Journal of Scientific Research & Reports
2(2): 719-740, 2013; Article no. JSRR.2013.019

SCIENCEDOMAIN *international*
www.sciencedomain.org

Macros Development for Create the Reusable Programs and Generate Customize Result in Clinical Data Reporting

Vasanth Kumar Kunitala [1*], Praveen Kumar Jinka[1], Varun Kumar[1] and Ahamed Kabeer[2]

[1]B·11, 2nd Floor Sector 65, Nodia - 201301 Uttar Pradesh 012-4810100, India.
[2]International drug discovery and clinical research, 8-2-293/82/J/105, plot no 105A, 2nd floor, near syndicate bank, journalist colony, jubilee hills, India.

Authors' contributions

This work was carried out in collaboration between all authors in wincere solutions pvt ltd. All authors in Wincere solutions SAS team was read and approved the final manuscript.

Research Article

Received 5th April 2013
Accepted 8th June 2013
Published 25th September 2013

ABSTRACT

The article is mainly describe the purpose of the macro facility and efficiency of macro-based applications in clinical domain for create, tests and provides resolution to bugs, defects and other changes to the SAS macro library. It mainly focus on fundamental concept of program flow, tokenization, %INCLUDE statement, %LET statement, macro triggers, macro statements, macro variables, global and local symbol tables, automatic macro variables, macro variable reference, substitution within a macro statement, substitution within a SAS literal, unresolved reference, substitution within SAS code, referencing macro variables, combining macro variables with text. It will give the brief idea about macro functions, defining a macro, macro compilation, calling a macro ,macro storage, macro parameters, SYMPUT routine, creating a series of macro variables, creating macro variables in SQL, the need for macro-level programming, conditional processing, monitoring macro execution, macro syntax errors, parameter validation, developing macro-based applications, iterative processing, the SYMPUTX routine, rules for creating and updating variables, rules for resolving variables, multiple local tables. This article shows complete concept of the macro system.

**Corresponding author: E-mail: kunitala.kumar@wincere.com;*

Keywords: Program flow; developing macro-based applications; macro triggers; macro statements; macro variables; global and local symbol tables.

DEFINITION

SQL: Structured Query Language; SAS: Statistical analysis Software; SCL: SAS Component Language.

1. INTRODUCTION

1.1 Purpose of the Macro Facility

The macro facility is a text processing facility for automating and customizing flexible SAS code. The macro facility supports, symbolic substitution within SAS code, automated production of SAS code, dynamic generation of SAS code and conditional construction of SAS code. The macro facility enables you to create and resolve macro variables anywhere within a SAS program. It can write and call macro programs (macros) that generate custom SAS code. Automatic macro variables, which store system information, can be used to avoid hard-coding of these values. User-defined macro variables enable you to define a value once and then substitute that value as often as necessary within a program. Macro programs can conditionally execute selected portions of a SAS program based on user-defined conditions. The macro facility can generate SAS code repetitively and substitute different values with each of the iteration.

1.2 Program Flow

A SAS program can be any combination of DATA steps and PROC steps, Global statements, SAS Component Language (SCL), Structured Query Language (SQL), SAS macro language. When you submit a program it was copied in to a memory called the *input stack*. Input Stack submits 1) Command 2) Stored Process 3) Batch or Non-interactive Submission. Once SAS code is in the input stack a component of SAS called the *word scanner* reads the text in the input stack, character by character, left-to-right, top-to-bottom and breaks the text into fundamental units called *tokens*. The word scanner passes the tokens, one at a time to the appropriate compiler as the compiler demands. The compiler requests tokens until it receives a semicolon, performs a syntax check on the statement and repeats this process for each statement. SAS suspends the compiler when a step boundary is encountered, executes the compiled code if there are no compilation errors and repeats this process for each step. Combining Macro Variables with Text.

Example was shown in Table 1.

Table 1. Combining macro variables with text

1)Data-Driven Applications	**3)Combining Macro Variables with Text**

data c h hy n w wa ;
set Mpd.Nc;
select(location);
when("coll") output c;
when("hos") output h;
when("hyd") output hy;
when("nzb") output n;
when("wae") output w;
when("war") output wa;
otherwise;end; run;
%let site=coll;
title "Good project result shown patient's CENTER &site ";
proc tabulate data=Mpd.Nc;
where location = "&site";
class full_name code Location;
var amount;
table full_name*Location*code*amount;
run;
 	
2)Macro Variable Name Delimiter
%let l=Mpd;
%let graphics=g;
%let table=Nc;
%let a=amount;
proc &graphics.chart data=&l..&table;
hbar code / sumvar=&a;
run;
proc &graphics.plot data=&l..&table;
plot &a*code;
run;
	%let table=Nc;
	proc chart data=Mpd.&table;
	hbar code / sumvar=amount;
	run;
	proc plot data=Mpd.&table;
	plot amount*code;
	run;
	%let table=N;
	%let set=c;
	proc chart data=Mpd.&table&set;
	hbar code / sumvar=amount;
	run;
	proc plot data=Mpd.&table&set;
	plot amount*code;
	run;
	%let table=N;
	%let set=c;
	%let c=code;
	%let a=amount;
	proc chart data=Mpd.&table&set;
	hbar &c/sumvar=&a;
	run;
	proc plot data=Mpd.&table&set;
	plot &a*&c;
	run;
	%let graphics=g;
	%let table=Nc;
	%let a=amount;
	proc &graphics.chart data=Mpd.&table;
	hbar code / sumvar=&a;
	run;
	proc &graphics.plot data=Mpd.&table;
	plot &a*code;
	run;

1) Tokenization

The word scanner recognizes four classes of tokens: 1) Literal tokens:- A *literal token* is a string of characters enclosed in single or double quotes. Examples: 'Any text' or "Any text". The string is treated as a unit by the compiler. 2) Number tokens:- *Number tokens* can be integer numbers, including SAS date constants, floating point numbers, containing a decimal point and/or exponent. Examples: 3, 3, 3.5, -3.5, '01jan2002'd, 5E8, 7.2E-4 3) Name tokens: *Name tokens* contain one or more characters beginning with a letter or underscore and continuing with underscores, letters, or numerals. Format and informat names contain a period. Examples: infile, _n_, item3, univariate, dollar10.2. 4) Special tokens:- *Special tokens* can be any character, or combination of characters, other than a letter, numeral, or underscore. Examples: *** / + - ** ; $ ( ) . & %** @ # **= ||** A token ends when the word

scanner detects, the beginning of another token, a blank after a token. Blanks are not tokens they are delimit tokens. The maximum length of a token is 32,767 characters.

