



Carpenter's Complete Guide to the SAS[®] Macro Language

Second Edition

Art Carpenter



Contents

Acknowledgments	xi
Preface	xiii
About the Author	xv
About This Book	xvii
New in the Macro Language	xxi

Part 1 Macro Basics

Chapter 1 Introduction 3

1.1	Macro Facility Overview	3
1.2	Terminology	4
1.3	Macro Execution Phases	5
1.4	Referencing Environments or Scopes	8
1.5	Chapter Summary	9

Chapter 2 Defining and Using Macro Variables 11

2.1	Naming Macro Variables	12
2.2	Defining Macro Variables	13
2.3	Using Macro Variables	13
2.4	Displaying Macro Variables	16
2.5	Resolving Macro Variables	19
2.5.1	Using the macro variable as a suffix	19
2.5.2	Using the macro variable as a prefix	20
2.5.3	Appending two macro variables to each other	21
2.6	Automatic Macro Variables	22
2.6.1	&SYSDATE, &SYSDATE9, &SYSDAY, and &SYSTIME	22
2.6.2	&SYSLAST and &SYSDSN	23
2.6.3	&SYSERR and &SYSCC	24
2.6.4	&SYSPARM	25
2.6.5	&SYSRC	26
2.6.6	&SYSSITE, &SYSSCP, &SYSSCPL AND &SYSUSERID	27
2.6.7	&SYSMACRONAME	27
2.7	Defining and Using Macro Variables in a PROC SQL Step	27
2.7.1	Counting observations in a data set	28
2.7.2	Building a list of values	28
2.8	Removing Macro Variables	30
2.9	Chapter Summary	30
2.10	Chapter Exercises	31

Chapter 3 Defining and Using Macros 33

- 3.1 Defining a Macro 34
 - 3.1.1 Creating a macro 35
 - 3.1.2 Using a macro to comment a block of code 36
- 3.2 Invoking a Macro 37
- 3.3 System Options Used with the Macro Facility 38
 - 3.3.1 General macro options 38
 - 3.3.2 Debugging options 39
 - 3.3.3 Autocall facility options 41
 - 3.3.4 Compiled stored macros 42
 - 3.3.5 Macro search order 44
 - 3.3.6 Memory control options 44
- 3.4 Display Manager Command-Line Macros 45
- 3.5 Statement- and Command-Style Macros 46
 - 3.5.1 Turning on system options 46
 - 3.5.2 Defining statement- and command-style macros 47
- 3.6 Controlling System Initialization 47
 - 3.6.1 Using AUTOEXEC.SAS 47
 - 3.6.2 Controlling AUTOEXEC execution 49
- 3.7 Chapter Summary 50
- 3.8 Chapter Exercises 50

Chapter 4 Macro Parameters 53

- 4.1 Introducing Macro Parameters 54
- 4.2 Positional Parameters 54
 - 4.2.1 Defining positional parameters 54
 - 4.2.2 Passing positional parameters 55
- 4.3 Keyword Parameters 56
 - 4.3.1 Defining keyword parameters 57
 - 4.3.2 Passing keyword parameters 57
 - 4.3.3 Documenting your macro 58
- 4.4 Choosing between Keyword and Positional Parameters 59
 - 4.4.1 Selecting parameter types 59
 - 4.4.2 Using keyword and positional parameters together 60
- 4.5 Chapter Summary 61
- 4.6 Chapter Exercises 61

Part 2 Using Macros

Chapter 5 Program Control through Macros 65

- 5.1 Macros That Invoke Macros 66
 - 5.1.1 Passing parameters between macros 66
 - 5.1.2 Controlling macro calls 70
 - 5.1.3 Nesting macro definitions 71
- 5.2 Using Conditional Macro Statements 72
 - 5.2.1 Executing macro statements 73
 - 5.2.2 Building SAS code dynamically 74
 - 5.2.3 Using the IN operator 76
- 5.3 Iterative Execution Using Macro Statements 77
 - 5.3.1 %DO block 78
 - 5.3.2 Iterative %DO loops 80
 - 5.3.3 %DO %UNTIL loops 83
 - 5.3.4 %DO %WHILE loops 84
- 5.4 Macro Program Statements 85
 - 5.4.1 Macro comments 86
 - 5.4.2 %GLOBAL and %LOCAL 88
 - 5.4.3 %GOTO and %label 91
 - 5.4.4 Using %SYSEXEC 93
 - 5.4.5 Creating macro %WINDOWS 94
 - 5.4.6 Abnormal termination of macro execution with the %ABORT statement 95
 - 5.4.7 Normal termination of macro execution with the %RETURN statement 96
- 5.5 Chapter Summary 96
- 5.6 Chapter Exercises 97

Chapter 6 Interfacing with Data 101

- 6.1 Using the SYMPUT Routine 102
 - 6.1.1 First argument of SYMPUT 104
 - 6.1.2 Second argument of SYMPUT 104
 - 6.1.3 SYMPUT example 105
 - 6.1.4 Using SYMPUTX 105
- 6.2 Using a SAS Data Set As a Control File 107
 - 6.2.1 Macro variable values 107
 - 6.2.2 Assigning macro variable names as well as values 113
- 6.3 Macro Variable Forms Used in Dynamic Programs 116
 - 6.3.1 Elements of a dynamic program 116
 - 6.3.2 Indirect macro variable references 118
 - 6.3.3 Sources of control information 118

6.4	Moving Text from Macro to DATA Step Variables	119
6.4.1	Assignment and RETAIN statements	119
6.4.2	Using the SYMGET function	120
6.4.3	Using the RESOLVE function	122
6.5	Doing More with the SQL Step	127
6.5.1	Placing a single value into a single macro variable	128
6.5.2	Creating more than one macro variable	128
6.5.3	Placing a list of values into a series of macro variables	129
6.5.4	Using the SQL dictionary tables	132
6.5.5	Automatic SQL generated macro variables	133
6.6	Execution of Macro Code Using CALL EXECUTE	134
6.6.1	Executing non-macro code	135
6.6.2	Executing macro code	135
6.6.3	Timing issues	138
6.7	Chapter Summary	141
6.8	Chapter Exercises	142

Chapter 7 Using Macro Functions 143

7.1	Quoting Functions	145
7.1.1	Using the %BQUOTE function	147
7.1.2	%STR	149
7.1.3	Considerations when quoting	150
7.1.4	Basic types of quoting functions and why we care	155
7.1.5	A bit about the %QUOTE and %NRQUOTE functions	156
7.1.6	%UNQUOTE	156
7.1.7	%SUPERQ	157
7.1.8	Quoting function summary	158
7.1.9	Marking and quoting mismatched symbols with the %STR and %QUOTE functions	159
7.2	Text Functions	160
7.2.1	%INDEX	160
7.2.2	%LENGTH	161
7.2.3	%SCAN and %QSCAN	162
7.2.4	%SUBSTR and %QSUBSTR	164
7.2.5	%UPCASE and %QUPCASE	165
7.3	Evaluation Functions	166
7.3.1	Explicit use of %EVAL	166
7.3.2	Implicit use of %EVAL	167
7.3.3	Using %SYSEVALF	168
7.4	Using DATA Step Functions and Routines	170
7.4.1	Using %SYSCALL	171
7.4.2	Using %SYSFUNC and %QSYSFUNC	172

7.5	Building Your Own Macro Functions	178
7.5.1	Introduction	178
7.5.2	Building the function	179
7.5.3	Using the function	182
7.6	Other Useful Macro Functions	183
7.6.1	One-liners	183
7.6.2	Functions for the DATA step	189
7.6.3	Macro functions with logic	191
7.7	Chapter Summary	197
7.8	Chapter Exercises	197

Chapter 8 Using Macro References with the SAS Component Language (SCL) 199

8.1	The Problem Is...	200
8.2	Using Macro Variables	201
8.2.1	Defining macro variables	202
8.2.2	Macro variables in SUBMIT blocks	202
8.2.3	Using automatic macro variables	203
8.2.4	Passing macro values between SCL entries	204
8.2.5	Using &&VAR&I macro arrays in SCL programs	204
8.3	Calling Macros from within SCL Programs	205
8.3.1	Run-time macros	205
8.3.2	Compile-time macros	205
8.4	Chapter Summary	207

Part 3 Advanced Macro Topics, Utilities, and Examples

Chapter 9 Writing Dynamic Code 211

9.1	Elements of Dynamic Programs	212
9.1.1	Logical branches	213
9.1.2	Iterative step execution	213
9.1.3	Building statements	214
9.2	Writing Applications without Hard-coded Data Dependencies	216
9.2.1	Generalized and controlled repeatability	217
9.2.2	Setting up control files	218
9.2.3	Building macro variables from control files	220
9.3	Using &&VAR&I Constructs as Macro Arrays	221
9.3.1	Resolving &&VAR&i	222
9.3.2	Stepping through a list of data sets	222

9.3.3	Checking for observations with duplicate KEY values	222
9.3.4	Coordinating two macro variable lists	224
9.4	Building SAS Statements Dynamically	227
9.4.1	Using the DATA _NULL_ and %INCLUDE	228
9.4.2	Using the CALL EXECUTE routine	231
9.4.3	Using macro lists rather than macro arrays	233
9.5	Using SASHELP Views	234
9.5.1	Overview of the SASHELP views	235
9.5.2	Using a view	236
9.6	Chapter Summary	237

Chapter 10 Controlling Your Environment 239

10.1	Operating System Operations	240
10.1.1	Copy members of a catalog	240
10.1.2	Write the first <i>N</i> lines of a series of flat files	242
10.1.3	Storing system clock values in macro variables	244
10.1.4	Checking for write access	245
10.1.5	Appending unknown data sets	246
10.1.6	Making a directory	251
10.1.7	Executing a series of SAS programs	252
10.1.8	Using %SYSGET to access system variables	255
10.2	Controlling Your Output	255
10.2.1	Combining titles	256
10.2.2	Renumbering listing pages	260
10.2.3	Coordinating titles (or footnotes)	261
10.3	Adapting Your SAS Environment	262
10.3.1	Maintaining system options	262
10.3.2	Building and maintaining formats	265
10.3.3	Working with libraries and directories	268
10.4	Coordinating with the Output Delivery System (ODS)	269
10.4.1	Why we might need to automate with macros	270
10.4.2	Controlling ODS locations	270
10.4.3	Using ODS to control the destination	271
10.4.4	Graphics devices	271
10.4.5	Using WEBFRAME with GRSEG catalog entries	272
10.4.6	Building an index with PROC PRINT	274
10.4.7	Creating drill-down graphs and charts	275
10.5	Chapter Summary	277

Chapter 11 Working with SAS Data Sets 279

- 11.1 Creating Flat Files 280
 - 11.1.1 Column-specified flat file 280
 - 11.1.2 Creating comma-delimited files 284
- 11.2 Subsetting a SAS Data Set 286
 - 11.2.1 Selection of top percentage using SQL 287
 - 11.2.2 Selection of top percentage using the POINT option 288
 - 11.2.3 Random selection of observations 288
 - 11.2.4 Building a WHERE clause dynamically 292
- 11.3 Checking the Existence of SAS Data Sets 295
- 11.4 Working with Data Set Variables 296
 - 11.4.1 Create a list of variable names from the PDV 296
 - 11.4.2 Create a list of variable names from an ID variable 304
 - 11.4.3 Creating individual macro variables from an existing list 304
 - 11.4.4 Counting words within a macro variable 306
 - 11.4.5 Placing commas between words 308
 - 11.4.6 Quoting words in a list 311
 - 11.4.7 Check for existence of variables 313
 - 11.4.8 Remove repeated words from a list 314
 - 11.4.9 Working with a list of class variables 316
- 11.5 Counting Observations in a Data Set 319
 - 11.5.1 Using %SYSFUNC and ATTRN 320
 - 11.5.2 Controlling observations in PRINT listings 321
 - 11.5.3 Using a DATA _NULL_ step 323
 - 11.5.4 Using SQL 324
- 11.6 Chapter Summary 326

Chapter 12 Building and Using Macro Libraries 327

- 12.1 Library Overview 328
 - 12.1.1 Using %INCLUDE as a macro library 329
 - 12.1.2 Using compiled stored macros 330
 - 12.1.3 Using the autocall facility 331
- 12.2 Macro Library Essentials 332
 - 12.2.1 The macro library search order 332
 - 12.2.2 Establishing a macro library structure and strategy 333
 - 12.2.3 Interactive macro development 334
 - 12.2.4 Modifying the SASAUTOS system variable 335
- 12.3 SAS Autocall Macros 335
 - 12.3.1 %VERIFY 337
 - 12.3.2 %LEFT 338
 - 12.3.3 %CMPRES 339
 - 12.3.4 %LOWCASE 340

- 12.3.5 %TRIM 342
- 12.3.6 %DATATYP 343
- 12.3.7 %COMPSTOR 344
- 12.4 Chapter Summary 345

Chapter 13 Miscellaneous Macro Topics 347

- 13.1 Other Specialized Tasks 348
 - 13.1.1 &&&VAR – Using triple-ampersand macro variables 348
 - 13.1.2 Totals based on a list of macro variables 349
 - 13.1.3 Selecting elements from macro arrays 351
 - 13.1.4 Calculating permutations 352
 - 13.1.5 Checking if a macro variable exists 354
 - 13.1.6 Extending the use of %SYMDEL 355
- 13.2 Doubly Subscripted Macro Arrays 357
 - 13.2.1 Subscript resolution issues 358
 - 13.2.2 Naming row and column indicators 358
 - 13.2.3 Using the &&&VAR&I variable form 361
 - 13.2.4 Using the %SCAN function to identify array elements 363
- 13.3 Programming Smarter 368
 - 13.3.1 Efficiency issues 368
 - 13.3.2 Programming with style 371
 - 13.3.3 Debugging your macros 373
 - 13.3.4 The DATA step versus the macro language 375
 - 13.3.5 Little things with a big bite 380
- 13.4 Working with Macro Parameter Buffers 389
 - 13.4.1 Calling a macro using /PARMBUFF 390
 - 13.4.2 Using the PARMBUFF macro switch 391
- 13.5 Understanding Recursion in the Macro Language 393
- 13.6 Determining Macro Variable Scopes 395
- 13.7 Chapter Summary 397

Appendix 1 Exercise Solutions 399

Appendix 2 Utilities Locator 415

Appendix 3 Example Locator 419

Appendix 4 Macro Locator 425

Appendix 5 Example Data Sets 431

Glossary 439

Bibliography 443

Index 463

① Macro Definition
② Data Step Compile
③ Data Step Execution

Chapter 1

Introduction

- 1.1 Macro Facility Overview 3
- 1.2 Terminology 4
- 1.3 Macro Execution Phases 5
- 1.4 Referencing Environments or Scopes 8
- 1.5 Chapter Summary 9

This chapter introduces you to the fundamentals of the SAS macro language, and it includes an overview and some of the terminology of the language. Because the behavior of macros is different from that of code that is written for Base SAS, sections are also included on macro execution and how SAS sees and uses macros.

SEE ALSO

Stroupe (2003) and First (2003b) both do a nice job of providing an introduction to several of the basic concepts of the macro language.

1.1 Macro Facility Overview

The *SAS Macro Facility* is a tool within Base SAS software that contains the essential elements that enable you to use macros. The macro facility contains a *macro processor* that translates macro code into statements that can be used by SAS, and the macro language. The *macro language* provides the means to communicate with the macro processor.

The macro language consists of its own set of commands, options, syntax, and compiler. While many macro statements have similarities to the statements in the DATA step, you must understand the differences in behavior in order to effectively write and use macros.

The macro language provides tools that allow you to

- pass information between SAS steps
- dynamically create code after the user submits the program for execution
- conditionally execute DATA or PROC steps
- create generalizable and flexible code.

The tools made available through the macro facility include macro (or symbolic) variables, macro statements, and macro functions. These tools are included as part of the SAS code, or program, where they are detected when the code is sent to the SAS Supervisor for execution.

First and foremost, you should always keep in mind that the macro facility is a source code generator. Whether you are substituting the name of a data set or you are having the macro language write a complex DATA step, the macro facility will be writing code. It works with text as input and writes source code as output.

1.2 Terminology

The statement and syntax structure that is used by the macro facility is known as the *macro language* and like any language it has its own terminology. The SAS user who understands the programming language used in Base SAS, however, will discover quickly that much of the syntax and content of the macro language is familiar.

The following terms will be used throughout this book.

text

a collection of characters and symbols that can contain variable names, data set names, SAS statement fragments, complete SAS statements, or even complete DATA and PROC steps. Text forms the primary building blocks used by the macro language.

macro variable

the names of macro variables are almost always preceded by an ampersand (&) when used in SAS code. Macro variables are generally used to store text.

macro program statement

these statements control what actions take place during the macro execution. Like Base SAS language statements, they start with a keyword that in the macro language is always preceded by a percent sign (%), and are often syntactically similar to statements used in the DATA step.

macro facility

the software responsible for interpreting and executing macro language statements and elements.

macro references

when encountered by the SAS parser, these references to the macro language invoke the macro facility. The symbols & and % are used to designate these elements.

macro

also known as a macro program, a macro is a stored collection of macro language statements and text.

macro expression

one or more macro variable names, text, and/or macro functions combined together by one or more operators and/or parentheses. Macro expressions are very analogous to the expressions used in Base SAS programming.

macro function

predefined routines for processing text in macros and macro variables. Many macro functions are similar to functions used in the DATA step

operators

symbols that are used for comparisons, logical operation, or arithmetic calculations. The operators are the same ones used in base language comparisons.

automatic macro variable

special-purpose macro variables. These are automatically defined and provided by SAS. These variable names should be considered as reserved.

open code

SAS program statements that exist outside of any macro definition. Not all macro statements can be used in open code.

resolving macro references

during the resolution process, elements of the macro language (or references) are replaced with text.

When SAS statements are submitted for processing, they are broken up into their component parts so that SAS can understand them. This is done by the *word scanner*. The basic component parts are known as *tokens*. There are several types of these tokens, but two have special meaning to the macro language. These two tokens are the percent sign (%) and the ampersand (&). These tokens are macro processor *triggers*. When the word scanner detects one of these macro triggers (followed by a letter or underscore), the macro processor is invoked. The statement is then turned over to the macro facility for processing.

MORE INFORMATION

You can find additional terminology in the glossary.

SEE ALSO

Burlew (1998, Chapter 2) includes more information on macro language terminology.

1.3 Macro Execution Phases

When you run a SAS program, it is executed in a series of DATA and PROC steps, one step at a time. GLOBAL statements (for example, TITLE, FOOTNOTE, %LET), which can exist outside of these steps, are executed immediately when they are encountered. For each step, SAS first checks to see if macro references exist. *Macro references* may be macro variables, macro statements, macro definitions, or macro calls. If the program does not contain any macro references, then processing continues with the DATA or PROC step processor. If the program does contain macro references, then the macro processor intercepts and resolves them prior to execution. The resolved macro references then become part of the SAS code that is passed to the DATA or PROC step processor.

When code is passed to the SAS supervisor, the following takes place for each step:

- Global statements are executed.
- Macro definitions are compiled and stored until they are called.
- A check is made to see if there are any macro statements, macro variables, or macro calls. If there are, then
 - macro variables are resolved
 - called macros are executed (resolved)
 - macro statements are executed.
- The DATA or PROC step that contains resolved macro references (if there were any) is compiled and executed.

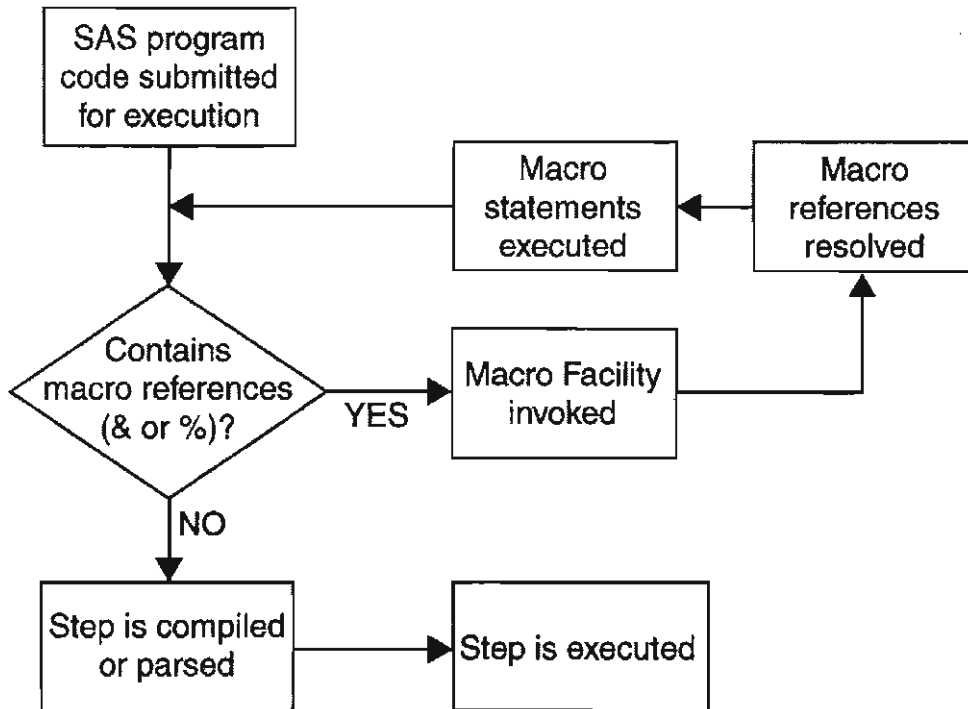
The following diagram is very much a simplification of the process described above. In fact, the SAS parser handles some of these steps in what is essentially a simultaneous manner. For ease of understanding the basic process, however, a simplification is called for, and an understanding of this process is absolutely essential if the programmer is to make full use of the macro language.

The most important information to glean from the diagram is the timing of the execution and resolution of macro language elements relative to Base SAS language elements. Not understanding this relationship causes new users to the macro language to ask questions like

- Can macro %IF statements be used interchangeably with DATA step IF statements?
- In a DATA step why can't I assign the value of a DATA step variable to a macro variable by using the %LET statement?
- Why can't I use a DATA step IF to conditionally execute a %LET?
- Why don't data set variables have values when using them in %IF statements?

Each of these questions can be answered by looking at this diagram. It shows the interaction between the macro and Base SAS language as well as the order that the different types of statements are executed.

The diagram is written as if it applies to each individual step. In fact, the same process applies at the statement and word level as well.



It is important for you to keep in mind that if there are Macro references in your code (% or &), these will be resolved **before** the step is even compiled.

Macro %IFs and DATA step IFs are not interchangeable because the %IF can **never** compare values of variables on the Program Data Vector (PDV). Indeed, the PDV does not yet exist when the %IF is executed. The %IF statement is described in Section 5.2.

For the same reason, you cannot conditionally assign a value to a macro variable using an IF statement and the %LET statement, because the %LET is a macro statement and is therefore executed long before the IF statement is even compiled. The %LET statement is first described in Section 2.2.

MORE INFORMATION

Additional comparisons between the macro language and the DATA step are made in Section 13.3.4.

SEE ALSO

SAS Macro Language: Reference, First Edition (pp. 14–19 and 33–41) and *SAS Macro Language: Reference, Version 8* (pp. 10–16 and 30–36) contain a detailed discussion of how SAS processes statements with macro activity. A very readable and detailed explanation of the internal processes of the macro facility from the SAS developer's point of view is offered by O'Connor (1998).

Jaffee (1999) restates this process in simple terms. Burlew (1998, Chapters 2 and 5) spends all of Chapter 2 and some of Chapter 5 discussing several variations on this series of events. A lot of detail is also included in Chapter 3 of *SAS Macro Language: Reference, First Edition*. First (2001) shows an example of the process as well as the timing of events.

1.4 Referencing Environments or Scopes

Unlike the values of data set variables, the values of macro variables are stored in memory in *symbol tables*. Each macro variable's definition in the symbol table is also associated with a *referencing environment* or *scope*, which is determined by where and how the macro variable is defined. There are two types of environments for macro variables: global and local. The terms "referencing environment" and "scope" are interchangeable, however "scope" is currently the preferred term.

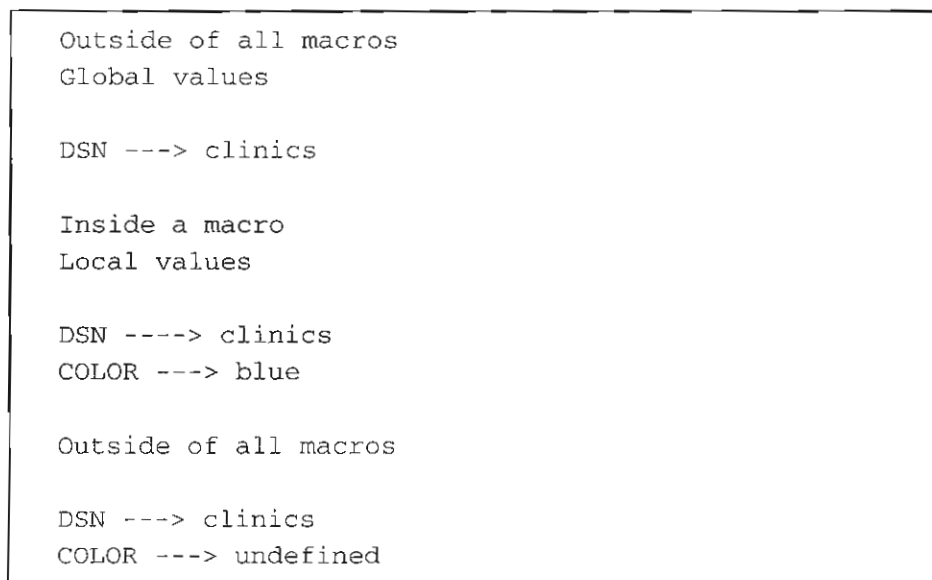
A *global* macro variable has a single value available to all macros within the program. Macro variables that are defined outside of any macro will be global.

