Macros - Basic

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: info@destinycorp.com
Web: www.destinycorp.com

Introduction to Macros in SAS

Why learn macros? In brief, they provide the only solution to certain programming goals. Further, they greatly reduce the amount of work required to achieve some goals. These facts will become clear as the curriculum continues.

Macros are used to:

- Change code easily, making code more flexible.
- Conditionally generate and execute code.
- Generate repetitive code.
- Pass variable information across Step boundaries.

A macro is stored text that can be referenced (or "called") using another, shorter name. In effect the macro environment allows the programmer to take one set of text and convert it to another text string according to well defined rules.

The macro environment allows a programmer to write customized syntax, which can be reused whole or modified with different parameters to produce varied output. Macros allow the programmer to set different values for data sets, variables, statistics, etc. each time the syntax runs. Rather than sift through hundreds of lines of code to find and change select references, the macro is coded with parameters, which need to be changed once each time the syntax runs.

As one example of where macros might be used consider the situation where you wish to include a summary statistic, such as average salary, in a title statement. The average salary could be stored in a data set, but how would the title pick up the value? Alternatively, the programmer could 'hard code' the value into the title. But what happens if the data set values for salary change? The programmer would have to be involved interactively every time the program runs. Hardly satisfactory!

In contrast, the value of average salary could be saved as a macro variable. The title statement can then reference the macro variable which would resolve to the value of average salary.

Macro Variables

In this module, we discuss two special characters:

- ampersand (&)
- percent (%).

Both characters have special functionality within the macro environment.

Naming conventions are to be followed for macro names, macro bundle names, and parameter names. Although these terms are still undefined, bear in mind the following when assigning names:

- Any valid SAS name according to the Operating System and SAS version is generally acceptable
- SAS has reserved some names for other use. Do not use the following words when programming in macros:

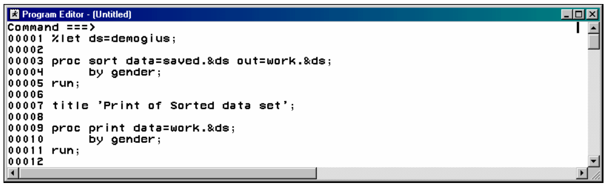
ABEND	ABORT	ACT	ACTIVATE	BQUOTE
BY	CLEAR	CLOSE	CMS	COMANDR
COPY	DEACT	DEL	DELETE	DISPLAY
DMIDSPLY	DMISPLIT	DO	EDIT	ELSE
END	EVAL	FILE	GLOBAL	GO
GOTO	IF	INC	INCLUDE	INDEX
INFILE	INPUT	KEYDEF	LENGTH	LET
LIST	LISTM	LOCAL	MACRO	MEND
METASYM	NRBQUOTE	NRQUOTE	NRSTR	ON
OPEN	PAUSE	PUT	QSCAN	QSUBSTR
QUOTE	QUPCASE	RESOLVE	RETURN	RUN
SAVE	SCAN	STOP	STR	SUBSTR
SUPERQ	SYSEXEC	SYSGET	SYSRPUT	THEN
TO	TSO	UNQUOTE	UNSTR	UNTIL
UPCASE	WHILE	WINDOW		

An attempt to call a macro by a reserved name will result in a warning message. The macro will be neither compiled nor available for use.

%Let Statements

A %*let* statement is one method used to create a macro variable. Within a %*let* statement the name of the macro variable is to the left of the equal sign. Everything to the right of the equal sign up to but not including the semi-colon, except leading and trailing blanks, is the value of the macro variable. We will discuss the %*let* statement in greater detail in a subsequent chapter.

We wish to consider the following example to demonstrate use of the ampersand and percent symbols.

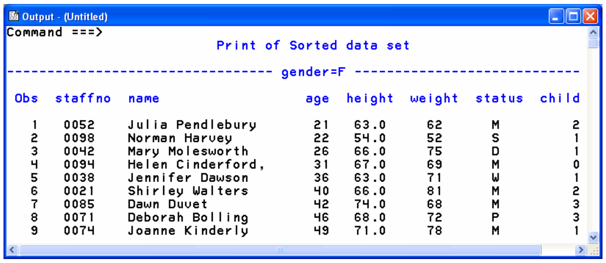


```
Command ===>
00001 %let ds=demogius;
00002
00003 proc sort data=saved.&ds out=work.&ds;
00004       by gender;
00005 run;
00006
00007 title 'Print of Sorted data set';
00008
00009 proc print data=work.&ds;
00010       by gender;
00011 run;
00012
```

Line	Comment
00001	Define a macro variable called *ds* with its value. The presence of a % followed by a non-blank character triggers the macro facility. A *%let* tells the macro facility that a macro variable is to be defined. The *%let* statement can appear anywhere within a SAS program to define one macro variable at a time.
00003 00009	Use *&ds* (ampersand+macro variable name) to invoke the value.

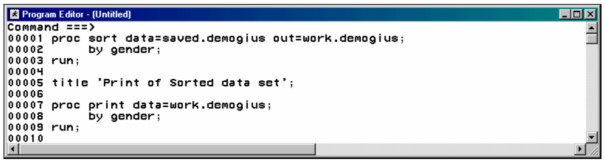
When the SAS Supervisor sees an ampersand followed by a non-blank character, the macro facility is activated. In turn, the macro facility determines the value for the macro variable and passes the value back on to the input stack.

A partial listing of the output is displayed below.



```
Command ===>
                         Print of Sorted data set

-------------------------------- gender=F ------------------------------

 Obs   staffno   name                    age   height   weight   status   child

   1    0052     Julia Pendlebury         21    63.0      62       M          2
   2    0098     Norman Harvey            22    54.0      52       S          1
   3    0042     Mary Molesworth          26    66.0      75       D          1
   4    0094     Helen Cinderford,        31    67.0      69       M          0
   5    0038     Jennifer Dawson          36    63.0      71       W          1
   6    0021     Shirley Walters          40    66.0      81       M          2
   7    0085     Dawn Duvet               42    74.0      68       M          3
   8    0071     Deborah Bolling          46    68.0      72       P          3
   9    0074     Joanne Kinderly          49    71.0      78       M          1
```

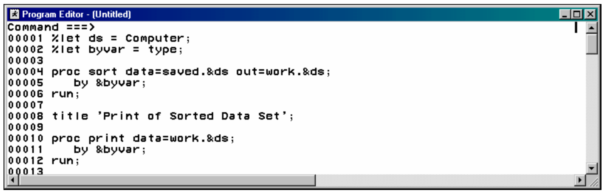
The code executes as if the programmer had submitted the following:



```
Command ===>
00001 proc sort data=saved.demogius out=work.demogius;
00002      by gender;
00003 run;
00004
00005 title 'Print of Sorted data set';
00006
00007 proc print data=work.demogius;
00008      by gender;
00009 run;
00010
```

What is the advantage of using the macro *%let* statement over direct coding? Consider how easily changes can be made should a new data set be used. It would also be easy to use a different By variable in the By statement.

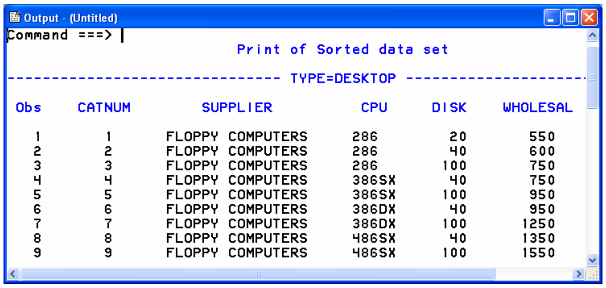
The following program shows easy code changes with macros.



```
Command ===>
00001 %let ds = Computer;
00002 %let byvar = type;
00003
00004 proc sort data=saved.&ds out=work.&ds;
00005    by &byvar;
00006 run;
00007
00008 title 'Print of Sorted Data Set';
00009
00010 proc print data=work.&ds;
00011    by &byvar;
00012 run;
00013
```

Line	Comment
00001 00002	The *%let* statements now refer to an entirely different data set and By variable.

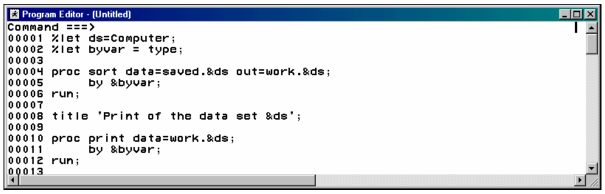
A partial listing of the new output is displayed below.



```
Command ===>
                         Print of Sorted data set

-------------------------------- TYPE=DESKTOP ---------------------

 Obs    CATNUM         SUPPLIER          CPU      DISK     WHOLESAL

   1       1       FLOPPY COMPUTERS     286        20        550
   2       2       FLOPPY COMPUTERS     286        40        600
   3       3       FLOPPY COMPUTERS     286       100        750
   4       4       FLOPPY COMPUTERS     386SX      40        750
   5       5       FLOPPY COMPUTERS     386SX     100        950
   6       6       FLOPPY COMPUTERS     386DX      40        950
   7       7       FLOPPY COMPUTERS     386DX     100       1250
   8       8       FLOPPY COMPUTERS     486SX      40       1350
   9       9       FLOPPY COMPUTERS     486SX     100       1550
```

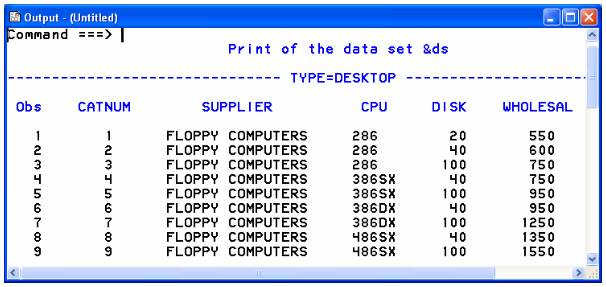
Macro Variables Within Title or Footnote Statements

Let us now enclose the title statement within single quotes. This will cause the macro within the title to not resolve properly.



```
Command ===>
00001 %let ds=Computer;
00002 %let byvar = type;
00003
00004 proc sort data=saved.&ds out=work.&ds;
00005      by &byvar;
00006 run;
00007
00008 title 'Print of the data set &ds';
00009
00010 proc print data=work.&ds;
00011      by &byvar;
00012 run;
00013
```

Line	Comment
00008	The title statement includes a call to macro *ds.*

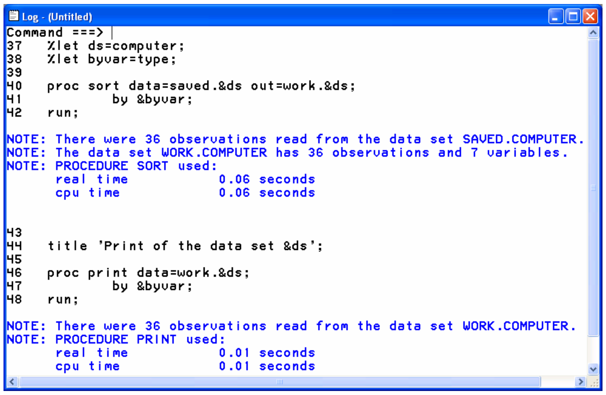


```
Command ===>
                         Print of the data set &ds

-------------------------------- TYPE=DESKTOP ---------------------

 Obs    CATNUM         SUPPLIER          CPU      DISK     WHOLESAL

   1       1       FLOPPY COMPUTERS     286        20        550
   2       2       FLOPPY COMPUTERS     286        40        600
   3       3       FLOPPY COMPUTERS     286       100        750
   4       4       FLOPPY COMPUTERS     386SX      40        750
   5       5       FLOPPY COMPUTERS     386SX     100        950
   6       6       FLOPPY COMPUTERS     386DX      40        950
   7       7       FLOPPY COMPUTERS     386DX     100       1250
   8       8       FLOPPY COMPUTERS     486SX      40       1350
   9       9       FLOPPY COMPUTERS     486SX     100       1550
```

The Output shows that the title statement still reads 'Print of the data set &ds'.