2) The %INCLUDE statement

The %INCLUDE *statement* copies SAS statements from an external file to the input stack. It is a global SAS statement. And it is not a macro language statement. It can be used only on a statement boundary. The contents of the external file are placed on the input stack. The word scanner then reads the newly inserted statements. In %INCLUDE statement using SOURCE2 option requests inserted SAS statements to appear in the SAS log.

3) Macro triggers

During word scanning, two token sequences are recognized as macro triggers:1) %*name-token* a macro statement, function, or call 2) &*name-token* a macro variable reference. The word scanner passes macro triggers to the macro processor, which requests additional tokens as necessary and performs the action indicated.

4) Macro statements

It begins with a percent sign (%) followed by a name token and end with a semicolon. It must represent macro triggers and these are executed by the macro processor.The %PUT statement writes text to the SAS log and writes to first column of the next line. It writes a blank line if no text is specified and does not require quotes around text. It is valid in open code (anywhere in a SAS program).

5) Macro variables

Macro variables store text, including complete or partial SAS steps or complete or partial SAS statements. Macro variables are referred to as *symbolic variables* because SAS programs can refer macro variables as symbols for additional program text.

1.3 Global Symbol Table

Macro variables are stored in an area of memory called the *global symbol table.* When SAS is invoked, the global symbol table is created and initialized with automatic macro variables. User-defined macro variables can be added to the global symbol table. Macro variables in the global symbol table are global in scope (available any time). They have a minimum length of 0 characters (*null value*). They have a maximum length of 65,534 (64K) characters. It stores numeric tokens as character strings.

Automatic Macro Variables are system-defined. There are created at SAS invocation are global (always available) and are assigned values by SAS. It can be assigned values by the user in some cases. Some automatic macro variables have fixed values that are set at SAS invocation: 1) SYSDATE date of SAS invocation DATE7. 2) SYSDATE9 date of SAS invocation DATE9. 3) SYSDAY day of the week of SAS invocation 4) SYSTIME time of SAS invocation. 5) SYSSCP abbreviation for the operating system: OpenVMS, WIN and HP etc. 6) SYSVER release of SAS software being used.

Some automatic macro variables have values that change automatically based on submitted SAS statements: SYSLAST displays the name of most recently created SAS data set in the

form of *libref.nam*e. If no data set has been created, the value is _NULL_. SYSPARM displays text specified at program invocation. Example: Writes the names and values of all automatic macro variables to the SAS log using the _AUTOMATIC_ argument of the %PUT statement. The macro variables SYSDATE, SYSDATE9, and SYSTIME stores as character strings, not SAS date or time values.

1.4 Macro Variable Reference

Macro variable references begin with an ampersand (&) followed by a macro variable name and represent macro triggers. These are also called as *symbolic* references and it can appear anywhere in the program and are passed to the macro processor. When the macro processor receives a macro variable reference, it searches the symbol table for the macro variable and it places the macro variable's value on the input stack and issues a warning to the SAS log if the macro variable is not found in the symbol table.

1) Substitution within a macro statement

When a macro trigger is encountered, it is passed to the macro processor for evaluation. The macro variable reference triggers the macro processor to search the symbol table for the reference. The macro processor resolves the macro variable reference by substituting its value. The macro processor executes the %PUT statement and writes the resolved text to the SAS log.

2) Substitution within a SAS literal

If we need to reference a macro variable within a literal, enclose the literal in double quotes. The word scanner continues to tokenize literals enclosed in double quotes, permitting macro variables to resolve. The word scanner does not tokenize literals enclosed in single quotes, so macro variables do not resolve.

SAS statements are passed to the compiler. The macro trigger is passed to the macro processor. The macro processor searches the symbol table. The resolved reference is passed back to the input stack. Word scanning continues. The double-quoted string is passed to the compiler as a unit. When a step boundary is encountered, compilation ends and execution begins.

3) Unresolved reference

Example: Reference a non-existent macro variable. The macro trigger is passed to the macro processor for evaluation. The macro processor writes a warning to the SAS log when it cannot resolve a reference. If the macro processor cannot resolve a reference, it passes the tokens back to the word scanner and the word scanner passes them to the compiler.

4) Substitution within SAS code

Example: Generalize PROC PRINT to print the last created data set, using the automatic macro variable SYSLAST.SAS statements are passed to the compiler. When a macro trigger is encountered, it is passed to the macro processor for evaluation. The *macro variable reference* triggers the macro processor to search the symbol table for the reference the macro processor resolves the macro variable reference, passing its resolved value back to

the input stack. Word scanning continues. A step boundary is encountered. Compilation ends and execution begins.

5) The %LET statement

The %LET statement creates a macro variable and assigns it a value. General form of the %LET statement: *variable* follows SAS naming conventions. If *variable* already exists, its *value* is overwritten. If *variable* or *value*s contain macro triggers, the triggers are evaluated before the assignment is made.

Value can be any string: 1) maximum length is 65,534 (64K) characters 2) minimum length is 0 characters (*null value*) 3) numeric tokens are stored as character strings 4) mathematical expressions are not evaluated 5) the case of *value* is preserved 6) quotes bounding literals are stored as part of *value* 7) leading and trailing blanks are removed from *value* before the assignment is made. %PUT _user_; Displays all user-defined macro variables in the SAS log. %put _all_; Displays all user-defined and automatic macro variables in the SAS log. The SYMBOLGEN system option writes macro variable values to the SAS log as they are resolved. The default option is NOSYMBOLGEN. The %SYMDEL statement deletes one or more user-defined macro variables from the global symbol table. Because symbol tables are stored in the memory so delete macro variables when they are not longer needed.

