ABSTRACT

A macro that can take more or less arguments when it is called than formally specified is often referred to as a varargs macro. SAS macro language supports the development of varargs macros because of its simplicity and flexibility in macro programming. The goal of this paper is to describe how to develop SAS varargs macros with three different approaches. It first explains how to create and use a typical SAS varargs macro. Then it examines two alternative methods for constructing a SAS varargs macro. The pros and cons of three approaches are discussed. Some special issues with SAS varargs macros are further explored. Numerous examples are presented to illustrate how to make the best use of SAS varargs macros in integrated clinical study analysis and reporting.

1. INTRODUCTION

In clinical studies, data with the same characteristics are often pooled together to form standard datasets; variables in the same category are usually analyzed with the same statistical method and reported with tables of same style. SAS Macros or macro functions are commonly used to encapsulate the programming codes for those recurring clinical problems. As a generic mechanism, SAS macros are often developed with a predetermined number of parameters. Sometimes, however, you may find that the number of arguments that are passed to a macro cannot be determined until the macro is called; sometimes they are better determined at run time. Consider a debugging macro that prints a group of variables in different datasets to the output window. Declaring the macro with a fixed number of arguments makes it virtually useless; at one time, you want to examine only a single variable in one dataset whereas at another time you need to examine a wide range of variables across multiple datasets under different conditions. It makes a lot of sense for the debugging macro to take arbitrary number of meaningful arguments at the invocation; otherwise you would end up with writing a separate debugging macro call for every variable in a dataset. You can come up with many such scenarios from time to time.

It can't have escaped your notice that some SAS Data Step functions such as sum(), and cats() accept as many actual parameters as the caller chooses to pass. For examples:

```sas
Sum1 = sum(2,4,1); /* add up 3 values */
Sum2 = sum(5,8,0,3,4, 10, 2, 3, 4, 10); /* add up multiple values */
S1 = Cats("One", "Two");/* Concatenates 2 strings */
S2 = Cats("One", "Two", "Three", "and More");/* Concatenates multiple strings */
```

Those functions have a variable argument list. They are often referred to as varargs functions (varargs is short for variable arguments). Varargs functionality is a favorite feature in modern programming languages such as Perl, Python, C#, Java, and Splus [2][3][4][5][6]. Lots of their built-in function libraries, or APIs, are developed by using this popular feature. Base SAS hasn't allowed users to create their own varargs functions in DATA Step yet, but SAS macro language does support the development of varargs macros [1]. In many clinical studies, especially the studies that involve Integrated Summary of Safety (ISS) and/or Integrated Summary of Efficacy (ISE), you will realize this feature can be very attractive due to its simplicity and flexibility in coding.

In this paper, I will first describe three different approaches to writing SAS varargs macros, and explore their pros and cons. I then will discuss some special issues with SAS varargs macros. Finally I explore the methods that make SAS varargs macros flexible, easy to implement and simple to use for integrated clinical study analysis and reporting. Numerous examples are provided to help you to understand and use varargs techniques better.

2. WRITING TYPICAL SAS VARARGS MACROS

As you know, a SAS macro can be defined as follows:

```sas
%MACRO macro-name(formal-parameter-list)</option(s)>;
  <macro statements>
    . . .
```
The keyword %MACRO introduces a macro definition. It must be followed by the macro name, and the parenthesized list of formal parameters separated by commas. If you would like, one or more options such as CMD, PBUFF, and STORE can be appended after a forward slash. The macro body consists of macro statements that start after the semicolon and end before the keyword %MEND. A SAS macro must be defined first and then be called. Once the macro is executed, the actual parameters (arguments) to the macro call become local macro variables of the called macro. Usually an ordinary SAS macro cannot accept more actual parameters than it is specified in its definition. If you do so, you get error messages. However, SAS macro language does provide a syntactic mechanism that allows you to have a macro that takes a variable number of arguments. Declaring and using a typical SAS varargs macro involves three steps:

- **Declare** a typical varargs macro with option PBUFF or PARMBUFF;
- **Implement** the varargs macro by writing code to access the individual variable arguments and perform the required functionality with them;
- **Call** the varargs macro with variable-length argument list.