Local macro variables have values that are available only within the macro in which they are defined. Since macros can call other macros, there may be multiple levels of nested local tables.

Because each macro creates its own local scope, macro variable values that are defined in one macro may be undefined within another. Indeed, macro variable names need not be unique even among nested macros. This means that the specific value associated with a given macro variable may depend on how the macro variable is used in the program.

When macro calls are nested, their associated local symbol tables are also nested. This means that macro variables known to one macro may also be known within the macros that it calls.

In the following schematic, the macro variable DSN is defined globally and is, therefore, also known inside of the shaded macro. The macro variable COLOR, however, is only defined inside of the shaded macro and is not known outside of the macro.



MORE INFORMATION

You can control the referencing environment for a macro variable through the use of the %GLOBAL and %LOCAL statements, which are described in Section 5.4.2.

SEE ALSO

Extensive examples can be found in *SAS Guide to Macro Processing, Version 6, Second Edition* (pp. 37–54) and the newer *SAS Macro Language: Reference, First Edition* (pp. 50–66).

Papers that specifically cover referencing environments include Bercov (1993) and Hubbell (1990).

An example of a macro variable that takes on more than one value at the same time is given in Carpenter (1996, p. 1637).

1.5 Chapter Summary

You can think of the macro facility as a part of SAS that passively waits to be evoked. If your SAS code contains no macros and no references to macro variables or macro statements, then the macro facility is not used. When the code does contain macro language references, the macro facility wakes up, intercepts the job stream, interprets or executes the macro references, and then releases its control.

The macro facility is made up of two primary components. The *macro processor* provides the ability to compile and execute the *macro language* statements that you use to write macros.



Chapter 2

Defining and Using Macro Variables

- 2.1 Naming Macro Variables 12
- 2.2 Defining Macro Variables 13
- 2.3 Using Macro Variables 13
- 2.4 Displaying Macro Variables 16
- 2.5 Resolving Macro Variables 19
 - 2.5.1 Using the macro variable as a suffix 19
 - 2.5.2 Using the macro variable as a prefix 20
 - 2.5.3 Appending two macro variables to each other 21
- 2.6 Automatic Macro Variables 22
 - 2.6.1 &SYSDATE, &SYSDATE9, &SYSDAY, and &SYSTIME 22
 - 2.6.2 &SYSLAST and &SYSDSN 23
 - 2.6.3 &SYSERR and &SYSCC 24
 - 2.6.4 &SYSPARM 25
 - 2.6.5 &SYSRC 26
 - 2.6.6 &SYSSITE, &SYSSCP, &SYSSCPL AND &SYSUSERID 27
 - 2.6.7 &SYSMACRONAME 27
- 2.7 Defining and Using Macro Variables in a PROC SQL Step 27
 - 2.7.1 Counting observations in a data set 28
 - 2.7.2 Building a list of values 28
- 2.8 Removing Macro Variables 30
- 2.9 Chapter Summary 30
- 2.10 Chapter Exercises 31

For most SAS programmers, their first encounter with the macro language is through the use of macro variables. Indeed, macro variables are very powerful all by themselves. Even if you know nothing else about the macro language other than the information contained in this chapter, you will be able to accomplish a great deal.

This chapter introduces macro variables by showing you how they are named, defined, and used in SAS programs. These symbolic variables can be used as a part of any SAS program. Macros and other macro statements need not be present for you to take advantage of the power of macro variables.

SEE ALSO

An introductory tutorial to various aspects of the macro language can be found in Leighton (1997).

Information about macro variables and their resolution, debugging information, and simple examples can be found in Widawski (1999 and 2002).

2.1 Naming Macro Variables

Macro variables, which are also known as symbolic variables, are not data set variables. Instead, macro variables belong to the SAS macro language, and once they are defined, they can take on many different values during the execution of a SAS program.

You can use the same basic rules to name macro variables as are used to name data set variables:

- A name can be between one and 32 characters in length.
- A name must begin with a letter or underscore (_).
- Any combination of letters, numbers, and underscores can follow the first character.

The following are basic rules that apply to the use of macro variables:

- Text that is stored in macro variables can range in length from 0 to 32K bytes for Version 6, and up to 64K bytes starting in Version 7. The exact number of bytes available is dependent on the host system.
- Reference macro variables inside or outside of a macro by prefixing the macro variable's name with an ampersand (&).
- When referencing macro variables, you can put a period immediately after the name. The period can be used to avoid any confusion that might occur when concatenating text onto the resolved value of the macro variable.
- The macro processor replaces, or substitutes, the name of the symbolic variable with its value.

Another important difference between DATA step variables and macro variables is that there **are** reserved names for macro variables, macro names, and macro labels. As a general rule you should avoid names that have other usages within the macro language. For example, since there is a %EVAL function, do not create a macro named %EVAL. Also, there are a number of automatic macro variables (see Section 2.6) and most of them start with the letters SYS. You should not create a macro variable with the same name as an automatic macro variable, and I recommend that when you name your macro variable you avoid using names that start with SYS altogether.

SEE ALSO

Appendix 2, “Reserved Words in the Macro Facility,” (p. 285) in *SAS Guide to Macro Processing, Second Edition* and Appendix 1, “Reserved Words in the Macro Facility,” in *SAS Macro Language: Reference, First Edition* (p. 273) both provide a full list of reserved names.

2.2 Defining Macro Variables

One of the easiest ways to define a macro variable is through the %LET statement. (Macro language statements always start with a %). This statement works much like an assignment statement in the DATA step.

The %LET statement is followed by the macro variable name, an equal sign (=), and then the text value to be assigned to the macro variable. Notice that quotation marks are not used. Unlike data set variables, macro variables are neither character nor numeric; they always just store text. While learning the macro language, SAS programmers familiar with DATA set variables, might find it easier to think of them as character. Because SAS knows that whatever is to the right of the equal sign is to be assigned to the macro variable, quotes are not needed. Indeed, when they are used they become part of the value that is stored.

The syntax of the %LET statement is

```
%LET macro-variable-name = text-or-text-value;
```

The following statement assigns the text string `clinics` to the macro variable `DSN`:

```
%LET dsn = clinics;
```

If the %LET statement is outside of any macro, its value will be available throughout the entire program, and it is said to be a global macro variable. On the other hand, if the macro variable is defined inside of a macro it may be local, and its value will only be available within that macro. Sections 1.3, 5.4.2, and 13.6 discuss these issues in more detail.

2.3 Using Macro Variables

You could use the following SAS program to determine the contents and general form of the data set `WORK.CLINICS`. It uses `PROC CONTENTS` and `PROC PRINT` (limiting the print to the first ten observations).

```
PROC CONTENTS DATA=CLINICS;
  TITLE 'DATA SET CLINICS';
  RUN;

PROC PRINT DATA=CLINICS (OBS=10);
  RUN;
```

Macro variables are especially useful when you generalize programs. The previous program works for only one data set. If you want to apply it to a different data set, you will need to edit it in three different places. This is trivial in this situation, but edits of existing production programs can be a serious problem in actual applications.

Fortunately, the program can be rewritten and generalized. ❶ The %LET statement defines the macro variable. ❷ A macro variable (&DSN) replaces the data set name. The program becomes

```
%LET DSN = CLINICS; ❶

PROC CONTENTS DATA=&dsn; ❷
  TITLE "DATA SET &dsn"; ❸
RUN;
PROC PRINT DATA=&dsn ❷ (OBS=10);
RUN;
```

To change the data set name, you still need to edit the %LET statement. At least it is now a simpler task. Examples later in the book (see Chapter 4, “Macro Parameters”) show easier and even more general ways of accomplishing this same sort of thing.

Notice that in the rewritten code, quotes in the TITLE statement ❸ were changed from single to double quotes. Macro variables that appear inside of a quoted string will not be resolved unless you use double quotes (“”).

You can change the value of a macro variable simply by issuing a new %LET statement. The most recent definition will be used at any given time.

The period, or dot, can be used to terminate the name of the unresolved macro variable. Although the macro variable name &DSN is interchangeable with &DSN., most macro programmers add the period only when it is needed to minimize confusion.

The macro language does not support the concept of a missing value. Unlike data set variables, macro variables can actually contain nothing. In the macro language this is often referred to as a *null value*. The %LET statement does not store non-embedded blanks, so each of the following pairs of %LET statements will store the same value (in this case the value stored in &NADA is actually nothing — null).

```
%let nada =;
%let nada =      ;

%let dsn =clinics;
%let dsn =      clinics  ;
```

If you do wish to store a blank, as opposed to a null value, then you will need to use a quoting function. These are described in Section 7.1.

The code in the previous PROC CONTENTS/PROC PRINT example might be useful during the debugging phase of program development. For ease of use it would be best if you set up the code so that it could be turned on or off at the flip of a debugging switch. The following code will execute exactly as it did in the previous example, because the macro variable &DEBUG ❶ has been assigned a null value (remember that null values are less than blank values; they truly are nothing).

```
%LET DSN = CLINICS;
%LET DEBUG =; ❶
&DEBUG PROC CONTENTS DATA=&dsn;
&DEBUG    TITLE "DATA SET &dsn";
&DEBUG    RUN;

&DEBUG PROC PRINT DATA=&dsn (OBS=10);
&DEBUG    RUN;
```


In each of these statements &DEBUG resolves to a null value and the statements are executed just as they were in the previous example. It is as if each of the &DEBUG variables were not even there.

```
PROC CONTENTS DATA= CLINICS;
  TITLE "DATA SET CLINICS ";
  RUN;

PROC PRINT DATA= CLINICS (OBS=10);
  RUN;
```

However, when you redefine &DEBUG as it is in the following code ❷, each of the statements becomes an asterisk-style comment:

```
%LET DSN = CLINICS;
%LET DEBUG = *; ❷
&DEBUG PROC CONTENTS DATA=&dsn;
&DEBUG   TITLE "DATA SET &dsn";
&DEBUG   RUN;

&DEBUG PROC PRINT DATA=&dsn (OBS=10);
&DEBUG   RUN;
```

The resolved code becomes:

```
* PROC CONTENTS DATA= CLINICS;
*   TITLE "DATA SET CLINICS ";
*   RUN;

* PROC PRINT DATA= CLINICS (OBS=10);
*   RUN;
```

The /* style comment can also be used in a similar way; however, because of the way that the macro language is interpreted, extra care must be exercised. This time the macro variable &DEBUG will take on the value of "/" when the code is to be commented.

```
%LET DSN = CLINICS;
%LET DEBUG = /;
&DEBUG* *;
  PROC CONTENTS DATA=&dsn;
  TITLE "DATA SET &dsn";
  RUN;
  PROC PRINT DATA=&dsn (OBS=10);
  RUN;
*/ *;
```

When &DEBUG is null (blank or empty), a valid * style comment appears before and after the block of code; otherwise, when &DEBUG is set to a slash (/), as is shown above, the block of code becomes surrounded by a /* --- */ style comment. Notice the use of the asterisk style comments to protect the syntax when &DEBUG is set to null.

This technique would, of course, fail if there were already `/* --- */` style comments within the block of code to be commented (this style of comment cannot be nested). Also changing the example to store `/*` in `&DEBUG`

```
%LET DSN = CLINICS;
%LET DEBUG = /*;
&DEBUG *;
```

would also fail, because the `%LET` statement would not store the anticipated values. (The `/*` symbols have special meaning and are not stored. See Section 7.1).

MORE INFORMATION

For other ways to use macros to comment out blocks of code, see Section 3.1.2.

SEE ALSO

The DATA step debugger is switched on and off using a macro variable described in Riba (2000).

2.4 Displaying Macro Variables

The `%PUT` statement, which is analogous to the DATA step `PUT` statement, writes text and the current values of macro variables to the SAS System LOG. As a macro statement the `%PUT` statement (unlike the `PUT` statement) does not need to be inside of a DATA step. The following two SAS statements comprise a complete (albeit silly) program:

```
%LET dsn = clinics;
```

```
%PUT ***** selected data set is &dsn; write text and current value of &dsn
```

Notice that, unlike the `PUT` statement, the text string is not enclosed in quotes. The quotes are not needed because, unlike in the DATA step, the macro facility does not need to distinguish between variable names and character strings. Everything is a text string, a macro language reference, or some other macro language trigger. The macro language can easily recognize macro variables, for instance, since they are preceded by an ampersand.

Because macro statements are executed before the DATA step statements are even compiled (see the figure in Section 1.3), you may need to get used to their execution order. The following DATA step is also rather silly, but is included to illustrate this point. It contains both a `%PUT` and a `PUT` statement, and both are inside of a DO loop. The associated LOG illustrates the differences between these two statements. The distinction between the `%PUT` and the `PUT` statements is an important one.

```
data _null_;
  do j = 1 to 5;
    put j ' Placed by PUT';
    %put j ' Placed by macro PUT';
  end;
run;
```

The LOG reads as shown:

```

1      data _null_;
2          do j = 1 to 5;
3              put j ' Placed by PUT';
4              %put j ' Placed by macro PUT';
5          end;
6      run;

1 Placed by PUT
2 Placed by PUT
3 Placed by PUT
4 Placed by PUT
5 Placed by PUT
NOTE: The DATA statement used 1.26 seconds.

```

- ❶ Notice that the %PUT statement is executed (and writes to the LOG) as soon as it is encountered, and because there are quotes in the statement they are also displayed. Also, %PUT does not recognize the *j* as a variable name so it just includes it as part of the rest of the text that is to be displayed.

There are several options that can be used on the %PUT statement. When the following options are used, without specifying other text, information about your symbol tables is written to the LOG. These include:

all

lists all macro variables in all referencing environments.

automatic

lists all of the macro variables that are automatically defined by SAS. The variables can vary by site, operating system, and version of SAS. Automatic macro variables are described further in Section 2.6, “Automatic Macro Variables.”

global

lists user-created macro variables that will be available in all of the referencing environments.

local

lists user-defined macro variables that are available only in the current or local referencing environment. This option must be used from within a macro.

user

lists all of the user-created macro variables in each of the referencing environments. This option can be especially useful during the debugging process for complicated macros.

When any of these options are used with the %PUT, the LOG will indicate the macro variable's symbol table❷, its name❸, and its value❹.

The next two statements create the LOG that follows:

```
%LET dsn = clinics;
%PUT _all_;
```

```
1      %let dsn = clinics;
2      %put _all_;
GLOBAL② DSN clinics
AUTOMATIC AFDSID 0
AUTOMATIC AFDSNAME
AUTOMATIC AFLIB
AUTOMATIC AFSTR1
AUTOMATIC AFSTR2
AUTOMATIC FSPBDV
AUTOMATIC SYSBUFFR
AUTOMATIC SYSCMD
AUTOMATIC SYSDATE③ 16JAN97
AUTOMATIC SYSDAY Thursday④
AUTOMATIC SYSDEVIC
AUTOMATIC SYSDSN          _NULL_
AUTOMATIC SYSENV FORE
AUTOMATIC SYSERR 0
AUTOMATIC SYSFILRC 0
AUTOMATIC SYSINDEX 0
AUTOMATIC SYSINFO 0
AUTOMATIC SYSJOBID 0000008671
AUTOMATIC SYSLAST _NULL_
AUTOMATIC SYSLCKRC 0
AUTOMATIC SYSLIBRC 0
AUTOMATIC SYSMENV S
AUTOMATIC SYSMSG
AUTOMATIC SYSPARM
AUTOMATIC SYSPBUFF
AUTOMATIC SYSRC 0
AUTOMATIC SYSSCP WIN
AUTOMATIC SYSSCPL WIN_32S
AUTOMATIC SYSSITE 0028569001
AUTOMATIC SYSTIME 19:07
AUTOMATIC SYSVER 6.11
AUTOMATIC SYSVLONG 6.11.0020P092795
```

Messages that mimic those written to the LOG by the Base language can also be generated by using the %PUT statement. When the %PUT is followed by ERROR:, WARNING:, or NOTE: the text associated with the %PUT will be written to the LOG in the color appropriate to that

message. Under the default settings in an interactive environment, the following %PUT statement generates a red error message in the log:

```
%PUT ERROR: Files were not copied as expected.;
```

The keywords must be capitalized, must immediately follow the %PUT statement, and must be immediately followed by a colon.

SEE ALSO

The additional options for the %PUT statement are described fully in Chapter 4, “SAS Macro Language,” (pp. 95–98) in *SAS Software Changes and Enhancements, Release 6.11*, and in *SAS Macro Language Reference, First Edition*, p. 204.

2.5 Resolving Macro Variables

Prior to the execution of the SAS code, macro variables are resolved. The resolved values are then substituted back into the code. It is, therefore, important to understand the rules associated with how macro variables are resolved.

The use of single macro variables, as shown in the following example, is fairly straightforward:

```
%LET SEX=MALE;
DATA &SEX;
  SET CLINICS;
  WHERE SEX="&SEX";
RUN;
```

This code will resolve to

```
DATA MALE;
  SET CLINICS;
  WHERE SEX="MALE";
RUN;
```

Remember that when the resolved value of a macro variable is to be within quotes (as in the above WHERE statement), you must use double, not single, quotes.

It is when you start combining macro variables with text and other macro variables that the fun begins. Macro variables can be concatenated to text or even to other macro variables.

2.5.1 Using the macro variable as a suffix

You can append a macro variable to SAS code that includes variables, text strings, and data set names. When resolved, the value of the macro variable is concatenated to the string that precedes it. In the macro language there is no concatenation operator. It is unnecessary, because there are only text and macro language elements. Once the macro language elements have been resolved, there is only text.

The following code contains the macro variable &SEX. In the second line, &SEX is appended to ONLY when resolved; ONLY&SEX becomes ONLYMALE.

```
%LET SEX=MALE;
DATA ONLY&SEX;
  SET CLINICS;
  WHERE SEX="&SEX";
RUN;
```

This code resolves to

```
DATA ONLYMALE;
  SET CLINICS;
  WHERE SEX="MALE";
RUN;
```

Because the macro facility can easily determine by the & where the macro variable name starts, there is no confusion when the macro variable is used as a suffix. This is not always true when using the macro variable as a prefix.

2.5.2 Using the macro variable as a prefix

A macro variable may also precede portions of SAS code. Since there are no concatenation operators in the macro language, text that is to be appended to the end of the resolved value of a macro variable can become confused with the macro variable name. To separate a macro variable name from the text that follows it, a macro variable reference can be followed by a period to designate the end of the variable name. In this example the period in the SET statement is necessary to avoid ambiguity:

```
%LET DSN=CLINICS;
%LET DSN1=OLDDATA;
DATA &DSN;
  SET &DSN.1 &DSN1;
RUN;
```

This resolves to

```
DATA CLINICS;
  SET CLINICS1 OLDDATA;
RUN;
```

It is important to note the difference between the &DSN.1 and &DSN1 macro variables. Any macro variable name (reference) can be followed by a period, and when present it can act as a delimiter. The period in &DSN.1 causes &DSN to be resolved to CLINICS with a 1 appended to the end. The period is seen as a delimiter that separates the variable name from the text that is to be appended to the resolved value of the macro variable. Without the period the 1 is seen as part of the macro variable name and &DSN1 is resolved to OLDDATA.

Sometimes the first character of the string that is to be appended to the macro variable is a period. As is shown in the previous code, the period will be used to concatenate the string and will not appear in the resolved text. To get around this you can use a double period (..) when a single period (.) is desired in the text, as shown:

```
%LET LIBREF=SASCLASS;
DATA &LIBREF..CLINICS;
  ...code not shown...
RUN;
```

This code resolves to

```
DATA SASCLASS.CLINICS;
    ...code not shown...
RUN;
```

The first period is seen as part of the macro variable name (&LIBREF.), and the second is just a character in the text (.CLINICS) that is appended to the resolved value of the macro variable.

2.5.3 Appending two macro variables to each other

You can join more than one macro variable to form a single result. Consider, for example, the following three macro variable definitions:

```
%LET DSN=CLINICS;
%LET N=5;
%LET DSN5=FRED;
```

Using the rules that are discussed in the previous section, various combinations of these macro variables will resolve as follows:

Combination	Resolves to
&DSN&N	CLINICS5
&DSN.&N	CLINICS5
&DSN..&N	CLINICS.5

The macro processor scans for macro variables and resolves them as encountered. In each of the previous macro variable combinations the resolution is possible in a single pass. When two or more ampersands (&) appear next to each other, successive passes or scans are required to make the final resolution. You can think of the double ampersand (&&) as a special reference that resolves to a single ampersand. This is demonstrated in the following combinations:

Combination	First Scan Resolves to	Second Scan Resolves to
&&DSN&N	&DSN5	FRED
&&&DSN&N	&CLINICS5	&CLINICS5

The macro variable reference &CLINICS5 does not exist and the following message is written to the LOG:

```
WARNING: Apparent symbolic reference CLINICS5 not resolved.
```

A common mistake is to assume that the resolution process proceeds from right to left as

```
INCORRECT:  &&DSN&N => &CLINICS5
```

This is incorrect because the && must be resolved to a single & before the &DSN is resolved. The correct resolution is

```
CORRECT:  &&&DSN&N => &DSN5 => FRED
```

For the same reasons, &&&DSN&N is taken as && &DSN &N, which resolves to &CLINICS5. This macro variable does not exist on the symbol table and therefore remains unresolved, and a warning is written to the LOG.

Using multiple ampersands establishes the ability to create a type of vector or array of macro variables. Examples of the use of this notation are shown in Section 5.3 and various other sections in this book.

SEE ALSO

Several examples and variations of the process of resolving macro variables can be found in Yindra (1998).

2.6 Automatic Macro Variables

Several macro variables are automatically created for you by the macro processor. You can use these variables as you would any other macro variable. Some of the more commonly used automatic variables are shown in the next sections.

The list of available automatic macro variables can vary quite a bit depending on your release of SAS and your operating environment.

Although some automatic macro variables are write-protected and cannot be modified by the user, others can be changed. As a general rule be cautious about modifying the values of any automatic macro variables. I am not saying do not do it, but just to be careful.

SEE ALSO

The *SAS Macro Language: Reference, First Edition* (pp. 22–23 and 156–157) and *SAS Macro Language: Reference, Version 8* (pp. 18–20 and 152–153) each have two tables listing many of the automatic variables.

A list and description of selected automatic macro variables is provided by First (2001b).

2.6.1 &SYSDATE, &SYSDATE9, &SYSDAY, and &SYSTIME

At the start of the current SAS session (for batch execution this may also be known as a “job”), the following four automatic macro variables are loaded; they note the day, date, and time of the start of the session:

SYSDATE	date that the session began executing (DATE7. form).
SYSDATE9	date that the session began executing displayed with a four-digit year (DATE9. form).
SYSDAY	day of the week that the session began executing.
SYSTIME	time of the day that the SAS session began executing (TIME8. form).

The following program converts the date (DATADATE) and time (DATATIME) for the second observation (BATCH=2) in the data set OLD to new values based on the values contained in the &SYSDATE and &SYSTIME macro variables.


```

data old;
  do batch=1 to 3;
    conc=2;
    datadate='02jan04'd;
    datatime = '09:00't;
    output;
  end;
  format datadate mmddyy10. datatime time5.;
run;

* Add time stamp when updating data;
data new;
  set old;
  if batch = 2 then do;
    conc=2.5;
    datadate="&sysdate"d;
    datatime="&systime"t;
  end;
run;

proc print data=new;
  title1 'Drug concentration';
  title2 "Mod date &sysdate";
run;

```

Notice that the &SYSDATE and &SYSTIME values are converted from text values to SAS date and time values by treating them as date/time constants. The resulting output is shown:

Drug concentration				
Mod date 05FEB97				
OBS	BATCH	CONC	DATADATE	DATATIME
1	1	2.0	01/02/2004	9:00
2	2	2.5	02/05/2004	15:30
3	3	2.0	01/02/2004	9:00

MORE INFORMATION

The use of macro variables in DATA step assignment statements is discussed further in Section 6.4.1.

2.6.2 &SYSLAST and &SYSDSN

&SYSLAST stores the name of the last data set that was created (or re-created). You can use this variable as you would use any data set name. This variable is useful when you are generating dynamic code, and you do not necessarily know the name of the data set that was just created.

&SYSDSN also stores the name of the last modified data set; however, the name is stored in two words with the library first.

SYSLAST name of the last SAS data set created with the library and data set name separated with a period (.).

SYSDSN name of the last SAS data set created with the library and data set name separated with at least one space.

If no data sets have been created in the current SAS session, the macro variables will contain the value `_NULL_`.