6) Referencing macro variables

We can reference macro variables anywhere in your program, including these special situations: Macro variable references adjacent to leading and/or trailing text: **text***&variable*, *&variable***text text***&variable***text**. Adjacent macro variable references:*&variable&variable.*

7) Combining macro variables with text

Place text immediately before a macro variable reference to build a new token. The word scanner recognizes the end of a macro variable reference when it encounters a character that cannot be part of the reference. A *period* (.) is a special delimiter that ends a macro variable reference and does not appear as text when the macro variable is resolved. Use another period after the delimiter period to supply the needed token [1].

1.5 Macro Functions

Macro functions 1) it have similar syntax as corresponding DATA step character functions 2) It have yield similar results 3) it is manipulates macro variables and expressions 4) it represent macro triggers 5) these are executed by the macro processor.6) Character comparisons are case sensitive.

1.6 Selected Character String Manipulation Functions

%UPCASE translates letters from lowercase to uppercase. The %UPCASE function translates characters to uppercase. *Argument* can be any combination of text and macro triggers.

%SUBSTR extracts a substring from a character string. The %SUBSTR function returns the portion of *argument* beginning at *position* for a length of *n* characters and returns the portion

of *argument* beginning at *position* to the end of *argument* when an *n* value is not supplied. You can specify *argument*, *positio*n, and *n* values using 1) constant text 2) macro variable references 3) macro functions 4) macro calls. It is not necessary to place *argument* in quotes because it is always handled as a character string by the %SUBSTR function.

%SCAN extracts a word from a character string. The %SCAN function 1) returns the *n*th word of argument, where words are strings of characters separated by delimiters 2) uses a default set of delimiters if none are specified 3) returns a null string if there are fewer than *n* words in argument. We can specify values for *argument*, *n*, and *delimiters* using 1) constant textb2) macro variable references3) macro functions 4) macro calls. The value of *n* can also be an arithmetic expression that yields an integer.

%INDEX searches a character string for specified text. %LENGTH returns the length of a character string or text expression.

Other functions: %SYSFUNC executes SAS Functions. The %SYSFUNC macro function executes SAS functions. *SAS function (argument(s))* is the name of a SAS function and its corresponding arguments. The second argument is an optional format for the value returned by the first argument. SYSDATE9 and SYSTIME represent the date and time the SAS session started. Most SAS functions can be used with %SYSFUNC some of the exceptions include:1)Array processing (DIM, HBOUND, LBOUND). 2) Variable information (VNAME, VLABEL, MISSING). 3) Macro interface (RESOLVE, SYMGET). 4) Data conversion (INPUT, PUT)5) Other functions (IORC, MSG, LAG, DIF). INPUTC and INPUTN can be used in place of INPUT. PUTC and PUTN can be used in place of PUT.

%EVAL performs arithmetic and logical operations. The %EVAL function 1) performs arithmetic and logical operations 2) truncates non-integer results 3)returns a character result 4) returns 1 (true) or 0 (false) for logical operations 5)returns a null value and issues an error message when non-integer values are used in arithmetic operations. Few Marcos functions an example was shown in Table 2.

%BQUOTE protects blanks and other special characters. The %BQUOTE function removes the normal meaning of special tokens that appear as constant text. Special tokens include: + - * /, < > = LT EQ GT AND OR NOT LE GE NE

The %BQUOTE function 1) protects (quotes) tokens so that the macro processor does not interpret them as macro-level syntax 2) enables macro triggers to work normally3) preserves leading and trailing blanks in its argument. Use PROC DATASETS to investigate the structure of the last data set created [2].

1.7 Defining a Macro

A *macro* or *macro definition* enables you to write *macro programs*. *Macro-name* follows SAS naming conventions. *Macro-text* can include any text, SAS statements or steps, macro variables, functions, statements, or calls, any combination of the above.

Table 2. Marcos functions examples

1)The %UPCASE Function
%let num=vk ;
proc means data=Mpd.Nc sum maxdec=0;
where code="%upcase(&num)";
var amount;
class full_name location;
title " &num project sponsor";
run;
2)The %SUBSTR Function
proc print data=Mpd.Nc ;
where project_begin_date between "01%substr(&sysdate9,3,3)2013"d and "&sysdate9"d;
title " blood collection time";
title2 "(as of &sysdate9)";
run;
3)The %SCAN Function
%let libref=%scan(&syslast,1);
%let dsname=%scan(&syslast,2,.);
proc datasets lib=&libref nolist;
title "Contents of Data Set &syslast";
contents data=&dsname;
run;
	4)The %BQUOTE Function
	%let text=%bquote(Pharmacokinetics of MPD);
	%put %bquote(&text is the value.);
	5)The %EVAL Function
	%put lastyr=%eval(1+6+3);
	6)The %SYSEVALF Function
	%put lastyr=%sysevalf(1+6.9+3.7);
	7)The %SYSFUNC Function
	%let firstyr=%sysfunc(today(),year4.);
	%let lastyr=%eval(&thisyr+3);
	proc print data=Mpd.Nc;
	where project_begin_date ge &lastyr and &firstyr;
	title1 "Blood sample collection &lastyr and &thisyr";
	title2 "(as of &sysdate9)";
	run;
	%put title "%sysfunc(today(),weekdate.)";

1) Macro compilation

When a macro definition is submitted, macro language statements are checked for syntax errors and compiled.SAS statements and other text are not checked for syntax errors and compiled. The macro is stored as an entry in a SAS catalog, the temporary catalog WORK.SASMACR by default. The MCOMPILENOTE=ALL option issues a note to the SAS log after a macro definition has compiled. The default setting is MCOMPILENOTE=NONE. The MCOMPILENOTE= option is new in SAS®9. Produces a list of compiled macros stored in the default temporary catalog WORK.SASMACR.

2) Calling a macro

A *macro call* 1) it causes the macro to execute2) it is specified by placing a percent sign before the name of the macro 3) it can be made anywhere in a program (similar to a macro variable reference) 4) it can represents a macro trigger 5) it is not a statement so semicolon is not required.