**STEP 1: DECLARATION**
A typical SAS varargs macro can be declared as follows:

```sas
%Macro MacroName(formal-parameter1, formal-parameter2,..., formal-parameter_n)/PBUFF;
```

The PBUFF (or PARMBUFF) option indicates that the macro may take an arbitrary number of parameters when it is called. Suppose you want to write a varargs macro that takes any number of datasets as parameters and prints their observations to the output window. The macro can be simply declared as follows:

```sas
%Macro Print1(DSN1)/PBUFF;
    < Macro body>
%Mend;
```

Where the formal parameter DSN1 is only a prototype of arguments that will be passed in.

**STEP 2: THE IMPLEMENTATION**
SAS macro language defines the syntax for declaring a typical varargs macro; however, the language itself provides no mechanism for such a macro to access its non-required arguments. When a varargs macro is invoked, two things occur:

- All required parameters are assigned to their corresponding local macro variables as given in the macro declaration
- All actual parameters (required parameters + additional parameters) are packed and saved into an automatic local macro variable called SYSPBUFF.

Therefore, a typical SAS varargs macro is actually equivalent to an ordinary macro with the special macro variable SYSPBUFF as a parameter.

However, you have to write your own code to programmatically scan the macro variable SYSPBUFF and perform the required functionality with the extricated parameters. So retrieving the actual parameters is a vital part in implementing a typical SAS varargs macro. Unfortunately, there is no standard way to do so. Below is one of the implementations of the macro %Print1, which uses built-in %scan macro to get variable arguments from automatic macro variable SYSPBUFF.

**Example 2.1**

```sas
%Macro Print1(DSN1)/PBUFF;
    %local args idx argv;
    %let args = %qsubstr(&syspbuff,2, %length(&syspbuff) - 2);
    %if %length(&args) eq 0 %then %let args = _last_
    %let idx = 1;
```
You may replace this varargs macro with an ordinary macro `%Print1(DSNList=)` which is a simulated approach to varargs macro that will be discussed later. The difference is that a typical varargs macro has the syntactic advantage that makes it easy to use, and simple to extend.

### STEP 3. CALLING VARARGS MACRO

You don't have to do anything special to invoke a varargs macro. Just call it as usual by putting the commas-separated arguments inside the macro parentheses. However, you can call the varargs macro with additional actual parameters or no parameters depending on the implementation. Below are three examples that use `%Print1`.

- /* No parameters, print the last used dataset. */
  %Print1
- /* Print selective variables in datasets demog and vital */
  %Print1(demog(drop=center), vital(keep= patid visit value))
- /* Print a few observations from datasets demog, vital, and AE */
  %Print1(demog(keep=patid where=(patid=1234)),
    vital(where=(pulse>100)),
    AE(where=(LabTest="Blood Chemistry")))

### 3. ALTERNATIVES

As you may have noticed, a typical SAS varargs macro is syntactically convenient for the caller, but it takes extra effort to implement. You have to write code to retrieve the variable arguments properly. Failing to do that can crash the varargs macro easily. However, following two alternative approaches can be used when appropriate.

#### SPECIFYING DEFAULT ARGUMENTS

Declaring a macro with default keyword parameters without PBUFF/PARMBUFF option enables you to create a macro that can be called with fewer arguments than it is defined. For example, you can define the following macro that can print variables in a dataset with a varying number of titles (up to 2 in this case) with this approach.

```sas
%Macro Print2(DSN, VARList=, Title1=, Title2=);
  Title1 &title1;
  Title2 &title2;
  proc print data=&DSN;
    var &VARList;
  run;
%Mend Print2;
```

If you want to print variables `PatientID` and `BirthDay` in dataset `DEMOG` with one title, you can omit the second title argument by making the following call.

```sas
%Print2(Demog, VarList=PatID  DOB, Title1="Demographic Dataset")
```

