Macros - Intermediate

Prepared by



International SAS[®] Training and Consulting

Destiny Corporation 100 Great Meadow Rd Suite 601 Wethersfield, CT 06109-2379 Phone: (860) 721-1684 1-800-7TRAINING Fax: (860) 721-9784 Email: <u>info@destinycorp.com</u> Web: www.destinycorp.com Copyright 2003

Overview

All the macro variables we have seen so far have been referenced using a single ampersand. This is known as direct referencing. These macros have resolved at 'the first attempt' or have not resolved at all.

Program Editor - m5_1	
Command ===> 00001 %let muar = test1; 00002 %put &muar	
00003	~
< · · · · · · · · · · · · · · · · · · ·	> .::

Line	Comment	
00001	The <i>%let</i> statement creates the macro variable <i>mvar</i> with value <i>test1</i> .	
00002	The %put statement calls the macro variable <i>mvar</i> <u>using a single ampersand</u> . The resolved value is written to the Log.	

However, various programming issues often require macro variables to be referenced using multiple ampersands. This is known as indirect referencing. This chapter examines how these macros are identified and processed.

Multiple Ampersands and Staggered Resolution

Consider the following code. What will &&dsn&n resolve to?

Program Editor - m5_2		
Command ===>		
00001 %let dsn=clinics;		
00002 %let clinics5=How did get here;		
00003 %let n=5;		
00004 %let dsn5=surprise;		
00005		
00006 %put &&dsn&n	_	
00007 '	~	
	>	

Multiple ampersands are resolved using the following strategy:

- Start at the left-hand side and group ampersands into two's.
- Each set of double ampersands (&&) is resolved to a single ampersand (&& => &)
- Each & followed by a character string is resolved as a macro variable
- A freestanding string remains unchanged.

Repeat the above steps until all ampersands have been removed.

The above code is processed as follows:

Line	Comment		
00001- 00004	%let statements create macro variables.		
00006	Call on &&dsn&n for	staggered r	esolution:
	First pass:	receives	&&dsn&n
		resolves to	: && → & dsn → dsn &n → 5
	Second pass:	receives	&dsn5
		resolves to	: surprise
🖹 Log - (Untitl	led)		
Command =			~
1 Xlet	t dsn=clinics;		
2 Alet 3 Xlet	t clinics5=How did t n=5:	get nere;	
	dsn5=surprise;		
5 6 Xput surprise	t &&dsn&n		

Now, how will &&&dsn&n resolve?

Program Editor - m5_3	
Command ===>	^
00001 %let dsn=clinics;	
00002 %let clinics5=How did I get here;	
00003 %let n=5;	
00004 %let dsn5=surprise;	
00005	
00006 %put &&&dsn&n	
00007	~
	2
	<u> </u>

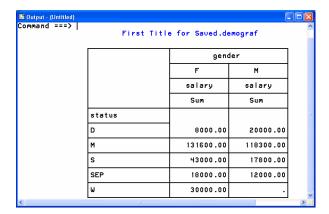
00001- 00004	%let statements create macro variables.		
00006	Call on &&&dsn&n for staggered resolution:		
	First pass:	receives	&&&dsn&n
		resolves to:	&& → & &dsn → clinics &n → 5
	Second pass:	receives	&clinics5
	here	resolves to	How did I get

🗒 Log	g - (Untitle	d)	
7 8 9 10	%let %let	dsn=clinics; clinics5=How did I get here;	
		&&&dsn&n get here	~
<			> .::

Examples of the form (&&&mvar) are rare whereas the generation of multiple macro variables using &&root&suffix to give &root1, etc. are quite common. Any number of ampersands can be used.

As a final example, changing the value of i causes different macro variables to be referenced. This is used to print different titles.

Program Editor - m5_4	×
Command ===>	~
00001 %let title1=First Title for Saved.demograf;	
00002 %let title2=Second Title for Saved.demograf;	
00003 %let title3=Yet Another Title for Saved demograf;	
00004	
00005 Xlet i=1;	
00006	
00007 title "&&title&i";	
00008 proc tabulate data=saved.demograf;	
00009 class status gender;	
00010 var salary;	
00011 table status, gender*salary;	
00012 run;	~
4	5
< III 7	2



Symbol Tables

This module examines the concept of symbol tables in greater depth. It has already been pointed out that macros live in symbol tables. It is critical to know which symbol table receives a macro.

Correct use of macro values requires the programmer to know how to change or avoid changing macro variable values in specific symbol tables.

This module develops the default rules for placing macro variables into global and local symbol tables. However, sometimes the default rules are not what the programmer needs. After examining default placement rules, this module looks at how to assure that macro variables are written to the symbol table of the programmer's choice.

Symbol Table Rules

At SAS invocation time, the Automatic Symbol Table (AST) is built containing most of the automatic macro variables.

An executing macro builds a symbol table local to it. This symbol table is deleted once the macro has ceased execution.

Just like the Data Step, the macro first compiles (when defined) and then executes (when called).

The %macro statement signals the start of the macro definition.

The macro processor takes all the code that follows until it reaches the *%mend* statement, compiling the code and saving it in the Work library.

The default location for a *%macro - %mend* bundle is the *Sasmacr* catalog in the Work library.

The 'compiled' form is a mixture of compiled statements and constant text.

Upon a macro call, the macro processor retrieves the compiled macro code from the Work library and executes it, placing any generated code upon the SAS input stack as text.

During macro execution, the macro processor may pause while text placed upon the input stack is processed. That is to say, once a full step is placed upon the input stack, there will be a pause while it is executed.

