PhUSE 2010

Paper CC05

APPEND, EXECUTE and MACRO

Jim Groeneveld, OCS Biometric Support, ‘s Hertogenbosch, Netherlands

SUMMARY

This paper starts with appending SAS® datasets using a data step or PROC APPEND and consequently ends with a **CALL EXECUTE proof** macro **%_Append**_. Drawbacks of PROC APPEND are discussed, the macro %_Append_ is proposed as the solution for those drawbacks and problems and solutions with running macros in general from CALL EXECUTE are presented.

PROC APPEND

When concatenating or appending datasets one may use a **data step** in which all to be appended datasets are specified on the **SET** statement at once. That looks relatively easy, but the drawback is that the lengths of character variables are taken from the first dataset or from the first dataset where a specific character variable occurs, only. On the other hand it preserves labels and (in)formats from new variables in the other dataset.

PROC APPEND with the **FORCE** option, apart from just appending one dataset to another one at a time, is even worse. It only takes the **PDV (Program Data Vector)** of the first (BASE=) dataset and any variables not present in that first dataset, though present in subsequently appended (DATA=) datasets, are discarded. This also happens with a data step and the OPEN=DEFER option on the SET statement.

PROC SQL concatenates datasets with the **OUTER UNION CORR** operator and keyword preserves all variables (and first defined labels) and the maximum character lengths in the datasets, but only keeps the (in)formats from the first dataset, not from the other one.

To overcome these drawbacks a macro %_Append_ has been written. It initially determines the maximum lengths (and deduced (in)formats) of all character variables involved and creates an empty overall PDV, a dataset with all variables defined, but without observations. Then it uses a data step to SET all datasets, starting with the empty PDV.

CALL EXECUTE

Dataset names to be appended may be contained as values of a character variable in another, reference dataset. Each record then specifies a different dataset name. These datasets have to be appended for every record of the reference dataset. The only way to run another data step or a procedure from a data step is, delayed, via CALL EXECUTE.

A prototype of the macro **%_Append**_ contained various conditional macro code, meant to run in between the SAS code, partially dependent on the contents of the SAS code processing the reference dataset. This prototype ran well if called directly. But it did not run correctly when called from a CALL EXECUTE statement because executing macro code from CALL EXECUTE is not delayed by a RUN or QUIT statement as usual, so all macro code runs before the SAS code, including the SAS code on which it should depend.

For that reason the macro has been rewritten to contain only SAS code, including CALL EXECUTE statements (and CALL SYMPUT statements) and macro code that only processes the macro input parameters. The current macro %_Append_ appends one dataset to another one without any loss of information. Of course same named variables in the datasets to be appended should always be of the same type.

ALTERNATIVES

Some other alternatives for the CALL EXECUTE – macro calling problem, in which macro code is dependent of preceding SAS code, are also presented, all based on delaying the macro execution until after the data step from which the CALL EXECUTE statement is issued. The basic form is:

```
CALL EXECUTE('%NRSTR(%TargetMacro)');
```

INTRODUCTION

PROC APPEND DRAWBACKS

PROC APPEND is a very nice proc to concatenate (two) datasets. Its general syntax is:

```
PROC APPEND BASE=<libref.>SAS-data-set <DATA=<libref.>SAS-data-set> <FORCE> <APPENDVER=V6>;
```

The advantage is that it can even be used if the BASE dataset does not (yet) exist; it will be created from the DATA dataset.

Unfortunately PROC APPEND only concatenates datasets correctly if they have the **same structure**, i.e. the same variables, variables of the same type and character variables of the same length. It (obviously) generates **errors** if same named variables in the BASE= and DATA= datasets are of a **different type** (numeric and character). PROC APPEND generates **warnings and errors**, while refusing to append, if same named character variables have **different lengths**, i.e. if corresponding character variables in the DATA= dataset have larger lengths. Finally it also generates warnings and errors if the (DATA=) datasets to be appended have **additional variables** that are not in the BASE= dataset.