4) Program flow

When the macro processor receives *%macro-name*, 1) it searches the designated SAS catalog (WORK.SASMACR by default) for an entry named *macro-name*. MACRO 2) it executes compiled macro language statements3) it sends any remaining text to the input stack for word scanning 4) it is pauses while the word scanner tokenizes the inserted text and SAS code executes 5) it resumes execution of macro language statements after the

SAS code executes. Macro Execution the MPRINT option writes to the SAS log the text sent to the SAS compiler as a result of macro execution.

1.8 Macro Storage

Macros are stored in the work library by default. The MSTORED system option enables storage of compiled macros in a permanent SAS library. The SASMSTORE= system option designates a permanent library to store compiled macros. The STORE option stores the compiled macro in the library indicated by the SASMSTORE= system option. The SOURCE option stores the macro source code along with the compiled code. The SOURCE option is new in SAS®9. In earlier releases, be sure to save the source code externally. Use a %COPY statement to access stored macro source code. COPY *macro-name* / SOURCE <*OUT*='*external file*'>; If the OUT= option is omitted, source code is written to the SAS log. The %COPY statement is new in SAS®9. A macro storages and parameters example was shown in Table 3.

1) Macro parameters

Macros can be defined with a *parameter list* of macro variables referenced within the macro. Parameter names and values are parenthesized or comma delimited. Parameter values can be any text, null values, macro variable references, or macro calls.

2) Local symbol tables

When a macro with a parameter list is called, the parameters are created in a separate symbol table called a *local symbol tabl*e. A local symbol table is created when a macro with a parameter list is called and deleted when the macro finishes execution. Macro variables in the local symbol table are available only during macro execution and therefore can be referenced only within the macro.

1.8 Types of Parameters

A) Positional parameters

Positional parameters use a one-to-one correspondence between parameter names supplied on the macro definition and parameter values supplied on the macro call.

B) Keyword parameters

A parameter list can include *keyword parameters*. General form of a macro definition with keyword parameters: Keyword parameters are assigned a default or null value after an equal (=) sign. *Keyword=value* combinations can be specified in any order and omitted from the call without placeholders. If omitted from the call, a keyword parameter receives its default value. To omit every keyword parameter from a macro call, specify *%macro-name*. Specifying *%macro-name* without the parentheses may not immediately execute the macro definition.

C) Mixed parameter lists

We can use a combination of positional and keyword parameters. In a mixed parameter list, positional parameters must be listed before keyword parameters on both the macro definition and the macro call. Macros application in data steps was shown in Table 4.

1.9 The SYMPUT Routine

The SYMPUT routine 1) it is an executable DATA step statement 2) it assigns to a macro variable any value available to the DATA step during execution time 3) it can create macro variables with static values dynamic (data dependent) values dynamic (data dependent) names The SYMPUT routine creates a macro variable and assigns it a value. General form of the SYMPUT routine: macro-*variable* is assigned as the character value of *text*. If *macro-variable* already exists, its value is replaced. If either argument represents a literal value, it must be quoted. We can copy the current value of a DATA step variable into a macro variable by using the name of a DATA step variable as the second argument to the SYMPUT routine. A maximum of 32,767 characters can be assigned to the receiving macro variable. Any leading or trailing blanks within the DATA step variable's value are stored in the macro variable. Values of numeric variables are converted automatically to character using the BEST12. format. We can use DATA step functions and expressions in the SYMPUT routine's second argument to left-align character strings created by numeric-to-character conversion and format data values, perform arithmetic operations on numeric data values. Example of Macro variables creation by SYMPUT Routine was shown in Table 5. The SYMPUTX routine automatically removes leading and trailing blanks from both arguments [3].

1) Creating a series of macro variables

To create a series of macro variables, use the SYMPUT or SYMPUTX routine with a DATA step variable or expression in *argument1*. Example: - **CALL SYMPUT** (*expression1*, *expression2*); expression1:- evaluates to a character value that is a valid macro variable name, unique to each execution of the routine. *expression2*:- value to assign to each macro variable. If we use the CALL SYMPUT function no macro triggers in the entire DATA step is passed to the compiler. The compiled DATA step executes after the RUN statement is encountered. The SET statement reads the first observation into the PDV. CALL SYMPUT evaluates the expressions and adds a macro variable to the symbol table.

2) Indirect references to macro variables

The Forward Rescan Rule:- Multiple ampersands preceding a name token denote an indirect reference that ends when a token encountered which cannot be part of a macro variable reference, that is, a token other than a name, an ampersand, or a period delimiter. The macro processor will re-scan an indirect reference, left to right, from the point where the multiple ampersands begin. Two ampersands (&&) resolve to one ampersand (&).Scanning continues until no more triggers can be resolved. Example (1):- title 2 "Taught by &&teach&crs"; Placing two ampersands at the start of the original token sequence alters the processing of the tokens and macro triggers. The CRS macro variable is an indirect reference to a TEACH macro variable. Example (1):- title1 "Schedule for &&&crsid"; the value of one macro variable exactly matches the name of another macro variable, three ampersands appear together in this indirect macro variable reference.

Table 3. Macro storages and parameters examples

1)Macro Storage

```
libname vasanth '.';
options mstored sasmstore=vasanth;
%let dsn=MPD.NC;
%let vars=AMOUNT;
%fee
%macro fee/ store source;
  proc print data=&dsn;
    var &vars;
  run;
%mend fee;
```

2)%*copy*

```
%copy fee / source;
%copy fee / source out='k';
```

3)Macro Parameters

```
%macro fee (dsn,vars);
  proc print data=&dsn;
    var &vars;
  run;
%mend ;
%fee(MPD.NC,AMOUNT)
```

4)Positional Parameters

```
%macro mpd(list, start, stop);
  %let start=%upcase(&start);
  %let stop=%upcase(&stop);
  proc freq data=mpd.nc;
    where project_begin_date between "&start"d and "&stop"d;
  table location / &list;
    title1 "patient's Enrollment from &start to &stop";run;
%mend;
options mprint;
%mpd(chisq,01jan2012,31dec2013)
%mpd(,01oct2012,31dec2013)
```