However, macro execution does one further thing. It creates a symbol table local to the executing macro for the duration of the execution of that macro. Once the execution of the macro has completed, the local symbol table is deleted.

In addition, we now know another way of defining macro variables – as parameters (either positional or keyword) in the definition of a macro. The macro variables defined in this way are always and only placed in the symbol table local to the macro. So, during the macro execution, there is access to two symbol tables - the local one and the global one. The local table is always searched before the global one. We shall return to this subject in the next module.

In this section, we illustrate the rules for the following:

- Writing to the various symbol tables during the execution of a macro
- Reading from the symbol tables.

There are several factors to keep in mind when working through the following examples.

These include the following:

How is the macro variable defined?

- 2. parameter,
- 3. otherwise....

Where is the macro variable defined?

- 1. In open code
- 2. Within a macro bundle

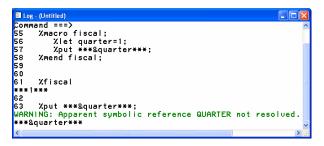
What is the name of the macro variable?

- 1. Same as an already-existing macro variable
- 2. A totally new name

Program Editor - m9_1		×
Command ===>		^
00003 Xmacro fiscal;		
00004 %let quarter=1;		
00005		
00006 %mend fiscal;		-
00007		
00008		
00009 Xfiscal		
00010		
00011 %put ***&quarter***;		_
00012		×
<	>	

Line	Comment
00004	The <i>%let</i> statement defines a macro inside a macro bundle.
	macro bundle.
00005	The %put statement tests the value of &quarter
	inside the macro bundle.
00011	The %put statements tests the value of
	&quarter after the macro bundle has finished
	executing.

The log is displayed below.



Explanation

When *%fiscal* executes a symbol table local to the macro *fiscal* is established. With the local symbol table in place, the *%let* statement does the following:

Checks the most local table for the existence of a macro variable called *quarter*.

If a macro variable called *quarter* is found, its value is overwritten in that symbol table.

If a macro variable called *quarter* is not found, the macro processor checks the next higher table (i.e., here, the global symbol table) for the existence of a macro variable called *quarter*.

If a macro variable called *quarter* is found in the next higher table, its value is overwritten in that table.

If a macro variable called *quarter* is not found, the process of checking the next higher symbol table continues. If a macro variable called *quarter* is never found, a macro variable is created and its value assigned in the most local table (i.e., the first table searched).

So, in this example the macro called *quarter* is established in the table local to *fiscal*.

The %put statements now read from the tables. During the execution of %fiscal, the macro processor will do the following:

- Search the local table for the presence of &quarter, find it and report on the value. The %put statement inside the macro fiscal will therefore write the value of quarter to the log.
- When %fiscal completes the local symbol table is deleted. Therefore *quarter* no longer exists. An error occurs when the %put statement outside the macro fiscal attempts to write a nonexistent macro.

Before Execution	During Execution	After Execution
Global Symbol Table	Global Symbol Table	Global Symbol Table
	Local Symbol Table	

FISCAL	
Quarter = 1	

Example 2

Program Editor - m9_2][X
Command ===>		^
00001 %let month=January;		
00002		
00003 %macro fiscal;		-
00004 %let quarter=1;		
00005		
00006 /put ***&month***;		
00007 Xmend fiscal;		
00008		
00009		
00010 Xfiscal		
00011		
00012 %put ***&quarter***;		
00013 %put ***&month***;		
	>	
		200

Line	Comment
00001	The %let statement defines a macro in
	open code.
00004	The %let statement defines a macro inside
	a macro bundle.
00005, 00006	Two %put statements test the value of
	&quarter and &month inside the macro
	bundle.
00012, 00013	Two %put statements test the value of
	&quarter and &month after the macro
	bundle has finished executing.

🗄 Log - (Untitled)	
Command ===>	^
86 Xfiscal	
1	
January	
87	
88 %put ***&quarter***;	
WARNING: Apparent symbolic reference QUARTER not resolve	ed.
&quarter	
89 %put ***&month***;	
January	~

Explanation

When the first *%let* statement executes in open code it writes to the global symbol table. Therefore, the macro variable *month* is defined in the global symbol table with a value of January.

When %*fiscal* executes another scope is established, which is the symbol table local to the macro *fiscal*. With the local symbol table in place, the second %*let* statement operates as described previously.

In this example then, the macro variable *month* is established in the global symbol table whereas *quarter* is established in the table local to *fiscal*.

The %put statements now read from the tables. During the execution of %fiscal, the macro processor will do the following:

- Search the local table for the presence of &month.
- Upon failing to find it, macro processor will now search the global table, find it, and report the value.
- Search the local table for the presence of &quarter, find it and report on the value.
- When *%fiscal* completes, the local symbol table is deleted, so that the final *%put* statements have only the global table left to search. So, *&month* is found but *&quarter* is not.

Before Execution	During Execution	After Execution
Global Symbol Table	Global Symbol Table	Global Symbol Table
Month = January	Month = January	Month = January
	Local Symbol Table FISCAL	
	Quarter = 1	

Example 3

Program Editor - m9_3	
Command ===>	^
00001 %let month=January;	
00002	=
00003 %macro fiscal;	
00004 %let month=1;	_
00005 %put ***&month***;	
00006 Xmend fiscal;	
00007	
00008	
00009 Xfiscal	
00010	
00011 %put ***&month***;	_
00012	~
	> .::

Comment
The %let statement defines a macro in open
code.
The %let statement defines a macro with the
same name inside a macro bundle.
The %put statement tests the value of &month
inside the macro bundle.
The %put statement tests the value of &month
after the macro bundle has finished executing.