In other words, within the title statement *&ds* did not resolve to 'Computer' as expected. Elsewhere within the program *&ds* resolved appropriately.

A check of the Log shows no syntax errors.



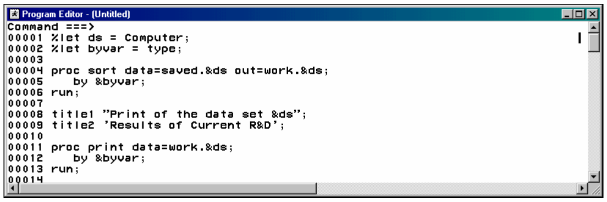
```
Command ===>
37    %let ds=computer;
38    %let byvar=type;
39
40    proc sort data=saved.&ds out=work.&ds;
41           by &byvar;
42    run;

NOTE: There were 36 observations read from the data set SAVED.COMPUTER.
NOTE: The data set WORK.COMPUTER has 36 observations and 7 variables.
NOTE: PROCEDURE SORT used:
      real time           0.06 seconds
      cpu time            0.06 seconds


43
44    title 'Print of the data set &ds';
45
46    proc print data=work.&ds;
47           by &byvar;
48    run;

NOTE: There were 36 observations read from the data set WORK.COMPUTER.
NOTE: PROCEDURE PRINT used:
      real time           0.01 seconds
      cpu time            0.01 seconds
```

This is one of the few times SAS distinguishes between single quotes (as used in the above program) and double quotes.

In brief:

- Use double quotes (or no quotes for titles and footnotes) if you want the macro to resolve.
- To let the ampersand remain – as with R&D – use single quotes

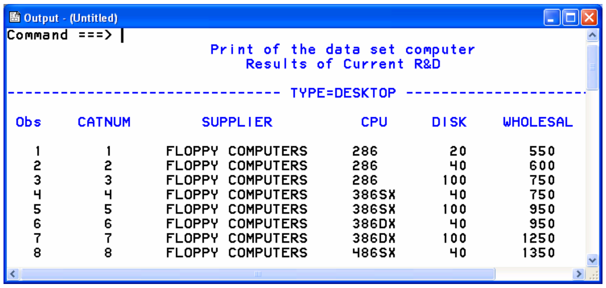
Refer now to the program with both options used properly.



```
Command ===>
00001 %let ds = Computer;
00002 %let byvar = type;
00003
00004 proc sort data=saved.&ds out=work.&ds;
00005    by &byvar;
00006 run;
00007
00008 title1 "Print of the data set &ds";
00009 title2 'Results of Current R&D';
00010
00011 proc print data=work.&ds;
00012    by &byvar;
00013 run;
00014
```

Line	Comment
00008	The title1 statement includes a call to macro *ds*. The macro will be resolved because the title statement uses <u>double quotes</u>. The same usage for double quotes applies to footnote statements.
00009	The title2 statement includes &D. SAS will not attempt to resolve it as a macro D because the title statement is in <u>single</u>

	quotes. The same usage for single quotes applies to footnote statements.



```
Command ===>
                     Print of the data set computer
                        Results of Current R&D

------------------------------ TYPE=DESKTOP --------------------

 Obs     CATNUM          SUPPLIER         CPU      DISK     WHOLESAL

   1        1     FLOPPY COMPUTERS     286        20        550
   2        2     FLOPPY COMPUTERS     286        40        600
   3        3     FLOPPY COMPUTERS     286       100        750
   4        4     FLOPPY COMPUTERS     386SX      40        750
   5        5     FLOPPY COMPUTERS     386SX     100        950
   6        6     FLOPPY COMPUTERS     386DX      40        950
   7        7     FLOPPY COMPUTERS     386DX     100       1250
   8        8     FLOPPY COMPUTERS     486SX      40       1350
```

What would happen if we placed the title2 statement in double quotes? How would SAS handle the request to resolve the macro *D*? The answer depends on a few different parameters.

- First, if previous coding had already defined a macro variable *D*, SAS would place its value into the title string.
- Second, if no macro variable *D* had been defined, SAS would leave the title string unresolved – Demonstration of Recent R&D – while giving a Warning message in the log.

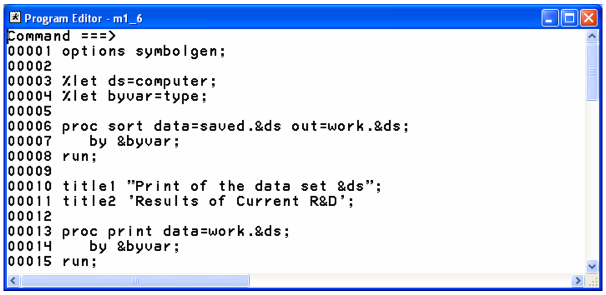
If the macro facility fails to find the current value for a macro variable, the following message appears on the SAS Log:

Warning: Apparent symbolic reference is not resolved.

SAS refers to macro variables as 'symbolics'. The Warning message states that the macro has been resolved.

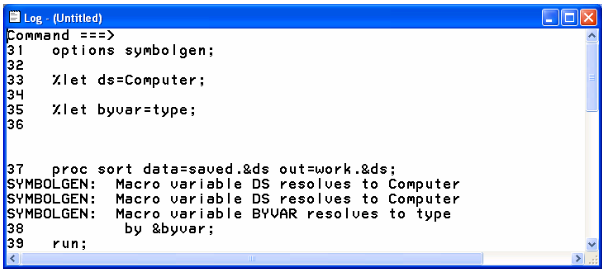
Symbolgen

The global option *Symbolgen* is one of several tools available which make it possible to examine the values of macros as they are processed. Consider the following example in which the previous code is run with this option in effect.



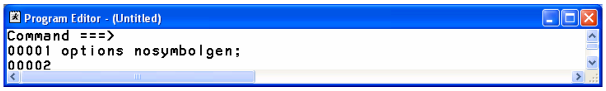
```
Command ===>
options symbolgen;

%let ds=computer;
%let byvar=type;

proc sort data=saved.&ds out=work.&ds;
   by &byvar;
run;

title1 "Print of the data set &ds";
title2 'Results of Current R&D';

proc print data=work.&ds;
   by &byvar;
run;
```

The log displays the resolved values of macros as the code is processed. This can be a useful aid in debugging a program.



```
Command ===>
31    options symbolgen;
32
33    %let ds=Computer;
34
35    %let byvar=type;
36


37    proc sort data=saved.&ds out=work.&ds;
SYMBOLGEN:  Macro variable DS resolves to Computer
SYMBOLGEN:  Macro variable DS resolves to Computer
SYMBOLGEN:  Macro variable BYVAR resolves to type
38         by &byvar;
39    run;
```

This option stays in effect until it is turned off.

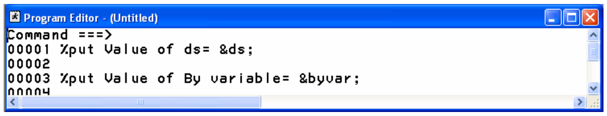


```
Command ===>
options nosymbolgen;

```

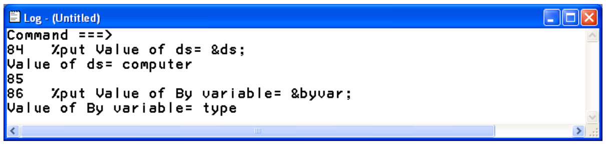
%Put Statement

A %*put* statement will write up to 132 characters of text to the log as well as the resolved value of a macro. This too can be useful in debugging a program or just as a tool to see the value of a macro.

For instance, we could examine the values of the macros *ds* and *byvar* with the following code:



```
Command ===>
%put Value of ds= &ds;

%put Value of By variable= &byvar;
```



```
Command ===>
84   %put Value of ds= &ds;
Value of ds= computer
85
86   %put Value of By variable= &byvar;
Value of By variable= type
```

Data vs. Macro Variables

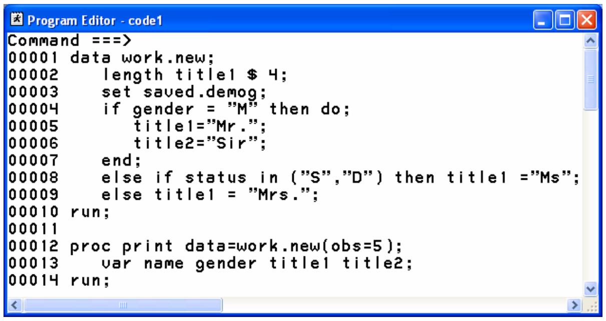
It is important to realize that macro variables are not the same as data set variables.

How are macro variables different from data set variables?

- They are created differently from data set variables.
- They are stored in symbol tables, not data sets.
- A macro variable has a single value whereas a data set variable can have multiple values depending on the loop of the data.

Creating Data Set Variables and Macro Variables

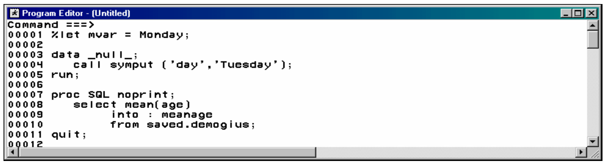
SAS can make a data set variable in many ways by using different syntax options within the Data Step.



```
Command ===>
00001 data work.new;
00002    length title1 $ 4;
00003    set saved.demog;
00004    if gender = "M" then do;
00005       title1="Mr.";
00006       title2="Sir";
00007    end;
00008    else if status in ("S","D") then title1 ="Ms";
00009    else title1 = "Mrs.";
00010 run;
00011
00012 proc print data=work.new(obs=5);
00013    var name gender title1 title2;
00014 run;
```

Line	Comment
00002	The length statement creates space in the PDV for a new data set variable.
00003	The Set statement reads the SAS data set saved.demog
00004-00009	The variables *title1* and *title2* are created by assignment.

Although at this point we have only discussed one technique, the *%let* statement, there are several ways in which macro variables can be created. Three techniques are illustrated in the following example.



```
Command ===>
00001 %let mvar = Monday;
00002
00003 data _null_;
00004    call symput ('day','Tuesday');
00005 run;
00006
00007 proc SQL noprint;
00008    select mean(age)
00009          into : meanage
00010          from saved.demogius;
00011 quit;
00012
```

Line	Comment
00001	The *%let* statement creates macro variable mvar.
00004	The *Call Symput* routine creates macro variable *day*.
00009	The Proc SQL *into clause* creates macro variable *meanage*.

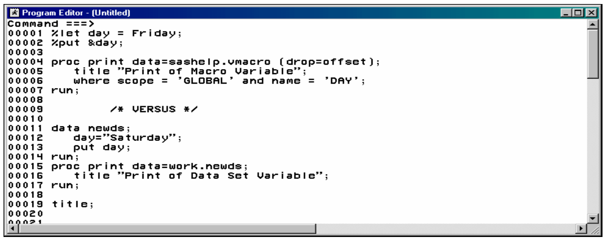
The point to remember: the syntax determines what is being created! There is no ambiguous case between a data set variable and a macro variable.

Where the Variables and Values Reside

Data set variables and macro variables can have the same name.

They will never replace each other's values, however.