5)Keyword Parameters

```
%macro mpd(c=, start=01jan2012, stop=31dec2013);
  %let start=%upcase(&start);
  %let stop=%upcase(&stop);
  proc freq data=mpd.nc;
    where project_begin_date between
          "&start"d and "&stop"d;
    table location / &c;
    title1 "patient's Enrollment from &start to &stop";
  run;
%mend;
options mprint;
%mpd(c=chisq)
%mpd(c=chisq,stop=31dec2013, start=01jan2012)
%mpd()
```

6)Mixed Parameter Lists

```
%macro mpd(c=, start=01jan2012, stop=31dec2013);
  %let start=%upcase(&start);
  %let stop=%upcase(&stop);
  proc freq data=mpd.nc;
    where project_begin_date between
          "&start"d and "&stop"d;
    table location /&c;
    title1 "patient's Enrollment from &start to &stop";
  run;
%mend;
options mprint;
%mpd(chisq)
%mpd(stop=31dec2013,start=01jan2012)
%mpd(NOCOL NOCUM,stop=31dec2013)
```

Table 4. Macros application in data steps

1)The DATA Step Interface

```
%let wt=55;
data vasanth;
  set mpd.i end=lasv;
  where weight=&wt;
  total+1;
  if gender='M' then place+1;
  if location then do;
    put total= place=;
    if place<total then do;
      %let foot=5ml Blood sample is collected;
    end;
    else do;
      %let foot=10ml Blood sample is collected;
    end;
  end;
run;
proc print data=vasanth;
   var PLACE TOTAL;
   title "amount of Blood sample is collected &wt";
   footnote "&foot";
run;
```

2)The SYMPUT Routine

```
A) %let wt=60;
data prednisolone;
   set mpd.i end=final;
   where weight=&wt;
  total+1;
  if gender='M' then place+1;
  if final then do;
    put total= place=;
    if place<total then do;
        call symput('foot','eligible weight of patients');
      end;
      else do;
        call symput('foot','not suitable weight of patients');
      end; end;
run;
```

```
B)%let wt=55;
data vasanth;
  set mpd.i end=final;
  where weight=&wt;
  total+1;
  if gender='M' then place+1;
    if final then do;
      call symput('nump',place);
      call symput('numstu',total);
      call symput('crsname',weight);
   end;
run;
%let numstu=&numstu         ;
%let crsname=&crsname;
%let nump=&nump             ;
proc print data=vasanth;
   var place total weight;
   title "5ml blood sample for &crsname (#wt)";
   footnote "Note: &nump out of &numstu Given";
run;
C)%let wt=55;
data vasanth;
  set mpd.i end=final;
  where weight=&wt;
  total+1;
  if gender='M' then place+1;
    if final then do;
      call symput('nump',trim(left(place));
      call symput('numstu',trim(left(total));
      call symput('crsname',trim(weight));
   end; run;
proc print data=vasanth;
   var place total weight;
   title "5ml blood sample for &crsname (#wt)";
   footnote "Note: &nump out of &numstu Given";run;
```

Use three ampersands when the value of one macro variable matches the entire name of a second macro variable. Scan sequence:- 1) reference(&&&crsid) 2) 1st scan(&c002) 3) 2nd scan only occurs when && is encountered (Structured Query Language). Placing three ampersands at the start of the original token sequence alters the processing of the tokens and macro triggers.

1.10 The SYMGET Function

Retrieve a macro variable's value during DATA step execution with the SYMGET function. General form of the SYMGET function: SYMGET(macro-variable) *macro-variable* can be specified as a character literal, DATA step character expression. A DATA step variable created by the SYMGET function is a character variable with a length of 200 bytes unless it has been previously defined. The SET statement reads the first observation into the PDV. The SYMGET function retrieves the macro variable value from the symbol table.

1.12 Creating Macro Variables in SQL

The SQL procedure INTO clause can create or update macro variables. ex:- SELECT col1, col2, . . . INTO: mvar1, :mvar2,...FROM table-expression; This form of the INTO clause does not trim leading or trailing blanks. The %LET statement removes leading and trailing blanks from TOTFEE. The INTO clause can create multiple macro variables per row when multiple rows are selected [4]. Ex:- SELECT col1, . .INTO: mvar1 - :mvarn,...FROM table-expression; The INTO clause can create macro variables for an unknown number of rows.1) Run a query to determine the number of rows and create a macro variable NUMROWS to store that number. 2) Run a query using NUMROWS as the suffix of a numbered series of macro variables. The INTO clause can store all unique values of a specified column into a single macro variable. General form of the INTO clause to create a list of unique values in one macro variable: SELECT col1, . . . INTO: mvar SEPARATED BY 'delimiter', . . .FROM table-expression; examples of Creating a Series of Macro Variables by SQL was shown in Table 6.

1.13 The Need for Macro-Level Programming

Suppose you submit a program every day to create registration listings for courses to be held later in the current month. Every Friday you also submit a second program to create a summary of revenue generated so far in the current month.

1) Conditional processing

You can perform conditional execution with %IF-%THEN and %ELSE statements. General form of %IF-%THEN and %ELSE statements: (%IF expression %THEN text; %ELSE text;) *expression* can be any valid macro expression. The %ELSE statement is optional. These macro language statements can only be used inside a macro definition. The text following keywords %THEN and %ELSE can be 1) it is a macro programming statement 2) it can constant text 3) it is an expression 4) it is a macro variable reference 4) it is a macro call. Macro language expressions are similar to DATA step expressions, except the following, which are **not** valid in the macro language: 1 <= &x <= 10, special WHERE operators, IN comparison operator (prior to SAS®9).Use %DO and %END statements following %THEN or %ELSE to generate texts that contain semicolons.

2) Monitoring macro execution

The MLOGIC system option displays macro execution messages in the SAS log, including 1) macro initialization 2) parameter values 3) results of arithmetic and logical operations 4) macro termination. General form of the MLOGIC|NOMLOGIC option: (OPTIONS MLOGIC; OPTIONS NOMLOGIC ;) The default setting is NOMLOGIC.

3) Macro syntax errors

If a macro definition contains macro language syntax errors, error messages are written to the SAS log and a no executable (dummy) macro is created. Store the production SAS programs in external files and copy those files to the input stack with %INCLUDE statements. Macro comparisons are case sensitive. The IN operator is new in SAS®9. The list of values is not enclosed in parentheses.