Assuming that the following PROC PRINT follows the code used in the previous example, it will produce the listing shown:

```
proc print data=&syslast;  
  title1 'Drug concentration';  
  title2 "Listing of &syslast";  
run;
```

Notice that the name of the data set stored in `&SYSLAST` includes the associated *libref*.

Drug concentration Listing of WORK.NEW				
OBS	BATCH	CONC	DATADATE	DATATIME
1	1	2.0	01/02/2004	9:00
2	2	2.5	02/05/2004	15:30
3	3	2.0	01/02/2004	9:00

2.6.3 &SYSERR and &SYSCC

It is sometimes handy to be able to determine if a particular PROC or DATA step executed successfully. Each step has a return code that measures the existence and severity of any errors. The return code is stored in `&SYSERR`, where it can be checked during the execution of the job.

SYSERR stores the return codes of PROC and DATA steps.

Since each step determines a new value for `&SYSERR`, it is reset at each step boundary.

SYSCC stores the overall session return code.

The system condition code is not reset at step boundaries and can be used to determine the error conditions across steps.

The various steps (procedures and DATA steps) do not all return the same set of values for `&SYSERR`. The value assigned to `&SYSERR` depends both on the type of error and the type of step. You may need to do some experimenting to determine the values returned for the step you are interested in. As a general rule successful steps assign a 0 to `&SYSERR`.

The following program fragment copies the data sets in one library (COMBINE) to another (COMBTEMP). It uses &SYSERR to detect if another user currently has write access to a data set in the library COMBINE (&SYSERR will take on a value of 5 or greater).

```
* Copy the current version of the COMBINE files
* to COMBTEMP;
proc datasets memtype=data;
    copy in=combine out=combtemp;
quit;
%put SYSERR is &syserr;
```

MORE INFORMATION

Section 10.1.4 shows a more sophisticated version of this example.

SEE ALSO

Beverly (2001) also discusses the automatic macro variable &SYSERR.

2.6.4 &SYSPARM

Usually used in the batch environment, this macro variable accesses the same value as is stored in the SYSPARM= system option and can also be retrieved using the SYSPARM() DATA step function. This option is most useful when its value is loaded when SAS is initially executed, and you may use it to pass a value into a program through the JCL or batch calling routines.

SYSPARM specifies a character string that can be passed into SAS programs.

The length of the character string is operating system dependent, but is fairly large (32k characters under Windows).

In the following example, you would like your programs to automatically direct your data to either a test or production library. To make this switch, assign &SYSPARM the value TST or PROD when you start the SAS session.

Assume that a VAX SAS session is initiated with

```
$ sas/sysparm=tst
```

A typical LIBNAME statement on VAX might be

```
libname projdat "usernode:[study03.gx&sysparm]";
```

The resolved LIBNAME statement becomes

```
libname projdat "usernode:[study03.gxtst]";
```

The syntax that you use to load a value into &SYSPARM depends on the operating environment that you are using. See the SAS Companion for your operating environment for more information. On Windows, `-sysparm tst` appears on the TARGET LINE in the Properties Window. In JCL, the option is used on the SYSIN card.

The DATA step function SYSPARM() can be used to retrieve the value of &SYSPARM. This function, however, returns the value &SYSPARM without resolving any embedded macro variable references. The differences between using &SYSPARM directly and the SYSPARM() function are demonstrated in the following example. Notice in this example that the value of &SYSPARM contains a macro variable reference.

Initialize SAS using the -SYSPARM option.

```
"C:\Sas\V8\Sas.exe" -sysparm &aaa; ❶
```

Once initialized, &SYSPARM can be used throughout the session.

```
%let aaa = AAAAA;
data try2;
a = "&sysparm"; ❷
b = sysparm(); ❸
run;
```

- ❶ Usually, as in this example, the value for &SYSPARM is set when SAS is first invoked. Since at this point the code has not even been sent to the word scanner, the macro processor is not called, and therefore, no attempt is made to resolve &AAA. As a result, &SYSPARM contains the characters &aaa.
- ❷ In the data set TRY2 the variable A is a character variable, which has a length of 5, and contains the value "AAAAA". Before a value can be assigned to the variable A, &SYSPARM is first resolved to &aaa. This is in turn resolved to AAAAA, and it is this value that is then stored in the data set variable A.
- ❸ While the variable B is also a character variable, it has a length of 200 (the default when using the SYSPARM function). Since the SYSPARM() function is executed during the DATA step execution phase, the value "&aaa" will be written directly to the variable B and no attempt will be made to resolve the macro reference.

If &SYSPARM contains more than 200 characters be sure to use the LENGTH statement to set the length of the variable created by the SYSPARM function, otherwise longer values will be truncated.

CAVEAT: Not all operating systems are the same. Consult your SAS Companion for details or limitations on the number of characters that can be passed into &SYSPARM.

MORE INFORMATION

Section 13.4 discusses the use of macro buffers.

SEE ALSO

Johnson (2001) has an example that parses several words out of a single &SYSPARM value. Wong (2002) shows examples of both the SYSPARM macro variable and the SYSPARM() function.

2.6.5 &SYSRC

The SYSRC macro variable captures the last return code from system operations that are executed following an X command, X statement, or the %SYSEXEC macro statement.

SYSRC indicates the last return code from your operating environment.

The value will be an integer, but it will vary according to the operating environment. Successful operations will not always return a value of 0. The SAS Companion for your operating system has additional information regarding what values can be returned to &SYSRC.

2.6.6 &SYSSITE, &SYSSCP, &SYSSCPL, and &SYSUSERID

These macro variables contain information about your site and operating system. Primarily, you can use them when a particular SAS job or application may be executed in multiple environments and its behavior needs to be altered accordingly.

SYSSITE	contains the current site number.
SYSSCP	gives the name of the host operating environment.
SYSSCPL	on some operating environments &SYSSCPL can be more specific than &SYSSCP.
SYSUSERID	for operating systems that require a login user ID, the user ID is placed in this macro variable. &SYSUSERID receives the value “default” when the operating system has not captured the user identification.

The following %PUT statement

```
%put &syssite &sysscp &sysscpl;
```

produces the following statement in the LOG when executed under the Windows ME operating environment.

```
13      %put &syssite &sysscp &sysscpl;
0053893001 WIN WIN_ME
```

SEE ALSO

&SYSSITE and &SYSSCPL are documented in *SAS Software: Changes and Enhancements, Release 6.10* (p. 36) and in *SAS Macro Language: Reference, First Edition* (pp. 257–260).

Davis (1997) and Hessel (1998) both include examples that use &SYSSCP.

2.6.7 &SYSMACRONAME

When a macro is executing, &SYSMACRONAME will contain the name of that macro. This macro variable can be very useful when you would like to document the progress of your application. If &SYSMACRONAME is used in open code (outside of any macro definitions), it will have a null value, and when macros are nested, &SYSMACRONAME will contain the name of the inner most macro.

2.7 Defining and Using Macro Variables in a PROC SQL Step

You can use macros and macro variables within a PROC SQL step in much the same way as in other PROC steps. The exception is the syntax related to creating macro variables within SQL.

Unlike most other procedures, statements within an SQL step are processed sequentially and are applied immediately. Furthermore, SQL statements are handled differently than the statements of other procedures and are expected to conform to specific syntax standards.

MORE INFORMATION

Section 6.5 provides additional examples of the use of macro variables with SQL.

SEE ALSO

Tassoni (1997) uses a macro to write SQL statements and Palmer (1997) uses macro variables in a PROC SQL step.

An introduction to and additional detail on the topic of the INTO: can be found in Zdeb (1999b) and Satchi (2000 and 2001).

Examples of SQL steps that create a series of macro variables can be found in Eddlestone (1997), Satchi (2000), and Casas (2002).

First (2001b), as well as Rajecki and Kahle (2000), both create a list of macro variables with a single INTO:.

2.7.1 Counting observations in a data set

The following example uses SQL to count the number of observations that contain a specified string in the table column CLINNAME. The string is placed in a macro variable (&CLN) and the SQL COUNT function is used to count the observations that match the WHERE clause.

```
%let cln = Beth;
proc sql noprint;
  select count(*)
    into :nobs ❶
      from clinics (where=(clinname="&cln")); ❸
quit;
%put number of clinics for &cln is &nobs; ❷
```

- ❶ The INTO clause is used to create the new macro variable NOBS. The colon informs the SELECT statement that the result of the COUNT function is to be written into a macro variable.
- ❷ Once created, the new macro variable is used in the same way as any other macro variable. Both macro variables are preceded by an ampersand in the %PUT statement.
- ❸ Notice that the colon in the FROM clause is used as a character comparison operator just as it is in the DATA step. SAS will not be confused by these two very different uses of the colon.

Within SQL, macro variables are also known as environmental variables. When being assigned values the name of the macro variable will follow an INTO and will be preceded by a colon. As is shown in this example, when macro variables are to be used within a SQL step the macro variable is preceded by the ampersand.

MORE INFORMATION

The example in Section 11.5.4 places this SQL step inside of a macro. More complex examples of SQL steps can be found in Sections 6.5 and 11.4.9.

2.7.2 Building a list of values

It is often useful to be able to build a macro variable that contains a list of values. For instance, the example in Section 11.4.9 creates a list of variable names. You may also wish to build a list that can be used with the IN operator, perhaps in a WHERE statement.

In the following example we want to create a subset of the student body (SCHOOL) based on the names in a particular class (CLASS):

```
data class;
  input @3 name $ 8. grade $1.;
  cards;
  Billy    B
  Jon      C
  Sally    A
  run;

data school;
  input @3 name $ 8. gradcode $1.;
  cards;
  Billy    Y
  Frank    Y
  Jon      N
  Laura    Y
  Sally    Y
  run;

proc sql noprint;
  select quote(name)
    into :clnames separated by ' ' ❶
    from class;
  quit;

data clasgrad;
  set school (where=(name in(&clnames))); ❷
  run;

proc print data=clasgrad;
  title 'Class Graduate Status';
  run;
```

- ❶ The macro variable &CLNAMES contains the quoted list of names found in the CLASS data set.
- ❷ The list of variable names stored in &CLNAMES is used with the IN operator.

The PROC PRINT creates the following output:

Class Graduate Status		
OBS	NAME	GRADCODE
1	Billy	Y
2	Jon	N
3	Sally	Y

SEE ALSO

A similar example can be found in *SAS Communications*, 1Qtr., 1997, page 48. Widawski (1997a) and White (2001) both create a list of file names using SQL and each places the list in a macro variable.

2.8 Removing Macro Variables

As you work with macro variables you may on occasion want to remove them from the **GLOBAL** symbol table. You can use the **%LET** statement to reset the value to null; however, that does not remove the variable. Fortunately, the macro statement **%SYMDEL**, which was new in Version 8, can be used to totally remove the global macro variable.

Syntax

```
%SYMDEL macro_variable_list;
```

In the following example the macro variables **&NADA** and **&DSN** are removed from the symbol table:

```
%symdel nada dsn;
```

Notice that the macro variable names **do not** include the ampersands. This statement expects macro variable names. If you used an ampersand, the resolved value would also need to be the name of a macro variable.

MORE INFORMATION

Interesting things can happen when **%SYMDEL** is used in ways other than specified above. See Section 13.1.6.

SEE ALSO

First (2001a) uses **%SYMDEL** to delete a list of macro variables.

2.9 Chapter Summary

Macro variables (sometimes also called symbolic variables) are very different from SAS data set variables. In the **DATA** step, data variables reside as part of the Program Data Vector. Macro variables are stored in a symbol table, are independent of all data sets, and do not depend on either the data set or the observation being processed.

You can use the **%LET** statement to define a macro variable. Macro variables can also be created and used within a **PROC SQL** step.

Macro variables that appear inside of a quoted string will not be resolved unless you use double quotes (" ").

Within a referencing environment, you can change the value of a macro variable at any time by issuing a new **%LET** statement. The most recent definition will be used at any given time.

You can use the **%PUT** statement to write text and macro variable values to the SAS System LOG.

SAS defines several macro variables automatically for the user. You can access these macro variables and use them in the same way that you can use any other macro variable. Because of possible conflicts with automatic macro variables, you should never create a macro variable with a name starting with the letters **SYS**.

2.10 Chapter Exercises

```
*****;
* The data set, SASCLASS.CLINICS, contains 80  *;
* observations and 20 variables. The following *;
* program will be used to complete the first  *;
* exercise in this chapter.                    *;
*****;

PROC PLOT DATA=SASCLASS.CLINICS;
  PLOT EDU * DOB;
  TITLE1 "YEARS OF EDUCATION COMPARED TO BIRTH DATE";
RUN;

PROC CHART DATA=SASCLASS.CLINICS;
  VBAR WT / SUMVAR=HT TYPE=MEAN;
  TITLE1 "AVERAGE HEIGHT FOR WEIGHT GROUPS";
RUN;
```

1. Rewrite the sample program that is presented above by adding at least one macro variable (%LET).
2. (True/False) Macro variables will not be resolved unless you use single quotation marks (').
3. Given the following macro variable definitions, how will the macro variable combinations be resolved? Take a guess first, and then use %PUT to check your answer.

%let dsn=clinic;	%let I = 3;	%let b = dsn;
%let lib = sasuser;	%let dsn3 = studydrq;	
&lib&dsn --> _____	&dsn&I --> _____	
&lib.&dsn --> _____	&&dsn&I --> _____	
&lib..&dsn --> _____	&dsn.&I --> _____	
&&b --> _____	&&b --> _____	

Extra Credit: Using the above macro variable definitions, what combination of &dsn and &I resolves to CLINIC.STUDYDRG? Use a %PUT statement to show your answers in the LOG.

4. What are automatic macro variables?
5. What is the purpose of &SYSPARM?



Chapter 3

Defining and Using Macros

- 3.1 Defining a Macro 34
 - 3.1.1 Creating a macro 35
 - 3.1.2 Using a macro to comment a block of code 36
- 3.2 Invoking a Macro 37
- 3.3 System Options Used with the Macro Facility 38
 - 3.3.1 General macro options 38
 - 3.3.2 Debugging options 39
 - 3.3.3 Autocall facility options 41
 - 3.3.4 Compiled stored macros 42
 - 3.3.5 Macro search order 44
 - 3.3.6 Memory control options 44
- 3.4 Display Manager Command-Line Macros 45
- 3.5 Statement- and Command-Style Macros 46
 - 3.5.1 Turning on system options 46
 - 3.5.2 Defining statement- and command-style macros 47
- 3.6 Controlling System Initialization 47
 - 3.6.1 Using AUTOEXEC.SAS 47
 - 3.6.2 Controlling AUTOEXEC execution 49
- 3.7 Chapter Summary 50
- 3.8 Chapter Exercises 50

This chapter introduces several of the simplest methods that you can use to create and use macros. Three types of macros are discussed as are several system options that can be used with macros. Several advantages and disadvantages of these types of macros are included.

SEE ALSO

Introductions and overviews to the macro language can be found in Cohen (1998), Jaffee (1999), First (2001a and 2001b), Williams (2001), and Whitlock (1999a and 2001a).

3.1 Defining a Macro

All macros are created using the two macro language statements; %MACRO and %MEND. Like the DO and END statements in the DATA step, these two macro statements always come in pairs.

Syntax

```
%MACRO macro-name;
... macro text ...
%MEND <macro-name>;
```

Every macro definition begins with a macro statement (%MACRO), which must contain a name for the macro. The %MEND statement closes the macro definition. This statement can optionally also include the name of the macro (optional, but a very good idea) for documentation reasons.

The macro text can include the following:

- constant text
- macro variables
- macro program statements
- macro expressions
- macro functions.

Constant text

The macro language treats constant text in much the same way as character strings are treated in the DATA step. Constant text is not evaluated, resolved, or even examined by the macro processor. It is just passed along. Constant text can include

- SAS data set names
- SAS variable names
- SAS statements
- SAS steps (like DATA and PROC steps)
- complete and partial SAS programs
- any combination of the above.

Macro variables

Macro variables are preceded in the macro text by an ampersand (&). Chapter 2, “Defining and Using Macro Variables,” includes a more complete discussion of the definition and use of macro variables.

Macro program statements

These statements are evaluated and executed when the macro is called. Many macro program statements must be contained inside of a macro and cannot exist in open code. Macro program statements, like DATA step statements, start with a keyword. Unlike

other SAS statements, however, macro program statement keywords are preceded with a percent sign (%). Sample macro program statements include

```
%let dsn = clinics;

%do i = 1 %to &yr;
```

Chapter 5, “Program Control through Macros,” introduces a number of these statements.

Macro expressions

Macro expressions perform the same tasks in the macro language as DATA step expressions do in Base SAS. The difference, of course, is that the macro expressions involve the evaluation and assignment of macro variables rather than DATA step variables. They can be used to determine conditional branches in the processing flow of your program, and to create new value assignments. The use of macro expressions in conditional processing is described in Section 5.2. Most conditional processing is accomplished with the %IF statement, which can be used to make decisions based on macro expressions. Two example %IF statements are shown here:

```
%if &nobs = 0 %then %do;

%if &cond = bad %then %badobs;
```

Macro functions

Macro functions operate on macro variables and constant text. Some of the more useful character functions available in the DATA step have analogous macro functions. There are also a number of macro functions that are unique to the macro language. Macro functions are described in Chapter 7, “Using Macro Functions.” The %UPCASE function shown here is an example of a macro function:

```
%let upper = %upcase(&name);
```

Many DATA step functions can double as macro functions through the use of the %SYSFUNC macro function that is described in Section 7.4.2.

3.1.1 Creating a macro

The easiest way to create a macro is to surround existing code with the %MACRO and %MEND statements. Remember that *every* macro definition must begin and end with these two statements; they come in pairs.

The following code creates a macro that looks at a data set. An existing program that contains a PROC CONTENTS and a PROC PRINT has been enclosed by the %MACRO and %MEND statements.

```
%LET DSN = CLINICS;

%MACRO LOOK;

PROC CONTENTS DATA=&dsn;
  TITLE "DATA SET &dsn";
RUN;

PROC PRINT DATA=&dsn (OBS=10);
  RUN;

%MEND LOOK;
```

When this pair of statements is encountered, the macro facility processes the enclosed statements and text. Even though this process is often referred to as macro compilation, non-macro text is not processed and is not compiled at this time. Syntax errors in macro statements are detected during this processing phase; however, syntax errors and any other problems with the non-macro code inside of the macro will not be detected until the macro is actually executed (see Section 3.2). After successful compilation the macro definition is stored as an entry in the SASMACR catalog. By default this catalog resides in the WORK library.

If your macro contains syntax errors in the macro code, your macro will not compile and will not be usable. You might see a message similar to the following in your LOG:

```
ERROR: A dummy macro will be compiled.
```

Remember that even though macros are processed by the macro facility when the %MACRO and %MEND statements are encountered, code that is enclosed in these two statements will not be executed until the macro is called (see Section 3.2).

3.1.2 Using a macro to comment a block of code

When evaluating SAS programs you may occasionally find blocks of code that have been effectively commented out by enclosing them with %MACRO and %MEND statements. In effect, a macro is defined, but it is never called. There is, however, a penalty associated with the use of this technique. This section will not only show you how to use this technique, but it will also help you to recognize the problem and to understand the issues associated with it.

Although the contents of a macro are processed when the definition of the macro is encountered, the macro is not executed until the macro is actually called. This means that you can comment out blocks of code by creating macros that you purposely do not call. This style of comment can even contain embedded comments that are defined by /* */.

The macro %COMMENT in the following code causes the second DATA step to be ignored:

```
DATA invert.SPECIE;
  INFILE 'SPECIES.mas';
  LENGTH SPCODE $ 5 SPNAME $ 40;
  INPUT  SPCODE 21-25 SPNAME;
  RUN;

%macro comment;
* Create the POSITION data set;
DATA DBMASTER.POSITION;
  INFILE POS;
  LENGTH POS $ 2;
  INPUT POS 1-2 /*COORD 4-8*/;
  FILE ERRS;
  IF _ERROR_ THEN PUT '-1';
  RUN;
%mend comment;
```

The following example is a more common application of this type of macro. Here, the macro is used to comment out debugging steps:

```
data new;
  set big;
  ...code not shown...
run;
```

```
%macro debugnew;
  proc print data=new (obs=5);
    title 'listing for NEW';
  run;
%mend debugnew;
```

During the debugging process, %DEBUGNEW can be executed to view the contents of the data set NEW.

As was mentioned earlier, this technique is not without its downside. The macros %COMMENT and %DEBUGNEW will be read, compiled, and stored in the SASMACR catalog for every run of the job that contains them. If %COMMENT appears in multiple places within the program, it will be recompiled each time it is encountered and additional computer resources will be expended. This can be somewhat inefficient. As a general rule this is not a good use of macros. That said, the inefficiencies are generally quite minor, and the technique sure can be handy from time to time.

MORE INFORMATION

Although asterisk-style comments can generally be used within a macro without problems, this style of comment can produce some surprises. Section 13.3.5 includes an example of a problem introduced by an asterisk-style comment.

SEE ALSO

The use of macros to comment out sections of code has also been discussed by Grant (1994, 1998), Stuelpner (1997), Flavin and Carpenter (2000), and Suhr (2001).

3.2 Invoking a Macro

Macros are invoked, or called, by placing a % in front of the macro name. Unlike SAS statements, the macro call does not need to be followed by a semicolon. The macro LOOK in Section 3.1.1 is called by

```
%LOOK
```

When a macro call is encountered during processing, the macro definition is loaded into the macro facility. Here macro statements are processed and any non-macro text is sent to the input stack, where it is further processed. When a macro call is detected, in this case %LOOK, the contents of the macro are in effect substituted for the macro name in the program. Because this happens before any other program statements in this step are evaluated, you can use the macro to contain statement fragments or complete steps.

In the following example, the macro %DEBUGNEW defines a PROC PRINT that you might want executed when you debug a program. (Section 3.1.2 contains additional discussion of %DEBUGNEW). As long as the macro call remains commented out, ❶ the macro is not executed. Notice that an asterisk-style comment ❷ is used to provide the semicolon for the commented macro.

```
data new;
  set big;
  ...code not shown...
run;
```

```

%macro debugnew;
proc print data=new (obs=5);
  title 'listing for NEW';
run;
%mend debugnew;
❶ *%debugnew      ❷ * uncomment to use debugnew;

```

You can also use a macro variable to control all debugging macros. In the following example, the macro variable `DEBUG` takes on the value of `*` when the debugging macros are to be turned off. The previous macro call becomes

```

%let debug = *;

...code not shown...

&debug %debugnew      * uncomment to use debugnew;

```

To execute the macro `%DEBUGNEW`, the macro variable `&DEBUG` is cleared or assigned a null text string:

```
%LET DEBUG =;
```

Although this is not a usual coding solution, the macro call itself can contain the name of a macro variable, which stores part of the macro name. The following macro call could be used to invoke the `%DEBUGNEW` macro:

```

%let version = new;
%debug&version

```

CAVEAT: In some versions of SAS (Release 5.18, Version 8.0, and Release 8.1), the macro processor will try to execute the `%DEBUG` macro before resolving the macro variable `&VERSION`. A further discussion of this problem can be found in Section 13.3.5. If you encounter this problem, one potential solution is to follow the macro call with a semicolon. The macro call becomes

```
%debug&version;
```

3.3 System Options Used with the Macro Facility

A number of SAS system options apply directly to the use of macros. They determine such things as whether or not the macro facility is available, how it is implemented, what debugging messages are to be printed, and if the macro autocall facility will be available.

3.3.1 General macro options

These system options control the overall use of the macro facility.

CMDMAC

Command-style macros enable the use of command line macro names that do not start with the `%` sign. Although not without value, this and the statement style of macros are carryovers from earlier releases of SAS and are rarely used. System resource

requirements are greatly increased through the use of these macros, and code can become more difficult to read and debug. By default this option is usually turned off (NOCMDMAC). Section 3.5 discusses this style of macro in more detail.

IMPLMAC

Statement style macros allow the use of macro names that do not start with the % sign. By default, the option is usually turned off (NOIMPLMAC). Section 3.5 discusses this style of macro in more detail.

MACRO

The MACRO system option, which you must specify at the invocation of SAS (or in a configuration file), determines if the macro facility is to be available. When you specify NOMACRO, you remove the ability to use the macro facility. As a general rule this option is already turned on and you rarely have to use it.

MERROR and SERROR

When using and writing macros, the debugging process is often difficult, even when these two options are turned on. Consequently, they will generally be left on whenever working with macros. MERROR enables the macro facility to display a warning in the LOG when a macro call is not resolved, and SERROR displays a warning in the LOG when a macro variable reference is not resolved.

3.3.2 Debugging options

Debugging a macro can be, under the best of conditions, difficult. The LOG is often very cryptic when it presents error messages that deal with macros, macro code, or macro variables. You can use several options that are specifically designed for use when debugging macros.

MPRINT

SAS code that is generated by a macro is generally not displayed in the LOG. The MPRINT option displays the text or SAS statements that are generated by macro execution, one statement per line, with macro variable references resolved.

MLOGIC

Macros often are designed with logical branches based on %IF-%THEN/%ELSE statements and %DO loop executions. MLOGIC traces the macro logic and follows the pattern of execution. The resolved result of an %IF statement is displayed in the LOG as true or false. Macro invocation, macro completion, and %DO loop evaluations are noted. This option is especially useful for nested macros.