They live in different structures (a symbol table vs. a data set or the PDV) and they are called for in different ways.



```
Command ===>
00001 %let day = Friday;
00002 %put &day;
00003
00004 proc print data=sashelp.vmacro (drop=offset);
00005    title "Print of Macro Variable";
00006    where scope = 'GLOBAL' and name = 'DAY';
00007 run;
00008
00009          /* VERSUS */
00010
00011 data newds;
00012    day="Saturday";
00013    put day;
00014 run;
00015 proc print data=work.newds;
00016    title "Print of Data Set Variable";
00017 run;
00018
00019 title;
00020
```

Line	Comment
	The Macro Variable *DAY*
00001	The *%let* statement defines macro variable *day* as *Friday*.
00002	The *%put* shows the value of macro variable *day* in the Log.
00004-00007	The Proc Print of sashelp.vmacro shows the macro variable in the Output window.
	The Data Set Variable *DAY*
00012	The data set character variable *day* is created through assignment.
00013	The Put statement shows the data set variable value in the Log.
00015-00017	The Proc Print of the data set shows the data set value of *day*.

Later, we will see how to pass a data set variable value directly into a macro variable.

Summary

- To define a macro variable, use the following syntax:

%let <macro_variable> = <value>;

Example:

%let ds = demograf;

- To use a macro variable, precede the macro variable name with an ampersand (&)

Example:

Proc print data = &ds;

- If the macro variable is to be used in quotes, as in a title, be sure to use double quotes to get the value to resolve.

Title "Report of Data Set &ds";

- Use the global option symbolgen to see the resolved values of macros in the log.

- Use a %put statement to write the resolved value of a macro to the log:

%put The value of the macro = &ds;

Differences between data set variables and macro variables:

- Data set variables exist in data sets; macro variables exist in symbol tables.

- Data set variables are created only in data steps or proc steps; macro variables can be created with statements such as %let that are not part of a step.

- A data set variable can have multiple values, whereas a macro variable can have only one value.

- To get the value of a data set variable, you just use the name of the variable; to resolve a macro variable, you must precede the name of the variable with an ampersand (&)

Automatic Macro Variables

Invoking SAS creates a set of automatic macro variables (assuming that SAS system options enable the macro facility).

These macro variables and their values are held in the Automatic Symbol Table (AST). The AST and its variables can only be deleted by exiting out of SAS.

The automatic macro variables exhibit some distinct features:

The values are set at the start of the SAS session. Therefore, the automatic macro variables *sysdate*, *systime*, and *sysday* are set when launching SAS.

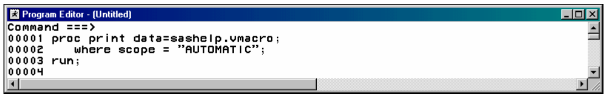
Some automatic macro variables – such as *sysdate*, *systime*, and *sysday* – are read only (R/O) and therefore cannot be changed by the programmer.

Some automatic macro variables are read-write (R/W). These can be changed by the programmer.

Determining the Values of Automatic SAS Variables

Automatic macro variables can be viewed using any of the following three syntax options.

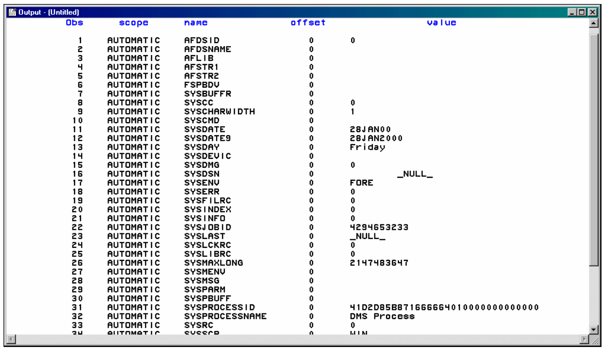
Option 1



Program Editor - (Untitled)

```
Command ===>
00001 proc print data=sashelp.vmacro;
00002     where scope = "AUTOMATIC";
00003 run;
00004
```

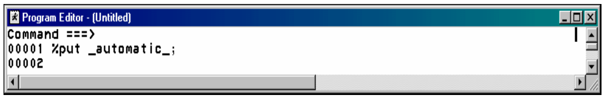
Line	Comment
00002	The variable “scope” refers to the referencing environment or table. To see other variables, use "USER", "LOCAL", "GLOBAL" in place of "AUTOMATIC"



Output - (Untitled)

```
            Obs      scope        name                 offset           value

              1    AUTOMATIC    AFDSID                   0      0
              2    AUTOMATIC    AFDSNAME                 0
              3    AUTOMATIC    AFLIB                    0
              4    AUTOMATIC    AFSTR1                   0
              5    AUTOMATIC    AFSTR2                   0
              6    AUTOMATIC    FSPBDV                   0
              7    AUTOMATIC    SYSBUFFR                 0
              8    AUTOMATIC    SYSCC                    0      0
              9    AUTOMATIC    SYSCHARWIDTH             0      1
             10    AUTOMATIC    SYSCMD                   0
             11    AUTOMATIC    SYSDATE                  0      28JAN00
             12    AUTOMATIC    SYSDATE9                 0      28JAN2000
             13    AUTOMATIC    SYSDAY                   0      Friday
             14    AUTOMATIC    SYSDEVIC                 0
             15    AUTOMATIC    SYSDMG                   0      0
             16    AUTOMATIC    SYSDSN                   0              _NULL_
             17    AUTOMATIC    SYSENV                   0      FORE
             18    AUTOMATIC    SYSERR                   0      0
             19    AUTOMATIC    SYSFILRC                 0      0
             20    AUTOMATIC    SYSINDEX                 0      0
             21    AUTOMATIC    SYSINFO                  0      0
             22    AUTOMATIC    SYSJOBID                 0      4294653233
             23    AUTOMATIC    SYSLAST                  0      _NULL_
             24    AUTOMATIC    SYSLCKRC                 0      0
             25    AUTOMATIC    SYSLIBRC                 0      0
             26    AUTOMATIC    SYSMAXLONG               0      2147483647
             27    AUTOMATIC    SYSMENV                  0
             28    AUTOMATIC    SYSMSG                   0
             29    AUTOMATIC    SYSPARM                  0
             30    AUTOMATIC    SYSPBUFF                 0
             31    AUTOMATIC    SYSPROCESSID             0      41D2D85B8716666640100000000000000
             32    AUTOMATIC    SYSPROCESSNAME           0      DMS Process
             33    AUTOMATIC    SYSRC                    0      0
             34    AUTOMATIC    SYSSCP                   0      WIN
```

Option 2



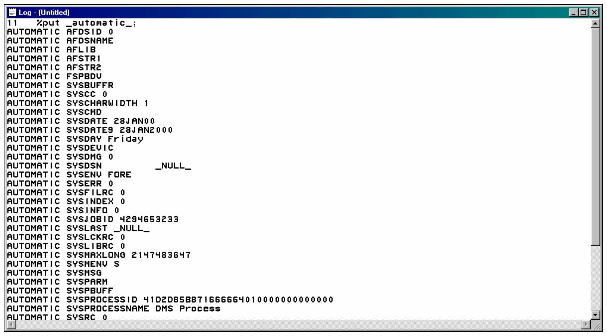
Program Editor - (Untitled)

```
Command ===>
00001 %put _automatic_;
00002
```

Line	Comment
00001	Writes the values of the Automatic macros to the log. Can also use _global_, _all_, _local_ and _user_ as needed.

Scope

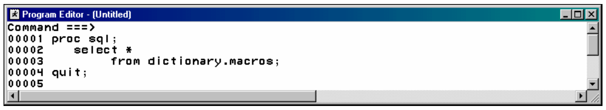
all	Returns all macros in all scopes
automatic	Returns all macros in the Automatic Global Table. These macro variables are available anywhere in programming.
global	Returns programmer-defined macros in the Global Symbol Table. These macro variables are available anywhere in programming.
local	Returns programmer-defined macros available only in the current or local scope
user	Returns a list of all the programmer-defined macro variables in the scopes. Useful for debugging.



Log - (Untitled)

```
11    %put _automatic_;
AUTOMATIC AFDSID 0
AUTOMATIC AFDSNAME
AUTOMATIC AFLIB
AUTOMATIC AFSTR1
AUTOMATIC AFSTR2
AUTOMATIC FSPBDV
AUTOMATIC SYSBUFFR
AUTOMATIC SYSCC 0
AUTOMATIC SYSCHARWIDTH 1
AUTOMATIC SYSCMD
AUTOMATIC SYSDATE 28JAN00
AUTOMATIC SYSDATE9 28JAN2000
AUTOMATIC SYSDAY Friday
AUTOMATIC SYSDEVIC
AUTOMATIC SYSDMG 0
AUTOMATIC SYSDSN         _NULL_
AUTOMATIC SYSENV FORE
AUTOMATIC SYSERR 0
AUTOMATIC SYSFILRC 0
AUTOMATIC SYSINDEX 0
AUTOMATIC SYSINFO 0
AUTOMATIC SYSJOBID 4294653233
AUTOMATIC SYSLAST _NULL_
AUTOMATIC SYSLCKRC 0
AUTOMATIC SYSLIBRC 0
AUTOMATIC SYSMAXLONG 2147483647
AUTOMATIC SYSMENV S
AUTOMATIC SYSMSG
AUTOMATIC SYSPARM
AUTOMATIC SYSPBUFF
AUTOMATIC SYSPROCESSID 41D2D85B8716666640100000000000000
AUTOMATIC SYSPROCESSNAME DMS Process
AUTOMATIC SYSRC 0
```

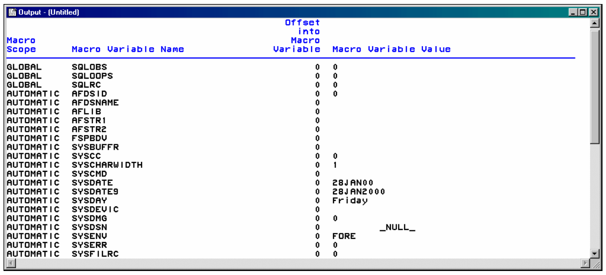
Option 3



Program Editor - (Untitled)

```
Command ===>
00001 proc sql;
00002     select *
00003          from dictionary.macros;
00004 quit;
00005
```

Shows the results in the Output window.



Output - (Untitled)

```
                                             Offset
                                               into
Macro                                         Macro
Scope       Macro Variable Name            Variable  Macro Variable Value
-------------------------------------------------------------------------
GLOBAL      SQLOBS                                0  0
GLOBAL      SQLOOPS                               0  0
GLOBAL      SQLRC                                 0  0
AUTOMATIC   AFDSID                                0  0
AUTOMATIC   AFDSNAME                              0
AUTOMATIC   AFLIB                                 0
AUTOMATIC   AFSTR1                                0
AUTOMATIC   AFSTR2                                0
AUTOMATIC   FSPBDV                                0
AUTOMATIC   SYSBUFFR                              0
AUTOMATIC   SYSCC                                 0  0
AUTOMATIC   SYSCHARWIDTH                          0  1
AUTOMATIC   SYSCMD                                0
AUTOMATIC   SYSDATE                               0  28JAN00
AUTOMATIC   SYSDATE9                              0  28JAN2000
AUTOMATIC   SYSDAY                                0  Friday
AUTOMATIC   SYSDEVIC                              0
AUTOMATIC   SYSDMG                                0  0
AUTOMATIC   SYSDSN                                0          _NULL_
AUTOMATIC   SYSENV                                0  FORE
AUTOMATIC   SYSERR                                0  0
AUTOMATIC   SYSFILRC                              0  0
```

Summary

- Quite a few macro variables are defined at the time your SAS session starts. These are available to use any place you can use macro variables.
- Some of the automatic macro variables have different values on different operating systems.
- Some of the automatic macro variables may not be available on all systems.
- To see a list of macro variables, use one of the following techniques:
  - Option 1:
    Proc print data = sashelp.vmacro;
     Where scope = "AUTOMATIC";
    run;
  - Option 2 (this will print in the log instead of the output window):
    %put _automatic_;
  - Option 3:

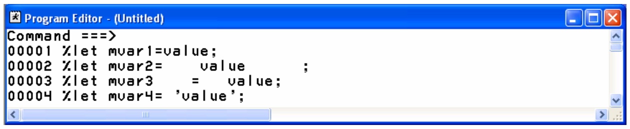
```
Proc sql;
 Select *
        From dictionary.macros
        Where scope =
"AUTOMATIC";
Quit;
```

- In options 1 and 3 above, "AUTOMATIC" may be replaced by "USER", "GLOBAL", or "LOCAL" to get customized lists. Note that since you are specifying the value of a variable, it is case-sensitive.