4) Parameter validation

Use the %INDEX function to check the value of a macro variable against a list of valid values. General form of the %INDEX function:(%INDEX(argument1, argument2)) The %INDEX function, searches *argument1* for the first occurrence of *argument2*, returns an integer representing the position in *argument1* of the first character of *argument2* if there is an exact match, returns 0 if there is no match.*argument1* and *argument2* can be, constant text, macro variable references, macro functions, macro calls.

5) Developing macro-based applications

If a macro-based application generates SAS code, use a four-step development approach. 1) Write and debug the SAS program without any macro coding. 2) Generalize the program by replacing hardcoded constants with macro variable references. Initialize the macro variables with %LET statements.3) Create a macro definition by placing %MACRO and %MEND statements around your program. 4) Convert %LET statements to macro parameters as appropriate. 5) Add macro-level programming statements such as %IF-%THEN.

2. ITERATIVE PROCESSING

2.1 Simple Loops

Many macro applications require iterative processing [5]. The iterative %DO statement can repeatedly, execute macro language statements, and generate SAS code. General form of the iterative %DO statement :(%DO index-variable=start %TO stop <%BY increment>; text %END;) %DO and %END statements are valid only inside a macro definition. *Index-variable* is a macro variable. *Index-variable* is created in the local symbol table if it does not already exist in an existing symbol table. *Start*, *stop*, and *increment* values can be any valid macro expressions that resolve to integers. %BY clause is optional (default *increment* is 1). *Text* can be constant text, macro variables or expressions, macro statements, macro calls. Examples: - Create a numbered series of macro variables. Display each macro variable in the SAS log by repeatedly executing %PUT within a macro loop. Partial SAS log with result of %put _local_; The _**local**_ argument of the **%PUT** statement lists the name and value of macro variables local to the currently executing macro. You can perform conditional iteration

in macros with %DO %WHILE and %DO %UNTIL statements. A %DO %WHILE loop, evaluates *expression* at the top of the loop before the loop executes, executes repetitively while expression is true. General form of the %DO %UNTIL statement: A %DO %UNTIL loop evaluates *expression* at the bottom of the loop after the loop executes, executes repetitively until expression is true, executes at least once. Simple loops and conditional iteration examples was shown in Table 7.

2.2 Global and Local Symbol Tables

The *global symbol table* is, created during the initialization of a SAS session or no interactive execution and initialized with automatic or system-defined macro variables. Finally it is deleted at the end of the session. Macro variables in the global symbol table are available anytime during the session. It can be created by your program or have values that can be changed during the session (except some automatic macro variables).You can create a global macro variable with a 1) %LET statement (used outside a macro definition) 2) DATA step containing a SYMPUT routine 3) SELECT statement containing an INTO clause in PROC SQL 4) %GLOBAL statement. The %GLOBAL statement creates one or more macro variables in the global symbol table and assigns them null values. It can be used inside or outside a macro definition. It has no effect on variables already in the global table.

2.3 The Local Symbol Table

A local symbol table is created when a macro with a parameter list is called or a local macro variable is created during macro execution deleted when the macro finishes execution. A local table is not created unless and until a request is made to create a local variable. Macros that do not create local variables do not have a local table. Local macro variables can be created and initialized at macro invocation (macro parameters) and created during macro execution. It can updated during macro execution and referenced anywhere within the macro. The memory used by a local table can be reused when the table is deleted after macro execution. Therefore use local variables instead of global variables whenever possible. In addition to macro parameters you can create local macro variables with any of the following methods used inside a macro definition1) %LET statement 2) DATA step containing a SYMPUT routine 3) SELECT statement containing an INTO clause in PROC SQL 4) %LOCAL statement. The SYMPUT routine creates local variables only if a local table already exists. The %LOCAL statement can appear only inside a macro definition. It can create one or more macro variables in the local symbol table and assigns them null values. It has no effect on variables already in the local table. Declare the index variable of a macro loop as a local variable to prevent the accidental contamination of macro variables of the same name in the global table or other local tables.

2.4 The SYMPUTX Routine

The optional *scope* argument of the SYMPUTX routine specifies where to store the macro variable CALL SYMPUTX (macro-variable, text, <scope>); G specifies the global symbol table. L specifies the most local of existing symbol tables, which might be the global symbol table if no local symbol table exists.

2.4 Rules for Creating and Updating Variables

When the macro processor receives a request to create or update a macro variable during macro execution, the macro processor follows these rules: check for does MACVAR already exist in the local table. If it is yes Update MACVAR with VALUE in the local table or if it is no check for does MACVAR already exist in the global table. If it is yes update MACVAR with VALUE in the global table. If it is no Create MACVAR and assign it VALUE in the local table.

2.5 Rules for Resolving Variables

To resolve a macro variable reference during macro execution the macro processor follows these rules &MACVAR check for Does MACVAR exist in the local table?. If it is yes retrieve its value from the local table. Or If it is no check for Does MACVAR exist in the global table?. If it is yes Retrieve its value from the global table or If it is no Give the tokens back to the word scanner. Issue warning message in SAS log: Apparent symbolic reference MACVAR not resolved. Total summary of macros concept was shown in the Table 8.

2.6 Multiple Local Tables

Multiple local tables can exist concurrently during macro execution. Call the OUTER macro. When the %LOCAL statement executes, a local table is created. A nested macro call can create its own local symbol table in addition to any other tables that may currently exist. The macro processor resolves a macro variable reference by searching symbol tables in the reverse order in which they were created 1) Current local Table 2) Previously created local Tables 3) global table. When the INNER macro finishes execution, its local table is deleted. Control passes back to the OUTER macro. When the OUTER macro finishes execution, its local table is removed. Only the GLOBAL table remains

3. ADVANTAGES

The macro facility can reduce program development time and maintenance time. If we want to use a program step for executing to execute the same Proc step on multiple data sets in large scale systems. We can accomplish repetitive tasks quickly and efficiently. A macro program can be reused many times. Parameters passed to the macro program customize the results without having to change the code within the macro program. Macros in SAS make a small change in the program and have SAS echo that change thought that program.