🖺 Log - (Untitled)	
Command ===>	^
1	
108	
109 %put ***&month***:	
1	~
<	2

Explanation

The sequence of logic is the same here as for the previous example:

Macro variable *month* with value *January* is written to the global table.

Upon macro execution, the processor checks the most local environment for a macro variable called *month*. Failing to find it, it

checks the next higher table(s). Upon finding a macro variable *month* in the global symbol table, the old value (*January*) is overwritten with the new value of *1*.

Both *%put* statements report from the global table, the 'inner' one checking the local table first.

Before Execution	During Execution	After Execution
Global Symbol Table	Global Symbol Table	Global Symbol Table
Month = January	Month = 1	Month = 1
	Local Symbol Table FISCAL	

Program Editor - m9_4	
Command ===>	^
00001 %macro stats(region=);	
00002 %put ***®ion***;	
00003 %mend stats;	
00004	
00005 %stats(region=CT)	
00006	
00007 %put ®ion	_
nnnno '	×
<	> .::

Line	Comment
00001	The keyword parameter defines a macro
	associated with the macro bundle.
00002	The %put statement tests the value of ®ion
	inside the macro bundle.
00007	The %put statements tests the value of ®ion
	after the macro bundle has finished executing.
🗒 Log - (Untitle	ed) 📃 🗆 🔀
Command ===>	

117	Xmacro stats(region=);	
118	Xput ***®ion***;	
119	Xmend stats;	
120		
121	%stats(region=CT)	
***CT	***	
122		
123	Xput ®ion	
WARNI	NG: Apparent symbolic reference REGION not resolved.	
®i	on	
<		2
		59

Explanation

Macro variables created as parameters are placed only and always in the table most local to the macro (except for read-write automatic macros, see example 7).

Before Execution	During Execution	After Execution
Global Symbol Table	Global Symbol Table	Global Symbol Table
	Local Symbol Table STATS Region = CT	

Example 5

■ Program Editor - m9_5	
Command ===>	^
00001 %let region=CT;	
00002	
00003 %macro stats(region=);	
00004 %put ***®ion***;	
00005 %mend stats;	
00006	
00007 %stats(region=AZ)	
00008	
00009 %put ***®ion***;	
00010	~
	> .::

Line	Comment
00001	The %let statement defines a macro in open
	code.
00003	The keyword parameter defines a macro of the
	same name associated with the macro bundle.
00004	The %put statement tests the value of ®ion
	inside the macro bundle.
00009	The %put statement tests the value of ®ion
	after the macro bundle has finished executing.

🖹 Log - (Untitled)	
Command ===>	^
130 %stats(region=AZ)	
AZ	
131	
132 %put ***®ion***;	3
CT	~
<	× .:

Explanation

Although the *%let* statement has already established the macro variable *region* in the global environment, the value is not overwritten by the parameter. Macro variables created as parameters are placed only and always in the table most local to the macro (except for read-write automatic macros, see example 7).

This example also illustrates the read sequence: the most local symbol table is always read first with the global and automatic tables last.

Before Execution	During Execution	After Execution
Global Symbol Table	Global Symbol Table	Global Symbol Table
Region = CT	Region = CT	Region = CT
	Local Symbol Table STATS	
	Region = AZ	

Example 6

Program Editor - m9_6	
Command ===>	
)0001 %let sysdate9=15apr2003;	
00002	
0003 %macro test6;	
)0004 %let sysdate9=01jan2003;	
00005	
)0006 Xput &sysdate9	
0007	
00008 Xmend test6;	
00009	
0010 Xtest6	
00011	
00012 Xput &sysdate9	

Line	Comment
00001	The <i>%let</i> statement attempts to define a macro variable using the name of a read-only automatic macro.

🖹 Log - (Untitled)
Command ===>
ERROR: Attempt to assign a value to a read-only symbolic variable
(SYSDATE9).
26 %let sysdate9=15apr2003;
27
28 Xmacro test6;
26 Xlet sysdate9=15apr2003; 27 28 Xmacro test6; 29 Xlet sysdate9=01jan2003; 30 31 Xput &sysdate9 32 33 Xmend test6;
30
31 Xput &sysdate9
32
33 Xmend test6;
34
35 Xtest6
ERROR: Attempt to assign a value to a read-only symbolic variable
(SYSDATE9).
04MAR2003
36
37 Xput &sysdate9
04MAR2003

Explanation

&sysdate9 is set by the system and is read only. The *%let* statement follows the standard write logic and the error message is given.

NOTE: Do not give your variables the same name as read-only automatic variables!

Before Execution	During Execution	After Execution
Automatic Symbol Table	Automatic Symbol Table	Automatic Symbol Table
Sysdate9 = 04MAR2003	Sysdate9 = 04MAR2003	Sysdate9 = 04MAR2003
Global Symbol Table	Global Symbol Table	Global Symbol Table

🔀 Program Editor - m9_7		×
Command ===>		^
00001 %put &syslast		
00002		
00003 %macro test7(syslast=);		
00004 Xput &syslast		
00005 %mend test7;		
00006		
00007 %test7(syslast=whatever)		
00008		
00009 %put &syslast		
00010	1	_
	1	

Line	Comment	
00001	The %put statement tests the value of &syslast, a read-write automatic macro variable, before the bundle executes.	
00003	The keyword parameter defines a macro with the	
	same name as the automatic macro.	
00004	The %put statement tests the value of &syslast inside the macro bundle.	
00009	The %put statement tests the value of &syslast after the macro bundle has finished executing.	
🗒 Log - (Untit		
Command ===>		
213 Xput &syslast		
WORK .DEMOGRAF		
214 215 Xmacro test7(syslast=);		
216 Xput &syslast		
217 Xmend test7;		
in shend tootiy		

217 Xmend test7; 218 219 Xtest7(syslast=whatever) WORK.whatever 220 221 Xput &syslast; WORK.whatever

Explanation

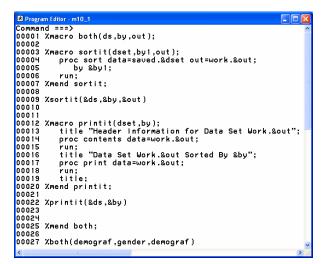
&syslast is set by the system but is a read-write macro variable. The parameter to the bundle has the same name. The read sequence is the same as that discussed: the most local table is read first with the global and automatic tables read last. The difference in this example is that the value of the automatic variable is overwritten.