Unlike the previous typical varargs macro approach, this approach requires you to predetermine the maximum number of variable arguments that will be used. This might not be an effective solution if you want to be able to support any number of arguments being passed in. However, this approach doesn’t require you to write code to retrieve the individual arguments, thus reducing your programming effort.
PACKING A VARIABLE ARGUMENT LIST INTO A KEYWORD PARAMETER

As mentioned before, a typical SAS varargs macro is actually equivalent to a macro with a special keyword parameter SYSPBUFF. Instead of using PBUFF option to provide a variable argument list, you can pack the variable arguments into one or more keyword parameters and pass them to a regular macro. For example, if you want to write a macro that calculates mean and median for a group of variables in a dataset and stacks the results vertically in an output dataset, you can write the macro as follows:

```sas
%Macro Summary1(DSN=, VARList=, OUTDSN=);
  %local idx j Var;
  %let idx = 1; %let Var = %scan(&VARList, &idx);
  %do %while(%length(&Var));
    proc univariate data=&DSN noprint;
      var &Var;
      output out=_out&idx mean=mean median=median;
    run;
    Data _out&idx;
    length VARID $64;
    set _out&idx;
    VARID="%upcase(&DSN._&VAR)";
    run;
    %let idx = %eval(&idx + 1);
  %let Var = %scan(&VARList, &idx);
  %end;
  %let idx = %eval(&idx - 1);
  %if &idx > 0 %then %do;
    Data &OUTDSN;
    set %do j = 1 %to &idx; _out&j %end; %str(;)
    run;
  %end;
%mend Summary1;
```

Then you may call it by:

```sas
/* Compute mean and median of variable height and weight. */
%Summary1(DSN=vital, VARList=height weight, OUTDSN=out1)
/* Compute mean and median of variable height, weight and pulse. */
%Summary1(DSN=vital, VARList=height weight pulse, OUTDSN=out1)
```

This approach is actually very popular, because it allows you to use keyword parameter(s) to replace a packed variable argument list. You, however, have to assemble the keyword parameters by yourself prior to invoking the macro. You thus lose some of the simplicity and elegance that the typical varargs syntax provides.

The three styles of varargs macros discussed above should provide you different trade-offs for efficiency, safety, and convenience. In real life, the three approaches can be intermingled to provide very flexible solutions to complex problems. You will see some examples shortly in section 5.

4. SPECIAL ISSUES WITH SAS VARARGS MACROS

There are four key issues you have to pay attention to while developing SAS varargs macros. You should

- Carefully design the calling convention of your varargs macros, and make them simple to use and easy to implement;
- Carefully write code to retrieve the individual arguments from the variable argument list and make sure they work well in a variety of situations;
- Carefully choose and use `scan` functions to get the right parameter values from variable argument list in the right form (either quoted or unquoted);
- Carefully document your varargs macros with sufficient examples for different usages.

CALLING CONVENTION

Although a SAS varargs macro can take an arbitrary number of actual parameters when it is called, the
implementation code must be able to figure out how many arguments your macro was actually called with, and how to reference them. Since there is no standard way to do this, it is very important for you to properly design the variable parameter list style and make sure the calling convention can be easily understood, implemented, and used.

Suppose there is a big clinical programming problem M that can be solved by breaking it up into a bunch of sub-problem P_i, i = 1, 2, ..., n. Below are four typical calling conventions or patterns that people often use depending on the nature of the big problem M.

- **Calling Pattern 1:** If for each sub-problem P_i there is a corresponding macro solution %P(C_1, C_2, ..., C_k, X_i), where C_1, C_2, ..., C_k are positional parameters shared by all macro solutions (value of k being the same for all cases), and X_i is a positional parameter specified for the sub-problem P_i, you can design a varargs macro for problem M with following calling convention:

  \%M(C_1, C_2, ..., C_k, X_1, X_2, ..., X_n)

  Varargs macro %M can call %P n times to solve the problem M. It can be declared as follows:

  \%Macro M(C_1, C_2, ..., C_k, X_1)/pbuff;

  Example 2.1 uses this calling convention with k=0.