Some of the errors indicated above can be avoided with the **FORCE** option, while generating **warnings** only, but character variable lengths will be **limited** to their lengths in the BASE= dataset, which determines the **PDV (Program Data Vector)** of the resulting concatenated BASE= dataset. This means that extended content of character variables in the DATA= dataset is lost. Furthermore PROC APPEND does **not add new variables** of a DATA= dataset to its result in the concatenated BASE= dataset. These variables are **lost**. Appending variables with different types yield missing appended values. Yet with the FORCE option there will be an appended **result**, however incomplete.

The problems with appending datasets, with variables of the same type, via PROC APPEND can be illustrated using the example:

```
* Three example datasets to be used throughout this document;
DATA One;
  LENGTH One $6;
  One = 'abcdef';
  Format One $6.; INFORMAT One $CHAR6.;
  LABEL One='This is One';
RUN;

DATA Two;
  LENGTH One Two $8;
  One = '12345678';
  Two = '12345678';
RUN;

DATA Three;
  LENGTH One $4 Two $12;
  Two = 'ABCDEFGHIJKL';
  Three = 3;
  FORMAT One 10.;
  LABEL One='New label One';
  FORMAT Two $CHAR12.; INFORMAT Two $12.;
  LABEL Three='This is Three';
RUN;

* copy One to Appended to start with;
DATA Appended;
  SET One;
RUN;
PROC APPEND BASE=Appended DATA=Two FORCE;
RUN;
PROC APPEND BASE=Appended DATA=Three FORCE;
RUN;
TITLE "twice PROC APPEND";
PROC PRINT DATA=Appended;
RUN;
```

The PROC PRINT result is:

```
twice PROC APPEND
Obs     One
1     abcdef
2     123456
3
```

The PROC CONTENTS result (a limited variable list written to a dataset and then PRINTed):

```
Obs NAME  TYPE LENGTH VARNUM     LABEL     FORMAT FORMATL FORMATD INFORMAT INFORML INFORMD
 1  One     2     6     1     This is One     $      6       0    $CHAR       6       0
```

from which it is clear that only the PDV of the first dataset in the series is kept; it is not extended.

PROC APPEND is **not generally useful or reliable** unless the dataset structures involved are identical. Nevertheless PROC APPEND is often used to concatenate any (unknowingly differing) datasets, clearly with **incorrect, unintended results**.

DATA STEP ALTERNATIVE

A **data step** can also concatenate datasets by just using the **SET** statement. Two or even more datasets have to be specified with the SET statement, while the dataset specified on the DATA statement can be any dataset, also the first one from the SET statement, which then gets overwritten. The general syntax of such a data step is:

```
DATA Appended;
  SET {list of datasets}; * any number of datasets, also the one to be Appended;
RUN;
```

The drawback of such a data step is that the initial **PDV** is still taken from the first dataset. Additional variables from subsequent datasets can be added to that PDV, but once a character variable is in the PDV its (maximum) **length is fixed**. Later specified character variables with larger lengths are **truncated** to the already stored lengths in the PDV. (Of course a LENGTH statement before the SET statement can fix that, but we do not know of the dataset's variables and character lengths.)

Using the same three datasets these problems are clearly illustrated by the example:

```
DATA Appended;
  SET One Two Three /* uncomment: OPEN=DEFER */;
RUN;
TITLE "data step SET statement";
PROC PRINT DATA=Appended;
RUN;
```

The PROC PRINT result is:

```
data step SET statement
Obs    One       Two         Three
 1     abcdef                  .
 2     123456    12345678      .
 3               ABCDEFGH      3
```

The PROC CONTENTS result (a limited variable list written to a dataset and then PRINTed):

```
Obs NAME  TYPE LENGTH VARNUM     LABEL     FORMAT FORMATL FORMATD INFORMAT INFORML INFORMD
 1  One     2     6      1   This is One   $          6      0     $CHAR       6      0
 2  Three   1     8      3   This is Three              0      0                 0      0
 3  Two     2     8      2                 $CHAR     12      0     $          12      0
```

from which it is clear that the lengths of character variables are being determined by the first dataset in which they occur.