- In option 2 above, _automatic_ may be replaced by _user_, _local_, _global_, or _user_.

%Let Statement to Create a Macro Variable
We have already seen the *%let* statement used to create a macro variable. As stated, within a *%let* statement the name of the macro variable is to the left of the equal sign. Everything to the right of the equal sign up to but not including the semi-colon, except leading and trailing blanks, is the value of the macro variable.

The following example illustrates these concepts.



```
Command ===>
00001 %let mvar1=value;
00002 %let mvar2=    value       ;
00003 %let mvar3    =    value;
00004 %let mvar4= 'value';
```

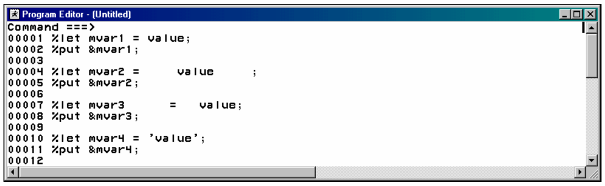
Line	Comment
00001	Macro variable definition does not include leading or trailing blanks.
00002	Macro variable definition contains leading and trailing blanks.
00003	Macro variable definition with leading blanks.
00004	Macro variable value enclosed in single quotes.

What do these macro variables resolve to?

Specifically, what about leading and trailing blanks?

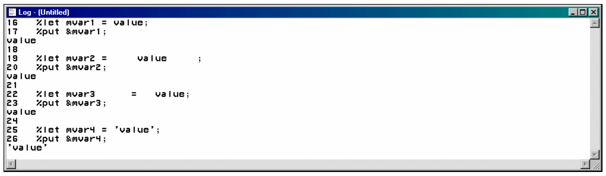
Do quotation marks become part of the value?

We will use the *%put* statement to examine the values of these macros.



```
Command ===>
00001 %let mvar1 = value;
00002 %put &mvar1;
00003
00004 %let mvar2 =     value     ;
00005 %put &mvar2;
00006
00007 %let mvar3      =   value;
00008 %put &mvar3;
00009
00010 %let mvar4 = 'value';
00011 %put &mvar4;
00012
```

Inspect the results of these paired *%let* and *%put* statements in the log.



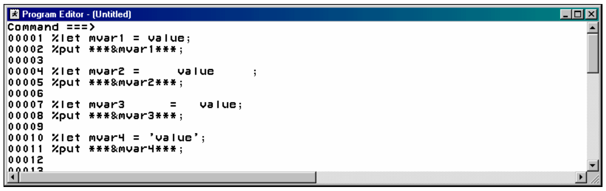
```
16    %let mvar1 = value;
17    %put &mvar1;
value
18
19    %let mvar2 =     value     ;
20    %put &mvar2;
value
21
22    %let mvar3      =   value;
23    %put &mvar3;
value
24
25    %let mvar4 = 'value';
26    %put &mvar4;
'value'
```

Line	Comment
after 17	Resolution of *&mvar1* to *value*
after 20	Resolution of *&mvar2* to *value* (no leading or trailing blanks)
after 23	Resolution of *&mvar3* (no leading blanks on the macro name)
after 26	Resolution of *&mvar4* to *'value'* (includes quotes)

In some cases, it is clear that the leading blanks were not included as part of the macro variable value. (In the Log, they are written flush with the left margin.)

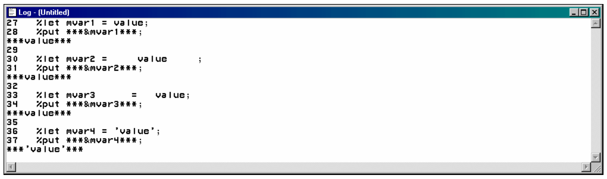
However, how can we test for trailing blanks?

To do so, insert a character string around the macro variable as shown.



```
Command ===>
00001 %let mvar1 = value;
00002 %put ***&mvar1***;
00003
00004 %let mvar2 =     value     ;
00005 %put ***&mvar2***;
00006
00007 %let mvar3      =   value;
00008 %put ***&mvar3***;
00009
00010 %let mvar4 = 'value';
00011 %put ***&mvar4***;
00012
00013
```

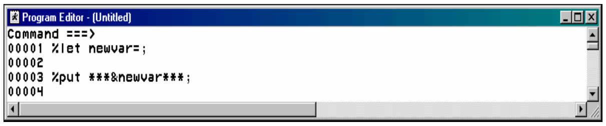
Inspect the Log again for the macro variable values.



```
27    %let mvar1 = value;
28    %put ***&mvar1***;
***value***
29
30    %let mvar2 =     value     ;
31    %put ***&mvar2***;
***value***
32
33    %let mvar3      =   value;
34    %put ***&mvar3***;
***value***
35
36    %let mvar4 = 'value';
37    %put ***&mvar4***;
***'value'***
```

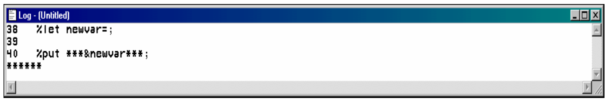
Null Values

One particular *%let* statement is used to assign a null value to the macro variable.



```
Command ===>
00001 %let newvar=;
00002
00003 %put ***&newvar***;
00004
```

Line	Comment
00001	*%let* statement creates a macro variable with null value. (This will be useful later for controlling which symbol table receives the macro value as well as eliminating unwanted macro variable values.)



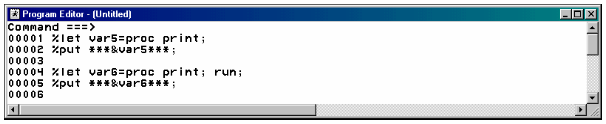
```
38    %let newvar=;
39
40    %put ***&newvar***;
******
```

%Let Statement – Another Look

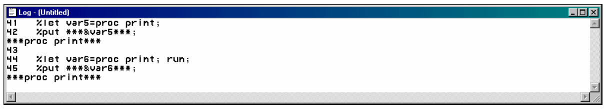
Now inspect the following *%let* statements.

Both will result in the same value for the macro variables.

What is the value? Why?



```
Command ===>
00001 %let var5=proc print;
00002 %put ***&var5***;
00003
00004 %let var6=proc print; run;
00005 %put ***&var6***;
00006
```



```
41    %let var5=proc print;
42    %put ***&var5***;
***proc print***
43
44    %let var6=proc print; run;
45    %put ***&var6***;
***proc print***
```

Line	Comment
after 42	The *%let* statement stops at semi-colon. *&var5* resolves to *proc print.*
after 45	The *%let* statement stops at first semi-colon. *&var6* resolves to *proc print.*

The semi-colon has special meaning associated with it. When *var5* is defined, the *%let* statement ends at the semi-colon. The value of macro variable *var5* is *proc print*.

The same process is followed for *var6* where the first semi-colon ends the *%let* statement. The semi-colon is not treated as 'just another' keystroke; it has functionality.

There are times when we need characters to be 'just another' character without functionality.

%STR and %NRSTR Functions

Two functions – *%str( )* and *%nrstr( )* – provide a means to remove the special meaning of certain characters:

%str( ) – eliminates the functionality of most special characters, except for the ampersand and percent symbols. It does not remove the meaning of apostrophes.

%nrstr( ) – eliminates the functionality of most special characters, including the ampersand and percent symbols. It does not remove the meaning of apostrophes.

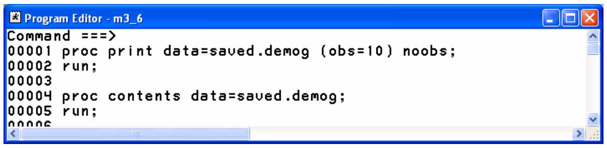
The following table illustrates the use of these two functions.

%let statement	Macro Variable Call	Macro Variable Value
%let Task1 = Proc print;	&Task1	Proc print
%let Task2=%str(Proc print;);	&Task2	Proc print;
%let Task3=%str(Proc print; run;);	&Task3	Proc print; run;
%let More=%str(&task1; run;);	&More	Proc print; run;
%let More1=%nrstr(&task1;run;);	&More1	&task1; run;

Macro Call	Comment
&Task1	Note that the semi-colon is not part of the resolved macro value.
&Task2	Note the semi-colon in the resolved macro value
&More	The macro variable *&task1* resolves in the *%str( )* function.
&More1	The macro variable *&task1* does not resolve in the *%nrstr( )* function.

Using the %STR Function

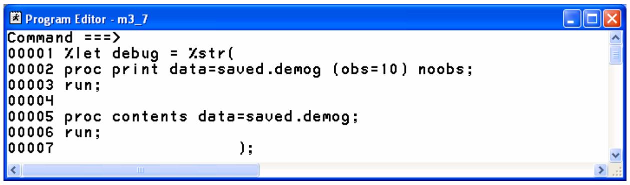
As an example of using the *%str* function consider the following code. Proc Print is used to print the first ten observations from a data set and then Proc Contents is used to display the descriptor part of the data set.



```
Command ===>
proc print data=saved.demog (obs=10) noobs;
run;

proc contents data=saved.demog;
run;
```

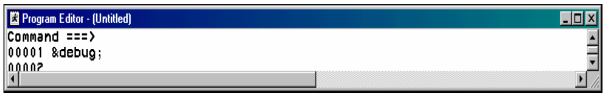
Assume you wished to run this code multiple times. One approach would be to type this code whenever needed.

Another approach is to make this entire code a macro variable. To include the semi-colon as part of the macro value the *%str* function is used.



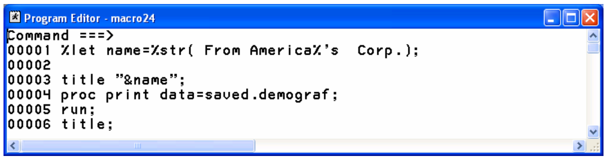
```
Command ===>
%let debug = %str(
proc print data=saved.demog (obs=10) noobs;
run;

proc contents data=saved.demog;
run;
                    );
```

Now, anytime the programmer wishes to use this code, a shorter syntax can be used.

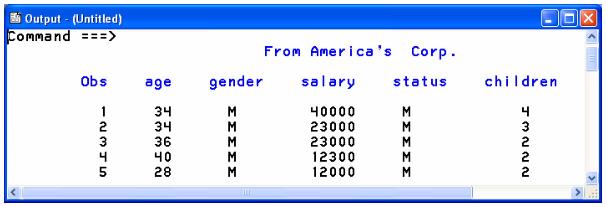


```
Command ===>
&debug;
```

Now consider a situation in which an apostrophe is part of a macro variable value. Neither the *%str* nor the *%nrstr* functions remove the functionality of the apostrophe. To remove this special meaning it is necessary to place a % sign in front of the apostrophe.