- **Calling Pattern 2:** If for each sub-problem P_i, there is a corresponding macro solution %P(C_1, C_2, ..., C_k, X_i=), where C_1, C_2, ..., C_k are positional parameters shared by all macro solutions (value of k being the same for all cases), and X_i= is a keyword parameter specified for the sub-problem P_i, you can design a varargs macro for problem M with following calling convention:

  \%M(C_1, C_2, ..., C_k, X_1=, X_2=, ..., X_n=)

  Varargs macro %M can call %P n times to solve the problem M. It can be declared as follows:

  \%Macro M(C_1, C_2, ..., C_k, X_1=)/pbuff;

  Examples 3.1 and 4.1 use this calling convention.

- **Calling Pattern 3:** If for each sub-problem P_i, there is a corresponding macro solution %P(A_i=, B_i=, ..., F_i=), where A_i=, B_i=, ..., F_i= are the required indexed keyword parameters for the sub-problem P_i, you can design a varargs macro for problem M with following calling convention:

  \%M(A_1=, B_1=, ..., F_1=,
  A_2=, B_2=, ..., F_2=,
  A_3=, B_3=, ..., F_3=,
  ..., 
  A_n=, B_n=, ..., F_n=)

  Varargs macro %M can call %P n times to solve the problem M. It can be declared as follows:

  \%Macro M(A_1=, B_1=, C_1=, ..., F_1=)/pbuff;

- **Calling Pattern 4:** If for each sub-problem P_i, there is a varargs macro solution %P(A_i=, B_i=, ..., F_i=, X_i1=, X_i2=, ..., X_im=), where A_i=, B_i=, ..., F_i=, are the required indexed keyword parameters for the sub-problem P_i, and X_i1=, X_i2=, ..., X_im=, ..., are variable argument list for sub-problem P_i, you can design a varargs macro to problem M with following calling convention:

  \%M(A_1=, B_1=, ..., F_1=, X_11=, X_12=, ..., X_1m=,
  A_2=, B_2=, ..., F_2=, X_21=, X_22=, ..., X_2m=,
  A_3=, B_3=, ..., F_3=, X_31=, X_32=, ..., X_3m=,
  ..., 
  A_n=, B_n=, ..., F_n=, X_n1=, X_n2=, ..., X_nm=)
Varargs macro %M can call %P \( n \) times to solve the problem M. It can be declared as follows if you like:

\[
\text{%Macro } \ M(A1=, B1=, C1=, \ldots, F1=, X11=, X12=) / pbuf; \\
\]

Example 5.1 in the next section uses this calling pattern.

You can certainly create your own calling patterns or use the combination of the four patterns above for your varargs macros on condition that the variable argument list can be easily parsed. As you may have noticed indexed keyword parameters are heavily used in above patterns except pattern 1. This is because the using indexed keyword parameter is a way to simulate arrays in macro language. It enhances the readability and usability of the calling code and at the same time makes variable parameters easy to retrieve with do loops. Calling pattern 1 can be used if your varargs macros act like DATA Step functions sum(), or cats().

**PARSING VARIABLE ARGUMENT LISTS**

There are three issues you have to deal with when you write codes to parse variable argument lists.

- Since a varargs macro can accept more or less actual parameters than the number of formal parameters declared, the varargs macro implementation should be written to deal with empty argument list, or argument list with fewer arguments declared.
- If a varargs macro has declared some positional parameters, you must decide where the first variable argument starts in the variable argument list.
- The varargs macro implementation must be able to determine when a variable argument list is exhausted. This can be done by using a special value at the end of variable argument list, such as empty value, or special characters. A developer can also use a required argument that indicates the number of variable arguments passed in.