Using the option **OPEN=DEFER** with the SET statement causes the PDV to be taken from the **first dataset only**, just like with PROC APPEND. All datasets should have the **same structure** or an **error** will occur. This option causes to keep only variable One in the final Appended dataset. In any case trying to append same named variables with different types results in an error.

Concatenating datasets in the data step preserves **labels and (in)formats** though from the first dataset where they have been defined, which is good, but **not sufficient**.

PROC SQL ALTERNATIVE

PROC SQL can concatenate datasets using the **OUTER UNION CORR** operator and keyword. An example is:

```
PROC SQL;
  CREATE TABLE OneTwo AS
    SELECT * FROM One
    OUTER UNION CORR
    SELECT * From Two;
  CREATE TABLE OneTwo3 AS
    SELECT * FROM OneTwo
    OUTER UNION CORR
    SELECT * FROM Three;
QUIT;
TITLE "twice PROC SQL";
PROC PRINT DATA=OneTwo3;
RUN;
```

This preserves all variables (and their labels in the first dataset where defined) and the maximum character lengths in the datasets, but only keeps the (in)formats from the first dataset, **not from the other one**, even if there are no (in)formats defined in the first dataset. This is somewhat opposite to the data step approach in that PROC SQL is able to extend the length of character variables, but **loses the (in)formats** from the second dataset. Both methods have different drawbacks.

The PROC PRINT result is:

```
twice PROC SQL
Obs    One       Two             Three
 1     abcdef                      .
 2     123456    12345678          .
 3               ABCDEFGHIJKL      3
```

The PROC CONTENTS result (a limited variable list written to a dataset and then PRINTed):

```
Obs NAME  TYPE LENGTH VARNUM     LABEL     FORMAT FORMATL FORMATD INFORMAT INFORML INFORMD
 1  One     2     8      1   This is One     $       6       0     $CHAR       6       0
 2  Three   1     8      3   This is Three           0       0                 0       0
 3  Two     2    12      2                           0       0                 0       0
```

(Note that in all PROC CONTENTS listings, presented in this paper, the variables are sorted alphabetically by name.)

In this example the value of variable One has been truncated to 6 characters because of the format defined for it in the first dataset. Internally the full value (8 characters) yet is stored. But redefinition of its format in the other dataset does not have any effect. This is acceptable, but **(in)formats** in the other dataset, whether for **existing or new variables**, are entirely **disregarded**, which is not desired for new variables. Of course formats can always be (re)defined later by the user.

Furthermore PROC SQL generates an error if attempting to concatenate datasets with same named variables of a different type. This is acceptable.

SAS MACRO %_APPEND_

A solution to concatenate datasets without loss of data (and metadata as far as possible) has been developed as the SAS macro %_Append_. It **keeps all** occurring variables in all specified datasets while forcing a length for character variables equal to the **maximum occurring length**. (It correctly does not allow discrepant variable types, but some (future) solution to that problem could even be thought of by converting numeric values to their character equivalent using a certain format.) The macro defines the **(in)formats** of character variables from the **maximum lengths** and those of numeric variables from the **firstly named (in)format** encountered or if there are no named (in)formats from the **maximum width** specified in any dataset. **Variable labels** from their first occurrence in the datasets are kept as well.