Before Execution	During Execution	After Execution
Automatic Symbol	Automatic	Automatic
Table	Symbol Table	Symbol Table
Syslast =	Syslast =	Syslast =
work.demograf	work.whatever	work.whatever
Global Symbol Table	Global Symbol Table	Global Symbol Table

Nested Macros

A nested macro refers to a macro invoked within another macro. Nested macros allow increased flexibility and control over program flow.

There are two basic structures for creating nested macros. In the first technique, one macro is completely defined within a second macro.



Such a structure will work, but is inefficient. The inner macros are stored as text instead of being compiled. Each time the outer macro executes the inner macros are compiled.

A more efficient technique is to define each macro separately and then invoke the compiled macros:

🗷 Progra	am Editor - m10_2	×
Commai	nd ===>	^
00001	%macro sortit(dset,by1,out);	
00002		
00003	by &by1	
00004	run;	
00005	Xmend sortit;	=
00006		
00007	%macro printit(dset,by);	
00008	title "Header Information for Data Set Work.&out";	
00009	proc contents data=work.&out	_
00010	run;	
00011		
00012		
00013		
00014	title;	
	Xmend printit;	
00016		
	Xmacro both(ds,by,out);	
00018		
00019		
	Xmend both;	
00021		
	%both(demograf,gender,demograf)	~
<	>	1.1

Because all the macros are defined separately, they all compile. Here are the referencing environments:

When the outer macro, *both*, is executing, the referencing environment is:

• The symbol table local to *both* plus the global table.

When the middle macro, *sortit*, is executing, the referencing environment is:

• The symbol table local to *sortit*, the symbol table local to *both*, plus the global table.

When the inner macro, *printit*, is executing, the referencing environment is:

• The symbol table local to *printit*, the symbol table local to *sortit*, the symbol table local to *both*, plus the global table.

Note that as a macro's execution terminates, the symbol tables are deleted and the referencing environment is reduced again.

Controlling the Placement of Macro Variables

So far, we have seen default rules for placement of macro variables into symbol tables. These rules help the programmer determine the value passed to the program. Often the default rules permit efficient coding possibilities.

In contrast, there are times when the default rules for placement will work against the goals of the programmer.

A macro variable might go global and replace a value needed for reference in later portions of the program.

Conversely, the programmer might wish to confine a macro variable to a local symbol table. This strategy would help minimize the size of the global table over the course of the SAS session. Local tables are destroyed after the macro finishes executing.

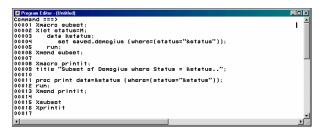
Restricting as many macro variables to the local tables as possible helps increase efficiency in the SAS session.

This section examines situations where the default rules for placement of macro variables are at odds with programming goals.

The programmer seeks to keep a macro variable in the local table where otherwise it would go to the global table. The reverse case can also be true: the macro variable should be written to the global table but would be written locally by default.

Directing a Macro Variable to the Global Table

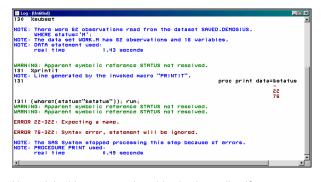
Sometimes a variable will be made local when the programmer seeks to place it in the GST. To allow its value to be used in another step, consider the following code:



Line Comment

00002 By default, the *%let* statement assigns the macro variable *status* to the most local table. The macro variable is unavailable during the execution of *%printit*.

The Log window shows the reason the program fails to print and create the correct title.



How might this programming objective be realized?

🕱 Program Editor - (Untitled)	
Command ===>	-
00001 %global status;	_
20000	
00003 Xmacro subset;	
00004 %let status=S;	
00005 data &status	
00006 set saved.demogius (where=(status="&status"));	
00007 run;	
00008 %mend subset;	
00009	
00010 %macro printit;	
00011 title "Subset of Demogius where Status = &status";	
00012	
00013 proc print data=&status (where=(status="&status"));	
00014 run;	
00015 %mend printit;	
00016	
00017 Xsubset	
00018 %printit	
00019	
00020 %put _global_;	
00021	
00022 title;	
00023	- Contraction of the second se
	 //.

Line	Comment
00001	The <i>%global</i> statement directs the creation of a macro variable (in this case <i>status</i>) in the global
	table. The value of the variable is null.
00004	The <i>%let</i> statement inside <i>%subset</i> follows the default rules. The GST receives the value for <i>status</i> , just as the programmer had intended.

Directing a Macro Variable to the Local Table

The programmer may also want a macro variable to be written to the local symbol table.

This step would assure that a value in the global symbol table would not be overwritten.

Also, the global table would not become cluttered with macro variables used on a one-time basis.