SYMBOLGEN

When you use this option, a message is printed in the LOG whenever a macro variable is resolved. This option is very useful when you trace macro variable references with multiple ampersands, for example, &&DAT&I. SGEN can be used as an alias for this option.

MFILE

Similar to MPRINT, the MFILE option can be used to write out the resolved macro code to a file. To take advantage of this option, use a FILENAME statement with a *fileref* of MPRINT, and be sure to turn on both the MPRINT and MFILE options.

```
filename mprint 'c:\mymacros\maccode.sas';
options mprint mfile;
```

As each subsequent macro is called, the resultant code is written to the file MACCODE.SAS. You can turn off the writing of these statements by issuing either a NOMPRINT or a NOMFILE option. Since only the resolved code is written, macro statements like %LET and %PUT will not be shown.

MPRINTNEST and MLOGICNEST

When you are using nested macros (macros that call macros), MPRINT and MLOGIC only show the innermost level, and it is difficult to discern the hierarchy of calling macros. Available in SAS 9, MPRINTNEST and MLOGICNEST overcome this limitation by listing each of the calling macros in order. MPRINTNEST and MLOGICNEST require the use of MPRINT and MLOGIC, respectfully.

MCOMPILENOTE

This SAS 9 option can be used to instruct the macro facility to write a note to the LOG each time you compile a macro. There are three levels for this option. The syntax is as follows:

```
options mcompilenote=<none|noautocall|all>;
```

NONE prevents any compilation notes from going to the LOG.

NOAUTOCALL prevents notes associated with the compilation of autocall macros.

ALL writes a note to the LOG for each compiled macro.

Macros do not have to compile successfully in order to generate a NOTE. In fact, this is one of the advantages of this option as macros that result in compile errors will have a specific note written to the LOG.

Using the Options

If you include the following two statements near the start of your programs, you will then be able to turn these options on or off at any point in the debugging process:

```
OPTIONS NOMPRINT NOMLOGIC NOSYMBOLGEN;
*OPTIONS MPRINT MLOGIC SYMBOLGEN;
```

When you program in the interactive mode or from the display manager, both statements are needed. This is because any option that you set (or turn on) remains in effect for the duration of that SAS session or until you reset it. Simply deleting the OPTIONS statement will not turn off the options. Of course this is not an issue if you are debugging in batch mode.

MORE INFORMATION

A more flexible version of the `OPTIONS` statements shown in this section can be found in Section 4.4.2.

Section 13.3.3 contains suggestions and things to look for when debugging your macros.

SEE ALSO

A number of SUGI papers address various aspects of the debugging process. Although some are based on Version 5 options, these papers can still provide good insight: Frankel (1991); Gilmore and Helwig (1990); O'Connor (1991); Phillips, Walgamotte, and Drummond (1993).

Chapter 10, "Macro Facility, Error Messages and Debugging," (pp. 111–130) in *SAS Macro Language: Reference, First Edition* discusses a variety of topics that relate to the debugging and troubleshooting of macros.

Litzinger, Brooks, and Riddle (2002) as well as Edgington and Zhou (2002) have examples that use the `MPRINT` and `MFILE` options to generate code.

Heaton (2001) shows some more sophisticated approaches to controlling (turning on and off) these debugging options.

3.3.3 Autocall facility options

The autocall facility enables you to call macros that have been stored as SAS programs. Using this facility enables you to create libraries of macros that you define only once, while also making these macros available to all of your programs or even to other programmers.

The autocall macro facility stores the source code for SAS macros in external files that, taken together, form an autocall library. The library is an aggregate storage location such as a directory that contains files (or members). The macro definition and the file containing that definition must have the same name. Generally, an autocall library contains individual files or members, each of which contains one macro definition. On platforms such as UNIX the matching of names may be case sensitive.

Options that are associated with the autocall facility include

MAUTOSOURCE

controls the availability of the autocall facility. When in effect, autocall libraries (specified with `SASAUTOS=`) are included in the search for the macro definition. Usually this option is turned on by default.

MRECALL

controls whether or not the autocall libraries will be searched again when a macro is not found. Typically this option is turned off (the default) and is really needed only when using multiple shared autocall libraries and one or more libraries may be occasionally unavailable. During macro development a library can become unavailable when an attempt to compile a macro is unsuccessful. The `NOMRECALL` prevents a second attempt. To the user the autocall library seems to be broken. Restarting SAS clears the way for a second attempt; however, it may be easier to set this option to `MRECALL` during macro development.

SASAUTOS=

specifies the libraries or locations for collections of macros that will be searched when a macro is called. The following **OPTIONS** statement defines a location for the autocall library in the Windows operating environment:

```
options SASAUTOS="d:\caltasks\macros";
```

You can specify more than one library in the **SASAUTOS=** option, and you can use *filerefs* instead of the actual specification of the library location. The following example specifies three macro libraries:

```
filename grp5mac 'c:\group5\macros';
filename prj5Amac 'c:\group5\prjA\macros';
options mautosource sasautos=(prj5amac, grp5mac, sasautos);
```

In addition to any autocall libraries that you might create, a number of macros have been included in the autocall library supplied with your SAS software. Some of these macros have been described in Section 12.3, "SAS Autocall Macros." Be sure to include the automatic *fileref* **SASAUTOS** in your list of autocall libraries. If you do not do so, you will be unable to access any of the autocall macros supplied by SAS.

Depending on your version of SAS, the automatic *fileref* **SASAUTOS** may not be available on the UNIX platform.

MORE INFORMATION

Section 13.4 goes into detail on various aspects of building and using macro libraries. An example of the use of the **SASAUTOS=** option is shown in Section 3.6.1.

SEE ALSO

Further discussion of the autocall facility can be found in Tindall and O'Connor (1991), O'Connor (1992), Carey and Carey (1996, p.142), Carpenter and Smith (2001b), and Carpenter (2001a).

The management of large macro libraries can be problematic, and Bryant (1997) includes a macro that will identify and list the macros in a system. Peszek and Troxell (2000) present a macro that generates an HTML management solution that is built using JAVASCRIPT.

Chapter 9, "Storing and Reusing Macros," (pp. 105–110) in *SAS Macro Language: Reference, First Edition* provides additional information on autocall macros.

Jennifer Price (1998) gives some ideas on setting up an autocall library.

3.3.4 Compiled stored macros

The compiled stored macro facility takes the concepts that were discussed in the previous section regarding the autocall macro facility one step further. Autocall libraries store text or source code that defines macros. This text can be called (by calling the macro) and at that point it is compiled and executed. Using the autocall library for macros that are called over and over in **different SAS sessions** will necessitate that the macro be compiled in each session.

The compiled macro facility enables you to compile the macro one time and then store the compiled version. When called, the macro is already compiled so you save time and resources.

The `MSTORED` and `SASMSTORE` system options, which are analogous to the ones used with the autocall facility, must be turned on so that you can use compiled macros. Remember that during macro compilation only macro statements are compiled; simple macro references and non-macro text are not evaluated until macro execution.

MSTORED

turns on the ability to use the facility. Most sites have this turned off (`NOMSTORED`) by default.

SASMSTORE=

specifies the *libref* (not a *fileref* like in `SASAUTOS=`) that contains a SAS catalog named `SASMACR`. This catalog is analogous to `WORK.SASMACR`, which temporarily stores macros compiled in the current session.

During normal macro operations, you may notice that SAS temporarily compiles macros (known as *session compiled macros*) and places them in `WORK.SASMACR`. This means that the *libref* that you use with `SASMSTORE=` cannot be `WORK`.

To store the compiled version of a macro, use the `STORE` option on the `%MACRO` statement:

```
options mstored sasmstore=mydir;

%macro doit(dsn,var1) / store;
...code not shown...
```

The macro `%DOIT` will be compiled and stored as the entry `DOIT` in the catalog `MYDIR.SASMACR`.

CAVEAT: Although the macro is compiled, the source code is not saved and cannot be reconstructed from the compiled version of the macro.

MORE INFORMATION

Section 12.1 goes into detail on various aspects of building and using macro libraries.

The option `SOURCE`, which can be used to save source code when compiling macros, is discussed in Section 12.1.2.

SEE ALSO

You can find an easy-to-read and thorough description of compiled stored macros in O'Connor (1992).

A brief introduction to compiled stored macros is in *SAS Macro Language: Reference, First Edition* (pp. 108–109), Carey and Carey (1996, p. 223), Carpenter and Smith (2001b), and Carpenter (2001a).

Davis (1997) includes a description and example of the use of compiled stored macros.

Section 13.4 discusses how to set up and use compiled stored macro libraries

3.3.5 Macro search order

When a macro is called, the macro facility must search for the definition of that macro. It first checks the catalogs that contain compiled macros and then checks the autocall macros that have not yet been compiled. The search order for a called macro is as follows:

1. **work.sasmacr catalog**

This catalog contains macros that are compiled during the current SAS session.

2. **libref.sasmacr catalog**

This catalog contains stored compiled macros (see Section 3.3.4). The **MSTORED** and **SASMSTORE=** options must be in effect to use stored compiled macros.

3. **autocall library**

When the **MAUTOSOURCE** option is in effect, each library of source programs listed in the **SASAUTOS=** option is searched.

Macros in the autocall libraries are never inspected by the SAS Macro Facility until the macro is called. The macro is then compiled (and its compiled version is loaded into **WORK.SASMOCR**) and executed.

It is important to remember the order in which macros are stored, compiled, and re-executed. If you use the display manager during the debugging process, a change to a macro definition stored in a program in the autocall library (**SASAUTOS=**) will not change the compiled version in **WORK.SASMOCR**. Re-execution of the macro will not implement changes unless the compiled version of the macro has been eliminated from the **WORK.SASMOCR** catalog or you recompile the macro so that the **WORK.SASMOCR** catalog is updated using the version in the autocall library.

MORE INFORMATION

Deleting entries directly from the **SASMOCR** catalog should force the recompilation of the macro; however, it may have unintended consequences (see Section 12.2.3).

3.3.6 Memory control options

Typically, macro symbol tables and therefore the values of macro variables are stored in memory. When the memory required to store the value of a macro variable is not available, SAS will instead write the macro variable to a catalog (under Windows the catalog is named **WORK.SAS0st0**). In this catalog each macro variable is a separate entry (that is, it has an entry type of **MSYMTAB**).

MVARSIZE

specifies the maximum size that an individual macro variable can take on before it is written to disk.

MSYMTABMAX

specifies the maximum memory available for all symbol tables. When this value is exceeded the macro variables are written to disk.

By adjusting the values of these options, you can control where macro variables and symbol tables will be written. Usually these options are not of general concern, but they can be useful if you have either large symbol tables or large macro variables and you are limited either in available memory or available disk space.

SEE ALSO

DiIorio (1999) uses the MVARSIZE option to force macro variables into a catalog where they can be removed.

3.4 Display Manager Command-Line Macros

Usually macros are executed from within programs, but they can also be written to help us do our work in the display manager. Command-line macros can contain any statement that you enter on a command line, and they can then be called from the command line or assigned to a specific key.

The following string of commands can be assigned to a key in the KEYS window. When executed together, they will switch to the program editor, recall the last submitted code, and maximize the window:

```
pgm; recall; zoom on;
```

These same commands can be placed in a macro:

```
%macro pgm;
  pgm;
  recall;
  zoom on;
%mend pgm;
```

Notice that the semicolons are included to chain the commands together. Entering the macro call %PGM on the command line will execute the macro and the display manager will see a series of commands. The call to the macro can also be assigned to a hot key, as is shown in the following portion of the KEYS window:

```
F9      :ts
F11     command bar
F12     zoom
SHF F1  subtop
SHF F2  %pgm
SHF F3
SHF F6
```

Before the macro can be executed, you must have defined it. If you are going to add the macro call to the KEYS definitions, consider adding the macro definition to the autocall library (see Section 3.3.3 for more information on using the Autocall Facility).

The command-line macro can include other macro statements, macro parameters, and macro logic. If you find yourself doing a series of command-line commands over and over again, try putting them in a macro.

SEE ALSO

Additional information on command-line macros can be found in Carey and Carey (1996, pp. 69, 296) and in *SAS Screen Control Language: Reference, Version 6, Second Edition* (p. 160).

The *Observations* article by Johnson and Gilman (1993, pp. 50–54) provides detailed information on command-line macros and the use of macros in PMENUs.

3.5 Statement- and Command-Style Macros

All of the macros used in this book (exclusive of this section) are what is known as *named-style macros*. When using named-style macros, the macro is always called by preceding its name by a percent sign (%) and parameters are passed within parentheses (see Chapter 4, "Macro Parameters," for more information). Statement-style and command-style macros do not require the use of the preceding %.

Statement-style macros enable you to create macro calls that *look* like SAS statements. Command-style macros are used to mimic display manager command-line commands.

As a general rule, few programmers use either of these macro styles. There are two very good reasons why they are not often used:

- The absence of the % in the macro call makes the code more difficult to read and recognize as a macro call because it is not what programmers are used to seeing.
- Substantial additional system resources are needed to scan for the macro names. This can slow the execution of all jobs, even those that do not contain statement-style macro references.

SEE ALSO

Additional information on statement-style macros can be found in *SAS Guide to Macro Processing, Version 6, Second Edition* (pp. 91–100) and in *SAS Macro Language: Reference, First Edition* (p. 133).

Although available in Version 5, command-style macros were not available in Version 6 until Release 6.07. Additional examples and syntax can be found in SAS Technical Report P-222, *Changes and Enhancements to Base SAS Software, Release 6.07* (pp. 309–310) and in *SAS Macro Language: Reference, First Edition* (p. 133).

The general inefficiency of statement- and command-style macros is briefly mentioned in *SAS® Macro Language: Reference, First Edition* (p. 133).

A few approaches to avoid are discussed in Carpenter (1996 and 1998b) and in Carpenter and Smith (2001b).

3.5.1 Turning on system options

Normally, the system options that are required to use these macro styles are turned off. You can turn them on by using the `IMPLMAC` and `CMDMAC` system options (the default is `NOIMPLMAC` and `NOCMDMAC`). The following `OPTION` statement turns on statement-style macros:

```
options implmac;
```

You can initiate the ability to use command-style macros if you use

```
options cmdmac;
```

3.5.2 Defining statement- and command-style macros

Macros for both of these macro styles are defined in the usual way using the %MACRO statement. Defined macros will be named-style unless the CMD **❶** or STMT option is used on the %MACRO statement. The macro described in Section 3.4 could be redefined as this command-style macro:

```
option cmdmac;
%macro zpgm / cmd; ❶
  pgm;
  recall;
  zoom on;
%mend zpgm;
```

After the macro is created, you can enter ZPGM (not %ZPGM) on the command line to execute the macro:

```
Command ==> zpgm
```

Most users are accustomed to using the % sign with macro calls and, as was mentioned earlier, statement-style macros often cause additional confusion. Command-style macros can make more sense in developer controlled applications.

3.6 Controlling System Initialization

Whenever SAS is started, you can direct it to automatically execute a SAS program that you can design to set up your environment. Although you can call this initialization program anything that you want, by default the name is AUTOEXEC.SAS. The file can be executed through the use of the AUTOEXEC initialization option.

The use of the AUTOEXEC.SAS is especially useful to macro programmers as it gives you a way to automatically set up a customized environment. This can include global macro variables and macro libraries.

SEE ALSO

Consult the SAS Companion for your operating system for specific details on the default locations and naming conventions for the AUTOEXEC.SAS program. Wang (2003) has an example that customizes a Windows shortcut using the AUTOEXEC.

3.6.1 Using AUTOEXEC.SAS

The AUTOEXEC.SAS program is just like any other SAS program. By default the file usually resides in the same location as the SAS.EXE executable file. If you create a program named AUTOEXEC.SAS and you place it in this location, SAS will find and execute it without any other intervention on your part.

The following is a customized autoexec program for the XYZ project:

```
*****;
* autoexec.sas
*
* Initialize SAS for the XYZ Project
*
* Created: Art Carpenter
*          09/27/2003
* Modified:
* Date      Initials    Note
*****;
* Establish GLOBAL Macro variables for this application;
%global tst path project;

* Set up the general path for this application;
%let path = d:\clintrials\ct113; ❶

* tst    test (tst) or production ( );
  %let tst = ;          * Production; ❷
*%let tst = tst;      * test environment;

* Project designation;
%let project = XYZ; ❸

*****;
* Define the location for the macros;
options sasautos=("&path\&project&tst\pgms\sasmacro" sasautos); ❹

* Define the Librefs and Filerefs for this project;
%libnames ❺

* Execute the first screen for this application;
dm "af cat = appls.&project..userid.program" af; ❻
```

- ❶ Root portion of all library and file references for this study.
- ❷ Switch to enable branching between the parallel test and production environments.
- ❸ Project code.
- ❹ Taken together, the macro variables &PATH, &PROJECT, and &TST form a complete, but highly adjustable, method for pointing to any location. The SASAUTOS= option is used to designate an autocall library.
- ❺ Call the macro that specifies all the *librefs* and *filerefs* for the project.
- ❻ Start SAS/AF application. The APPLS library was defined by the %LIBNAMES macro.

3.6.2 Controlling AUTOEXEC execution

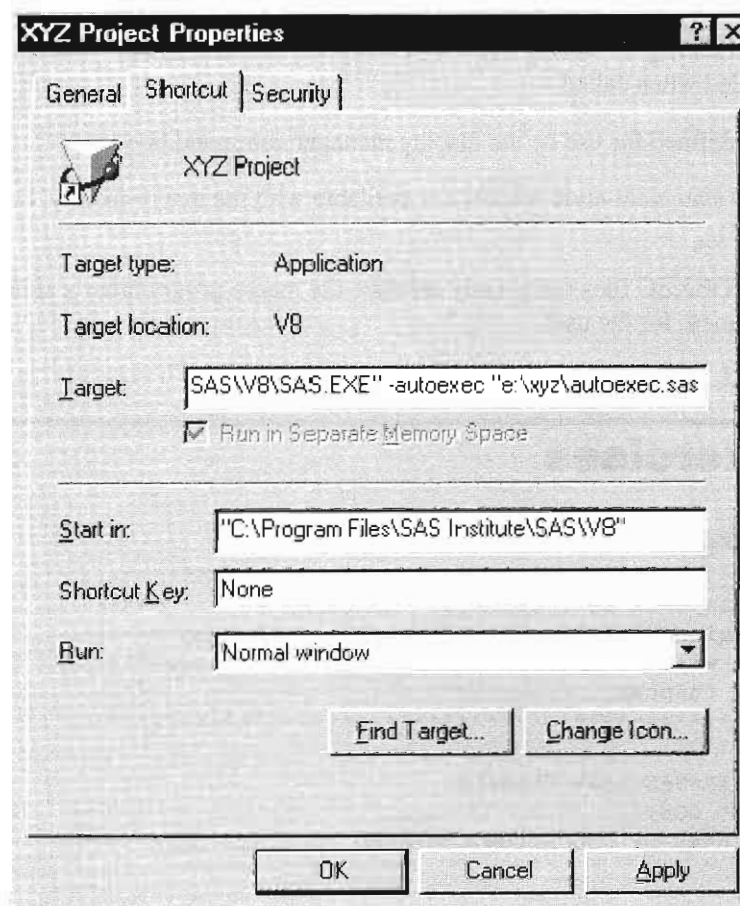
When SAS is first initialized, options can be applied that will direct SAS to execute a specific AUTOEXEC file. This means that the file can take on any name, and it does not need to be AUTOEXEC.SAS. The -AUTOEXEC initialization option specifies the file to be executed.

Under MVS this option can be specified on the //SYSIN card.

When executing SAS from batch, the UNIX option might look something like this:

```
sas -autoexec /home/justina/autoexec.sas
```

Under Windows it is specified by editing the properties of the SAS icon, as shown here:



SEE ALSO

Fehd (2001) has three examples of AUTOEXEC programs and discusses their differences. Jennifer Price (1998) gives some general guidelines on setting up an AUTOEXEC.

3.7 Chapter Summary

Macros are defined using two macro language statements: `%MACRO macro-name` and `%MEND <macro-name>`. Text is then enclosed between these two statements. Macros are invoked or called by placing a `%` in front of the macro name.

During macro debugging the information written to the LOG is often very cryptic and the error messages are often difficult to decipher. Several options that are specifically designed for use with macros may be useful during the writing, debugging, and processing of macros. These include `MPRINT`, `MLOGIC`, and `SYMBOLGEN`.

The autocall facility enables you to define and store macros for later use, including the use in programs in different SAS sessions. SAS software is shipped with some predefined macros in the autocall library.

You can increase efficiency by storing macros in compiled stored libraries. Macros stored here will not be recompiled when called.

Macros can also be defined for use on the display manager command line.

Command-style and statement-style macros are available with the macro facility, but their use is not recommended.

The use of the `AUTOEXEC` files can greatly enhance the macro programmer's ability to build a customized environment for the user.

3.8 Chapter Exercises

Sample Program

```
*****;
**** The class data set, CLINICS, contains 80 ****;
**** observations and 20 variables. The following ****;
**** program will be used to complete the exercises ****;
**** in this chapter. ****;
*****;

PROC PLOT DATA=SASCLASS.CLINICS;
  PLOT EDU * DOB;
  TITLE1 'YEARS OF EDUCATION COMPARED TO BIRTH DATE';
RUN;

PROC CHART DATA=SASCLASS.CLINICS;
  VBAR WT / SUMVAR=HT TYPE=MEAN;
  TITLE1 'AVERAGE HEIGHT FOR WEIGHT GROUPS';
RUN;
```

1. Convert the sample program into a macro and execute it. Examine the SAS LOG.
2. In the macro program that you wrote in the previous exercise, use the options `MLOGIC`, `MPRINT`, and `SYMBOLGEN` separately and in combination to see how the SAS LOG changes.

3. (True or False) To invoke a macro, a percent sign (%) is placed before the macro name.
4. Create an autocall library.
 - a) Place the macro that you created in question 1 into the autocall library.
 - b) Delete the compiled version of this macro from the SASMACR catalog.
 - c) Execute the macro.
 - d) Verify that the compiled macro is now in the SASMACR catalog.



Chapter 4

Macro Parameters

- 4.1 Introducing Macro Parameters 54
- 4.2 Positional Parameters 54
 - 4.2.1 Defining positional parameters 54
 - 4.2.2 Passing positional parameters 55
- 4.3 Keyword Parameters 56
 - 4.3.1 Defining keyword parameters 57
 - 4.3.2 Passing keyword parameters 57
 - 4.3.3 Documenting your macro 58
- 4.4 Choosing between Keyword and Positional Parameters 59
 - 4.4.1 Selecting parameter types 59
 - 4.4.2 Using keyword and positional parameters together 60
- 4.5 Chapter Summary 61
- 4.6 Chapter Exercises 61

Macros are made more powerful and flexible when information can be transferred to them through the macro call. This chapter expands on the uses of macro variables by detailing their use as parameters that can be passed into and between macros. The chapter introduces two types of macro parameters, and their relative merits are discussed.

SEE ALSO

There is a very good overview of beginning macro issues in Whitlock (1999a).

4.1 Introducing Macro Parameters

In Chapter 2, "Defining and Using Macro Variables," macro variables were introduced and defined through the use of the %LET macro statement. Unfortunately, %LET becomes cumbersome and often limiting when it is used to make the values of macro variables available to a macro. If we use %LET in the global space, the macro variables will be available to all macros in the session; however, we will be restricted to using the specified names in all our macros. If we use the %LET within the macro definition, we will need to edit the macro each time we want to change a variable's value. Macro parameters are used to overcome these limitations.

Macro parameters enable you to define macro variables without using the %LET statement. Macro parameters are used to pass values or text strings into a macro, where they are assigned to a macro variable on the local symbol table. Since these parameters are given names in the %MACRO statement, the names do not have to correspond to similar values on the global symbol table. The assignment of values to the parameters is made when the macro is called, not when the macro is coded. The following sections describe this process.

There are two types of parameters: *positional* (Section 4.2) and *keyword*, which is also known as *named* (Section 4.3).

4.2 Positional Parameters

Positional parameters derive their name from the fact that they are defined using a specific position on the %MACRO statement. When the macro is called, the value is passed using that same corresponding position in the macro call as it has in the macro definition (%MACRO statement).

4.2.1 Defining positional parameters

Positional parameters are defined by listing the macro variable names that are to receive the parameter values in the %MACRO statement. When parameters are present, the macro name is followed by a comma-separated list of macro variables that are enclosed in a pair of parentheses.

The following version of %LOOK uses the %LET to establish two global macro variables (&DSN and &OBS):

```
%LET DSN = CLINICS;
%LET OBS = 10;
%MACRO LOOK;
  PROC CONTENTS DATA=&dsn;
    TITLE "DATA SET &dsn";
  RUN;

  PROC PRINT DATA=&dsn (OBS=&obs);
    TITLE2 "FIRST &obs OBSERVATIONS";
  RUN;
%MEND LOOK;
```

We can easily convert this macro so that it uses positional parameters rather than relying on the %LET. The following version of %LOOK has two positional parameters, and it is more flexible:

```

%MACRO LOOK(dsn,obs) ;
    PROC CONTENTS DATA=&dsn;
        TITLE "DATA SET &dsn";
    RUN;

    PROC PRINT DATA=&dsn (OBS=&obs);
        TITLE2 "FIRST &obs OBSERVATIONS";
    RUN;
%MEND LOOK;

```

The only difference in these two versions of %LOOK is in the %MACRO statement. The parameters enable us to create &DSN and &OBS as local macro variables, and we are not required to modify the macro itself.