**CHOOSING PROPER SCAN FUNCTION**

The SAS built-in macro %scan() and %qscan() is often used to retrieve individual comma-separated argument lists passed in the macro variable SYSPBUFF. For parameters with quoted texts, it is suggested that you use %qscan() to locate them. It is important to remember that not all actual parameter values can be retrieved properly with %scan, or %qscan(). If you have an actual parameter that contains commas enclosed in quotation marks, you may have to use %sysfunc(scanq()) or %qsysfunc(scanq()) instead of %scan() or %qscan(). The reason is that new DATA Step function scanq() is able to find the \( n \)th parameter in the variable argument list by ignoring commas that are enclosed in quotation marks. For example, below is a robust and flexible version of macro %Print2 that prints any number of titles that might contain commas in quotation marks.

**Example 4.1**

```sas
%Macro Print3(DSN=, VARList=, Title1=) / PBUFF;
%local args argv var idx title;
%let args =&SYSPBUFF;
%if %length(&DSN) eq 0 %then DSN= _last_; \\
%let idx = 3; \\
%let argv = %qsysfunc (scanq(&args, &idx, %str((,)))); \\
%do %while (%length(&argv)); \\
  %let title = %qscan(&argv, 2, %str(=)); \\
  title%eval(&idx -2) &title; \\
  %let idx = %eval(&idx + 1); \\
  %let argv = %qsysfunc (scanq(&args, &idx, %str((,)))); \\
%end; \\
Proc print data=&DSN width=minimum; \\
  Var &VARList;
Run;
%Mend Print3;
```

You can call %Print3 by

```sas
%Print3(DSN=DEMOG, VARList=patid gender dob,
```
Sometimes, you even have to write your own version of scan macro functions to deal with special parameter values, such as values that have commas enclosed in balanced brackets.

DOCUMENTING YOUR VARARGS MACROS

SAS varargs macros should be well documented for its proper use because formal parameters defined with varargs macros only provide a prototype of variable arguments. Calling convention and one or more examples should be provided to illustrate its various usages.

5. ENHANCING YOUR CLINICAL MACROS WITH VARARGS TECHNIQUES

Varargs macros are widely used in building standard macro libraries for clinical study analysis and reporting. Though it is not a trivial task to develop a good varargs macro, it does bring lots of benefits to its users when it is well developed and documented:

- **Two birds with one stone.** You can solve two or more problems with just one varargs macro instead of one call for each problem. For example, if you want use %Print3 to print variables in three different datasets with different titles, you might call %Print3 three times, but with the %PrintN described below, you only need to call %PrintN once.

- **Cleaner and shorter code.** A varargs macro allows you to encapsulate the lengthy and complex solution to an array of similar clinical problems in a macro definition so that you can invoke it to solve any size of problems simultaneously in existing or future clinical studies. This kind of coding raises the level of abstraction from implementation details no matter what size a problem has, and at the same time greatly reduces the verbosity of code. It is especially useful for ISS/ISE studies that have to pool data together from different study datasets.

- **More reliable and reusable code.** With varargs techniques, you can extend your current macro solutions that deal only with one variable, one dataset, or one study to universal solutions that can deal with multiple variables, multiple datasets, and even multiple studies. This kind of code reuse (of the solutions contained in the macros) helps you in the long run since you are repeatedly using the tried and tested code.

- **More scalable code.** One particularly nice feature about a varargs macro is scalability. A well-developed varargs macro can be invoked to solve an array of problems of the same type with a single macro call; on the other hand, a large problem can be scaled down to a number of smaller workable problems with a list of well-designed variable arguments.

Below I will show two more examples of varargs macros that can be mimicked in clinical study analysis and reporting: one is an extension of macro %Summary1, and the other is a more generic version of macro %Print2.