Piper	m Editor - (Untitled)	- 🗆 ×
	%let gender=F;	18
	Xput 1***&gender***;	- 1 H
00003	······	
	Xmacro genderM:	
	Xput 2***&gender***;	
	Xlet gender=M;	
00007	proc means data=saved.demogius mean:	
00008	var salary;	
00009	where gender = "&gender";	
00010	Xput 3***&gender***;	
00011	run;	
00012	Zmend genderM;	
00013		
	Xput 4***&gender***;	_
00015		
	Xmacro genderF;	
00017	Xput 5***&gender***;	
00018	proc means data=saved.demogius mean;	
00019	var salary;	
00020	where gender = "%gender";	
00021	%put 6***&gender***;	
22000	run;	
00023	Xmend genderF;	
00025	%put 7***&gender***;	
	ZaenderM	
	Zgendern Zput 8***&aender***:	
00029	Aput B***@genuer ***;	
	ZoenderF	
	Xput 9***&gender***;	
00032	npar energenaeren;	-
1		
_		· · //

Line	Comment
00001	The % <i>let</i> statement in open code creates the macro variable <i>gender</i> in the GST with value <i>F</i> .
00006	The <i>%let</i> statement replaces the GST value of <i>gender</i> with <i>M</i> . This value is what the programmer wanted for the <i>%genderM</i> macro bundle.
00020	The &gender resolves to <i>M</i> , not <i>F</i> as the programmer had wanted.

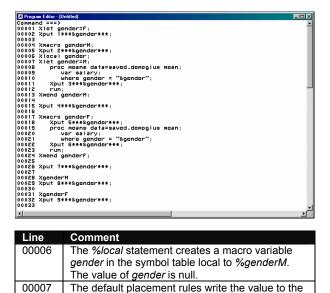
- 0 ×

132	%let gender=F;	8×
133	<pre>%put I ***&gender***;</pre>	
1***	F***	
134		
135	Xmacro genderM;	
136	%put 2****&gender***;	
137	Xlet gender=M;	
138	proc means data=saved.demogius mean;	
139	var salary;	
140	where gender = "%gender";	
141	Xput 3***&gender***;	
142	run;	
143	Xmend genderM;	
144		
145	Xput 4***&gender***;	
4***	F***	
146		
147	Xmacro genderF;	
148	Xput 5***&gender***;	
149	proc means data=saved.demogius mean;	
150	var salary;	
151	where gender = "%gender";	100
152	Xput 6***&gender***;	100
153	run;	
154	Xmend genderF;	100
155		
156 7***	Aput 7***&gender***;	
157	***	
		100
2***	KgenderM	100
	r # # #	100
3***	M***	-1
•		•
		_ ////



The default rules have worked against the goals of the program.

The global macro variable gender was given a new value. To assure that the value of a macro variable is written to the local table, include syntax as shown.



Overview

This module looks at how the Data Step can create macro variables, often out of a data set variable. The *Call Symput* routine is used to create a macro variable from within the data step.

the GST remains F.

local symbol table. The macro variable gender in

In addition, this module looks at the *Symget* function for bringing a macro variable into the Data Step.

Call Symput Routine

Of all the macro syntax options, Call Symput is one of the most important and most useful. With Call Symput, we have another way of creating a macro variable and giving it a value, this time during data step execution.

The syntax for call symput is:

Call symput (macro variable name, macro value);

Note:

- Call Symput is used exclusively in the Data Step.
- Call symput works at data step execution time (not compile time).

• The macro variable is not available until the data step creating it completes execution (i.e., until after the *run* statement).

Another issue to be aware is if the arguments are in quotes or not. Briefly, any argument enclosed in quotes is taken as the literal name or value. Any argument not in quotes is treated as a variable and the argument is the value of that variable.

Several examples follow illustrating this last point.

Example 1

Program Editor - Utivitie()
Image: Compared Emiliary Compared Emiliory Compared Emiliary Compared Emiliary Compared Emiliary C

Line Comment

00002 Both arguments are in quotes. Argument one creates a macro variable *newmacro* with the value *hello*.

D Burgers Addres (Unnum) I Command ==>> >>>>>>>>>>>>>>>>>>>>>>>>>>>>		
Line	Comment	
00003	First argument unquoted and second argument quoted. Argument one references the data set variable 'x', which has the value of 'mvar'. Argument one creates a macro variable <i>mvar</i> with the value <i>greetings</i> .	

Example 3



Line Comment 00004 Neither argument is in quotes. Arguments one and two reference data set variables 'x' and 'y' respectively. Both data set variables have values. Argument one creates a macro variable *mvar* with the value *Monday*.

Example 4

Program Editor -1 Command ===> 00001 data 00002 y = 00003 cal 00004 run; 00005 %put * 00006 %put *	null_: ≛ "Monday"; symput("x",y);
Line	Comment
00003	Only the first argument is in quotes. Argument two references a data set variable 'y' with value

'Monday'. Argument one creates a macro variable 'x' with the value *Monday*.

These are the four variations in the Call Symput routine.

Now note the timing of creating the macro variable with Call Symput.

😫 Program Editor - (Untitled)	
Command ===>	
00001 data _null_;	
00002 z1 = "January";	
00003 call symput("month",z1);	
00004 %put 1***&month***;	
00005 run;	
00006	
00007 %put 2***&month***;	
00008	•
	•

Line	Comment
00004	The %put inside the Data Step will not resolve
	since the macro variable has not been created.
00007	The %put after the Data Step will resolve.

Using the Call Symput Routine

Application of the Call Symput routine displays its programming power.

To illustrate, let's create macro variables and values for frequencies of distinct values in a data set.



is determined by the value in the status variable.
The macro value is determined by the value in the
count variable.