4. DISADVANTAGES

SAS code generated by macro techniques does not compile or execute faster than any other SAS code but depends on the efficiency of the underlying SAS code regardless of how the SAS code was generated.

Table 5. Macro variables creation by SYMPUT routine

1)The SYMPUT Routine

```
%let wt=55;
data vasanth;
  set mpd.i end=final;
  where weight=&wt;
  total+1;
  if gender='M' then place+1;
    if final then do;
      call symput('nump',place);
      call symput('DATE',PUT(project_begin_date,
MMDDYYY));
      call symput('crsname',put(fee*(total-place),dollar8.));
    end;
run;
proc print data=vasanth;
   var place total weight;
   title "5ml blood sample for &crsname (#wt)";
   footnote "Note: &nump out of &numstu Given";
run;
%let wt=55;
data vasanth;
  set mpd.i end=final;
  where weight=&wt;
  total+1;
  if gender='M' then place+1;
    if final then do;
      call symput('nump',place);
      call symput('DATE',PUT(project_begin_date,
mmddyy10.));
      call symput('crsname',put(amount*(total-
place),dollar8.));
    end;
run;
proc print data=vasanth;
   var project_begin_date place total weight amount ;
   title "5ml blood sample for &DATE (#wt)";
   footnote "Note: &nump out of &nump Given";
run;
```

2)The SYMPUTX Routine

```
%let start=02FEB2012;
%let stop=02JUN2013;
proc freq data = mpd.i;
  where project_begin_date between "&start"d and "&stop"d;
  table code*location /
              out=stats (rename=(count=ENROLLMENT));
run;
data _null_;
  set stats end=last;
  amount+1;
  full_name+enrollment;
  if last;
  call symputx('amount',amount);
  call symputx('full_name',put(full_name/amount,4.));
run;
%put_user_;
options nolabel;
proc gchart data=stats;
  vbar3d location / patternid=midpoint cframe=w shape=c
               sumvar=enrollment type=mean mean ref=&amount;
  title1 "Report from &start to &stop";
  title2 h=2 f=swiss "Students this period: " c=b "&amount";
  footnote1 h=2 f=swiss "Enrollment average: " c=b "&full_name";
run;
```

Table 6. Creating a series of macro variables by SQL and conditional processing examples

1) Creating a Series of Macro Variables

```
data _null_;
  set mpd.i;
  call
symput('full_name'||left(patientid),trim(full_name));
run;
%let patientid = 1119;
proc print data=mpd.i noobs;
  where location= &l;
  var code patientid location full_name;
  title1 "code for &patientid";
  title2 "patientid by &&full_name&patientid";
run;
```

2) Indirect References to Macro Variables

```
data _null_;
  call symputx(patientid,full_name);
run;
%put _user_;
%let patientid =1119;
proc print data=mpd.i noobs;
  where location= &l;
  var code patientid location full_name;
  title1 "code for &patientid";
  title2 "patientid by &&&patientid";
run;
```

3)The SYMGET Function

```
data _null_;
  set mpd.i;
  call symput('full_name'||left(patientid),
                          trim(full_name)); run;
data SUJECTSNAMES ;
  set mpd.i;
  length SUJECTSNAMES $ 20;
```

4)Create a macro variables in SQL Procedure INTO Clause

A) proc sql noprint;

```
  select sum(amount) format=dollar10.
           into :totfee
    from mpd.i;
quit;
/* %put totfee=&totfee;*/
/*totfee=   $13,778*/
```

B) proc sql noprint;

```
  select distinct location into :Location
       separated by ' '
  from mpd.i;
quit;
/* %put Location =&Location ;*/
/*Location =coll hos hyd nzb wae war*/
```

5)Conditional Processing

%macro *daily*;

```
  proc print data=perm.all noobs n;
    where put(begin_date,monyy7.)="%substr(&sysdate9,3,7)"
         and begin_date ge "&sysdate9"d;
    var student_name student_company paid;
    title "Course Registration as of &sysdate";
   run;
%mend daily;
```

%macro *weekly*;

```
  proc means data=perm.all maxdec=0 sum;
    where put(begin_date,monyy7.)="%substr(&sysdate9,3,7)"
         and begin_date le "&sysdate9"d;
    class begin_date location course_title;
    var fee;
    title "Revenue for Courses as of &sysdate9";
  run;
```

SUJECTSNAMES=symget(('full_name'||left(patientid), trim(full_name)); **run**;	**%mend** weekly; **%macro *reports***; %***daily*** %if &sysday=Friday %then %***wee*** **%mend** reports;

Table 7. Simple loops and conditional iteration examples

1)Simple Loops	**2)Generating Data-Dependent Steps**

A)data _null_;
set perm.schedule end=no_more;
call symputx('teach'||left(_n_),teacher);
if no_more then call symputx('count',_n_);
run;
%macro *putloop*;
%do i=**1** %to &count;
%put TEACH&i is &&teach&i;
%end;
%mend putloop;
%macro B)readraw(first=**1999**,last=**2005**);
%do year=&first %to &last;
data year&year;
infile "raw&year..dat";
input course_code $4.
location $15.
begin_date date9.
teacher $25.;
run;
proc print data=year&year;
title "Scheduled classes for &year";
run;
%end;
%mend readraw;
%***readraw***(first=**2000**,last=**2002**)
	%macro printlib(lib=WORK,obs=**5**);
	%let lib=%upcase(&lib);
	data _null_;
	set sashelp.vstabvw end=final;
	where libname="&lib";
	call symputx('dsn'||left(_n_),memname);
	if final then call symputx('totaldsn',_n_);
	run;
	%do i=**1** %to &totaldsn;
	proc print data=&lib..&&dsn&i(obs=&obs);
	title "&lib..&&dsn&i Data Set"; run;%end;
	%mend printlib;
	%***printlib***(lib=PERM
	3)Conditional Iteration
	%macro values(text,delim=*);
	%let i=1;
	%let value=%scan(&text,&i,&delim);
	%if &value= %then %put Text is blank.;
	%else %do %while (&value ne);
	%put Value &i is: &value;
	%let i=%eval(&i+1);
	%let value=%scan(&text,&i,&delim);
	%end;**%mend** values; %***values***(&sitelist)