An experimental prototype version of the macro consists of both macro code and SAS code, in which a part of the macro code (after RUN; or QUIT; statements) is **dependent** on the preceding SAS code. The macro analyses both datasets (as far as the BASE= dataset exists initially) using PROC CONTENTS and stores many variable attributes, like

- the name (Name)
- the type (Type)
- the (character) length (Length)
- the sequence number in the dataset (Varnum)
- the label (Label)
- the format name (Format)
- the format width (FormatL)
- the format decimals (FormatD)
- the informat name (Informat)
- the informat width (InformL)
- the informat decimals (InformD)

It uses these **attributes of all variables** of the involved datasets to build a **new PDV** with all variables, labels and maximised character lengths, formats and informats. New variables from the second (DATA=) dataset are added after those from the first (BASE=) one. This takes the bulk of the core processing code. Once the PDV, an **empty dataset** (without observations), has been built, the datasets to be concatenated are appended to it with the SET statement in a simple datastep (which could as well be done using PROC SQL or PROC APPEND).

The PROC PRINT result is:

```
twice macro %_Append_
Obs    ONE         TWO             THREE
 1     abcdef                        .
 2     12345678    12345678          .
 3                 ABCDEFGHIJKL      3
```

The PROC CONTENTS result (a limited variable list written to a dataset and then PRINTed):

```
Obs NAME  TYPE LENGTH VARNUM     LABEL     FORMAT FORMATL FORMATD INFORMAT INFORML INFORMD
 1  ONE     2     8      1   This is One     $        8      0     $CHAR       8      0
 2  THREE   1     8      3   This is Three            0      0                 0      0
 3  TWO     2    12      2                   $       12      0     $          12      0
```

As can be seen all **meta information** of the original datasets is **kept** as much as possible or adapted as optimal as possible. Note the redefinition of the (in)format of character variable One, based on its contents in dataset Two (in contrast to the result of the data step above). Furthermore, the variable names have been converted to upper case, which is necessary to sort and merge the meta information of both datasets by the variable name during processing.

The general macro call is:

```
%_Append_ (Base=Base_dataset, Data=Data_dataset) /* (ending semicolon generally redundant) */
```

CALL EXECUTE

As already indicated the initial macro %_Append_ uses SAS code and intermediate macro code partially dependent on preceding SAS code results. This generally works fine, but **not** if the macro is being called from a CALL EXECUTE statement.

However, it may be necessary to call the macro USING CALL EXECUTE from an **implicit** (data step) or **explicit loop**. For example, the names of datasets to consecutively append to the first one may be stored in a reference dataset. For every record of that dataset the macro has to be called and the multiple appends have to take place after the data step processing the reference dataset ends. To such a purpose example code is:

```
DATA Reference;
  INPUT Dataset $16.;
  CARDS;
dataset1
dataset2
dataset3
;
RUN;

DATA _NULL_;
  SET Reference;
  CALL EXECUTE ('%_Append_ (Base=Appended, Data=' || TRIM(Dataset) || '); ' );
RUN;
```

The resolution of the call to the macro itself should be delayed by enclosing it in **single** quotes, preventing immediately running the generated SAS code (other data steps) inside the current data step (yielding errors or unintended results anyway). Yet once the macro runs **all macro code** (macro statements building SAS code to run) is **executed immediately**, while the generated **SAS code** only runs **delayed**, after the data step processing the Reference dataset has ended. This is all right if the macro code is intended to generate the same SAS code in all instances, e.g. just checking the input parameters and generating fixed loops and so on. But if there is macro code that should run in between SAS code, e.g. after RUN; of data or PROC steps, and should generate subsequent SAS code **dependent on the results** of previous steps then this construct will **not work** as intended and yield either errors or unintended results.

Note that CALL SYMPUT statements, assigning values to macro variables, is SAS code, not macro code, and is processed from the concerning data steps. Furthermore macro variable references (&MacVar) are resolved where they occur, as intended. So, SAS code involving CALL SYMPUT statements and later references to those assigned macro variables are being processed correctly wherever they occur in the SAS code, even from CALL EXECUTE.