Another advantage of using the local symbol table whenever possible is that it minimizes the risk of macro variable collisions (see Sections 5.4.2, 7.5.1, and 13.3.5).

4.2.2 Passing positional parameters

Because the parameters are positional, the first value in the macro call is assigned to the macro variable that is listed first in the macro statement's parameter list. When you have multiple parameters, you need to use commas to separate their values. If you want to pass a value that contains a comma, you will need to use one of the quoting functions that are described in Section 7.1.

The macro call for %LOOK could be

```
%LOOK(CLINICS,10)
```

You do not have to give all parameters a value. Alternative invocations of the %LOOK macro might include

```

%LOOK()
%LOOK(CLINICS)
%LOOK(,10)

```

Macro variables that are not assigned a value will resolve to a null string. Thus, the macro call %LOOK(,10) resolves to

```

PROC CONTENTS DATA=;
    TITLE "DATA SET ";
RUN;

PROC PRINT DATA= (OBS=10);
    TITLE2 "FIRST 10 OBSERVATIONS";
RUN;

```

The resolved code contains syntax errors, and it will not run. Be careful to construct code that will resolve to what you expect, and when possible anticipate and code around problems like this one.

The following macro sorts a data set with one to three BY variables. The macro would not be needed or used if there was not at least one BY variable—indeed, syntax errors would be generated if that were the case.

```
%MACRO SORTIT(DSN,BY1,BY2,BY3);
  PROC SORT DATA=&DSN;
    BY &BY1 &BY2 &BY3;
  RUN;
%MEND SORTIT;
```

The macro call %SORTIT (CLINICS, LNAME, FNAME) resolves to

```
PROC SORT DATA=CLINICS;
  BY LNAME FNAME;
RUN;
```

Because undefined parameters result in a null string, &BY3 is dropped from the resolved code. This technique enables us to create generalized code that will be syntactically correct at execution time. The macro %SORTIT will work correctly for one, two, or three BY variables. It will, of course, generate errors if no BY variables are passed into it.

The previous definition of %SORTIT could be made more flexible by creating a single macro variable to hold all of the BY variables. &BYLIST replaces &BY1, &BY2, and &BY3 in the following example of %SORTIT:

```
%MACRO SORTIT(DSN,BYLIST);
  PROC SORT DATA=&DSN;
    BY &BYLIST;
  RUN;
%MEND SORTIT;
```

Now, a call to %SORTIT could be

```
%SORTIT(CLINICS,CLINNO LNAME FNAME SSN)
```

This call resolves to

```
PROC SORT DATA=CLINICS;
  BY CLINNO LNAME FNAME SSN;
RUN;
```

Notice that there are **no** commas between the names of the BY variables in the macro call. These four variables become a text string that forms the single definition to the macro variable &BYLIST. In this definition of %SORTIT, the user is not limited to three BY variables.

4.3 Keyword Parameters

Parameters may be designated as *keyword* in the %MACRO statement. Unlike positional parameters, keyword parameters may be used in any order and may be assigned default values. Keyword parameters are especially useful when there are large numbers of parameters or when the names of the parameters themselves will help the user with the correct specification of the macro call.

4.3.1 Defining keyword parameters

Keyword parameters are designated by following the parameter name with an equal sign (=). Default values, when present, follow the equal sign. You can use keyword parameters to redefine the %LOOK macro from Section 4.2.2 like this:

```
%MACRO LOOK (dsn=CLINICS,obs=) ;

    PROC CONTENTS DATA=&dsn;
        TITLE "DATA SET &dsn";
    RUN;

    PROC PRINT DATA=&dsn (OBS=&obs);
        TITLE2 "FIRST &obs OBSERVATIONS";
    RUN;

%MEND LOOK;
```

In this version of %LOOK, the macro variable &DSN will have a default value of CLINICS, while &OBS does not have a default value. If a value is not passed to &OBS, &OBS will take on a null value, in much the same way as a positional parameter will when it is not provided a value.

4.3.2 Passing keyword parameters

Macro variables that are not assigned a value resolve to their default value or to a null value when you do not specify a default. In Section 4.2.2 the macro call (using positional parameters) %LOOK(,10) results in syntax errors because the data set name (the first parameter) is not provided. When you use the macro %LOOK that is defined with keyword parameters in Section 4.3.1, the macro call %LOOK(OBS=10) resolves to

```
PROC CONTENTS DATA=CLINICS;
    TITLE "DATA SET CLINICS";
RUN;

PROC PRINT DATA=CLINICS (OBS=10);
    TITLE2 "FIRST 10 OBSERVATIONS";
RUN;
```

Because the macro call %LOOK(obs=10) did not include a definition for &DSN, the default value of CLINICS was used. The resulting code eliminates the syntax errors that were generated in the %LOOK example in Section 4.2.2. However, since &OBS does not receive a default value, syntax errors will still result if a value is not provided for &OBS. Unlike in this example, as a general rule it is a good idea to provide appropriate default values for all your parameters so that the macro will work correctly regardless of which parameters are specified by the user.

4.3.3 Documenting your macro

There are a couple of documentation and style techniques that you can apply that will make your macros more readable, as well as easier to use.

- When you have more than two or three parameters, consider lining up the parameters (one per line).
- Use descriptive parameter names that will help the user remember what each parameter does.
- Supply default values whenever reasonable defaults are available.
- Copy the list of parameters in the %MACRO statement to build a comment that explains each parameter (include a list of acceptable values when appropriate).

The following macro contains 11 keyword parameters. A full list of the parameters, the default values, and comments for each parameter is included as a part of the macro.

```
%macro schoen2(data = ,
               event = status,
               outsch = schr,
               outbt = schbt,
               vref = yes,
               points = yes,
               df = 4,
               alpha = .05,
               rug = no ,
               no = no );

*****
* Macro parameters with default values.
data = ,           name of the analysis data set.
event = status,    event variable for survival status at
                   exit, 1=event, 0=censored, currently named STATUS
outsch = schr,     name of output data set containing Schoenfeld
                   residuals.
outbt = schbt,     name of output data set containing the scaled
                   Schoenfeld residuals(Bt).
vref = yes,        indicator to control plotting of a vertical
                   reference line at y=0. Values are yes(default)
                   and no.
points = yes,      Indicates whether to plot the actual data points.
                   Default is yes.
df = 4,            degrees of freedom for smoothing process.
                   Possible (integral) values are 3 to 7. Default
                   is 4.
alpha = .05,       confidence coefficient for plotting standard
                   error bars. Default is .05.
rug = no           indicator to control plotting of rug of x values.
no = no            Turn on and off macro debugging
                   no    debugging is off
                   blank debugging is on
```

```

*****;
OPTIONS &no.symbolgen &no.mprint &no.mlogic;
%put *** Entering SCHOEN2 macro ***;

...code not shown...

%put *** Leaving SCHOEN2 macro ***;
%mend schoen2;

```

This section of comments is easily adapted when documenting the macro for the users.

4.4 Choosing between Keyword and Positional Parameters

When you are writing macros you will need to decide whether to use positional or keyword parameters. Some macro programmers always use one style or the other, while others make the determination based on the program itself. While there is no right or wrong way of selecting the type of parameters, there are some things that you might want to take into consideration.

4.4.1 Selecting parameter types

Macros with keyword parameters tend to be easier to document and to use, although there is a bit more typing. In most of my more recent macros I have tended to use keyword parameters. Only rarely do I mix positional and keyword parameters.

Generally I decide which type of parameters to use based on some combination of the following questions.

Who will be using the macro?

If the macro is going to be used by anyone other than the initial programmer, then keyword parameters have the advantages of ease of use and flexibility.

How many parameters are being passed?

If the macro only has two or three parameters, then it should be fairly easy for the user to get all of the parameters correctly specified and in the correct order. More than three and the chances of mistakes on the part of the macro user are magnified.

Do any of the parameters need to have defaults?

If default values are required, then it is much easier to supply them using keyword parameters than it is to infer them by using programming statements such as %IF-%THEN/%ELSE.

MORE INFORMATION

Section 13.3.2 discusses other issues related to macro programming style.

4.4.2 Using keyword and positional parameters together

Although you may specify both keyword and positional parameters in the %MACRO statement, the list of positional parameters must precede the list of keyword parameters.

In the following version of the macro %LOOK, there is one positional and one keyword parameter. The keyword parameter, OBS, has the default value of 10.

```
%MACRO LOOK(dsn,obs=10);
  PROC CONTENTS DATA=&dsn;
    TITLE "DATA SET &dsn";
  RUN;

  PROC PRINT DATA=&dsn (OBS=&obs);
    TITLE2 "FIRST &obs OBSERVATIONS";
  RUN;
%MEND LOOK;
```

The macro call %LOOK(CLINICS) would have the same result as %LOOK(CLINICS,OBS=10) and would resolve to

```
PROC CONTENTS DATA=CLINICS;
  TITLE "DATA SET CLINICS";
RUN;

PROC PRINT DATA=CLINICS (OBS=10);
  TITLE2 "FIRST 10 OBSERVATIONS";
RUN;
```

When developing a series of nested macros (macros that call macros that call macros...), it can be very tiresome to edit individual macros during debugging. One approach is to use comments so that you can turn on (or off) the system options that control the LOG (see Section 3.3.2). A more flexible version of the OPTIONS statement is shown next and is controlled with a keyword parameter.

In the portions of the macros shown, there is at least one keyword parameter (&NO) that is assigned a default value of NO. In the OPTIONS statement ❶ &no.mprint will default to nomprint. If you need to debug %PRECOMP, then call the macro as %PRECOMP(1234,NO=) and &no.mprint will resolve to mprint.

```
%macro precomp(subject,no=no);
  options ps=47 pageno=1 ls=80;
  options &no.mprint &no.symbolgen &no.mlogic; ❶

  ...code not shown...

  * Create the location data set;
  %locate(&subject,no=&no) ❷

  ...code not shown...

%mend precomp;
```

Notice that %LOCATE ② uses the same system, and it also passes the positional parameter &SUBJECT. If the &NO is blank for %PRECOMP (MPRINT is turned on), then it will be automatically turned on for %LOCATE as well.

SEE ALSO

Heaton (2001) shows a more sophisticated method of turning on and off debugging in a macro.

4.5 Chapter Summary

You can make macros more flexible by adding the ability to pass parameter values directly into the macro. There are two types of parameters: positional and keyword (named). When you use parameters, be careful to construct code that will resolve to what you expect.

Parameter names are determined in the macro definition. Although keyword parameters may be used in any order and may be assigned default values, the order of positional parameters is fixed.

Macro variables that are not assigned a value resolve to their default value or to a null value when no default has been specified.

4.6 Chapter Exercises

Sample Program

```
*****;
**** The class data set, CLINICS, contains 80 ****;
**** observations and 20 variables. The following ****;
**** program will be used to complete the exercises ****;
**** in this chapter. ****;
*****;

PROC PLOT DATA=SASCLASS.CLINICS;
  PLOT EDU * DOB;
  TITLE1 'YEARS OF EDUCATION COMPARED TO BIRTH DATE';
RUN;

PROC CHART DATA=SASCLASS.CLINICS;
  VBAR WT / SUMVAR=HT TYPE=MEAN;
  TITLE1 'AVERAGE HEIGHT FOR WEIGHT GROUPS';
RUN;
```

1. What are the two types of parameters that are available in macros?
2. (True or False) You can specify keyword parameters in any order.
3. Change the macro that you wrote in Chapter 3 Exercises 1 and 2 to include at least one positional and one keyword (named) macro parameter (do not use %LET). Then execute the macro by passing parameters into the macro.

4. How many positional parameters are found in this statement?

```
%MACRO TOOL(DSIN,DSOUT,STOP=500);
```

5. Why must keyword parameters follow positional parameters?

6. What is wrong with the following macro code?

```
MACRO MYCOPY;
```

```
    PROC COPY IN=WORK OUT=MASTER;
```

```
        SELECT PATIENTS;
```

```
    RUN;
```

```
%MEND COPY;
```



Part 2

Using Macros

Chapter 5 **Program Control through Macros** 65

Chapter 6 **Interfacing with the DATA Step** 101

Chapter 7 **Using Macro Functions** 143

Chapter 8 **Using Macro References with the SAS Component Language (SCL)** 199



Chapter 5

Program Control through Macros

- 5.1 Macros That Invoke Macros 66
 - 5.1.1 Passing parameters between macros 66
 - 5.1.2 Controlling macro calls 70
 - 5.1.3 Nesting macro definitions 71
- 5.2 Using Conditional Macro Statements 72
 - 5.2.1 Executing macro statements 73
 - 5.2.2 Building SAS code dynamically 74
 - 5.2.3 Using the IN operator 74
- 5.3 Iterative Execution Using Macro Statements 77
 - 5.3.1 %DO block 78
 - 5.3.2 Iterative %DO loops 80
 - 5.3.3 %DO %UNTIL loops 83
 - 5.3.4 %DO %WHILE loops 84
- 5.4 Macro Program Statements 85
 - 5.4.1 Macro comments 86
 - 5.4.2 %GLOBAL and %LOCAL 88
 - 5.4.3 %GOTO and %label 91
 - 5.4.4 Using %SYSEXEC 93
 - 5.4.5 Creating macro %WINDOWS 94
 - 5.4.6 Abnormal termination of macro execution with the %ABORT 95
 - 5.4.7 Normal termination of macro execution with the %RETURN 96
- 5.5 Chapter Summary 96
- 5.6 Chapter Exercises 97

A great deal of the power of the SAS macro language comes from its ability to utilize conditional logic as well as a substantial number of macro statements and functions. This chapter introduces several of the more commonly used macro programming statements.

Examples are also included in this chapter that will show you how to write programs that use macros as building blocks by having macros invoke other macros.

5.1 Macros That Invoke Macros

It is not unusual for macros to call other macros and in the process to pass values to the macro that is being called. Often values defined in one macro will be used to affect the flow of logic and execution within a different macro. When a macro calls another macro, the call is said to be *nested* within the macro. It is often advantageous to nest macro calls (however, it is only very rarely appropriate to nest macro definitions).

MORE INFORMATION

Sections 5.1.3 and 13.3.1 discuss in more detail the nesting of macro definitions.

5.1.1 Passing parameters between macros

Consider the two macros %LOOK and %SORTIT (see Sections 4.2 and 4.3). These macros have one common parameter (&dsn), and it might be nice to have a utility or controlling macro to do both steps at once. Consider the macro %DOBOTH shown here:

```
%MACRO DOBOTH;
    %SORTIT (CLINICS, LNAME, FNAME)
    %LOOK (OBS=10)
%MEND DOBOTH;

%MACRO LOOK (dsn=CLINICS, obs=);
    PROC CONTENTS DATA=&dsn;
        TITLE "DATA SET &dsn";
    RUN;
    PROC PRINT DATA=&dsn (OBS=&obs);
        TITLE2 "FIRST &obs OBSERVATIONS";
    RUN;
%MEND LOOK;

%MACRO SORTIT (DSN, BY1, BY2, BY3);
    PROC SORT DATA=&DSN;
        BY &BY1 &BY2 &BY3;
    RUN;
%MEND SORTIT;
```

Notice that the macro %DOBOTH is defined before the macros that it calls. The order in which the macros are defined does not matter as long as each macro is defined before it is called. The calls to %SORTIT and %LOOK in the %DOBOTH macro are not executed until %DOBOTH itself is executed.

The macro call to %DOBOTH will in turn call the macros %SORTIT and %LOOK and these will be resolved as

```
PROC SORT DATA=CLINICS;
  BY LNAME FNAME;
RUN;
PROC CONTENTS DATA=CLINICS;
  TITLE "DATA SET CLINICS";
RUN;
PROC PRINT DATA=CLINICS (OBS=10);
  TITLE2 "FIRST 10 OBSERVATIONS";
RUN;
```

In the example shown above, the macro %DOBOTH has no parameters. Therefore it would be necessary to change or edit the definition of %DOBOTH in order to alter how it calls %SORTIT and %LOOK. %DOBOTH would be much more flexible if the parameters for the macros %SORTIT and %LOOK could be passed directly through the call to %DOBOTH.

The following example contains a rewritten macro %DOBOTH. This version of the macro is written to receive the parameters that are to be used by %SORTIT and %LOOK. Notice that the parameters for the calls to %SORTIT and %LOOK are all %DOBOTH macro variables. The values of these macro variables are specified in the call to %DOBOTH and their resolved values are passed on to %SORTIT and %LOOK.

```
%MACRO DOBOTH(d,o,b1,b2,b3);
  %SORTIT(&d,&b1,&b2,&b3) ❶
  %LOOK(&d,&o) ❷
%MEND DOBOTH;

%MACRO LOOK(dsn,obs);
  PROC CONTENTS DATA=&dsn;
    TITLE "DATA SET &dsn";
  RUN;
  PROC PRINT DATA=&dsn (OBS=&obs);
    TITLE2 "FIRST &obs OBSERVATIONS";
  RUN;
%MEND LOOK;

%MACRO SORTIT(DSET,BY1,BY2,BY3);
  PROC SORT DATA=&DSET; ❸
    BY &BY1 &BY2 &BY3; ❹
  RUN; ❺
%MEND SORTIT;
```

Notice that the macro variable names in %DOBOTH need not be the same as the ones used in %SORTIT and %LOOK. The macro call

```
%DOBOTH(CLINICS,10,LNAME,FNAME) ❻
```

resolves to

```
%SORTIT(CLINICS,LNAME,FNAME,) ❶
%LOOK(CLINICS,10) ❷
```

which, in turn, resolves to

```
PROC SORT DATA=CLINICS; ❸
  BY LNAME FNAME; ❹
RUN; ❺
```

```

PROC CONTENTS DATA=CLINICS;
  TITLE "DATA SET CLINICS";
RUN;
PROC PRINT DATA=CLINICS (OBS=10);
  TITLE2 "FIRST 10 OBSERVATIONS";
RUN;

```

The macro variables in %DOBOTH, (for example, &D, &O, and &B1), are resolved before the calls to %SORTIT and %LOOK are made. Consequently, it does not matter that the names of the macro variables in %DOBOTH are different from the ones that are used in the other two macros.

It might help you to visualize the process of macros passing values to other macros if we look at the above calls in more detail.

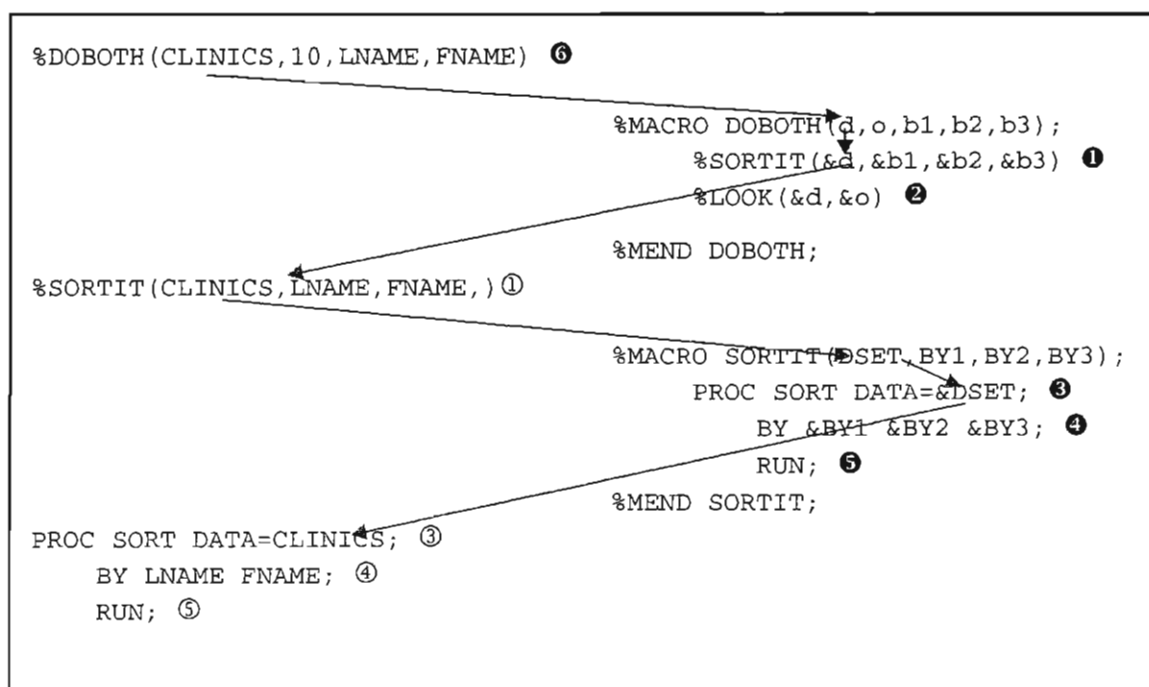
The following table shows a portion of the sequence of operations that take place when the macro %DOBOTH is called.

Step#	Line#	Code	What happens
1	⑥	%DOBOTH(CLINICS,10, LNAME, FNAME)	The macro %DOBOTH is called.
2			Macro variables within %DOBOTH receive values, e.g., &D receives the value of CLINICS
3	①	%SORTIT(&d,&b1,&b2,&b3)	The call for macro %SORTIT contains macro variable references. These references are resolved before the macro is called.
4	①	%SORTIT(CLINICS,LNAME, FNAME,)	The macro %SORTIT is called.
5			Macro variables within %SORTIT receive values, e.g., &DSET receives the value of CLINICS.
6	③	PROC SORT DATA=&DSET;	The PROC statement contains macro variable reference that must be resolved before the statement is submitted for execution.
7	③	PROC SORT DATA=CLINICS;	&DSET is replaced by CLINICS and the statement is ready to be submitted.
8	④	BY &BY1 &BY2 &BY3;	The BY statement contains macro variable references that must be resolved before the statement is submitted for execution.

Table (continued)

Step#	Line#	Code	What happens
9	④	BY LNAME FNAME ;	&BY1, &BY2, and &BY3 are replaced with their respective values and the statement is ready to submit (&BY3 is null).
10	⑤	RUN;	This statement contains no macro references and is submitted directly.
11			%SORTIT is complete and the next line in %DOBOTH is executed.
12	②	%LOOK(&d,&o)	The call for macro %LOOK contains macro variable references. These must be resolved before the macro is called.
		<<< remaining steps not shown >>>	The process of resolution and execution for %LOOK is similar to that for %SORTIT.

For the SORT, the process of the resolution of the data set CLINICS can also be viewed as:



5.1.2 Controlling macro calls

Often it is useful to create a SAS program that consists simply of calls to macros. This enables you to easily control, through macro calls, the execution of selected portions of the program.

The macro calls shown next execute macros that are either defined earlier in a portion of the program (not shown here) or are stored in a macro library (see Sections 3.3.3, 3.3.4, and 13.4). Although you do not have the definitions of the macros available to look at here, it is instructive to look at how the macro calls are set up.

`%SETUP` ❶ ❷ creates a data set (for example, `NEW1`) based on some criteria in the second and third parameters. `%ALL` ❸ then combines all of the `NEW` data sets (for example, `NEW1`, `NEW2`) that were created by the series of calls to `%SETUP` into the permanent data set `FINAL.COMBINE`. The two analysis macros (`%CORREL` ❹ and `%ANOVA` ❺) then operate on `FINAL.COMBINE` to do the selected analyses.

...code not shown...

```
*****;
**** CREATE DATA AS REQUIRED*****;
%MACRO DOIT;
  ❶ %SETUP(NEW1,15,32) * GATHER DATA FOR WORK.NEW1;
    %SETUP(NEW2,33,40) * GATHER DATA FOR WORK.NEW2;
  ❷ *%SETUP(NEW3,41,72) * GATHER DATA FOR WORK.NEW3;
  ❸ %ALL(FINAL.COMBINE) * COMBINE DATA FOR ANALYSIS;
%MEND DOIT;
%DOIT * CREATE THE ANALYSIS DATA SET;

**** ANALYZE COMBINED DATA *****;
❹%CORREL * CORRELATION OF P/C VARS;
❺%ANOVA * ANALYSIS OF VARIANCE (P/C);
```

Using this arrangement of statements gives you a lot of flexibility. You can use comments to eliminate calls to macros that are not required (as is the third call to `%SETUP` ❷) and all control is located within ten lines of code rather than spread throughout an entire program. Section 3.2 contains more information on commenting out macro calls. This type of arrangement of macro calls can also serve as documentation of what was executed, what the parameters were when executed, and in what order the macros were called.