Table 8. Summary of macros concept

1) IMPORTANT POINTS IN MACROS

- Macro variables in the global symbol table 65,534 (64K)
- SYMPUT routine creates a macro variable A maximum of 32,767 characters can be assigned to the receiving macro variable
- SYMPUT routine creates a macro variable Any leading or trailing blanks within the DATA step variable's value are stored in the macro variable
- The SYMPUTX routine automatically removes leading and trailing blanks from both arguments
- Values of numeric variables are converted automatically to character using the BEST12. format
- The word scanner recognizes four classes of tokens:-

literal tokens2) Number tokens:- 3) Name tokens: 4) special tokens: -

- Some automatic macro variables have fixed values that are set at SAS invocation: SYSDATE SYSDATE9 SYSSCP SYSVER
- Automatic macro variables have values that change automatically based on submitted SAS statements: SYSLAST SYSPARM
- Macro variables SYSDATE, SYSDATE9, and SYSTIME store character strings, not SAS date or time values.
- A *period* (.) is a special delimiter that ends a macro variable reference and does not appear as text when the macro variable is resolved.

2) OPTIONS IN MACROS*(smmsmmsssso)-10*

1) SOURCE2 option requests inserted SAS statements to appear in the SAS log
2) MCOMPILENOTE=ALL/NONE option issues a note to the SAS log after a macro definition has compiled
3) MEMRPT Specifies that memory usage statistics be displayed on the SAS Log.
4) MERROR: SAS will issue warning if we invoke a macro that SAS didn't find. Presents Warning Messages when there are misspellings or when an undefined macro is called.
5) SERROR: SAS will issue warning if we use a macro variable that SAS can't find.
6) MLOGIC: SAS prints details about the execution of the macros in the log.
7) MPRINT: Displays SAS statements generated by macro execution are traced on the SAS Log for debugging purposes.
8) SYMBOLGEN: SAS prints the value of macro variables in log and also displays text from expanding macro variables to the SAS Log.
9) MSTORED system option enables storage of compiled macros in a permanent SAS library.
10) SASMSTORE= libref , system option designates a permanent library to store compiled macros.
11) STORE option stores the compiled macro in the library indicated by the SASMSTORE= system option.

<table>
<tr><td colspan="2">12) SOURCE option stores the macro source code along with the compiled code
13) OUT= option is omitted, source code is written to the SAS log.</td></tr>
<tr><td>3) MACRO FUNCTIONS(<i><u>ussssenilp)-10</u></i>
%UPCASE(<i>argument</i>);
%SUBSTR(<i>argument</i>, <i>position</i> <,<i>n</i>>);
%SCAN(<i>argument</i>, <i>n</i> < , <i>delimiters</i>>);
%SYSEVALF(<i>expression</i>) ;
%SYSFUNC<i>(SAS function(argument(s))<,format>)</i>;
%EVAL(<i>expression</i>) ;
%BQUOTE(<i>argumen</i>t);
%INDEX(argument1, argument2);
%LENGTH(argument1, argument2);
%PUTN(source ,<i>format</i>);
4) STATEMENT IN MACROS<i><u>(ipcllmsg)-8</u></i>
➢ %INCLUDE Statement; It is a global SAS statement. It is not a macro language statement.
➢ %PUT statement;
➢ %COPY statement;
➢ %LOCAL statement;
➢ %LET statement;
➢ %MACRO and %MEND statements;
➢ %SYMDEL statement ; deletes one or more user-defined macro variables from the global symbol table
➢ %GLOBAL statement;</td><td>Calling a Macro <i>:- %macro-name</i>
6) Create local macro variables with any of the following methods:-<i><u>5</u></i>
Inside a macro definition
1) %LET statement
2) DATA step containing a SYMPUT routine(The SYMPUT routine creates local variables only if a local table already exists)
3) SELECT statement containing an INTO clause inSQL
4) %LOCAL statement.
5) parameters are created in a separate symbol table called a <i>local</i> symbol table
5) Create a global macro variable with any of the following methods:-<i><u>4</u></i>
1) %LET statement (used outside a macro definition)
2) DATA step containing a SYMPUT routine
3) SELECT statement containing an INTO clause in PROC SQL
4) %GLOBAL statement.
CALL SYMPUT (<i>macro-variable</i>,, <i>expression2</i>);
CALL SYMPUTX(<i>macro-variable</i>, <i>expression2</i>);
SYMGET(macro-variable);</td></tr>
</table>

5. CONCLUSION

The macro facility is a tool for customizing SAS and for minimizing the amount of program code you must enter to perform common tasks. Automatic and User-Defined macro variables help to make SAS programs more easily customizable. SAS Macros are used to make redundant tasks much easier to implement, and can be customized to individual tasks. The SAS macro facility continues to have incremental improvements with each new SAS release. When we used these macros concept in proper way in this huge clinical data handle we can reduce program development time and maintenance time to analysis validate and report the clinical data.

CONTACT INFORMATION

For your comments and questions are valued and encouraged. Contact the author with Gmail:- vasanthmph@gmail.com, **phone no:-** 9133080276.

COMPETING INTERESTS

Authors have declared that no competing interests exist.

REFERENCES

1. Oftware.ncsu.edu/sas-macro-language-1-essentials.
2. Available: http://support.sas.com/edu/schedules.
3. SAS Macro Language 1: Essentials, 4, Back links to support.sas.com, 6. vrcgi.com, Programmers '*Refer*, 43, 2012-06-05.
4. Title, SAS SQL 1: Essentials: Course Notes. Authors, Davetta Dunlap, Mark Jordan. Contributor, SAS Institute. Publisher, SAS Institute, 2009. ISBN, 1607642425.
5. Delwich, Lora D, Susan J. Slaughter. 2003. The Little SAS® Book: A Primer, Third Edition. Cary, NC: Sas Institute Inc.doc.

Peer-review history:
The peer review history for this paper can be accessed here:
http://www.sciencedomain.org/review-history.php?iid=250&id=22&aid=2055