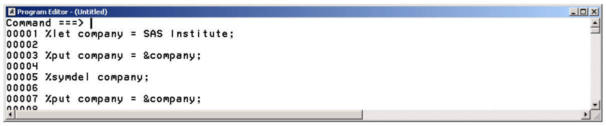
```
Command ===>
%let name=%str( From America%'s  Corp.);

title "&name";
proc print data=saved.demograf;
run;
title;
```



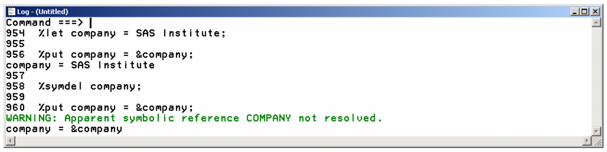
```
Command ===>
                    From America's  Corp.

     Obs    age    gender    salary    status    children

      1      34      M       40000       M          4
      2      34      M       23000       M          3
      3      36      M       23000       M          2
      4      40      M       12300       M          2
      5      28      M       12000       M          2
```

Removing Macro Variables from the Global Symbol Table
Once a macro variable is created, that is written to the Global Symbol Table, it exists until it is explicitly deleted. Symbol tables will be discussed in greater detail in a subsequent chapter. For present considerations, it is necessary to know that a macro variable created by a %*let* statement in open code (outside a macro bundle) is written to the Global Symbol Table.

Symdel is used to delete macro variables. *Symdel* can be used either as a macro statement or as a call routine in a data step. The macro statement form is illustrated below.



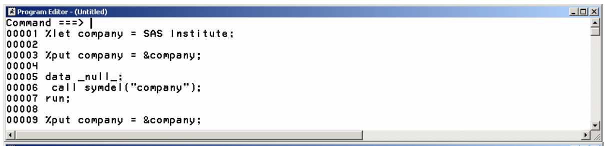
```
Command ===>
00001 %let company = SAS Institute;
00002
00003 %put company = &company;
00004
00005 %symdel company;
00006
00007 %put company = &company;
00008
```



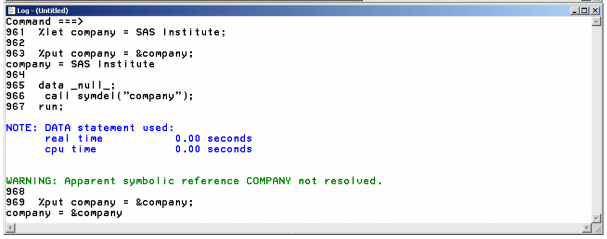
```
Command ===>
954  %let company = SAS Institute;
955
956  %put company = &company;
company = SAS Institute
957
958  %symdel company;
959
960  %put company = &company;
WARNING: Apparent symbolic reference COMPANY not resolved.
company = &company
```

Line	Comment
00001	*%let* statement creates a macro variable.
00003	*%put* statement prints the label and variable in the log.
00005	*%symdel* statement deletes the macro variable from the global symbol table.
00007	*%put* statement attempts to print the variable after the deletion; the log shows that the variable no longer exists.

The call routine form of *Symdel* is illustrated below.



```
Command ===>
00001 %let company = SAS Institute;
00002
00003 %put company = &company;
00004
00005 data _null_;
00006  call symdel("company");
00007 run;
00008
00009 %put company = &company;
```



```
Command ===>
961  %let company = SAS Institute;
962
963  %put company = &company;
company = SAS Institute
964
965  data _null_;
966   call symdel("company");
967  run;

NOTE: DATA statement used:
      real time           0.00 seconds
      cpu time            0.00 seconds


WARNING: Apparent symbolic reference COMPANY not resolved.
968
969  %put company = &company;
company = &company
```

Line	Comment
00001	*%let* statement creates a macro variable.
00003	*%put* statement prints the label and variable in the log.
00005 – 00007	The *symdel* call routine deletes the macro variable from the global symbol table. Notice the macro variable reference – no ampersand, but enclosed in quotes (either single or double quotes will work)
00009	*%put* statement attempts to print the variable after the deletion; the log shows that the variable no longer exists.

Resolution Considerations

Macro programming relies on invoking a macro value by naming and resolving the macro variable.

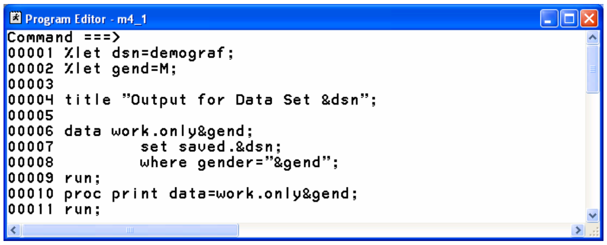
Various programming issues often require a macro variable to be embedded in another character string. In this chapter we examine how these macros are identified and processed.

Macro Variable as a Suffix

We have seen that an ampersand followed by a non-blank character is treated as a macro variable name and resolved.

How is a macro variable name identified and processed when it is part of another character string? As part of another character string a macro variable is resolved and the result is concatenated with the remaining character string. A blank, an ampersand (&) or a dot (.) indicate the end of a macro variable name.

Consider the following example:



```
Command ===>
00001 %let dsn=demograf;
00002 %let gend=M;
00003
00004 title "Output for Data Set &dsn";
00005
00006 data work.only&gend;
00007         set saved.&dsn;
00008         where gender="&gend";
00009 run;
00010 proc print data=work.only&gend;
00011 run;
```

Line	Comment
00006	Name of data set being created is work.onlyM.
00007	Data set being read is saved.demograf
00008	Where clause resolves to: where gender="M";

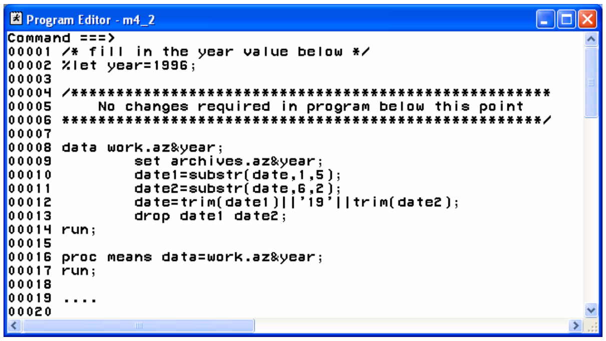
Consider a second example:

A SAS library contains data sets named after a state's two-letter abbreviation and the fiscal years from 1990 to 1999. Thus, AZ1990, AZ1991, … AZ1999 are examples of the data set names.

You must change the date structure in each of the 10 data sets as well as manipulate each data set to produce statistics and graphic output.

How can macros provide quick access to making changes accommodating each of the ten data sets?

The solution is a simple application of the *%let* statement.



```
Command ===>
00001 /* fill in the year value below */
00002 %let year=1996;
00003
00004 /*******************************************************
00005      No changes required in program below this point
00006 *******************************************************/
00007
00008 data work.az&year;
00009         set archives.az&year;
00010         date1=substr(date,1,5);
00011         date2=substr(date,6,2);
00012         date=trim(date1)||'19'||trim(date2);
00013         drop date1 date2;
00014 run;
00015
00016 proc means data=work.az&year;
00017 run;
00018
00019 ....
00020
```

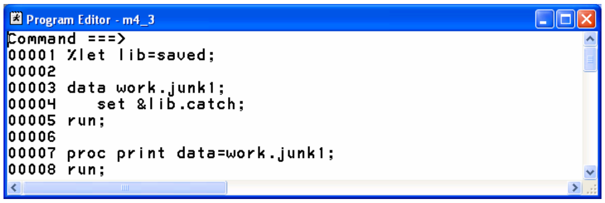
Line	Comment
00001	Use comments to provide user assistance.
00002	The *%let* statement allows the year value to change throughout the program. In this case the value 1996 has been entered.
00004	Stop the user from making changes where unnecessary.

All changes are consolidated at the top of the program.

Rather than changing every reference to the data set all through the entire length of the program, we can reference the data set name through a macro variable and generate the correct name each time.

Macro Variable as a Prefix

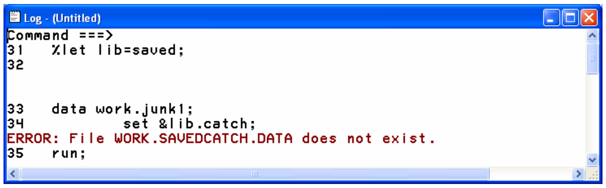
In the next example an attempt is made to use a macro variable in front of other text.



```
Command ===>
00001 %let lib=saved;
00002
00003 data work.junk1;
00004     set &lib.catch;
00005 run;
00006
00007 proc print data=work.junk1;
00008 run;
```

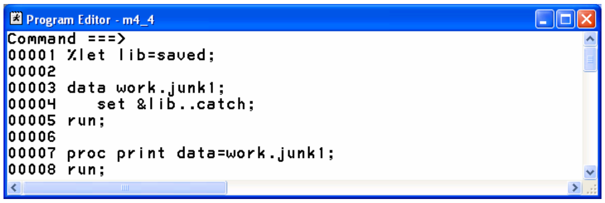
Line	Comment
00004	Resolves to savedcatch

As the log indicates this results in an error.



```
Command ===>
31    %let lib=saved;
32


33    data work.junk1;
34              set &lib.catch;
ERROR: File WORK.SAVEDCATCH.DATA does not exist.
35    run;
```

As part of the process to identify the end of the macro name the dot (.) is absorbed. Therefore, when the macro resolves, the dot (.) is removed and not left behind as part of the remaining character string.
By adding a second dot (.) the macro resolves properly. The first dot (.) operates as discussed while the second dot (.) is simply part of the remaining character string.



```
Command ===>
00001 %let lib=saved;
00002
00003 data work.junk1;
00004     set &lib..catch;
00005 run;
00006
00007 proc print data=work.junk1;
00008 run;
```

Line	Comment
00004	Resolves to saved.catch

As another example:

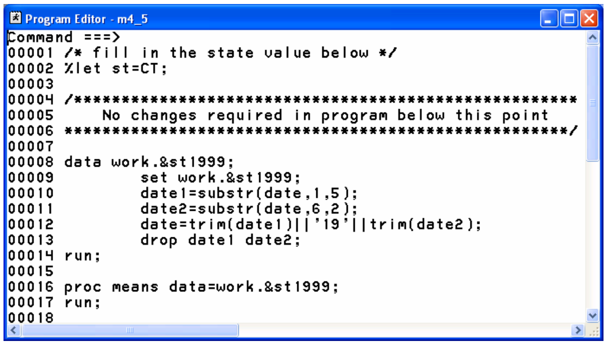
A SAS library contains data sets named after each state's two-letter abbreviation and the fiscal year 1999. Thus, AZ1999, CT1999, HI1999, NY1999 are examples of the data set names.

You must change the date structure in each of the 50 data sets as well as manipulate the data set to produce statistics and graphic output.

How can macros provide quick access making changes accommodating each of the fifty data sets?

At first, it would seem that the *%let* statement would work.

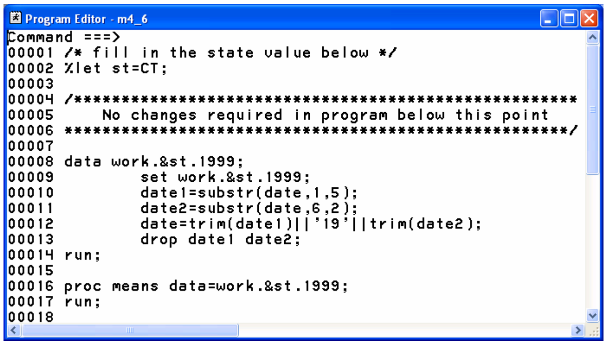
Note: This program demonstrates a programming error.



```
Command ===>
00001 /* fill in the state value below */
00002 %let st=CT;
00003
00004 /*****************************************************
00005      No changes required in program below this point
00006 *****************************************************/
00007
00008 data work.&st1999;
00009          set work.&st1999;
00010          date1=substr(date,1,5);
00011          date2=substr(date,6,2);
00012          date=trim(date1)||'19'||trim(date2);
00013          drop date1 date2;
00014 run;
00015
00016 proc means data=work.&st1999;
00017 run;
00018
```

The program soon shows a problem. Rather than search for the macro variable *st*, SAS searches for the macro variable *st1999*.