### Example 5.1

Suppose you want to extend macro %Summary1 so that it can calculate the mean and median for a varying number of variables in multiple study datasets and stack all results vertically in one output dataset. Some typical usages of %SummaryN in an ISS/ISE study are showed as follows:

**Case 1:**

```
%SummaryN(DSN1=Study1.Vital, VARList1=pulse, OUTDSN=OutdDSN);
```

**Case 2:**

```
%SummaryN(DSN1=Study1.Vital, VARList1=pulse,  
            DSN2=Study2.Vital, VARList2=height weight,  
            OUTDSN=OutdDSN);
```

**Case 3:**

```
%SummaryN(DSN1=Study1.Vital, VARList1=pulse,  
            DSN2=Study2.Vital, VARList2=height weight,  
            DSN3=Study3.Vital, VARList3=pulse height weight,  
            OUTDSN=OutdDSN);
```
You might have noticed that the mean and median calculation problem on large number of variables in different study datasets are decomposed into several smaller problems that can be solved by %Summary1, and all the results that are then combined by macro %SummaryN at the end to give the final result.

**Example 5.2.** Suppose you want to create a generic macro called %PrintN that can print an arbitrary number of variables in a group of clinical datasets. Each printed dataset can have varying titles (up to 2 in this case) that might include commas in enclosed quote marks. Below are some usages of %PrintN that will print variables in different clinical datasets with different titles:

**Case 1:**
%PrintN(DSN1=Demog, Vars1= Patid Gender, Title11="Print Patid and Gender in Dataset Demog")

**Case 2:**
%PrintN(
  DSN1=Demog, Vars1= Patid Gender, Title11="Print Patid and Gender in Dataset Demog",  
  DSN2=Vital, Vars2= Height Weight Pulse, Title21="Print Height, Weight and Pulse",  
  Title22="in Dataset Vital")

**Case 3:**
%PrintN(
  DSN1=Demog, Vars1= Patid Gender, Title11="Print Patid and Gender in Dataset Demog",  
  DSN2=Vital, Vars2= Height Weight Pulse, Title21="Print Height, Weight and Pulse",  
  Title22="in Dataset Vital",  
  DSN3=LAB, Vars3= Test TestType Result Unit,
One of the implementations of %PrintN is listed as follows

`%Macro PrintN(DSN1=, VARList1=, Title1=, Title2=)/PBUFF;`  
`/* 1. Decompose the variable argument list into local macro variables */`  
`%local idx argv args i var N;`  
`%let args =&syspbuff;`  
`%let idx = 1;`  
`%let N = 0;`  
`%let argv =%qscan(&args, &idx, %str((,)));`  
`%do %while (%length(&argv));`  
`%let var=%scan(&argv, 1, %str(=));`  
`%let &var = %qscan(&argv, 2, %str(=));`  
`%if %upcase(%substr(&var, 1, 3)) EQ DSN %then %let N = %eval(&N + 1);`  
`%let idx = %eval(&idx + 1);`  
`%let argv =%qscan(&args, &idx, %str((,)));`  
`%end;`  
`/* 2. Print all datasets */`  
`%let i = 1;`  
`%do %while (&i LE &N);`  
`%print2(DSN=&&DSN&i, VARList=&&VARList&i,`  
`    Title1=&&Title&i.1, Title2=&&Title&i.2);`  
`%let i = %eval(&i + 1);`  
`%end;`  
`%Mend PrintN;`  

Note that %PrintN is implemented by extending %Print2.

6. CONCLUSION  
This paper describes the way to declare and develop typical SAS varargs macros, and the way to document and call them properly, as well as other two simulated methods that offer similar functionality. This should help you to develop better, cleaner, and more flexible SAS macros in no time for clinical study analysis and reporting. As a professional SAS programmer, if you need to extend the functionality of an existing SAS macro, or prepare a SAS macro for its increasing uses in a variety of situations, or simply would like your SAS macro to be as flexible as possible, one thing you should look into is developing and using varargs macros.

ACKNOWLEDGMENTS  
The author wish to thank Dr. Izabella Peszek, and Roberge, Sylvianne B. for their kind comments and constructive suggestions during the preparation of this paper.

REFERENCE  

CONTACT INFORMATION  
Your comments and questions are valued and encouraged. Contact the author at:

Lei Zhang
Merck & Co., Inc.
RY34-A320  P.O. Box 2000

George Li
Amgen Inc.
One Amgen Center Drive
SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® Indicates USA registration.

Other brand and product names are trademarks of their respective companies.