Analyze the following additional programs to see the power of the Call Symput routine.

Example 1:

Create output as shown. It requires both a string and numeric value reflecting the average salary of the data set.

Format the string to use in a title statement.

Use the numeric value in a Where statement to subset the top half earners into a new data set.

🔀 Program Editor - (Untitled)	
Command ===>	
00001 proc means data=saved.demogius mean noprint;	
00002 var salary;	
00003 output out=work.summary (keep=meansal) mean=meansal;	
00004 run;	
00005	
00006 data _null_;	
00007 set work.summary;	
00008 call symput('meansal',meansal);	
00009 call symput('avgsal',put(meansal,dollar10.2));	
00010 run;	
00011	
00012 title "Average Salary is &avgsal";	
00013 title2 'Subset of Population with greater than average income.';	
00014	
00015 data work.tophalf;	
00016 set saved.demogius (where=(salary gt &meansal)	
00017 keep= name salary staffno);	
00018 run;	
00019	
00020 proc print data=work.tophalf;	
00021 var name staffno salary;	
00022 format salary dollar12.2;	
00023 run;	
00024	-
	• •

Line	Comment
00001-00004	Proc Means is used to calculate the mean of the variable salary. An output data set is created (work.summary) with a variable (meansal) that is the mean of salary.
00006-00010	The data step is used to read in the data set created by Proc Means. Call symput is used to create macro variables meansal and avgsal.
00012	Macro variable avgsal is used in Title statement.
00016	Macro variable meansal is used in where clause to subset data.

	ut - (Untitled)			
	ge Salary is \$21,382.19.			
Subse	t of Population with gre	ater than a	verage income.	
Obs	NAME	STAFFNO	SALARY	
ubs	NHME	STHEFNU	SHLHRY	
1	Julia Pendlebury	0052	\$25.410.00	
2	Helen Cinderford.	0094	\$25,200.00	
3	Mark Chapel	0019	\$30.360.00	
- 4	Julio Jennings	0084	\$25.700.00	
5	David Dulley	0066	\$29,700.00	
6	Dawn Duvet	0085	\$28,975.00	
7	Brian Ellows	0089	\$24,500.00	
8	Deborah Bolling	0071	\$37,600.00	
9	Terence Lafter	00102	\$35,200.00	
10	Alan Postlethwaite	0034	\$33,520.00	
11	Celia Freebody	0055	\$23,023.00	
12	Agnes Fortesque-Smyt	0051	\$25,410.00	
13	Susan McGrath	0017	\$53,970.00	
14	David C. Andersen	0079	\$22,950.00	
15	Elaine M. Allen	0060	\$26,050.00	
16	Carl M. Fischer	0031	\$47,520.00	
17	Deborah Randolph	0077	\$27,950.00	
18	Lois Barr	0065	\$35,650.00	
19	Mark Mancini	00101	\$29,200.00	
20	Pamela Mignt	0090	\$29,650.00	-
4				

(and so it

Line	Comment
00011	Each loop of the incoming SAS data set
	work stats creates a new macro. The macro name

Example 2:

Create macro variables to show average age by gender values. Show one average age for males, another for females.

🕱 Program Editor - (Untitled)	- O X
Command ===>	
00001 proc sort data=saved.demogius	-
00002 out=work.demoglus;	
00003 by gender;	
00004 run;	
00005	
00006 proc means data=work.demogius mean noprint;	
00007 var age;	
00008 by gender;	
00009 output out=work.agestats (keep=gender mean	
where=(gender ne " "))	
00011 mean=mean;	
00012 run;	
00013	
00014 data_null_;	
00015 set work.agestats;	
00016 call symput(gender,left(mean)); 00017 run:	
00017 run; 00018	
00019 %put ***&f***:	
00013 Aput ***&r***; 00020 Xput ***&m***;	
00021 / / / / / / / / / / / / / / / / / / /	
	<u> </u>
	▶ //.

Line	Comment
00001-00003	The <i>d</i> ata set saved.demogius is sorted by gender.
00006-00012	Proc Means calculates the mean age for each gender. An output data set is created (work.agestats) with two observations; the first with the mean age for Females and the second with the mean age for Males.
00014- 00017	The data step reads in the data set containing the mean age values. Call symput is used to create a macro variable. The name of the macro variable is given by the value of the gender variable. The value of the macro variable is given by the value of the macro variable is given by the value of the variable mean. Because there are two observations in work agestats, two macro variables are created, one for Females and the second for Males.
00019-	%put is used to write the value of the
00020	macro variables to the Log

🗮 Log - (Untitled)	
Command ===>	^
60 61 %put ***&f***;	
61 %put ***&f***;	
***37.2 ***	
62 %put ***&m***;	
***38.78125 ***	
	~
2	

Example 3:

The idea is to archive observations more than 30 days old. This example could be adapted to any dynamic file (say one under FSEDIT control) where it was important to move old observations into some archive or backup file.