A disadvantage of using the asterisk-style comment is that the comment is stored as part of the compiled macro. Comments defined with `/* ... */` and macro comments (see Section 5.4.1) are excluded from the compiled macro and can result in a savings in compilation time and storage size. Macro `%DOIT` could be rewritten as

```
%MACRO DOIT;
  %SETUP(NEW1,15,32) /* GATHER DATA FOR WORK.NEW1*/
  %SETUP(NEW2,33,40) /* GATHER DATA FOR WORK.NEW2*/
  /*%SETUP(NEW3,41,72) /* GATHER DATA FOR WORK.NEW3*/
  %ALL(FINAL.COMBINE) /* COMBINE DATA FOR ANALYSIS*/
%MEND DOIT;
```

The comments in this definition of %DOIT will be stripped off and the macro compiler will only see

```
%MACRO DOIT;
    %SETUP(NEW1,15,32)
    %SETUP(NEW2,33,40)
    %ALL(FINAL.COMBINE)
%MEND DOIT;
```

5.1.3 Nesting macro definitions

A nested macro definition occurs when the %MACRO through %MEND statements are contained within the definition of another macro. Almost always when you see this in a program, it is because the programmer does not truly understand how macro definitions are stored. It is only **very rarely** necessary or even appropriate to nest macro definitions.

In the following pseudo code a simple example of nested macros could be written as

```
%macro abc;
    ...code not shown...

    %macro def;
        ...code not shown...
    %mend def;
    %def
        ...code not shown...
    %mend abc ;
%abc
```

recompile every time when abc is called

Every time the macro %ABC is executed, the macro %DEF will be recompiled. There is no need for this. By moving the definition of the embedded macro (%DEF) so that it is not contained within the %MACRO and %MEND of the macro %ABC, it will be compiled only once. It does not matter which macro is defined first as long as both are defined before %ABC is called, as follows:

```
%macro abc;
    ...code not shown...
    %def
        ...code not shown...
    %mend abc;
%macro def;
    ...code not shown...
%mend def;
%abc
```

Now that I have said to never nest macro definitions, one legitimate use might be if the macro %ABC defined some conditions that were necessary in order to create a customized version of %DEF. Here again, though, I would tend to pass these conditions into %DEF as parameters, if at all possible, and use macro logic to perform the customization.

MORE INFORMATION

Another coding technique, in which nested macro definitions **might** have some utility, would be when using a macro to “comment” out a block of code. Section 3.1.2 discusses this technique in more detail. Additional discussion on nesting macro definitions can be found in Section 13.3.1.

SEE ALSO

The literature is replete with examples of nested macro definitions. This author found none that showed nesting when it was necessary.

5.2 Using Conditional Macro Statements

IF-THEN/ELSE processing in the DATA step enables you to write programs that will act conditionally based on values contained in variables in the Program Data Vector (PDV). The macro %IF-%THEN and %ELSE statements are analogous to the DATA step IF-THEN/ELSE statements. However, they are not constrained to the DATA step and do **not** operate on DATA step variables.

Macro %IF-%THEN and %ELSE statements may appear anywhere inside of a macro and are used to conditionally

- assign values to macro variables
- execute other macros
- create SAS code and SAS statements.

Syntax

```
%IF expression %THEN result-text ;
%ELSE result-text ;
```

Most of the same rules that apply to the DATA step IF-THEN/ELSE statements apply here as well. Three primary differences that you must keep in mind when using %IF-%THEN/%ELSE statements are

- They **do not** operate on the values of DATA step variables. Macro statements and macro references are resolved before **any** data are read. Macro expressions will **never** directly evaluate the value of a variable on the PDV.
- They control program flow and code construction, **not** the flow of the execution of a DATA step. Again it is very important to remember that whatever the macro statements do they will be resolved and executed long before the SAS program is compiled and executed.
- Comparisons are made on literal text strings and quotes are generally not used. For example, `X='X'` would be false.

MORE INFORMATION

Section 1.3 discusses the phases of macro execution and highlights the relationship between macro language and DATA step execution.

There is often initial confusion on the roles of the IF and %IF statements. Section 13.3.4 also addresses the differences.

SEE ALSO

Virgile (1997) uses the CALL EXECUTE routine to set up conditional macro logic outside of macros.

5.2.1 Executing macro statements

Like the IF statement's *action* in the DATA step, the *result-text* in the macro %IF takes place when the *expression* is true. The construction of the *expression* is similar to what it would be for the IF statement; however, it will **not**, as was hinted at earlier, be testing values of DATA step variables. This has been repeated multiple times because this is a major point of confusion for many first-time users.

Most of the standard DATA step comparison operators are available in the macro %IF expressions. One exception is the IN operator, which is not available; However, starting in SAS 9.1 a functional equivalent will be available. The macro IN operator is discussed in Section 5.2.3.

In the following example, the %IF tests the value of the macro variable &CITY. Notice that the value ALBANY is not enclosed in quotes as it would be if CITY were a character variable on the PDV. Remember macro variables are neither character nor numeric, but store text. Because macro variable names are always preceded by an ampersand (&), quotes are not needed. In this example, when the value ALBANY is stored in &CITY, the macro variable &STATE will be set to NEW YORK. For all other values of &CITY the second %LET statement is executed and &STATE will be set to CALIFORNIA.

```
%IF &CITY=ALBANY %THEN %LET STATE=NEW YORK;
%ELSE %LET STATE=CALIFORNIA;
```

Unlike in the DATA step where the programmer must be concerned about the length of a variable when assigning it values, in the macro language this is not an issue. In the previous example, although NEW YORK is coded first and is shorter than CALIFORNIA, there will be no truncation issues.

The macro %DOBOTH in Section 5.1.1 will fail with a syntax error if no BY variable values are passed into the macro. It would be nice if the %SORTIT macro was only executed if at least one BY variable is not blank. In the revised %DOBOTH below, it is assumed that if any BY variables are to be listed, at least one of them will be placed in &B1, and an %IF statement is then used to conditionally execute the %SORTIT macro:

```
%MACRO DOBOTH (d, o, b1, b2, b3);
    %IF &B1 ^= %THEN %SORTIT (&d, &b1, &b2, &b3);
    %LOOK (&d, &o)
%MEND DOBOTH;
```

Notice that the expression used in the above example would not work in a DATA step IF statement, since there is nothing on the right side of the equal sign. Because macro variables can contain a null value (remember that this is different from a blank), and because quotes are not needed, this syntax is correctly interpreted. But it does look strange. To make it look less strange, some programmers rewrite the %IF as

```
%IF "&B1" ^= "" %THEN %SORTIT (&d, &b1, &b2, &b3);
```

Using double quotes on both sides of the comparison operator is not recommended, and generally shows a poorer level of understanding of how the %IF comparison actually works. Also, there is a tendency to place a blank between the two double quotes on the right, but doing this will include the blank as part of the characters being compared and this would probably not be

desirable. When quotes are included they become a part of the text that is to be compared, and for this reason, one could not use single quotes on the right side and double quotes on the left.

In each of the previous examples the result-text is a macro statement or a macro call that is then immediately executed. When the result-text does not contain any macro variables or calls, the result-text is "left behind" and becomes part of the SAS code that will be compiled and executed.

MORE INFORMATION

Special considerations might be needed when comparing numeric values. See Sections 5.2.3, 7.3.2 (%EVAL) and 7.3.3 (%SYSEVALF) for more details.

Refinements of the above code that compares a macro variable to a null value can be found in Sections 7.1.2, 11.4.3, and the answer to question 4 in the Chapter 7 exercises (Q7.4).

5.2.2 Building SAS code dynamically

You can use the %IF to conditionally insert SAS code into a program. This enables you to create programs that are not fully defined until they are actually executed. In the following example, the data set that is named in the SET statement is not known until the parameter (&STATE), which is passed into %DASTEP, is evaluated.

```
%MACRO DASTEP(STATE);
DATA SUBHOSP;
  SET
    %IF &STATE=CA %THEN CAHOSP;
    %ELSE AZHOSP;
  ; ❶
  WHERE DATE>'19JUN96'D;
  RUN;
%MEND DASTEP;
```

Semicolon management

Note the use of semicolons in the statements shown above. Each macro statement ends with a semicolon. At first glance, it seems that these semicolons will end the SET statement or at least confuse the DATA step compiler. This will not be a problem, however, because all macro references and macro level statements will be resolved before this DATA step is compiled. Any semicolons that are associated with macro statements will no longer be present.

In this example, the semicolon that is used with the SET statement is on its own line ❶ (just after the %ELSE statement). It could also have been on the same line as the %ELSE. It was placed on its own line to make it easier for the programmer to read. SAS does not prefer one location over the other.

For the above macro definition, %DASTEP(AZ) returns the following code:

```
DATA SUBHOSP;
  SET
  AZHOSP
  ; ❶
  WHERE DATE>'19JUN96'D;
  RUN;
```

The section of code in the following example is one of several DATA steps in the macro %SENRATE. Macro %IF-%THEN/%ELSE logic is used to determine how the variable SEX is

to be defined. The DATA step will create one of a series of tables that are to be written to the library SENRATE. For table SENRATE.RTSEN5 (&NUM=5), the proper value for the variable SEX already exists in the variable GENDER. For all other values of &NUM; the PUT function is used to create the variable SEX.

```
%macro senrate(num);

...code not shown...

data senrate.rtsen&num;
set ratedata&num;
* Table 5 has sex already defined in GENDER;
* Table 3 & 7 use a format to define SEX;
%if &num=5 %then rename sex = gender; ❷
%else sex = put(gender, $sexdef.); ❷
; ❸

...code not shown...

%mend senrate;
```

- ❷ The semicolon in the %IF statement ends the %IF statement and not the assignment statement. The semicolon in the %ELSE statement ends the %ELSE and not the assignment statement. At this point neither of the assignment statements has their semicolon. Technically, until the %IF-%THEN/%ELSE executes, neither of the assignment statements has even been created. After the %IF-%THEN/%ELSE executes, only one of the assignment statements will have become part of the DATA step, and it is this assignment statement that receives the semicolon ❸.
- ❸ This semicolon will be used to close the assignment statement that results from the execution of the %IF-%THEN/%ELSE ❷.

For &NUM=3, the resulting DATA step would be

```
...code not shown...
data senrate.rtsen3;
set ratedata3;
* Table 5 has sex already defined in GENDER;
* Table 3 & 7 use a format to define SEX;
sex = put(gender, $sexdef.)
; ❸
```

Using a “hanging” semicolon as was done in the previous two examples is not unusual, but there are better coding solutions. One is to use the %DO block which is described in 5.3.1.

Misplacement of the semicolon can have some other unintended consequences. One might attempt to put the base language semicolon at the end of the base language statements ❹ rather than following the %ELSE statement:

```
%macro senrate(num);

...code not shown...

data senrate.rtsen&num;
set ratedata&num;
* Table 5 has sex already defined in GENDER;
* Table 3 & 7 use a format to define SEX;
```

```
%if &num=5 %then rename sex = gender;;④
%else sex = put(gender, $sexdef.);

...code not shown...

%mend senrate;
```

- ④ This causes a macro language error, because although we want the first semicolon to close the assignment statement and the second to close the %IF (or %ELSE) this is **not** what happens. The scanner detects the first semicolon and uses it to close the %IF. Although the second semicolon is not part of a macro language statement, the scanner knows that when a %ELSE follows a %IF, the %ELSE will be the next statement, and the existence of the second semicolon precludes that possibility. Now when the scanner finds the %ELSE it has no corresponding %IF (they have become disassociated by the second semicolon in the %IF) and the %ELSE causes an error.

MORE INFORMATION

The %DO block also is often used for semicolon management (see Section 5.3.1 for another version of the above example). The discussion of macro %DOBOTH in Section 5.3.1 also includes comments on the management of the semicolon.

SEE ALSO

Leighton (1997) contains several examples of dynamic code building.

The topics of semicolon management and, specifically, the use of the double semicolon are addressed by Yindra (1998).

5.2.3 Using the IN operator

The DATA step IN operator is not available in the macro language; however, starting in SAS 9.1, what is essentially the equivalent will be available. As with the DATA step you can use it to search for an item within a list of items.

Syntax

```
%IF item in target_list %THEN action;
```

```
%IF item # target_list %THEN action;
```

The IN operator can be replaced by a pound sign (#).

During evaluation the expression, which uses an implied %EVAL function (see Section 7.3.2), will return a 1 if *item* is found as one of the values in *target_list*. Otherwise it returns a 0. In the following %IF statement the expression is true because the value **aa** is in the list:

```
%if aa in aa bb cc dd %then %do;
```

Notice that the list is separated by spaces and that the second item are the letters `in`. The following would return a 0 (false) because the list will be seen as a single item (there are no spaces):

```
%if aa in aa,bb,cc,dd %then %do;
```

Having a space as the only possible list delimiter could be inconvenient, consequently the system option `MINDELIMITER` can be used to change the default delimiter from a space to some other character. In the next example the delimiter is set to a comma, and the expression in the `%IF` would now be true:

```
options mindelimiter=',';
%if aa in aa,bb,cc,dd %then %do;
```

In actual use the individual values in the list may be macro variables or the list itself might be a macro variable that contains the list. You will need to be careful when you create the list to make sure that you correctly specify the delimiter.

```
%macro testin(var,varlist);
  %if &var in &varlist %then
    %put Found |&var| in |&varlist|;
%mend testin;

%testin(aa, aa bb cc) ❶
%testin(AA, aa bb cc) ❷
```

❶ This will result in a true expression; however, ❷ will not since the test is case sensitive.

The `IN` operator requires an explicit `%EVAL` (see Section 7.3.1), when it is used outside of the `%IF` statement.

MORE INFORMATION

The construction of macro variables that contain lists is discussed in a number of sections within this book.

5.3 Iterative Execution Using Macro Statements

The `%DO` block, iterative `%DO`, `%DO %UNTIL`, and `%DO %WHILE` statements in the macro language are very similar to the corresponding statements that are used in the DATA step. Like the `%IF`, however, these statements are not confined to the DATA step. You can use them anywhere inside of a macro.

All forms of the `%DO` statement must be matched with an `%END`.

5.3.1 %DO block

The simplest form of the macro %DO statement is the %DO block, and it is analogous to the DATA step's DO block. Remember that the %DO statement is always paired with an %END.

Syntax

```
%DO;
...text...
%END;
```

The %DO block is most commonly needed when you want the result-text of an %IF-%THEN statement to contain multiple macro calls, multiple macro statements, or even multiple SAS statements.

You also can use the %DO block to help with semicolon management when you dynamically create code. The following examples remove the “hanging” semicolon problem that was shown in Section 5.2.2:

The code in the example that creates the data set SUBHOSP could be rewritten several ways to control the semicolon issue. The following code uses the %DO to enclose the semicolon:

```
%MACRO DASTEP (STATE) ;
  DATA SUBHOSP;
  SET
    %IF &STATE=CA %THEN %DO;
      CAHOSP;
    %END;
    %ELSE %DO;
      AZHOSP;
    %END;
WHERE DATE>'19JUN96'D;
RUN;
%MEND DASTEP;
```

Of course if you are going to use this type of solution, which is no longer dynamically creating the SET statement, you might as well enclose the whole SET statement inside of the %DO.

```
%MACRO DASTEP (STATE) ;
  DATA SUBHOSP;
  %IF &STATE=CA %THEN %DO;
    SET CAHOSP;
  %END;
  %ELSE %DO;
    SET AZHOSP;
  %END;
WHERE DATE>'19JUN96'D;
RUN;
%MEND DASTEP;
```

In a similar manner, the %DO block can be used to enclose the assignment statements in the %SENRATE macro (Section 5.2.2).

```
%macro senrate(num);

...code not shown...

data senrate.rtsen&num;
set ratedata&num;
* Table 5 has sex already defined in GENDER;
* Table 3 & 7 use a format to define SEX;
%if &num=5 %then %do; rename sex = gender; %end;
%else %do; sex = put(gender, $sexdef.); %end;

...code not shown...

%mend senrate;
```

It is easier to follow the logic in this code than it was in Section 5.2.2 because the %DO blocks contain complete SAS statements.

Along the same lines, macro %DOBOTH in Section 5.2.1 can be greatly simplified by eliminating the calls to %SORTIT and %LOOK. The new macro becomes

```
%MACRO DOBOTH(dsn,obs,by1,by2,by3);

  %IF &BY1 ^= %THEN %DO; ❶
    PROC SORT DATA=&DSN;
      BY &BY1 &BY2 &BY3;
    RUN;
  %END;

  PROC CONTENTS DATA=&dsn;
    TITLE "DATA SET &dsn";
  RUN;

  PROC PRINT DATA=&dsn
  %IF &OBS>0 %THEN %DO; ❷
    (OBS=&obs);
    TITLE2 "FIRST &obs OBSERVATIONS"
  %END;
  ; ❸
  RUN;
%MEND DOBOTH;
```

- ❶ The PROC SORT step will not be included in the final code when the &BY1 macro variable is null.
- ❷ In the PROC PRINT, &OBS is checked before the OBS= option is included in the PROC statement. This allows the insertion of a special title.
- ❸ Notice that the semicolon that follows the last %END statement will close either the TITLE or PROC statement, depending on the value in &OBS.

The following program conditionally executes blocks of SAS code. The macro %DOIT is used to create the temporary data set TDATA. The difficulty lies in the fact that there are two styles of input data (&STYLE) that must be handled differently. &STYLE is checked and the appropriate code is inserted into the program.

```

%macro doit;
* a macro is required because conditional macro logic statements
* are used;

* test for type of data and set up accordingly.;
%if &style=OLD %then %do;
    * read the data for a specific test;
    proc sort data=old.&dsn out=tdata;
    by varno age;
    where temp=&temp & age le &maxage;
    run;
%end;
%else %do;
    data tdata (keep=varno canno temp age depvar test package);
    set cps.dest (keep=base no &regvar.1-&regvar.6
                  temp test package);
    array can &regvar.1-&regvar.6;
    length age varno 8;
    where base="&dsn";

    ...code not shown...

    proc sort data=tdata;
    by varno age temp;
    run;
%end;
%mend doit;
%doit

```

The %DOIT macro is required because you can only use %IF statements within a macro.

5.3.2 Iterative %DO loops

Although the form of the iterative %DO is similar to the DO statement, it differs in that

- the %WHILE and %UNTIL specifications cannot be added to the increments
- increments are integer only
- only one specification is allowed.

Syntax

```

%DO macro-variable = start %TO stop <%BY increment>;
... text ...
%END;

```

The iterative %DO defines and increments a macro variable. In the following example, the macro variable &YEAR is incremented by one starting with &START and ending with &STOP ❶. Although the DATA step also has a variable YEAR, the two will not be confused. The incoming data sets are named YR95, YR96, and so on. These are read one at a time, and they are appended to the all-inclusive data set ALLYEAR ❷.

```

%MACRO ALLYR (START, STOP);
    %DO YEAR = &START %TO &STOP; ❶
        DATA TEMP;
            SET YR&YEAR;
            YEAR = 1900 + &YEAR;
        RUN;
        PROC APPEND BASE=ALLYEAR DATA=TEMP; ❷
        RUN;
    %END;
%MEND ALLYR;

```

The macro call %ALLYR (95, 97) generates the following code:

```

DATA TEMP;
    SET YR95;
    YEAR = 1900 + 95;
RUN;
PROC APPEND BASE=ALLYEAR DATA=TEMP;
RUN;

DATA TEMP;
    SET YR96;
    YEAR = 1900 + 96;
RUN;
PROC APPEND BASE=ALLYEAR DATA=TEMP;
RUN;

DATA TEMP;
    SET YR97;
    YEAR = 1900 + 97;
RUN;
PROC APPEND BASE=ALLYEAR DATA=TEMP;
RUN;

```

You can greatly simplify this code by taking better advantage of the %DO loop. Rather than having the macro create separate DATA and APPEND steps for each year, you can build the code dynamically.

```

%MACRO ALLYR (START, STOP);
    DATA ALLYEAR;
        SET
            %DO YEAR = &START %TO &STOP;
                YR&YEAR (IN=IN&YEAR)
            %END;;
        YEAR = 1900
            %DO YEAR = &START %TO &STOP;
                + (IN&YEAR* &YEAR)
            %END;;
    RUN;
%MEND ALLYR;

```


This time the call to %ALLYR (95, 97) produces the following:

```
DATA ALLYEAR;
  SET
    YR95 (IN=IN95)
    YR96 (IN=IN96)
    YR97 (IN=IN97)
  ;

  YEAR = 1900
    + (IN95*95)
    + (IN96*96)
    + (IN97*97)
  ;

RUN;
```

In this code, the value of YEAR is assigned by taking advantage of the IN= data set option. The values of IN95, IN96, and IN97 will be either true or false (1 or 0), and only one of the three will be true at any given time.

As in the previous example, the following macro, %READNEW, reads a series of data sets and builds the SET statement dynamically. In this example, however, the %DO loop is placed in the macro call rather than within the macro itself. In the call to %READNEW, the %DO loop is written as the value of the &DSIN parameter. This simplifies the code in the macro but makes the macro call more complex.

```
*  SOFT BENTHOS ABUNDANCE SUMMARY TABLES BY SPECIES;

%MACRO READNEW (DSIN=, DSOUT=);
DATA &DSOUT;
SET &DSIN;
  BY SURVEY; RETAIN SDATE;
  KEEP SPCODE DEPTH XLOC PERIOD DATE TCNTAB;

...code not shown...

%MEND READNEW;

*****;
%MACRO DOIT;
* Use selected surveys
*;
%READNEW (DSOUT=NEW1, DSIN=%DO I=16 %TO 18; DBBIO.SUR&I %END;)
%READNEW (DSOUT=NEW2, DSIN=%DO I=42 %TO 54; DBBIO.SUR&I %END;)
%READNEW (DSOUT=NEW3, DSIN=%DO I=55 %TO 72; DBBIO.SUR&I %END;)
%ALL
%MEND DOIT;

%DOIT          RUN;
```

CAVEAT: It is likely that the macro's author intended to pass the %DO loop into the macro where it would build the SET statement. This is **not**, however, what happens. Because in the macro call the parameter &DSIN contains the macro statement %DO, the loop will be executed and resolved before the resulting parameter value is passed to %READNEW. In the first call to %READNEW, it is the resolved value of the %DO loop that is passed. The resultant macro call will be the same as if the call had been coded as

```
%READNEW(DSOUT=NEW1,DSIN=DBBIO.SUR16 DBBIO.SUR17 DBBIO.SUR18)
```

Notice that no macro `%DO` statements were passed. Only their resolved values were passed. If you really do need to pass values that contain the symbols `%` and `&`, you may need to use quoting functions (see Section 7.1).

You must be careful when using statements like this in a macro call because the resolved text must be an appropriate parameter value. In this case everything worked out okay and the correct result was obtained, if not in the way the programmer expected.

It is also interesting to note that the three calls to `%READNEW` are contained within the macro `%DOIT`. If this had not been the case, (that is, if the calls to `%READNEW` had been in open code) the `%DO` statements would have caused an error because they can only be used within a macro. It is likely that the author of this macro expected the `%DO` to be inserted into `%READNEW` and then only surrounded the calls to `%READNEW` with the `%DOIT` definition when the first attempt did not work as anticipated.

SEE ALSO

Beverly (1999) introduces the `%DO` and Leighton (1997) includes examples of `%DO` loops that are used to build code dynamically.

5.3.3 %DO %UNTIL loops

Continuous loops or blocks of code can also be executed without the use of incremental variables. The `%DO %UNTIL` statement executes a block repeatedly until the specified condition is met. Be careful when using `%DO %UNTIL` as it is possible to construct an infinite loop. Because the `%DO %UNTIL` statement contains a logical macro expression and the loop continues to execute until the expression is true, be sure that the condition can be met.

Syntax

```
%DO %UNTIL(expression);
... text ...
%END;
```

When the expression in the parentheses is evaluated and the expression is false, the `%DO` statement is executed again. The loop continues to execute until the expression is true. Like the DATA step `DO UNTIL` statement, the expression is evaluated at the end of the loop. (This is sometimes expressed as “evaluation at the bottom of the loop.”) This means that the loop always executes at least once and will not execute again if the expression becomes true during the execution of the loop.

The following example rewrites the iterative `%DO` statement found in the first example in Section 5.3.2. The values of the two macro variables `&YEAR` and `&STOP` are compared to determine if the loop should execute again ❶. Even though the macro variable `&YEAR` in the comparison is not defined until the statement after the `%DO %UNTIL` statement ❷, there will be no error because the evaluation takes place at the bottom of the loop.

```
%MACRO ALLYR(START,STOP);
  %LET CNT = 0;
  %DO %UNTIL(&YEAR >= &STOP); ❶
    %LET YEAR = %EVAL(&CNT + &START); ❷
  %END;
```

```

DATA TEMP;
    SET YR&YEAR;
    YEAR = 1900 + &YEAR;
RUN;

PROC APPEND BASE=ALLYEAR DATA=TEMP;
RUN;

%LET CNT = %EVAL(&CNT + 1);
%END;
%MEND ALLYR;

```

- ❶ The `>=` comparison operator (rather than just an `=`) prevents an infinite loop when `&START` is larger than `&STOP` in the macro call. Because the comparison is evaluated at the bottom of the loop, the loop will execute at least once.
- ❷ This example uses the `%EVAL` macro function, which is discussed in more detail in Section 7.3.1. The `%EVAL` function is used to perform arithmetic operations within the macro language. A special function is required because the macro language does not distinguish between numeric and non-numeric values.

SEE ALSO

Tassoni, Chen, and Chu (1997) contains an example of a `%DO %UNTIL` statement.