The program must be revised to end the macro name explicitly with a dot (.).



```
Command ===>
00001 /* fill in the state value below */
00002 %let st=CT;
00003
00004 /*****************************************************
00005      No changes required in program below this point
00006 *****************************************************/
00007
00008 data work.&st.1999;
00009          set work.&st.1999;
00010          date1=substr(date,1,5);
00011          date2=substr(date,6,2);
00012          date=trim(date1)||'19'||trim(date2);
00013          drop date1 date2;
00014 run;
00015
00016 proc means data=work.&st.1999;
00017 run;
00018
```

Line	Comment
00008, 00009, 00016	The macro variable call of *&st.* will resolve to the value desired.

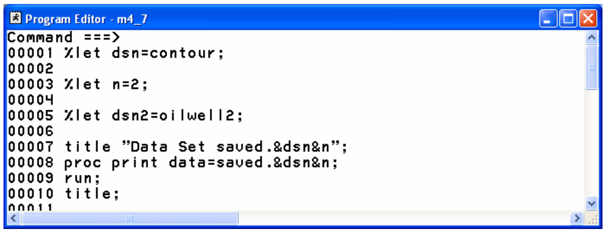
Remember that SAS uses the dot for other syntax reasons such as:

Usage of Dot	Example
Multiple-level names	Source.data Source.project.data.entry
First-dot / Last-dot	If first.gender = 0 then ….;
Formats / Informats	Dollar12.2
Missing numerics	Where salary ne .;
Naming output	Put ‘<img src=”mypic.gif”>’;

Append Two Macro Variables Together

Often two or more macro variables are incorporated into a single character string. Again, the issue is how is the macro variable name identified and processed.

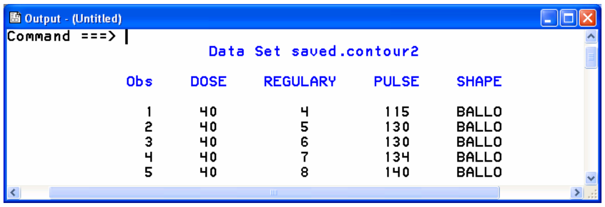
In the following example the macro variables *dsn* and *n* are used together to identify the data set to be printed. Which data set is printed?



```
Command ===>
00001 %let dsn=contour;
00002
00003 %let n=2;
00004
00005 %let dsn2=oilwell2;
00006
00007 title "Data Set saved.&dsn&n";
00008 proc print data=saved.&dsn&n;
00009 run;
00010 title;
00011
```

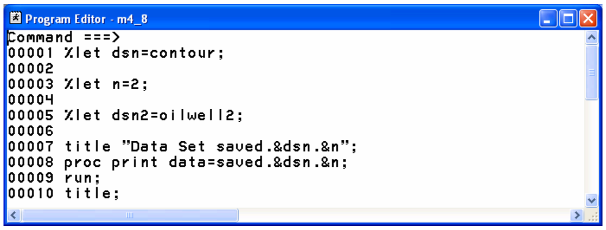
The ampersand (&) indicates the beginning of a macro variable name. Therefore, *&dsn* is considered one macro variable name while &*n* refers to a second macro variable. The entire character string resolves to saved.contour2.

Line	Comment
00001, 00003, 00005	Create macro variables *dsn, n,* and *dsn2.*
00007, 00008	Resolves to saved.*contour2*



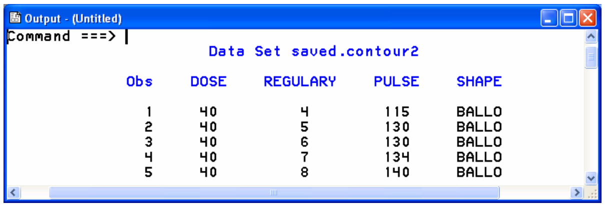
```
Command ===> |
                    Data Set saved.contour2

            Obs    DOSE    REGULARY    PULSE    SHAPE

              1     40         4        115     BALLO
              2     40         5        130     BALLO
              3     40         6        130     BALLO
              4     40         7        134     BALLO
              5     40         8        140     BALLO
```

In the following example a dot (.) indicates the end of the first macro variable name (&*dsn*). The dot (.) acts as a delimiter and is removed as part of the process to identify the macro variable name. Because a second macro variable name follows, the dot (.) is not needed. This character string resolves to saved.contour2 just as in the previous example.



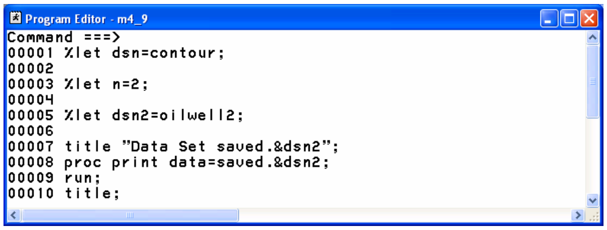
```
Command ===>
00001 %let dsn=contour;
00002
00003 %let n=2;
00004
00005 %let dsn2=oilwell2;
00006
00007 title "Data Set saved.&dsn.&n";
00008 proc print data=saved.&dsn.&n;
00009 run;
00010 title;
```

Line	Comment
00001, 00003, 00005	Create macro variables *dsn, n,* and *dsn2.*
00007, 00008	Resolves to saved.*contour2*



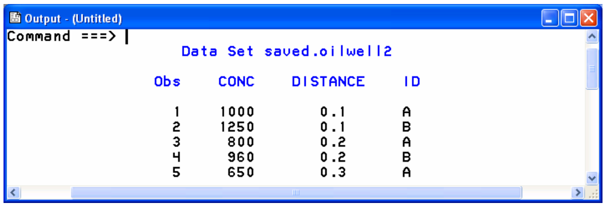
```
Command ===> |
                    Data Set saved.contour2

            Obs    DOSE    REGULARY    PULSE    SHAPE

              1     40         4        115     BALLO
              2     40         5        130     BALLO
              3     40         6        130     BALLO
              4     40         7        134     BALLO
              5     40         8        140     BALLO
```

Finally, consider the following code. Which data set is printed?



```
Command ===>
00001 %let dsn=contour;
00002
00003 %let n=2;
00004
00005 %let dsn2=oilwell2;
00006
00007 title "Data Set saved.&dsn2";
00008 proc print data=saved.&dsn2;
00009 run;
00010 title;
```

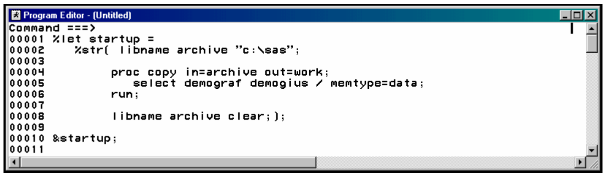
Line	Comment
00001, 00003, 00005	Create macro variables *dsn, n,* and *dsn2.*
00007, 00008	Resolves to saved.*oilwell2*



```
Command ===> |
                    Data Set saved.oilwell2

                Obs    CONC    DISTANCE    ID

                  1    1000       0.1      A
                  2    1250       0.1      B
                  3     800       0.2      A
                  4     960       0.2      B
                  5     650       0.3      A
```

Macro Code Buncles

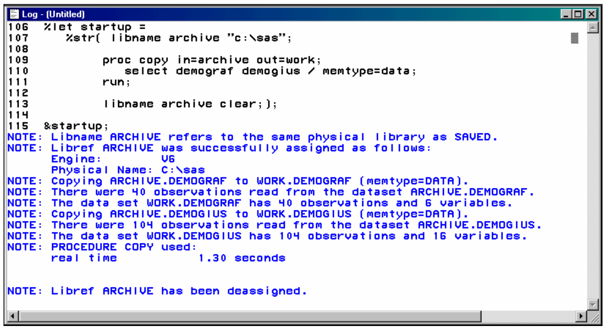
The regular job you submit to backup your data sets or produce your graphics can all be bundled up into a macro and then 'invoked' using one word – the name of the macro.



```
Command ===>
00001 %let startup =
00002    %str( libname archive "c:\sas";
00003
00004         proc copy in=archive out=work;
00005            select demograf demogius / memtype=data;
00006         run;
00007
00008         libname archive clear;);
00009
00010 &startup;
00011
```

Line	Comment
00001-00008	The *%let* statement used with the *%str* function to write a short program.
00010	The macro *startup* is invoked with *&startup;* (i.e., ampersand, name, semi-colon).

The log shows the results.

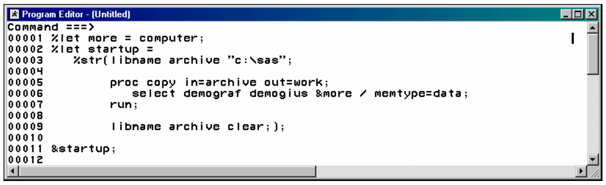


```
106   %let startup =
107      %str( libname archive "c:\sas";
108
109            proc copy in=archive out=work;
110               select demograf demogius / memtype=data;
111            run;
112
113            libname archive clear;);
114
115   &startup;
NOTE: Libname ARCHIVE refers to the same physical library as SAVED.
NOTE: Libref ARCHIVE was successfully assigned as follows:
      Engine:        V6
      Physical Name: C:\sas
NOTE: Copying ARCHIVE.DEMOGRAF to WORK.DEMOGRAF (memtype=DATA).
NOTE: There were 40 observations read from the dataset ARCHIVE.DEMOGRAF.
NOTE: The data set WORK.DEMOGRAF has 40 observations and 6 variables.
NOTE: Copying ARCHIVE.DEMOGIUS to WORK.DEMOGIUS (memtype=DATA).
NOTE: There were 104 observations read from the dataset ARCHIVE.DEMOGIUS.
NOTE: The data set WORK.DEMOGIUS has 104 observations and 16 variables.
NOTE: PROCEDURE COPY used:
      real time           1.30 seconds


NOTE: Libref ARCHIVE has been deassigned.
```

Reusing fixed programs is one advantage of a macro. Rather than retype a lengthy set of programming instructions, it is easier to create a macro program and call it with a single word.

The limiting feature of the previous example is its inability to adapt to changing needs. Suppose the programmer needs a different data set copied.

As currently written it would be necessary to add the name of the new data set to the Proc Copy select statement. Greater flexibility can be achieved by incorporating a macro variable into the select statement.

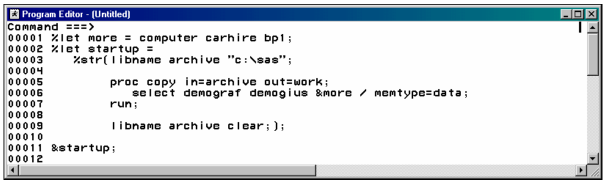


```
Command ===>
00001 %let more = computer;
00002 %let startup =
00003    %str(libname archive "c:\sas";
00004
00005          proc copy in=archive out=work;
00006             select demograf demogius &more / memtype=data;
00007          run;
00008
00009          libname archive clear;);
00010
00011 &startup;
00012
```

Line	Comment
00006	The *&more* macro variable makes it possible to specify additional data sets.