Program Editor - (Untitled)	- 🗆 🗡
Command ===>	
00001 data work.new;	
00002 input @1 date date9.	- 1 H
00003 reading1	
00004 reading2;	
00005 datalines;	
00006 01sep1996 102 150	
00007 19aug1996 98 143	
00008 05may1997 120 34	
00009 21aug1998 33 66	
00010 11may1996 13 67	
00011 run;	
00012	
00013 data work.oldrecs;	
00014 set work.new	
00015 if today() - date > 30 then output work.oldrecs;	
00016 run;	
00017	
00018 data _null_;	
00019 set work.oldrecs nobs=numobs;	
00020 call symput('append',left(numobs));	
00021 stop;	
00022 run;	
00023	
00024 %macro archive;	
00025 %if &append ne 0 %then %do;	
00026 proc append base=saved.arch data=work.oldrecs;	
00027 run;	
00028 Xend;	
00029 %else %put No archiving required;	
00030 Xmend archive;	
00031	
00032 Xarchive;	
00033	-
	•

Line	Comment
00001-00011	Create original data set.
00013-00016	Create work.oldrecs with observations

	more than 30 days old.	
00018- 00022	Use data set option NOBS to create variable NUMOBS. The value of NUMOE is the number of observations in data set work.oldrecs. Call symput is used to create a macro variable (append) with the number of observations in the data set work.oldrecs.	
00024- 00030	Macro bundle archive is created. If the value of the append macro is not zero (therefore there are observations in the work.oldrecs data set) the Proc Append code is generated and executed. Otherwise the Proc Append code is not generated and executed.	

Example 4:

Subset a data set so that each distinct value of a variable is written to a new data set bearing the value name.

📩 Program Editor - (Untitled)	_ 🗆 🗡
Command ===>	
00001 %macro split (inputds, byvar, prefix);	
20000	
00003 %* Create a list of distinct values of the byvar;	
00004 proc freq data=&inputds	
00005 tables &byvar / noprint out=work.numbys (keep=&byvar);	
00006 run;	
00007	
00008 %* Create a series of macro variables with count values;	
00009 data _null_;	
00010 set work.numbys end=x;	
00011 call symput('mvar' left(put(_n_,2.)),	
00012 left(put(&byvar,3.)));	
00013 if × then call symput('numobs',put(_n_,2.));	
00014 run;	
00015	
00016 %* Use the Data Step to name new data sets and subset;	
00017 data	
00018 %do i=1 %to &numobs	
00019 &prefix&&mvar&i	
00020 %end;	
00021 ; 00022 set &inputds	
00022 set &inputds 00023 %let else=:	
00023 %IET BISE=; 00024 %do i = 1 %to &numobs	
00025 & &else if &byvar=&&mvar&i then output &prefix&&mvar&i	
00025 Xist else=else;	
00026 XIEL EISE-EISE; 00027 Xend:	
00028 run:	
00030 Xmend split;	
00032 %split (saved.bp1, patient, ds)	
	<u> </u>

Line	Comment
00004-00006	Use Proc Freq to create a data set
	containing all unique values of variable defined by &byvar.
00009-00014	Create a series of macro variables called
	mvar1, mvar2, etc. The values of these
	macros are given by the value of the
	macro &byvar. The macro variable
	Numobs that is created has a value equal
	to the number of unique values of &byvar.
00017-	Use a %do loop to create the data set
00021	names in the data statement.
00023	Create macro variable else and set its
	value to null
00024-	Use a %do loop to generate a series of if,
00028	else if statements. Based on the value of
	&byvar the observation is written to the
	appropriate data set.

Example 5:

Let us assume that the programmer wants a title statement to spell out the results determined from data set summarization.

For example, if there are more males in the population than females, the title should read, "Males outnumber Females". If the reverse is true, the title should read, "Females outnumber Males". How can the programmer tell SAS which title to select?

🗷 Prog	am Editor - m12_11	
Commai	nd ===>	^
00001	Xmacro males;	
00002	title 'Males outnumber Females';	
00003	title2 "&m to &f":	
00004	%let order = descending;	
	Xmend males;	
00006		
	%macro females;	11 () () () () () () () () () (
00008		
	title2 "&f to &m";	
00010		
	%mend females;	
00012		
	%macro size;	_
00014	%global order;	
00015	options nodate nonumber; title:	
00015	title;	
00018	data _null_;	
00019	set saved.demogius (keep = gender	
00020	where =(gender ne " "))	
00021	end=end:	
00022	retain f m :	
00023	if gender = "F" then f+1;	
00024	else m+1;	
00025	,	
00026	if end then do:	
00027	call symput(gender, compress(put(f, 3.)));	
00028	call symput(gender, compress(put(m, 3.)));	
00029	end;	
00030	run;	~
<		> .:

🗷 Prog	ram Editor - m12_11	
Comma	nd ===>	^
00032	%if &m gt &f %then %males;	
00033	Zelse Zfemales;	
00034		
00035	proc sort data=saved.demogius	
00036	out=work.demogius;	
00037	by ℴ gender;	
00038	run;	-
00039		
00040	proc print data=work.demogius;	
00041	run;	
00042		
	Xmend size;	
00044		
00045	Xsize	~
<		> .:

Line	Comment
00001-00005	The %macro - %mend bundle males supplies the title to use if males outnumber females. If so, the data set is sorted in descending order to show the males at the top. The second title statement will show the actual values (derived below).
00007-00011	The %macro - %mend bundle females supplies the title to use if females outnumber males. If so, the data set is sorted in ascending order to show the females at the top. The second title statement will show the actual values (derived below).
00014	The %global statement creates a slot for the order macro. The value is supplied when the macro %male or %female is invoked conditionally below.
00016	Because titles are used, all previously existing titles are removed.
00022-00025	The sum statements F+1 and M+1 count the number of observations read into the Data Step.
00027-00029	The <i>Call Symput</i> routines create macro variables <i>F</i> and <i>M</i> at then end of the data step execution.
00033-00035	The % <i>if</i> % <i>then</i> condition invokes either % <i>males</i> or % <i>females</i> based on values of the macro variables. At this point the title statements are ready and the Global Symbol Table holds a value of either <i>descending</i> or < <i>null</i> > for the macro variable <i>order</i> .
00039	The <i>order</i> macro variable resolves from the Global Symbol Table.