5.3.4 `%DO %WHILE` loops

Like the `%DO %UNTIL` statement, the `%DO %WHILE` statement executes a block repeatedly. Unlike `%DO %UNTIL`, however, the `%DO %WHILE` loops are executed as long as or while the specified condition is met.

Syntax

```

%DO %WHILE(expression);
... text ...
%END;

```

Like the `DO WHILE` statement in the `DATA` step, the expression in `%DO %WHILE` is evaluated (at the top of the loop) before the loop is executed. This means that the loop will not automatically execute at least once as the `%DO %UNTIL` does.

The example in Section 5.3.3 could be rewritten as follows:

```

%MACRO ALLYR(START,STOP);
    %LET YEAR = &START;
    %DO %WHILE(&YEAR <= &STOP);

        DATA TEMP;
            SET YR&YEAR;
            YEAR = 1900 + &YEAR;
            RUN;
        PROC APPEND BASE=ALLYEAR DATA=TEMP;
            RUN;
    %END;
%MEND ALLYR;

```

```

      %LET YEAR = %EVAL(&YEAR + 1);
    %END;
  %MEND ALLYR;

```

SEE ALSO

Roberts (1997) uses the %DO %WHILE statement to decompose a macro variable list.

5.4 Macro Program Statements

The macro language is rich in program statements. Several of these have already been introduced and used in previous examples.

Statement	Section
%LET	2.1
%PUT	2.4
%SYMDEL	2.8
%MACRO	3.1
%MEND	3.1
%IF-%THEN	5.2
%ELSE	5.2
%DO	5.3.1 and 5.3.2
%END	5.3.1, 5.3.2, 5.3.3, and 5.3.4
%DO %UNTIL	5.3.3
%DO %WHILE	5.3.4

Additional macro program statements are introduced in the following subsections of this chapter.

Statement	Section
Macro comments	5.4.1
%GLOBAL	5.4.2
%LOCAL	5.4.2
%GOTO & %label	5.4.3
%SYSEXEC	5.4.4
%WINDOW	5.4.5
%DISPLAY	5.4.5

SAS 9 Statements	Section
%ABORT	5.4.6
%RETURN	5.4.7
%COPY	12.1.2

5.4.1 Macro comments

Macro comments behave in much the same way as the usual SAS asterisk-style comment. As a general rule, there is little advantage to using macro comments over standard comments. However, you might notice the following differences:

- The LOG will contain asterisk-style comments for the text that is generated by the macro when using the MPRINT system option (see Section 3.3.2). Macro comments and comments denoted by `/* ...*/` will not be shown in the LOG.
- Asterisk-style comments are stored as part of the macro text and, therefore, take up additional, albeit not many, resources.
- Macro comments can be used in places and ways that asterisk-style comments cannot be used.

Syntax

`%* comment text ;`

Macro comments and asterisk-style comments are complete statements, and as such they are tokenized. This means that they cannot contain embedded semicolons or unmatched quotation marks. Since the `/* ...*/` style comments are processed as individual characters (they are not tokenized) they can offer a bit more flexibility.

In the following example, the three styles of comments are used to document the macro `%DOIT`:

```
%macro doit;
/* This macro does nothing of interest;
 * This macro var &VAR remains unresolved;
/* Only the previous comment shows with MPRINT */
%mend doit;
```

When `%DOIT` is executed with the MPRINT system option turned on, the LOG shows only the asterisk-style comment following the macro call:

```
%doit

MPRINT(DOIT):  * This macro var &VAR remains unresolved;
```

Macro comments can be useful when you dynamically build SAS code. In Section 5.3.2, a `%DO` loop is used to build a SET statement, and a comment would provide helpful documentation. An asterisk-style comment could be used before the SET statement.

```
DATA ALLYEAR;
  * Include the data from the years of interest;
  * Use years from &start to &stop;
SET
%DO YEAR = &START %TO &STOP;
  YR&YEAR(IN=IN&YEAR)
%END;;
...code not shown...
```

For the years 94–96 the DATA step becomes

```
DATA ALLYEAR;
  * Include the data from the years of interest;
  * Use years from &start to &stop;
  SET
      YR94 (IN=IN94)
      YR95 (IN=IN95)
      YR96 (IN=IN96)
  ;
  ...code not shown...
```

The comments could not be placed just before the %DO statement as they would become embedded in the SET statement.

```
DATA ALLYEAR;
  SET
      * Include the data from the years of interest;
      * Use years from &start to &stop;
      %DO YEAR = &START %TO &STOP;
          YR&YEAR (IN=IN&YEAR)
      %END;;
  ...code not shown...
```

This code resolves to

```
DATA ALLYEAR;
  SET
      * Include the data from the years of interest;
      * Use years from &start to &stop;
      YR94 (IN=IN94)
      YR95 (IN=IN95)
      YR96 (IN=IN96)
  ;
  ...code not shown...
```

Since, of course, an asterisk-style comment cannot be embedded within other statements, this causes syntax errors when the DATA step is compiled.

You could, however, use macro-style comments to document the %DO process within what will become the SET statement.

```
DATA ALLYEAR;
  SET
      %* Include the data from the years of interest;
      %* Use years from &start to &stop;
      %DO YEAR = &START %TO &STOP;
          YR&YEAR (IN=IN&YEAR)
      %END;;
  ...code not shown...
```

The resulting DATA step code will contain

```
DATA ALLYEAR;
  SET
      YR94 (IN=IN94)
      YR95 (IN=IN95)
      YR96 (IN=IN96)
  ;
  ...code not shown...
```

You could also use a comment that is defined by `/* ... */`, as these comments can be embedded within other SAS statements.

5.4.2 %GLOBAL and %LOCAL

These statements create macro variables that are either available in all scopes or referencing environments (GLOBAL), or in a limited referencing environment (LOCAL). Unless first declared otherwise by these statements, macro variables are automatically assigned to a scope using a complex set of rules that are applied when the macro variables are first defined. The default assignment of macro variables to a scope and the issues associated with this assignment are described in more detail in Section 13.6.

When the %GLOBAL and %LOCAL statements are not used, macro variables are **generally**

- global when they are defined outside of a macro
- local to a macro when they are defined inside of a macro.

In the previous sentence the emphasis is on the word **generally**, because there are a number of important, and oftentimes subtle, exceptions to these two rules. The examples in this section follow these rules and they **do not** address these exceptions. The detailed rules for determining when a macro variable is to be global or local can be found in Section 13.6.

Referencing environments will not be a problem for you if you either define macro variables outside of all macros or pass macro variable values into and out of each macro as parameters.

Unfortunately these approaches often are not practical. For more complicated applications, it may be useful to create a macro variable in one process that you want to have available to other macros without having to physically pass it as a parameter. Understanding the scope of the macro variable becomes important for you at this point.

Syntax

```
%GLOBAL macro-variable-list;
%LOCAL macro-variable-list;
```

The %GLOBAL statement can be used in open code or within a macro definition, however the %LOCAL statement can only be used inside of a macro.

The following example does nothing except highlight the availability of macro variable values:

```
%let outside = AAA; ❶

%macro one;
  %global inone; ❷
```

```

    %let inone = BBB;
%mend one;

%macro two;
    %let intwo = CCC; ❸
%mend two;

%macro last;
    %one %two
    %put &outside &inone &intwo; ❹
%mend last;

%last

```

- ❶ &OUTSIDE is created outside of any macro. It is, therefore, a GLOBAL macro variable and is automatically available within any macro.
- ❷ The macro variable &INONE is globalized before it is defined. So, although it is defined inside of a macro (%ONE), it too is now available within other macros.
- ❸ &INTWO is created inside of %TWO and is not globalized. Its definition or even knowledge of its existence is limited to the macro %TWO.
- ❹ The %PUT statement in the macro %LAST demonstrates these availabilities. When %LAST is called, the following is written to the LOG:

```

WARNING: Apparent symbolic reference INTWO not resolved.
AAA BBB &intwo

```

%LOCAL seems to be used less frequently than %GLOBAL, and is almost certainly used less often than it should be. Most commonly, it is used as a means to protect the values of global macro variables. In the example in Section 5.3.3, the macro variable &YEAR is created in the macro %ALLYR. If %ALLYR is called within an application that has a globalized macro variable of the same name (&YEAR), %ALLYR will change the global value of &YEAR. By adding the %LOCAL statement in the following example, the globalized value of &YEAR will not be changed by the value local to %ALLYR. (As added protection &CNT is also included in the %LOCAL statement.)

```

%MACRO ALLYR(START,STOP);
    %local year cnt;
    %LET CNT = 0;
    %DO %UNTIL(&YEAR = &STOP);
        %LET YEAR = %EVAL(&CNT + &START);

        DATA TEMP;
            SET YR&YEAR;
            YEAR = 1900 + &YEAR;
        RUN;

        PROC APPEND BASE=ALLYEAR DATA=TEMP;
        RUN;

        %LET CNT = %EVAL(&CNT + 1);
    %END;
%MEND ALLYR;

```


The above example hints that it is possible for a macro variable to seem to have more than one definition at a given time depending on whether it is local or global. This dual (or should we say dueling) definition can be confusing, and you should avoid it when possible. It is possible to associate more than one value with the same variable name, because they are actually associated with different symbol tables. Since we do not have a way to name or point to the symbol table of interest, we have to keep track of the symbol tables ourselves. Usually this is not a major issue.

The following macros write two different values for &AA to the LOG. Resetting &AA in %INSIDE does not change its value in the %OUTSIDE macro.

```
%macro outside;
    %let aa = 5;
    %inside(3)
    %put outside &aa;
%mend outside;

%macro inside(aa);
    %put inside &aa;
%mend inside;

%outside
```

In this example, &AA is local to two different macros at the same time and has two different values at the same time. The LOG will contain

<pre>inside 3 outside 5</pre>

The following two examples further demonstrate the difference between these two scopes and the need to protect GLOBAL macro variables. In the following example, &BB is defined using a %LET statement outside of the macro definition, which by default makes it GLOBAL. The macro %INSIDE, however also assigns a value to &BB, therefore when %INSIDE is called the value of &BB is redefined **in the global symbol table** not the symbol table local to %INSIDE.

```
%macro inside(var);
    %let bb = &var;
    %put inside &bb;
%mend inside;

%* This LET statement makes &BB global;
%let bb = 5;
%inside(3)
%put outside &bb;
run;
```

&BB is global, so when it is redefined in %INSIDE the value of &BB is changed outside of the macro %INSIDE as well. The LOG will contain the following:

<pre>inside 3 outside 3</pre>

Adding a %LOCAL statement to %INSIDE protects the global value of the macro variable:

```
%macro inside(var);
    %local bb;
    %let bb = &var;
    %put inside &bb;
%mend inside;

%let bb = 5;
%inside(3)
%put outside &bb;
run;
```

In this example, &BB is again a global macro variable. However, because of the %LOCAL statement, its instance in %INSIDE is local to that macro. This means that although the macro variable &BB is redefined in %INSIDE, the value is not changed in the global symbol table (outside of %INSIDE). The LOG will contain the following:

```
inside 3
outside 5
```

5.4.3 %GOTO and %label

These two statements are included in this book because you might encounter them someday in someone else's code (warning: subtle author bias may be encountered in this subsection). These

statements, like other directed branching statements, enable you to create code that is **very** unstructured. So far (when I have tried hard enough), I have always been able to find better ways of solving a problem (both in coding SAS and in my personal life) other than by using GOTO and %GOTO type statements. My first choice is to use alternative logic, thereby avoiding the use of these statements.

Like the DATA step GOTO statement, %GOTO (or %GO TO) causes a logic branch in the processing. The branch destination will be a macro label (%label). Therefore, the argument associated with the %GOTO must resolve to a known %label.

Syntax

```
%GOTO label;
or
%GO TO label;

%label:
```

The *label* associated with the %GOTO statement must resolve to a macro label that you have defined somewhere within the macro. The label may be explicitly or implicitly named. In the following example, the label is named explicitly. The branch will be to the statement following the %NEXTSTEP: label.

```
%GOTO NEXTSTEP;
...code not shown...
```

```
%nextstep:
```

In code that uses %GOTO, it is not unusual for the %GOTO statement to include a label that contains a reference to a macro variable that must be resolved before the %GOTO is executed. In the following statement &STEP must resolve to a defined macro label—for example, NEXTSTEP—before the branch can take place:

```
%GOTO &STEP;
```

Because the macro label is preceded by a %, the new user often uses a % with the *label* in the %GOTO statement, as in this statement:

```
%GOTO %STEPTOT;
```

Rather than branching to the specified *%label*, however, a call to execute the macro %STEPTOT will be issued before the %GOTO can be executed. Generally, this will result in an error, but it can work if the macro %STEPTOT resolves to the name of a macro label.

In the following example, %GOTO is used to determine which of two DATA steps will be executed. The macro labels are explicitly defined in the %GOTO statements.

```
%macro mkwt(dsn);
  %* Point directly to the label;
  %if &dsn = MALE %then %goto male;
  data wt;
    set female;
    wt = wt*2.2;
  run;
  %goto next;
%male:
  data wt;
    set male;
  run;
%next:
%mend mkwt;
```

You can rewrite this example to use implicit labels that reflect the incoming macro variable (&DSN). This makes the use of the %IF unnecessary.

```
%macro make(dsn);
  %* Point indirectly to the label;
  %goto &dsn;
%female:
  data wt;
    set female;
    wt = wt*2.2;
  run;
%goto next;
%male:
  data wt;
    set male;
  run;
%next:
%mend make;
```

Admittedly, this is a rather simplistic case, but you can easily rewrite the previous examples to avoid the use of the %GOTO altogether:

```
%macro smart(dsn);
  %*AVOID GOTO WHEN POSSIBLE;
  data wt;
  set &dsn;
  %if &dsn=FEMALE %then wt = wt*2.2;;
  run;
%mend smart;
```

MORE INFORMATION

The %GOTO statement is used in %TRIM, an autocall macro supplied by SAS, which is discussed in Section 12.3.5. The macro %TF in Section 10.2.1 also uses the %GOTO.

SEE ALSO

The %GOTO statement and %label macro is used by Wang (2003) in a %WINDOW example. Lund (2003a) uses %GOTO to skip the execution of a macro.

4.4 Using %SYSEXEC

You can issue operating environment commands from within a macro or in open code if you use the %SYSEXEC macro statement. This statement is analogous to the X statement and replaces the earlier statements of %TSO, %CMS, and so on.

- Specified operating system commands are executed immediately.
- Any return codes are assigned to the automatic macro variable &SYSRC.

Syntax

%SYSEXEC *operating system command*;

Enter the operating system command just as you would enter it at your operating system prompt. The command is **not** enclosed in quotes. Everything between the %SYSEXEC and its semicolon is passed directly to the operating system for immediate execution.

In the following MVS statement, the partitioned data set member CLEANUP will be executed. The quotes are needed here because they are expected by the operating system:

```
%SYSEXEC ex 'userid.mytools.sascode(cleanup)';
```

The following %SYSEXEC statement is followed by a query to its return code, which is stored in the automatic macro variable &SYSRC:

```
%sysexec dir *.sas;
%if &sysrc = 0 %then %do; ....
```

If you have not used the X statement before on your operating system, you should consult the appropriate *SAS Companion* so that you will know what to expect in terms of behavior for %SYSEXEC and what values can be taken on by &SYSRC.

MORE INFORMATION

The example in Section 10.1.6 uses the %SYSEXEC statement to create a system directory under Windows.

SEE ALSO

The %SYSEXEC statement is compared to the X statement by Wang (2003).

5.4.5 Creating macro %WINDOWS

Through the use of the %WINDOW statement, the macro language provides the programmer with a tool that can be used to establish a basic user interface. Similar to the WINDOW statement in the DATA step, %WINDOW can be used to create and display message boxes and to collect information from the user that can then be placed into macro variables.

The %WINDOW statement can be used to

- display a window
- control window attributes including size and color
- make use of existing key and menu definitions
- display existing macro variable values
- define and assign values to macro variables.

Once a window has been defined with the %WINDOW statement, it can then be displayed by using the %DISPLAY statement. The %WINDOW and %DISPLAY statements can be used in open code.

Syntax

%WINDOW *window-name* <attributes and display characteristics>;

%DISPLAY *window-name* <display control options>;

The following macro defines and then displays a macro window, which prompts the user for the name of a data set:

```
%macro dsnprompt(lib=sasuser);
%* prompt user to for data set name;
%window verdsn color=white ❶
  #2 @5 "Specify the data set of interest" ❷
  #3 @5 "for the library &lib" ❸
  #4 @5 "Enter Name: "
      dsn 20 ❹ ATTR=UNDERLINE REQUIRED=YES ❺;

%display verdsn; ❻

proc print data=&lib..&dsn;
run;
%mend dsnprompt;

%dsnprompt(lib=sasclass)
```

When the %DSNPROMPT macro is executed, the VERDSN macro window will be defined and displayed.

- ❶ The VERDSN window will have a white background with no specifications for size.
- ❷ The text (in single or double quotes) is to be displayed at row 2 and column 5. The same notation for row (#) and column (@) is used as in the PUT and INPUT statements.
- ❸ Macro variables can be included in the text (see caveat below).
- ❹ The user is prompted for the name of a data set which is placed into &DSN.
- ❺ Attributes can be assigned to the display of the macro variable.
- ❻ Although defined, the VERDSN window is not displayed until the %DISPLAY statement is executed.

CAVEAT: In the above example the VERDSN window is defined when the macro is executed. It is during this definition phase that &LIB at ❹ is resolved. If &LIB is redefined at a later time (after the %WINDOW statement has executed), then %DISPLAY will still show the original value, **not** the current value of &LIB.

SEE ALSO

Many of the options associated with %WINDOW are introduced and discussed by Alden (2000) and Mace (1997, 1998, 2000, 2002, and 2003). These papers provide very nice overviews as well as detailed (and in most cases more sophisticated) examples of macro windows.

Gau (1999) presents an example of the %WINDOW to manage programs. Ren (1999), Parker (2000), Dynder, Cohen, and Cunningham (2000), Fahmy (2003), Huang (2003), Wang (2003), Parker (2003), and Rhoads and Letourneau (2002) each use a macro window to create user interfaces. Plath (2002) creates and executes a series of macro windows to collect information from the user.

Access control to macros is achieved through the use of the WINDOW statement in an example by Shilling and Kelly (2001).

5.4.6 Abnormal termination of macro execution with the %ABORT statement

Like the ABORT statement in the DATA step, %ABORT (available in SAS 9) can be used to control a program's continued execution based on factors that you can specify. When executed, this statement stops the execution of the current macro with an abnormal termination and, depending on its arguments and conditions of execution, can potentially stop the SAS job or SAS session. In each case an error message is written to the LOG.

Syntax

```
%ABORT <abend>return<n>>;
```

You might need to experiment with this statement and its arguments. In each case the current macro's execution is terminated. %ABORT without any arguments seems to be the most benign and with the ABEND argument the most severe.

In the following %ABORT statement the return code of 27 will be passed back to the operating environment.

```
%abort return 27;
```

As is the case with the DATA step ABORT statement, the severity of the termination will depend a lot on how and when the %ABORT is executed and under what operating environment (batch or interactive) the macro is executing. Since this is considered an abnormal termination, how your operating system treats abnormal terminations may also determine whether or not the whole SAS session is terminated.

MORE INFORMATION

The %ABORT statement is used in %CHKCOPY in Section 10.1.4.

5.4.7 Normal termination of macro execution with the %RETURN statement

Unlike the %ABORT statement, which results in an abnormal termination of a macro's execution, the %RETURN statement (available in SAS 9) can be used to cause normal termination of a macro. The consequences for using %RETURN as opposed to %ABORT are therefore much less severe.

Syntax

```
%RETURN;
```

Upon execution, which is usually done conditionally using an %IF, the current macro ceases executing and control is returned to the program immediately following the macro call. The following macro only operates on data sets with more than 24 observations:

```
%macro means(dsn);
  %if %obsent(&dsn) < 25 %then %return;
  proc means data=&dsn noprint;
    ...code not shown...
%mend means;
```

MORE INFORMATION

The %OBSCNT macro is described in Section 11.5.1.

5.5 Chapter Summary

The %IF-%THEN and %ELSE statements are similar to the IF-THEN/ELSE statements used in the DATA step, except that the macro %IF statement is not constrained to a DATA step. You can use the %IF-%THEN statement to conditionally insert SAS code into a program.

The %DO block, iterative %DO, %DO %UNTIL, and %DO %WHILE statements in the macro language are very similar to the corresponding statements used in the DATA step. Like the %IF statement, however, these statements are not confined to the DATA step and you can use them

anywhere inside of a macro. You must match each form of the %DO statement with an %END statement.

Continuous loops or blocks of code can also be executed, without the use of incremental variables, **until** or **while** certain conditions hold. The %DO %UNTIL statement executes a block repeatedly, until the specified condition is met; and the %DO %WHILE statement executes a block repeatedly, while the specified condition is met.

Macro comments give the macro programmer flexibility when documenting macros and macro statements.

The %GLOBAL and %LOCAL statements are used to specify the scope of macro variables. By controlling the scope of a macro variable you can control whether or not the value of a macro variable is currently available.

Similar to the X statement in Base SAS, the %SYSEXEC macro statement can be used to execute operating system commands.

The %WINDOW and %DISPLAY statements are used to build user interfaces that can directly create macro variables.

Other macro statements, such as %ABORT and %RETURN, give the programmer additional control over the processing of the macro.

5.6 Chapter Exercises

- Using the following macro (%PRINTT), construct the macro call that will invoke the PRINT procedure using a SAS data set called CLINICS:

```
%MACRO PRINTT(DSIN, PROC=) ;
    PROC &PROC DATA=&DSIN;
    RUN;
%MEND PRINTT;
```

- Use the %DO, %DO %UNTIL, or %DO %WHILE statement to construct a macro that will print the message "This is Test *number*" in the LOG ten times, each time displaying the test *number*: 1, 2, 3, 4, and so on.

Extra Credit: Solve this exercise three times, once for each type of %DO.

- (True or False) Every macro definition must end with a %MEND statement.
- (True or False) The %LET statement can be used inside as well as outside of a macro.
- Which statement(s) below is (are) syntactically incorrect? Why?
 - %GOTO %PRINT;
 - %GLOBLE SAVE;
 - %IF &SEX = M THEN %PUT MALE;

6. Write a macro that will be generic enough to handle the MEANS and UNIVARIATE procedures. Include the following minimum information as macro variables:
 - name of the procedure
 - title line #1
 - VAR statement for selection of numeric variables.
7. Convert the following PROC MEANS step to a macro that will either print a standard set of statistics or generate the means as shown, depending on whether or not the user specifies an output data set:

```
proc means data=sasclass.clinics noprint;
var ht wt;
output out=stats mean= max= / autoname;
run;
```

Extra Credit: Enable the user to specify the variable list and the statistics that will be printed or written to the new data set.

8. The following program uses the macro variable names &INDAT and &OUTDAT in both of the macros %REGTEST and %DOTESTS. When %REGTEST is called from within %DOTESTS are the values of these variables in %DOTEST changed? This question can be restated as “is a %LOCAL statement required?”

```
%macro regtest(indat,outdat);
  proc reg data=&indat;
    model count=distance;
    output out=&outdat r=resid;
  run;
%mend regtest;

%macro dotests(indat,outdat);
  data r1;
    set &indat;
    if station=1;
  run;
  %regtest(r1,r1out);

  data r2;
    set &indat;
    if station=2;
  run;
  %regtest(r2,r2out);

  data &outdat;
    set r1out r2out;
  run;
%mend dotests;

%dotests(biodat,bioreg)
```

9. In the macro %TRY, shown here, which %PUT statements will be executed?

```
%let a = AAA;
%macro try;
%put &a;
%if &a = AAA %then %put no quotes;
%if '&a' = 'AAA' %then %put single quotes;
%if 'AAA' = 'AAA' %then %put exact strings;
%if "&a" = "AAA" %then %put double quotes;
%if "&a" = 'AAA' %then %put mixed quotes;
%if "&a" = AAA %then %put quotes on one side only;
%mend;
%try
```

10. The macro %ALLYEAR (Section 5.3.2) does not meet Y2K programming standards.

Fix the macro assuming that the years range from 2000 to 2004.

Extra Credit: Alter %ALLYEAR to allow years from 1990 to 2004.