Notice that in the above example a single additional data set has been copied.

To copy several additional data sets simply list the data sets as the value of the macro variable.



```
Command ===>
00001 %let more = computer carhire bp1;
00002 %let startup =
00003    %str(libname archive "c:\sas";
00004
00005          proc copy in=archive out=work;
00006             select demograf demogius &more / memtype=data;
00007          run;
00008
00009          libname archive clear;);
00010
00011 &startup;
00012
```

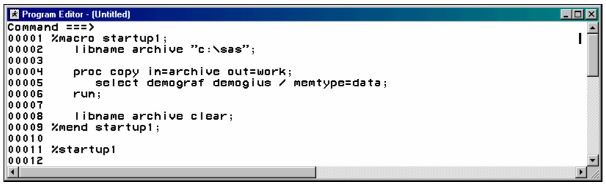
Line	Comment
00006	The *%let* statement defines multiple data sets to copy along with the ones already identified.

%Macro - %Mend

SAS can define a series of programming statements in an alternate manner.

Rather than use the *%let = %str( )* or *%let = %nrstr( )* syntax, begin the code with *%macro* statement and end it with *%mend* statement.

The rewritten program is shown below.



```
Command ===>
00001 %macro startup1;
00002    libname archive "c:\sas";
00003
00004    proc copy in=archive out=work;
00005       select demograf demogius / memtype=data;
00006    run;
00007
00008    libname archive clear;
00009 %mend startup1;
00010
00011 %startup1
00012
```

Line	Comment
00001-00009	A macro bundle is created using *%macro* and *%mend* syntax.
00011	The macro bundle *startup1* is invoked with *%startup1* (i.e., percent and name only – no semi-colon).

Notice that the macro call, *%startup1*, does not include a semi-colon.

There is no need here as a semi-colon has been generated by the macro call.

The code within the definition is complete, so no extra semi-colon is required.

%Macro - %Mend Notes

- A macro bundle – defined by *%macro - %mend* statements – must be defined before the named bundle can be called.

- Macro definitions start with the *%macro* statement that defines the name of the macro.

- The definition continues until the *%mend* statement.

- Including the macro name in the *%mend* statement is optional but advised.

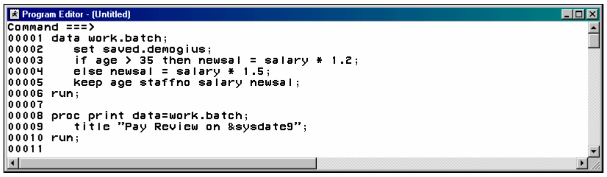
- The macro is called or invoked by typing the macro name (no semi-colon).

A macro may contain:

- Data and Proc step code

- Macro programming statements and functions

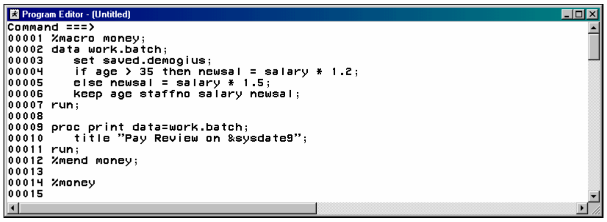
As a second example, consider the following code.



```
Command ===>
data work.batch;
   set saved.demogius;
   if age > 35 then newsal = salary * 1.2;
   else newsal = salary * 1.5;
   keep age staffno salary newsal;
run;

proc print data=work.batch;
   title "Pay Review on &sysdate9";
run;

```

How can we bundle this code so all we have to type to execute the code is a single word?

Use the *%macro - %mend* syntax.



```
Command ===>
%macro money;
data work.batch;
   set saved.demogius;
   if age > 35 then newsal = salary * 1.2;
   else newsal = salary * 1.5;
   keep age staffno salary newsal;
run;

proc print data=work.batch;
   title "Pay Review on &sysdate9";
run;
%mend money;

%money

```

The macro name may be omitted in the *%mend* statement because the SAS System will default to ending the last macro.

However, the *%mend* statement is critical. The macro processor takes control when a *%macro* statement is seen. Should the *%mend* be missing, everything from the *%macro* statement is regarded as being part of the open macro definition.

There are occasions when all the submitted code appears to be written to the log and nothing else - the code appears to be disappearing into a black hole! On such occasions, check for the absence of a *%mend* statement.

Macro Bundle Parameters

In the *%money* example, the data set names were fixed.   How can we write a macro invoking any name for the library and data sets involved?

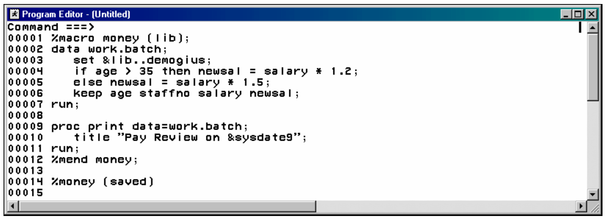
The way to do this is to pass parameters to the macro bundle. To use this method, the macro must be defined as requiring parameters.

Macro bundle parameters are created by listing variables in a set of parentheses next to the name of the macro bundle. The parameters are nothing more than macro variables available for use within the macro bundle. Within the bundle the macro variables are referenced just as we have seen before, with an ampersand followed by the macro variable name. When the macro is called the values for the parameters are specified in a set of parentheses next to the name of the macro.

There are two types of macro bundle parameters, positional and keyword.

Positional Parameters

The following example shows the creation of a single positional parameter.



```
Command ===>
00001 %macro money (lib);
00002 data work.batch;
00003    set &lib..demogius;
00004    if age > 35 then newsal = salary * 1.2;
00005    else newsal = salary * 1.5;
00006    keep age staffno salary newsal;
00007 run;
00008
00009 proc print data=work.batch;
00010    title "Pay Review on &sysdate9";
00011 run;
00012 %mend money;
00013
00014 %money (saved)
00015
```

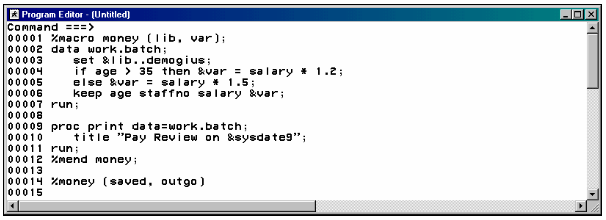
Line	Comment
00001	The positional parameter *lib* is established.
00014	The value ‘saved’ replaces each macro call for *&lib*.

The above example is logically equivalent to:

%let lib = saved;

Multiple parameters may be created. Multiple positional parameters are simply listed with a comma between parameters. For positional parameters variable names and associated values are determined by the position or order of the parameters.

In the following example two positional parameters are created.



```
Command ===>
00001 %macro money (lib, var);
00002 data work.batch;
00003    set &lib..demogius;
00004    if age > 35 then &var = salary * 1.2;
00005    else &var = salary * 1.5;
00006    keep age staffno salary &var;
00007 run;
00008
00009 proc print data=work.batch;
00010    title "Pay Review on &sysdate9";
00011 run;
00012 %mend money;
00013
00014 %money (saved, outgo)
00015
```

Line	Comment
00001	Two positional parameters are established, *lib* and *var* respectively.
00014	The value ‘saved’ is substituted for the first positional parameter – *lib* – and the value ‘outgo’ is substituted for the second positional parameter – *var.*

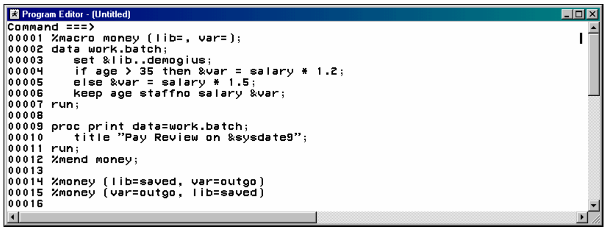
Your macro call must have a number of values matching the number of positional parameters.

Keyword Parameters

Keyword parameters are defined using the parameter name with an equal sign. This technique is preferred in certain situations because:

- It does not require defining and passing parameters in the same order.
- It allows default values to be used.

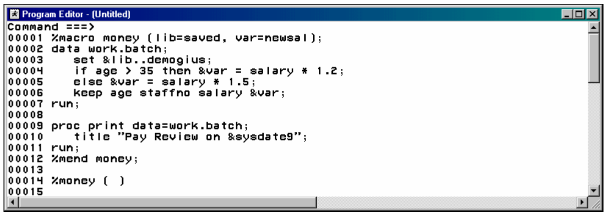
Two keyword parameters are created in the following example.



```
Command ===>
00001 %macro money (lib=, var=);
00002 data work.batch;
00003    set &lib..demogius;
00004    if age > 35 then &var = salary * 1.2;
00005    else &var = salary * 1.5;
00006    keep age staffno salary &var;
00007 run;
00008
00009 proc print data=work.batch;
00010    title "Pay Review on &sysdate9";
00011 run;
00012 %mend money;
00013
00014 %money (lib=saved, var=outgo)
00015 %money (var=outgo, lib=saved)
00016
```

Line	Comment
00001	Two keyword parameters are created – *lib* and *var*.
00014 00015	The values are assigned for the keyword parameters. The order of assigning values is not important with keyword parameters.

In the next example default values are assigned to the macro parameters.



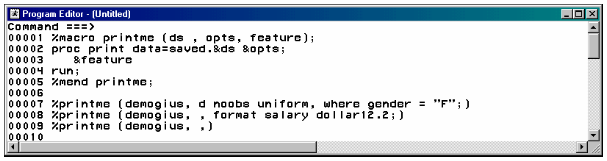
```
Command ===>
00001 %macro money (lib=saved, var=newsal);
00002 data work.batch;
00003    set &lib..demogius;
00004    if age > 35 then &var = salary * 1.2;
00005    else &var = salary * 1.5;
00006    keep age staffno salary &var;
00007 run;
00008
00009 proc print data=work.batch;
00010    title "Pay Review on &sysdate9";
00011 run;
00012 %mend money;
00013
00014 %money ( )
00015
```

Line	Comment
00001	Two keyword parameters are created, with default values assigned.
00014	The macro invocation uses the default parameters.

Parentheses must be used when invoking a macro with keyword parameters. If no values are being passed an empty set of parentheses is used.

Null Values

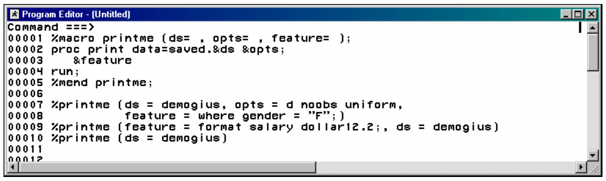
With positional parameters null values are assigned by using a comma as a 'placeholder':



```
Command ===>
00001 %macro printme (ds , opts, feature);
00002 proc print data=saved.&ds &opts;
00003    &feature
00004 run;
00005 %mend printme;
00006
00007 %printme (demogius, d noobs uniform, where gender = "F";)
00008 %printme (demogius, , format salary dollar12.2;)
00009 %printme (demogius, ,)
00010
```

Line	Comment
00001	Three positional parameters are defined.
00007	Each positional parameter is given a value.
00008	The second parameter – *opts* – lacks a value. A null value is given. No print options are defined.
00009	The second and third parameters – *opts* and *feature* – lack values. Null values are given.

A null value is assigned for keyword parameters by simply omitting the parameter.