Frequently Asked Questions About Call Symput

(a) To which symbol table does the macro variable belong?

Most macro variables created by the use of the Call Symput routine are placed in the global table. However, the variable will be placed in the nearest symbol table in the current referencing environment of the data step, providing that symbol table is not empty. If it is empty, it will be placed in the next higher symbol table, providing it is not empty and so on.

(b) When is the macro variable available for use?

The most common mistake with the use of the Call Symput routine is to forget that the macro variable is *only available after the data step completes execution!*

(c) How does the Call Symput format character values?

The default format is \$w. where w is the width of the variable. Hence trailing blanks may also be transferred. Avoid this by the use of the trim function with the second argument:

call symput('mvar1', trim(datavar));

(d) How does Call SYMPUT format numeric values?

The default format is BEST12. with the number being right justified. You may need to use the left and put functions to get your desired result:

call symput('mvar1', left(put(datavar, 3.));

Symget Function

The purpose of the *Symget* function is to pass values from a symbol table in the current referencing environment to the data step variable in the program data vector:

Symbol Table

mvar1	Value1
mvar2	Value2

Program Data Vector

T TOGTATH D				
VarA	VarB	VarC	VarD	VarE
	Value2			

Note several important features about the Symget function:

- It is used exclusively in the data step.
- It works at data step execution time (not compile time).
- Symget always retrieves a character variable.
- The data step variable created by symget is a character string with a default length of 200.

The syntax for the symget function is;

Variable=symget(argument);

Three types of arguments are accepted by symget:

- Name of a macro variable in single quotes.
- A data step character variable whose value is the name of a macro variable.
- A data step character expression.

Use of the Symget function is quite versatile.

🗷 Progra	1m Editor - m12_12	×
Commai	nd ===>	1
00003	%let value=M;	
00004	Xlet title="Mr.";	
00005		
00006	data work.demogius;	
00007	set saved.demogius (where=(status=symget('value')));	
80000	if gender = symget('value') then title = symget('title');	
00009	run;	
00010		
00011	proc print data=work.demogius;	
00012	var status gender title;	
00013	run;	6
1		

Line	Comment
00007	The Symget function used in a where option
00008	The Symget function used in a conditional assignment

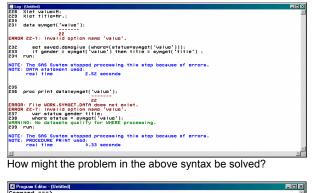
In contrast, consider the following attempt to use the Symget function.

Warning: This program contains syntax errors!

🕱 Program Editor - (Untitled)	- D ×
Command ===>	-
00001 %let value=M;	
00002 Xlet title=Mr.;	
00003	
00004 data sympet('value');	
00005 set saved.demogius (where=(status=sympet('value')));	
00006 if gender = symget('value') then title = symget('title') ;	
00007 run;	
00008	
00009 proc print data=sympet('value'):	
00010 var status gender title:	
00011 where status = symget('value');	
00012 run:	
00013	-
	•

Line	Comment
00004	The <i>Symget</i> function is incorrectly used to name the data set (which is done at compile time).
00009, 00011	The <i>Symget</i> function incorrectly used outside the Data Step.

The Log window shows several problems.





Line	Comment
00004, 00009, 00011	Unlike the <i>Symget</i> function, invoking a macro variable value using <i>&value</i> does not depend on the Data Step nor on execute time.

Dynamic Change to the Macro Variable Name

The most common use of the *Symget* function is to make dynamic changes to the macro variable name.

🗷 Program	Editor - m12_15	×
Command	===>	^
00001 2	(let mvar1 = first;	
	(let mvar2 = second;	Ξ
00003 2	(let mvar3 = third;	
00004	·	
00005 c	lata work.newds:	
00006	do i = 1 to 3;	
00007	datavar = symget('mvar' put(i,1.));	
00008	output;	
00009		
00010 r	run;	
00011		
00012 g	proc print data=work.newds;	
00013		~
<		

Line	Comment
00001-	The three %/et statement created sequentially
00003	named macro variables.
00006-	The doto; end; syntax provides the
00009	numbered portion of the macro variable
	names.

The distinction between the two syntax options of the Symget function is in the use of quotes!

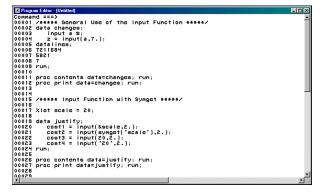
Placing the first argument in quotes indicates a literal; the string inside the quotes is the macro variable name.

When the first argument is not in quotes, it indicates a data set variable; the value of the data set variable supplies the name of the macro variable.

Creation of Numeric Variables

Another point to note is that the Symget function creates a macro that is a character string. What are the implications of this?

Consider the following syntax that appears to be quite similar. The syntax uses the Input function to convert a character variable to a numeric variable.



Line	Comment
00020-	The derived variables cost1 to cost4 seem to
00023	follow the same syntax. They do not. Explore the differences in the Input statements
	below.

Why these differences in results?

Remember that the Input function uses a character value as the first argument. Both Symget and '20' provide the correct variable type, and thus the correct justification (to the left). These two examples work as predicted.

In contrast, *&scale* and *20* are numeric values. The Log window states that numeric values have been converted to character. However, the automatic conversion used the Best12. format. As a result, the numbers were converted to strings as "20". What part of the string contributed to the values of cost1 and cost3? Answer: the first two (blank) spaces. As a result, the values for cost1 and cost3 are missing.

Remember the result from the Symget function is a character string of default length 200. It will be left justified and therefore works as a literal string.