PROBLEM DEMONSTRATION

The problem with mixed SAS and macro code, especially macro code, dependent on SAS code results, can basically be demonstrated with the example code:

```
%MACRO TargetMacro;
  DATA _NULL_;
    PUT '==1==';
  RUN;
  %PUT ==2==;
  DATA _NULL_;
    PUT '==3==';
  RUN;
  %PUT ==4==;
%MEND TargetMacro;

%PUT Directly called;
%TargetMacro /* semicolon redundant */

%PUT Via CALL EXECUTE;
DATA _NULL_;
  CALL EXECUTE('%TargetMacro');
RUN;
```

Running this example shows that the result of the macro when called directly is as intended, i.e. that the values 1 through 4 are written to the log in that order. However, when calling the same macro from CALL EXECUTE it can be seen from the log that the values 1 through 4 are written in the order 2-4-1-3, clearly demonstrating that the macro code runs before the SAS code.

REWRITTEN MACRO %_APPEND_

For that reason the macro %_Append_ has been rewritten to contain only SAS code (including its own CALL EXECUTE and CALL SYMPUT statements) and macro code that only processes the macro input parameters. The current CALL EXECUTE proof macro %_Append_ appends one dataset to another one **without any loss of information**. Of course same named variables in the datasets to be appended should always be of the same type.

Additionally the macro has an extra feature, the optional specification of an Out= dataset, causing the appended result of the Base= and the Data= datasets to be written to the Out= dataset. The macro is too large to be listed in this paper. Its latest version can be obtained from http://jim.groeneveld.eu.tf/ software/SASmacro/_Append_.zip

ALTERNATIVE

The macro %_Append_ has been rewritten to avoid the use of macro code that is dependent on previous SAS code results. How that has been done while maintaining the same functionality is not so much of interest here. Programming and reprogramming very much depends on the purpose of a macro and the way a coding solution is implemented.

Yet it may be difficult and time consuming to rewrite all macros that need to be called from a CALL EXECUTE statement, especially if they are quite **complex** and/or have been written by **someone else**. Furthermore, such a rewritten macro is actually quite another one (though with an unchanged purpose) and should be extensively **tested and (re)validated**.

Some other alternatives for the CALL EXECUTE – macro calling problem, in which macro code is dependent of preceding SAS code, are all based on **delaying the whole macro execution** (including the macro code) until after the data step from which the CALL EXECUTE statement is issued. The basic form to delay macro execution until after the data step, from which it is called, involves quoting the macro call with the **%NRSTR** macro quoting function and is:

```
CALL EXECUTE('%NRSTR(%TargetMacro)');
```

Alternative 1

The %NRSTR macro function **prevents immediate resolution** (and thus execution) of the macro call. This can be demonstrated using the example macro with the 4 values, defined previously:

```
%PUT Via CALL EXECUTE and %NRSTR(%%)NRSTR;
DATA _NULL_;
  CALL EXECUTE('%NRSTR(%TargetMacro)');
RUN;
```

The order of the values written to the log is now 1-2-3-4, just as intended. Note that the whole character expression of CALL EXECUTE still must be enclosed in **single** quotes; if double quotes are used, e.g.

```
CALL EXECUTE("%NRSTR(%TargetMacro)"); %* This is a faulty example;
```

the effect is the same as with single quotes without the %NRSTR macro function: immediate execution of macro code and postponed execution of SAS code, not an improvement.

This %NRSTR construct can be used with **any macro call** from CALL EXECUTE, whether or not containing SAS code dependent macro code. It is the **safest way** to call a macro and doing it like this would have made it unnecessary to rewrite the macro %_Append_ at all. But it needs a little more code and the awareness of the programmer to do that.

