (K < A(M)) Hi  = M – 1 ;
    When (K > A(M)) Lo = M + 1 ;
    Otherwise Do ;
      X_Found = M ;
      Leave ;
    End ;
  End ;
End ;
```

then the result of this, seemingly equivalent, code would be most miserable: In the case a match, it would loop forever!

Why? Because in this context, LEAVE (for reasons the designer has not shared) passes control past the END closing the SELECT statement. So, the only goal LEAVE achieves here is leaving Do-loop control loose.

5. Nested Loops and Dead Code

Unlike some other tongues, the SAS Data step does not limit the number of nested DO-loops and leaves the matter to the sanity of the programmer. In most programming situations, a good reason for really high nesting levels is difficult to imagine. There is one general situation, however, where nesting a potentially large number of loops can do a nice trick which allows either avoiding a macro, or significantly reducing its complexity.

Suppose we have arrayed variables A1-A7 and would like to form 4 series of one-way combinations of these variables: 1-element, 2-element, 3-element, and 4-element, and place them in an array C1-C7. Forming each series requires code with corresponding number of nested loops. For example, to generate 1-, 2-, and 3-element series, we would need the following piece of code, where N denotes the total number of elements to choose from:

```
x0 = 0;
do x1=x0+1 to n;
  c(1) = a(x1);
end;
do x1=x0+1 to n-1;
do x2=x1+1 to n-0;
  c(1) = a(x1);
  c(2) = a(x2);
end;
end;
do x1=x0+1 to n-2;
do x2=x1+1 to n-1;
do x3=x2+1 to n-0;
```

```
  c(1) = a(x1);
  c(2) = a(x2);
  c(3) = a(x3);
end;
end;
end;
```

The pattern here is quite clear, so the necessary code can be generated by a macro.

However, certain properties of a Do-loop allow doing it without the aid of the Macro language, by using what I term the "dead code approach". The idea is simple: To code a large number of nested loops (15, say) beforehand, and then compose Data step code making the necessary number of the innermost loops iterate just once. In the example above, the prewritten number of dead loops is limited to 9 just for the sake of brevity:

```
Data Allcombs ( Keep = C: ) ;
  Retain N 7 K 4 ;

  Array A (7) (11 13 17 19 23 29 31) ;
  Array C (7) ;
  Array X (9) ;
  Array F (9) _Temporary_ ;
  Array S (9) _Temporary_ ;

  Do Level = 1 To K ;
    Do J = 1 To Dim (F) ;
      F (J) = Level < J ;
      S (J) = (N - Level + J) * (Level => J) ;
    End ;
    Do X1 =      1 To       1  * F(1) + S(1) ;
    Do X2 = X1+1 To (X1+1) * F(2) + S(2) ;
    Do X3 = X2+1 To (X2+1) * F(3) + S(3) ;
    Do X4 = X3+1 To (X3+1) * F(4) + S(4) ;
    Do X5 = X4+1 To (X4+1) * F(5) + S(5) ;
    Do X6 = X4+1 To (X5+1) * F(6) + S(6) ;
    Do X7 = X6+1 To (X6+1) * F(7) + S(7) ;
    Do X8 = X7+1 To (X7+1) * F(8) + S(8) ;
    Do X9 = X8+1 To (X8+1) * F(9) + S(9) ;
      Do J = 1 To Level ;
        C (J) = A ( X(J) ) ;
      End ;
      Output ;
    End; End; End; End; End;
    End; End; End; End;
  End;
Run ;
```

In the program, above, LEVEL loops from 1 to 4 as required. At each level iteration, the coefficients affecting the TO-expressions are adjusted in such a way that all innermost loops deeper than LEVEL go through a single iteration, i.e. straight to the J-loop populating array C(*). In other words, at LEVEL=1, only the outermost loop iterates full cycle, while others, being bound from (x1+1) to (x1+1), (x2+1) to (x2+1), ... , (x9+1) to (x9+1), iterate just once merely passing control to the J-loop. At LEVEL=2, the loop-disabling pattern continues, but starting from (x2+1), and so on.

This code can be also be enveloped in a macro parameterized by K and N, but it is much easier to do from the macro perspective. Also note that the dead-code step will not execute as efficiently as a step containing only "live" instructions, but at least it offers a practical possibility if the entire program must be controlled from the Data step. For example, if K and N are supplied from each incoming observation along with the A-variables, dead code is about the only opportunity to accomplish the task in a non-convoluted manner.

V. CONCLUSION

The Data step Do-loop is a magnificent structure. It is the most powerful tool you can use to make the computer do the most of the work for you. And in some of its incarnations, such as the DOW-loop, it is beautiful to behold.

VI. REFERENCES

1. D. E.Knuth, *The Art of Computer Programming*, **2**.
2. D. E.Knuth, *The Art of Computer Programming*, **3**.
3. R. Sedgewick, Algorithms in C, Parts 1-4.
4. P.M.Dorfman. Table Look-Up by Direct Addressing: Key-Indexing– Bitmapping–Hashing. SUGI 26, 2001.
5. P.M.Dorfman. Table Look-Up Techniques: From Sequential Search to Key-Indexing. SESUG, 1999.
6. P.M.Dorfman. QuickSorting an Array. SUGI 26, 2001.
7. P.M.Dorfman. Manipulating Data: Elements of the Data Step language. SSU, 2001.

VII. TRADEMARKS

VIII. ACKNOLEDGEMENTS

Thanks to Ian Whitlock for the idea of the DOW-loop and not protesting too much against the acronym. Thanks to Dan Bruns and Gary Schlegelmilch for the invitation to do the DO. Thanks to all long-time SAS-L participants for tolerating my experiments (at times, extreme) in Do-loop programming.

IX. AUTHOR CONTACT INFORMATION

Paul M. Dorfman
4437 Summer Walk Court
Jacksonville, FL 32258
(904) 260-6509 (h)
(904) 905-5428 (o)
sashole@bellsouth.net
paul_dorfman@hotmail.com
paul.dorfman@bcbsfl.com