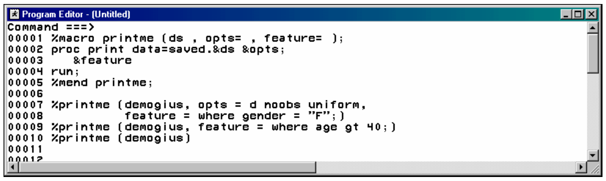


```
Command ===>
00001 %macro printme (ds= , opts= , feature= );
00002 proc print data=saved.&ds &opts;
00003    &feature
00004 run;
00005 %mend printme;
00006
00007 %printme (ds = demogius, opts = d noobs uniform,
00008           feature = where gender = ”F”;)
00009 %printme (feature = format salary dollar12.2;, ds = demogius)
00010 %printme (ds = demogius)
00011
00012
```

Line	Comment
00001	Three keyword parameters are defined without default values.
00007-00008	All three keyword parameters are given values. The order of the parameters is not important.
00009	The second parameter – *opts* – has not been listed. It receives a null value.
00010	The second and third parameters – *otps* and *feature* – are not listed and receive null values.

Combination of Positional and Keyword Parameters

If positional and keyword parameters are used together, the positional parameters must be listed first.



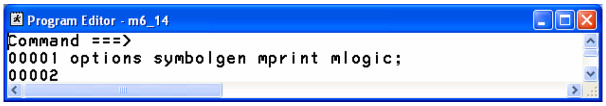
```
Command ===>
00001 %macro printme (ds , opts= , feature= );
00002 proc print data=saved.&ds &opts;
00003    &feature
00004 run;
00005 %mend printme;
00006
00007 %printme (demogius, opts = d noobs uniform,
00008           feature = where gender = ”F”;)
00009 %printme (demogius, feature = where age gt 40;)
00010 %printme (demogius)
00011
00012
```

Line	Comment
00001	Three parameters are defined, only one of which is positional and two are keyword.
00007-00008, 00009, 00010	The macro bundle call references the positional parameter(s) first. Any keyword parameters can be referenced only after the positional parameters are given.

Macro Debugging Options

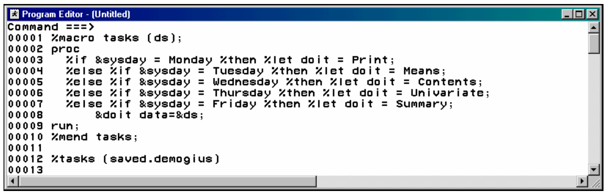
A major challenge of the programmer is to verify that syntax written by a macro and values passed are correct. As for assuring the latter, the programmer can insert a series of *%put* statements in the syntax while it is being written. If the predicted values match the displayed values, the program is in good shape. The SAS session can also use three system options to see more information about the processing of macro code and values.

While developing macros, consider using any of three options.



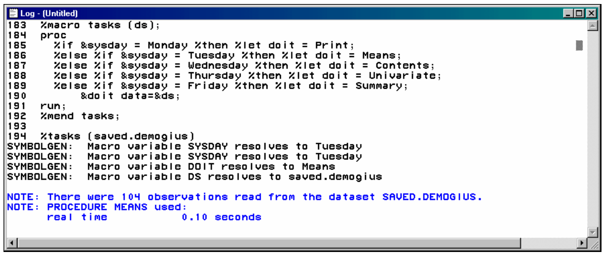
```
Command ===>
00001 options symbolgen mprint mlogic;
00002
```

Consider the following program with various macro options invoked one at a time.



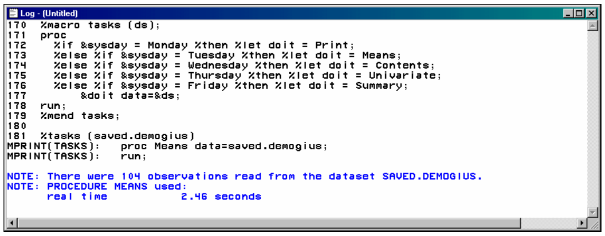
```
Command ===>
00001 %macro tasks (ds);
00002 proc
00003   %if &sysday = Monday %then %let doit = Print;
00004   %else %if &sysday = Tuesday %then %let doit = Means;
00005   %else %if &sysday = Wednesday %then %let doit = Contents;
00006   %else %if &sysday = Thursday %then %let doit = Univariate;
00007   %else %if &sysday = Friday %then %let doit = Summary;
00008       &doit data=&ds;
00009 run;
00010 %mend tasks;
00011
00012 %tasks (saved.demogius)
00013
```

Symbolgen shows the values of macro variables.



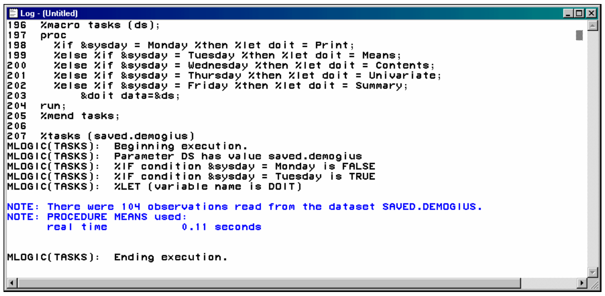
```
183   %macro tasks (ds);
184   proc
185     %if &sysday = Monday %then %let doit = Print;
186     %else %if &sysday = Tuesday %then %let doit = Means;
187     %else %if &sysday = Wednesday %then %let doit = Contents;
188     %else %if &sysday = Thursday %then %let doit = Univariate;
189     %else %if &sysday = Friday %then %let doit = Summary;
190         &doit data=&ds;
191   run;
192   %mend tasks;
193
194   %tasks (saved.demogius)
SYMBOLGEN:  Macro variable SYSDAY resolves to Tuesday
SYMBOLGEN:  Macro variable SYSDAY resolves to Tuesday
SYMBOLGEN:  Macro variable DOIT resolves to Means
SYMBOLGEN:  Macro variable DS resolves to saved.demogius

NOTE: There were 104 observations read from the dataset SAVED.DEMOGIUS.
NOTE: PROCEDURE MEANS used:
      real time           0.10 seconds
```

Mprint writes the code actually created by the macro syntax to the Log window.



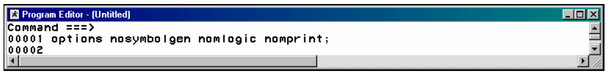
```
170   %macro tasks (ds);
171   proc
172     %if &sysday = Monday %then %let doit = Print;
173     %else %if &sysday = Tuesday %then %let doit = Means;
174     %else %if &sysday = Wednesday %then %let doit = Contents;
175     %else %if &sysday = Thursday %then %let doit = Univariate;
176     %else %if &sysday = Friday %then %let doit = Summary;
177         &doit data=&ds;
178   run;
179   %mend tasks;
180
181   %tasks (saved.demogius)
MPRINT(TASKS):   proc Means data=saved.demogius;
MPRINT(TASKS):   run;

NOTE: There were 104 observations read from the dataset SAVED.DEMOGIUS.
NOTE: PROCEDURE MEANS used:
      real time           2.46 seconds
```

Mlogic allows the programmer to trace the flow of the macro execution.



```
196   %macro tasks (ds);
197   proc
198     %if &sysday = Monday %then %let doit = Print;
199     %else %if &sysday = Tuesday %then %let doit = Means;
200     %else %if &sysday = Wednesday %then %let doit = Contents;
201     %else %if &sysday = Thursday %then %let doit = Univariate;
202     %else %if &sysday = Friday %then %let doit = Summary;
203         &doit data=&ds;
204   run;
205   %mend tasks;
206
207   %tasks (saved.demogius)
MLOGIC(TASKS):  Beginning execution.
MLOGIC(TASKS):  Parameter DS has value saved.demogius
MLOGIC(TASKS):  %IF condition &sysday = Monday is FALSE
MLOGIC(TASKS):  %IF condition &sysday = Tuesday is TRUE
MLOGIC(TASKS):  %LET (variable name is DOIT)

NOTE: There were 104 observations read from the dataset SAVED.DEMOGIUS.
NOTE: PROCEDURE MEANS used:
      real time           0.11 seconds


MLOGIC(TASKS):  Ending execution.
```

When not developing macros, efficiency considerations suggest turning off the macro debugging options.



```
Command ===>
00001 options nosymbolgen nomlogic noprint;
00002
```

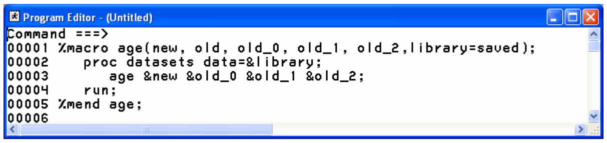
Optional - Variable Numbers of Parameters

Sometimes you may want to write a macro to contain a variable numbers of parameters.

For example, the *%age* macro as defined below can only process five data sets.

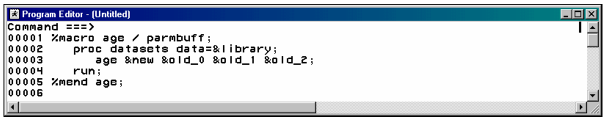
What if we wanted to write a utility macro so we could process any number of data sets?

A way to accomplish this is to use the *Parmbuff* option.



```
%macro age(new, old, old_0, old_1, old_2,library=saved);
   proc datasets data=&library;
      age &new &old_0 &old_1 &old_2;
   run;
%mend age;
```

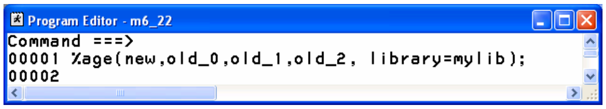
Define the macro in the normal way except for the / PARMBUFF option.



```
%macro age / parmbuff;
   proc datasets data=&library;
      age &new &old_0 &old_1 &old_2;
   run;
%mend age;
```

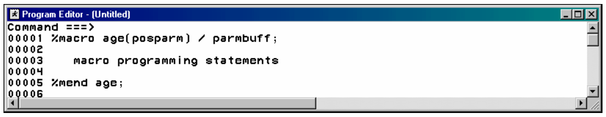
Here, all supplied parameters, including any special characters used, are assigned to the automatic local macro variable *&Syspbuff* which is then manipulated in the macro by macro programming statements.

The call displayed in the code displayed below gives a value to *&syspbuff* of *library=mylib,new,old_0,old_1,old_2.*



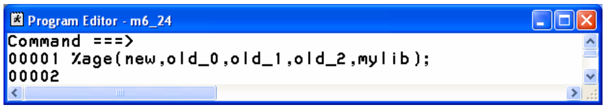
```
%age(new,old_0,old_1,old_2, library=mylib);
```

Parameters may also be included in the definition



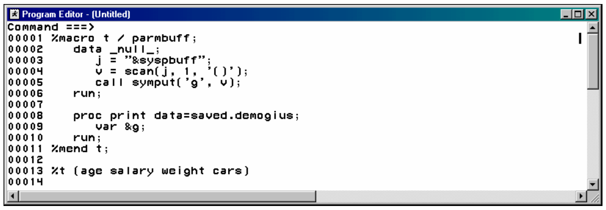
```
%macro age(posparm) / parmbuff;

   macro programming statements

%mend age;
```

In the above example, a different number of parameters can be supplied as long as there is at least one.



```
%age(new,old_0,old_1,old_2,mylib);
```

The call gives a value to *&syspbuff* of *mylib,new,old_0,old_1,old_2* and *&posparm* the value *mylib*.



```
%macro t / parmbuff;
   data _null_;
      j = ”&syspbuff”;
      v = scan(j, 1, ’()’);
      call symput(’g’, v);
   run;

   proc print data=saved.demogius;
      var &g;
   run;
%mend t;

%t (age salary weight cars)
```

Here the value (age salary weight cars) has been passed to *&syspbuff*.

The _null_ data step is required to remove the parentheses from around the variable names.

&Syspbuff is found in a symbol table local to the executing macro.