MORE ALTERNATIVES

Based on this basic example there are a few more constructions that can be applied:

Alternative 2

The first one is building a **calling** macro that calls the target macro using the delaying %NRSTR macro function:

```
%PUT Indirectly via CALL EXECUTE to a not delayed call to macro CallTargetMacro;
%MACRO CallTargetMacro;
  %NRSTR(%TargetMacro) /* semicolon superfluous, called macro contains final semicolon */
%MEND;
```

That **surrounding macro** may now be called from CALL EXECUTE without using the %NSTR macro function, logically still forcing the correct, intended course of the code and result:

```
DATA _NULL_;
  CALL EXECUTE('%CallTargetMacro');
RUN;
```

Alternative 3

Another logical alternative is, instead of calling the delayed target macro from another (surrounding) macro, to call it from a **macro variable**, actually to build the call as the value of a macro variable, as in the example:

```
%PUT Indirectly via CALL EXECUTE to a call to a macro variable CallTargetMacro;
%LET CallTargetMacro = %NRSTR(%TargetMacro);
DATA _NULL_;
  CALL EXECUTE('&CallTargetMacro');
RUN;
```

It can be logically understood that by delayed resolving the macro variable to its value the code resolves to its basic form.

Alternative 4

Yet another possible. correct alternative, where the quotes around the call to the target macro are in the macro value, is:

```
%PUT Indirectly via CALL EXECUTE to a call to an unquoted macro variable CallTargetMacro;
%LET CallTargetMacro = '%NRSTR(%TargetMacro)';
DATA _NULL_;
  CALL EXECUTE(&CallTargetMacro); %* &CallTargetMacro resolves to the quoted, delayed target
macro call;
RUN;
```

and …

Alternative 5

where the quotes around the call to the target macro are in the surrounding, calling macro:

```
%PUT Indirectly via CALL EXECUTE to an unquoted call to macro CallTargetMacro;
%MACRO CallTargetMacro;
  '%NRSTR(%TargetMacro)' /* semicolon superfluous, called macro contains final semicolon */
%MEND;
DATA _NULL_;
  CALL EXECUTE(%CallTargetMacro);
RUN;
```

Alternative 6

The macro call is contained in a **SAS character variable** assigned with single quotes (to avoid immediate execution):

```
%PUT Indirectly via CALL EXECUTE to a call to a SAS variable CallCode;
DATA _NULL_;
  CallTargetMacro = '%NRSTR(%TargetMacro)';
  CALL EXECUTE(CallTargetMacro);
RUN;
```

The practical use of these other than basic alternatives may not be quite evident, but they may once prove to be useful. In any case it has been shown that they work correctly indeed.

CONCLUSIONS

The **functionality** of PROC APPEND is **limited**, too limited for many purposes (appending datasets with different structures). It may cause the **loss of information** in subsequent datasets. Many users are not aware of that. Macro %_Append_ offers the **same functionality without loss of information** (variable structure and character variable length).

Macros called from CALL EXECUTE (between single quotes) have their **macro code executed immediately**, while the **execution of the generated SAS code is delayed**. This may cause **unintended behavior, erroneous results or SAS errors**, especially if some macro code is intended to run or to act dependent on the preceding SAS code results. Apart from designing and writing CALL EXECUTE proof macros such that all embedded macro code may safely run before the generated SAS code the general solution to run macros safely from CALL EXECUTE is to **delay the macro code resolution and execution** as well by specifying the macro call using the **%NRSTR macro quoting function**.

REFERENCES

1. SAS Institute Inc. 2004. Base SAS® 9.1.3 Procedures Guide. Cary, NC: SAS Institute Inc.
2. SAS Institute Inc. 2005. SAS ® 9.1.3 Language Reference: Dictionary, Third Edition. Cary, NC: SAS Institute Inc.Guide. Cary, NC: SAS Institute Inc.
3. http://jim.groeneveld.eu.tf/software/SASmacro/_Append_.zip or http://jim.groeneveld.eu.tf/_append_

CONTACT INFORMATION

Author Name :	Y. (Jim) Groeneveld	Work Phone :	+31 (0)73 523 6000
Company :	OCS Biometric Support	Fax :	+31 (0)73 523 6600
Address :	P.O. Box 3434	Email :	sasquestions@ocs-consulting.com
	5203 DK ‘s-Hertogenbosch	Web :	http://www.ocs-biometricsupport.com
	The Netherlands	Author’s website :	http://jim.groeneveld.eu.tf