Chapter 6

Interfacing with Data

- 6.1 Using the SYMPUT Routine 102
 - 6.1.1 First argument of SYMPUT 104
 - 6.1.2 Second argument of SYMPUT 104
 - 6.1.3 SYMPUT example 105
 - 6.1.4 Using SYMPUTX 105
- 6.2 Using a SAS Data Set As a Control File 107
 - 6.2.1 Macro variable values 107
 - 6.2.2 Assigning macro variable names as well as values 113
- 6.3 Macro Variable Forms Used in Dynamic Programs 116
 - 6.3.1 Elements of a dynamic program 116
 - 6.3.2 Indirect macro variable references 118
 - 6.3.3 Sources of control information 118
- 6.4 Moving Text from Macro to DATA Step Variables 119
 - 6.4.1 Assignment and RETAIN statements 119
 - 6.4.2 Using the SYMGET function 120
 - 6.4.3 Using the RESOLVE function 122
- 6.5 Doing More with the SQL Step 127
 - 6.5.1 Placing a single value into a single macro variable 128
 - 6.5.2 Creating more than one macro variable 128
 - 6.5.3 Placing a list of values into a series of macro variables 129
 - 6.5.4 Using the SQL dictionary tables 132
 - 6.5.5 Automatic SQL generated macro variables 133

6.6 Execution of Macro Code Using CALL EXECUTE	134
6.6.1 Executing non-macro code	135
6.6.2 Executing macro code	135
6.6.3 Timing issues	138
6.7 Chapter Summary	141
6.8 Chapter Exercises	142

Very often in macro language programs and applications you will want the data itself to drive the definitions of the macro variables and parameters. The data used to create the macro variable definitions might be the same data that is to be analyzed, or you might have some kind of control file that can be used to provide directions to your program. In either case, you need the ability to take information that is stored in a SAS data set or in some other format and convert it into values for macro variables. This chapter introduces various aspects of this process.

6.1 Using the SYMPUT Routine

In Section 1.3 it was shown that macro language references are resolved and executed long before the DATA step is compiled and executed. This means that you cannot use the %LET statement to assign a DATA step variable's value to a macro variable. In the following pseudo code, the programmer wants to assign the value of the variable FNAME to the macro variable &FIRSTN:

```
data new;
  set old;
  %let firstn = fname;
run;
```

Instead, because the %LET statement is executed long before the DATA step is even completely compiled, the macro variable &FIRSTN will contain the lowercase letters f-n-a-m-e. To get around this problem we need to be able to use the DATA step to place information onto the macro variable symbol tables. We can do this with SYMPUT, which is a DATA step call routine.

Like the %LET statement, the SYMPUT call routine can be used to assign a value to a macro variable. However, SYMPUT is **not** a macro level statement; instead, it is a DATA step routine. As such, it is used as part of a DATA step and enables you to directly assign values of data set variables to macro variables during DATA step execution.

The routine has two arguments, either of which may be a variable or a constant (such as a character string) or a combination of the two. The first argument identifies the name of the macro variable, and the second argument identifies the value to be assigned to it.

Syntax

```
CALL SYMPUT(macro_varname,value);
```

SYMPUT assumes that the second argument is character and will automatically convert it to character (with a message written to the LOG) if it is numeric.

It is not uncommon for users to try to access a macro variable in the same DATA step in which it is created. However, you **cannot** define a macro variable using SYMPUT and then attempt to

access that macro variable using an ampersand (&) in the same DATA step. This makes sense (see Section 1.3) when you remember that values are assigned to the macro variable through SYMPUT during **DATA step execution**. However, any macro variable references involving the use of an & are resolved before the compilation phase of the DATA step. And since the DATA step execution phase comes after the compilation of the entire DATA step is complete, the macro facility would be attempting to resolve a macro variable that would not be defined until the DATA step's execution phase. It is generally considered unfair to ask any language to resolve or use variables that have not yet been defined. As is demonstrated in the following simplistic example, this means that macro variable values cannot be obtained by referencing them with an & in the same step that they are created:

```
data new;
  set old;
  call symput('firstn',fname); ❷
  nickname = "❶&firstn"; ❸
run;
```

- ❶ Before this DATA step statement can even be compiled, the macro variable reference &FIRSTN must be resolved. Since this variable has not yet been defined by the CALL SYMPUT at this point, a warning message stating that &FNAME is not resolved will be written to the LOG.
- ❷ During DATA step **execution**, the macro variable FIRSTN receives the value contained in the DATA step variable FNAME.
- ❸ Since the &FIRSTN was unresolved during the macro resolution phase, the DATA step variable NICKNAME will contain the characters exactly like they are within the quotation marks (& and all).

MORE INFORMATION

CALL SYMPUT will create a global macro variable if it is used in open code. However, when it is used inside of a macro the macro variable will usually be local to that macro. The detailed rules for determining whether the macro variable will be global or local are described in Section 13.6.

It is very rare that you truly need to build a macro variable and then use it again in the same DATA step. After all, you have the value in a DATA step variable already, so why not use it? When you do need to access the macro variable symbol tables from within a DATA step, you will need to use either the SYMGET or RESOLVE functions. These are described in more detail in Sections 6.4.2 and 6.4.3.

SEE ALSO

An overview to the use of the SYMPUT routine can be found in Palmer (1998 and 2001).

6.1.1 First argument of SYMPUT

The first argument of SYMPUT is used to name the macro variable that is to be either created or assigned a value. The argument may be either a character string, DATA step variable, or a combination of the two. Each of the following three examples assigns a value to the macro variable &DSN1:

Character string

A character string enclosed in quotes creates a macro variable with the name that is enclosed in the quotes:

```
CALL SYMPUT('DSN1', argument2);
```

Character variable

When the first argument is a DATA step character variable, its value becomes the macro variable name:

```
macvar = 'DSN1';  
CALL SYMPUT(macvar, argument2);
```

Character variable and string used together

A character expression, which can be made up of any combination of character variables and strings, may be used to determine the macro variable name:

```
n=1;  
CALL SYMPUT('DSN' || LEFT(PUT(n,12.)), argument2);
```

In this case 'DSN' will have a '1' appended onto it forming 'DSN1'. Notice that the PUT function converts the numeric variable N to a character string that is left-justified and is appended using the concatenation operator (||).

6.1.2 Second argument of SYMPUT

The second argument contains the value that is to be assigned to the macro variable named in the first argument. As is the case with the first argument, the second argument can be a character string, variable name, or a combination of the two. In each of the following examples, the macro variable &DSN1 is assigned the value of CLINICS:

Character string

The character string becomes the value assigned to the macro variable:

```
CALL SYMPUT('DSN1', 'CLINICS');
```

Character variable

The value of the DATA step variable is assigned to the macro variable:

```
dataset='CLINICS';  
CALL SYMPUT('DSN1', dataset);
```

Character variable and string used together

The resolved value of the character expression is assigned to the macro variable:

```
PARTNAME='CLIN';
CALL SYMPUT('DSN1',TRIM(PARTNAME)||'ICS');
```

The TRIM function removes trailing blanks and is needed if PARTNAME has a length that is greater than the number of nonblank characters. If the length of PARTNAME was \$6, and you did not use TRIM, the character expression would become CLIN ICS, and this is what would be stored in &DSN1.

6.1.3 SYMPUT example

The TITLE1 statement in the following example contains a macro variable (&SEX) that is defined in the previous DATA step. &SEX will contain the value of the data set variable SEX. Because the SYMPUT routine is executed for each observation that is processed (each pass of the DATA step), this code will reassign the value of &SEX for each observation that meets the WHERE criteria. In this example the macro variable &SEX will contain the value of the DATA variable SEX in the last observation read from the data set CLINICS.

```
data regnl;
set clinics;
where reg='1';
call symput('sex',sex);
run;

title1 "Region 1 data for &sex";
```

In this example, the TITLE1 statement **cannot** be placed within the DATA step.

MORE INFORMATION

Each of the examples in Sections 6.2 and 6.3 also use the SYMPUT routine.

SEE ALSO

Landers and Bryher (1997) has several examples that highlight various behaviors of the SYMPUT routine under different conditions.

6.1.4 Using SYMPUTX

Starting in SAS 9 the SYMPUTX routine can be used to directly store numeric (or even character) values. This routine is very similar to SYMPUT; however, it has an optional third argument that can be used to specify the symbol table to which the macro variable is to be added.

Syntax

```
CALL SYMPUTX(macro_varname,value,<symbol_table_code>);
```

The optional third argument is a code that can be used to alter or control the destination symbol table. Although both SYMPUT and SYMPUTX by default generally write to the most local symbol table, this is not always the case (Section 13.6 discusses the rules for these assignments).

When the third argument is specified as one of the following codes, the macro variable will be written to one of three possible locations:

Code	Location
G	the global symbol table
L	the most local symbol table that currently exists
F	the most local symbol table that already contains this macro variable. If the variable does not yet exist, it will be written to the most local table that currently exists.

This routine will not write a message to the LOG when it receives a numeric second argument. In addition, it left-justifies and trims the converted value. In the following DATA step the value in the numeric variable X is written to &X1 and &X2:

```

16  data a;
17  x = 5;
18  call symput('x1',x);
19  call symputx('x2',x);
20  run;

```

NOTE: Numeric values have been converted to character values at the places given by:

(Line):(Column).

18:18 ❶

NOTE: The data set WORK.A has 1 observations and 1 variables.

NOTE: DATA statement used (Total process time):

real time 0.06 seconds

cpu time 0.02 seconds

```

21  %put |&x1|;
    |      5|
22  %put |&x2|;
    |5| ❷

```

Notice that SYMPUT generates a conversion NOTE ❶ and that the value of &X2, which is generated with SYMPUTX, is left-justified. ❷ Most current users have not yet transitioned to SAS 9; as they do I anticipate that SYMPUTX will replace SYMPUT in many coding situations.

MORE INFORMATION

The SYMPUTN function is similar to the SYMPUTX in that it can write numeric values without generating NOTES in the LOG; however, SYMPUTN is a SCL routine and is not available in the DATA step.

6.2 Using a SAS Data Set As a Control File

Often a number of macro variables need to be defined in order to control a process or series of programs. Rather than pass the values into macros through the use of parameters, the control information can be stored in SAS data sets. These data values can then be used to control a macro by creating macro variables with the SYMPUT routine.

Control data sets are most often used when the number of macro variables is large or when there are a great number of macro calls.

Control files often contain information about data sets. This data about data is also known as *metadata*, and there are a number of sources of metadata in addition to those that are constructed by the user. These include VIEWS provided by SAS.

MORE INFORMATION

A number of sections in this book demonstrate the use of metadata to build macro variables. In addition to the following sections (6.2.1 and 6.2.2), look in Sections 6.5.4 and 9.5.

SEE ALSO

Zhang, Chen, and Wong (2003) provide an overview of metadata within the context of a data warehouse.

6.2.1 Macro variable values

The following rather simplistic example illustrates the steps that are needed to control a macro using values that are contained in a data set. Two macro variables, &DSN and &OBS, are to be assigned values that are stored in the data set CONTROL using the variables DSNAME and NOBS.

```
DATA CONTROL;
    DSNAME='CLINICS';
    NOBS='5';
RUN;
%MACRO LOOK;
    DATA _NULL_;
        SET CONTROL;
        CALL SYMPUT('DSN',DSNAME);
        CALL SYMPUT('OBS',NOBS);
    RUN;
    PROC CONTENTS DATA=&DSN;
    RUN;
    PROC PRINT DATA=&DSN (OBS=&OBS);
    RUN;
%MEND LOOK;
```

SAS generates the following code when %LOOK is executed:

```
PROC CONTENTS DATA=CLINICS;
RUN;
PROC PRINT DATA=CLINICS (OBS=5);
RUN;
```

In the above example, the control file contains only the value of the macro variables and not the name of the macro variable. Because each variable in the data set has a distinct value, only one observation is needed.

In the previous example NOBS is defined as a character variable of length 1. If NOBS had been a numeric variable, then the SYMPUT routine would have converted its value to a character string before building the macro variable.

```
DATA CONTROL;
    DSNAME='CLINICS';
    NOBS=5;
RUN;
```

The resolved macro code then becomes

```
PROC CONTENTS DATA=CLINICS;
RUN;
PROC PRINT DATA=CLINICS (OBS=      5);
RUN;
```

Notice the large number of spaces that are inserted before the 5. The spaces result from the conversion of the numeric value to a character string by the SYMPUT function. The LOG also displays a conversion note: "Numeric values have been converted to character values." The spaces, but not the note, can be removed by using the TRIM and LEFT functions.

```
CALL SYMPUT('OBS',trim(left(NOBS)));
```

When placing a numeric value into a macro variable, the PUT function (or the related functions PUTN and PUTC) prevents the generation of the note and can be used to fully control the conversion process. The following statement converts NOBS into a character string, which is then left-justified and trimmed prior to building the macro variable &OBS.

```
CALL SYMPUT('OBS',trim(left(put(NOBS,3.))));
```

If it is too small, the format used in the PUT function can limit the size of the number to be converted. For instance, in the previous statement, NOBS could be no larger than 999. In practical applications there is no real penalty for using a format that is quite a bit larger than the largest number.

```
CALL SYMPUT('OBS',trim(left(put(NOBS,best12.))));
```

If you are using SAS 9 or later, the SYMPUTX routine (see Section 6.1.4) can be used to automatically trim, left-justify, and convert the numeric value. The statement becomes

```
CALL SYMPUTX('OBS', NOBS);
```

Control data sets may not always be available or indeed even needed. Often control parameters can be determined directly from the analysis data, thus reducing the need for a separate control data set. The following example produces a separate plot for each value of the variable REGION and places the region name in the title. A similar series of plots could have been generated by using the BY statement in PROC GPLOT along with the #BYVAL option. This method, however, allows more flexibility in more complex situations:

```
%MACRO PLOTIT;
    PROC SORT DATA=CLINICS;
        BY REGION;
    RUN;
```

```

* Count the unique regions and create
* a macro variable for each value.;
DATA _NULL_;
  SET CLINICS;
  BY REGION;
  IF FIRST.REGION THEN DO; ❶
    * Count the regions;
    I+1; ❷
    * Create char var with count (II). Allow;
    * up to 99 unique regions;
    II=LEFT(PUT(I,2.)); ❸
    * Assign value of region to a mac var;
    CALL SYMPUT('REG'||II,REGION); ❹
    CALL SYMPUT('TOTAL',II); ❺
  END;
RUN;

* Do a separate PROC GPLOT step for;
* each unique region;
%DO I=1 %TO &TOTAL; ❻
  PROC GPLOT DATA=CLINICS;
    PLOT HT * WT;
    WHERE REGION="&&REG&I"; ❼
    TITLE1 "Height/Weight for REGION &&REG&I"; ❼
  RUN;
%END;
%MEND PLOTIT;

```

This code will execute PROC GPLOT once for each unique value of the variable REGION. The macro variable &TOTAL counts the number of REGIONS and is used to set up a macro %DO loop (see Section 5.3.2). &®&I acts as a macro vector that contains the various values of the variable REGION (Section 2.5.3 discusses the resolution of double ampersand macro variables).

- ❶ FIRST.REGION is used to determine the unique values taken on by the variable REGION.
- ❷ The individual regions are counted using the SUM statement.
- ❸ A left-justified character representation of the counter is made.
- ❹ The macro variable ®4 is created when II='4'. The current value of REGION is assigned to the macro variable using CALL SYMPUT.
- ❺ The total number of regions is saved in &TOTAL.
- ❻ A %DO loop is used to step through the &TOTAL regions one at a time.
- ❼ The individual regions are identified by using the indirect reference &®&I.

Note that the SYMPUT function that assigns the value to &TOTAL is executed for each region. However, only the last assignment for the last observation really matters. This slight inefficiency could have been avoided by moving this statement outside of the DO block. Also, the DATA _NULL_ reads the entire data set (CLINICS) when you are really only interested in the unique

values of REGION. Both of these inefficiencies are corrected in the following version of %PLOTIT:

```
%MACRO PLOTIT;
  PROC SORT DATA=CLINICS
          OUT=REGCLN(KEEP=REGION)
          NODUPKEY;
    BY REGION;
  RUN;
  DATA _NULL_;
    SET REGCLN END=EOF;
    * Count the regions;
    I+1;
    * Create char var with count (II);
    II=LEFT(PUT(I,2.));
    CALL SYMPUT('REG'||II,REGION);
    IF EOF THEN CALL SYMPUT('TOTAL',II);
  RUN;
  %DO I=1 %TO &TOTAL;
    PROC GPLOT DATA=CLINICS;
      PLOT HT * WT;
      WHERE REGION="&&REG&I";
      TITLE1 "Height/Weight for REGION &&REG&I";
    RUN;
  %END;
%MEND PLOTIT;
```

If the programmer uses the %DO loop to encompass more than one PROC step, the resulting output would appear to be collated within BY groups. The programmer is then able to group the output from multiple procedures for each level of the BY groups without manually re-sorting. This technique effectively creates a pseudo, across-procedure %BY statement. The %DO loop shown here demonstrates this technique by generating all of the output for the first region before generating any output for the next region:

```
%DO I=1 %TO &TOTAL;
  PROC GPLOT DATA=CLINICS (WHERE= (REGION="&&REG&I") );
    ...code not shown...
  PROC PRINT DATA=CLINICS (WHERE= (REGION="&&REG&I") );
    ...code not shown...
  PROC REPORT DATA=CLINICS (WHERE= (REGION="&&REG&I") );
    ...code not shown...
  PROC TABULATE DATA=CLINICS (WHERE= (REGION="&&REG&I") );
    ...code not shown...
%END;
```

The following example creates a separate flat ASCII file for each combination of the variables STATION and DEPTH in the data set A1. STATION takes on the values of TS3, TS6, and so on, while DEPTH is measured in whole meters (0, 1, 2, and so on). The objective for the macro DOIT is to break up the data set into a series of flat files each of which will be named according to the data that it contains. By writing the macro using these techniques the programmer does not need to know what combinations of STATION and DEPTH will exist in the data when the macro is eventually executed.

```

%macro doit;

* Create the macro variables.
* One set for each STATION X DEPTH;
data _null_;
set al;
by station depth;
length ii $1 dd $2 fn $14;
if first.depth then do;
  i+1; ❶
  ii = left(put(i,2.));
  * Create a character value of the numeric depth;
  dd = trim(left(put(depth,3.)));
  * Construct the filename;
  fn = compress(station || dd || '.dat'); ❷
  call symput('i',ii); ❸
  call symput('d' || ii,dd); ❹
  call symput('sta' || ii,station); ❺
  call symput('fn' || ii,fn); ❻
end;
run;

* There will be &i files;
%do j=1 %to &i;
  filename toascii "&fn&j"; ❼
  * print the ascii files;
  data _null_;
  set al;
  where station="&sta&j" and depth="&d&j"; ❽
  cnt + 1;
  file toascii;
  if cnt=1 then put '***** ' "&fn&j";
  put @1 date mddy8. @10 aveday; ❾
run;
%end;
%mend doit;

%doit

```

- ❶ Count the unique combinations of STATION and DEPTH.
- ❷ Build the string variable that contains the name of the new ASCII file.
- ❸ &I contains the number of unique combinations.
- ❹ &D&i contains the *i*th depth, which was converted from a numeric value.
- ❺ &STA&i contains the *i*th station.
- ❻ &FN&i contains the *i*th ASCII file name.
- ❼ Build the FILEREF which depends on &J.
- ❽ In order to subset the data, use the WHERE statement to select observations based on specified values of STATION and DEPTH.
- ❾ Write the selected data to the ASCII file.

The DATA step that writes the ASCII files is executed &I times. This means that the data set A1 must also be read &I times. A potentially substantial improvement in efficiency could be achieved by recoding the macro to read A1 only once in the final step. Questions 2 and 3 in this chapter's exercises address this issue.

The following table illustrates how the macro variables will be resolved for the first four combinations of STATION and DEPTH in the data set A1:

STATION	DEPTH	&J	&&STA&J	&&D&J	&&FN&J
TS3	0	1	&STA1 TS3	&D1 0	&FN1 TS30.DAT
TS3	1	2	&STA2 TS3	&D2 1	&FN2 TS31.DAT
TS3	2	3	&STA3 TS3	&D3 2	&FN3 TS32.DAT
TS6	0	4	&STA4 TS6	&D4 0	&FN4 TS60.DAT

The double-ampersand macro variables form the macro equivalent of an array or vector of values. The macro language does not have an ARRAY statement, but the &&VAR&I combination works in much the same way. In the previous table, the &&FN&I combination can be viewed as a vector, as is shown here:

&J (subscript)	&&FN&J (array element)	element value
1	&FN1	TS30.DAT
2	&FN2	TS31.DAT
3	&FN3	TS32.DAT
4	&FN4	TS60.DAT

This table shows that for &J = 3, the macro variable &&FN&J resolves to &FN3, which further resolves to TS32.DAT.

Each of the examples in this section includes a DATA step much like the one shown below. These steps each include a numeric variable that counts some portion of the process ❶, a left-justified character representation of that counting variable ❷, and a call to the SYMPUT routine that uses that counting variable ❸. My programming habit has been to use the variable names "I" and "II" as these index variables, but there is no particular reason for doing so.

```
DATA _NULL_;
  SET REGCLN END=EOF;
  * Count the regions;
  I+1; ❶
  * Create char var with count {II};
  II=LEFT(PUT(I,2.)); ❷
  CALL SYMPUT('REG'||II,REGION); ❸
  IF EOF THEN CALL SYMPUT('TOTAL',II);
RUN;
```

Many macro programmers, however, choose to avoid creating the second DATA step variable and instead use the functions directly ❹ (often including the use of the TRIM function), as shown below.

```

DATA _NULL_;
  SET REGCLN END=EOF;
  * Count the regions;
  I+1;
  CALL SYMPUT('REG' || TRIM(LEFT(PUT(I,4.))) ④, REGION);
  IF EOF THEN CALL SYMPUT('TOTAL', TRIM(LEFT(PUT(I,4.))) ④);
RUN;

```

- ④ Remember the format in the PUT function restricts the maximum size of the counter. Using a overly large format has no “bad” ramifications and could provide a cushion if the counter (I – in this case) becomes larger than expected.

SEE ALSO

Carpenter and Callahan (1988) contains two related macros that you can use to split data into operational subsets for further processing. Similar approaches are taken in more sophisticated examples by Blair (1998), and Talbott and Westerlund (1998).

Wright (2001) uses a PROC FREQ to generate the control file.

Pass (1998, 2000) also controls the program flow for BY groups by breaking up the data into a series of steps. His approach, however, does not require the use of the &&VAR&I construct.

6.2.2 Assigning macro variable names as well as values

The control data set can be used to assign the name of the macro variable as well as its value. In this example, the DATA step variable MVARNAME contains the name of the macro variable, and VALUE has its intended value. Each observation in CONTROL (there are two) will be used to define one macro variable. Notice that only one SYMPUT is used, and there is no hard coding of macro variable names in the DATA _NULL_ step.

```

DATA CONTROL;
  MVARNAME='DSN';
  VALUE='CLINICS';
  OUTPUT;
  MVARNAME='OBS';
  VALUE='5';
  OUTPUT;
RUN;
%MACRO LOOK;
  DATA _NULL_;
    SET CONTROL;
    CALL SYMPUT(MVARNAME, VALUE);
  RUN;
  PROC CONTENTS DATA=&DSN;
  RUN;
  PROC PRINT DATA=&DSN (OBS=&OBS);
  RUN;
%MEND LOOK;

```

SAS generates the following code when %LOOK is executed:

```

PROC CONTENTS DATA=CLINICS;
RUN;
PROC PRINT DATA=CLINICS (OBS=5);
RUN;

```

Notice that neither argument in the SYMPUT statement is a quoted string. Both are data set variables, and their values provide the appropriate information (macro variable name and macro variable value).

Of course the author of this version of %LOOK had to know the name of the macro variables that were to be created (&DSN and &OBS). The following revised %LOOK macro, which uses the same control data set, overcomes this limitation by passing in the name of the macro variables as macro parameters.

```
DATA _NULL_;
  SET CONTROL;
  CALL SYMPUT (MVARNAME, VALUE) ;
RUN;
%MACRO LOOK (DAT, CNT) ;
  PROC CONTENTS DATA=%%&DAT;
  RUN;
  PROC PRINT DATA=%%&DAT (OBS=%%&CNT) ;
  RUN;
%MEND LOOK;
%LOOK (DSN, OBS)
```

When %LOOK is executed the macro reference %%&DAT will be resolved as follows:

	%&&DAT	
is seen as	%&	&DAT
	↓	↓
becomes	%	DSN
and	%DSN	
resolves to	CLINICS	

Remember that when you see a macro variable with three ampersands, it is a macro variable that holds the name of a macro variable that resolves to the value of interest. This is true whether it is in the form of %%&VAR or %%VAR&I (Section 6.3 discusses these two macro variable forms in more detail).

An indirect table lookup takes place when two steps are required to retrieve a given piece of information. In these cases you are often given a key or index that will enable you to look up information that you can in turn use to retrieve the information of interest. This is the case in the next example, which uses the data to both name the macro variable and provide its value.

We will be given a code for a particular family (every family has a unique code). The task is to find all families coming to a reunion (listed in the REUNION data set) that have the same MOTHER and FATHER as in the family of interest. The data set RELATION contains information on the members of each family that is designated with a family code (FAMILY). The two variables RELATION and NAME provide information on the relationships and names of the members. This data is then used to construct a filter that uses macro variables to subset a second data set (REUNION) that contains only names but not the family code.


```

data relation;
length relation name $8;
family = 'MC24';
relation='mom'; name='Sally'; f_origin='AL06'; output;
relation='dad'; name='Fred'; f_origin='MC22'; output;
relation='son'; name='Clint'; f_origin='MC24'; output;

family = 'MC25';
relation='mom'; name='Jane'; f_origin='BA06'; output;
relation='dad'; name='Clint'; f_origin='MC24'; output;
relation='daughter'; name='Laura'; f_origin='MC25'; output;
run;

data reunion;
length mother father son daughter $8;
father='Donald'; mother='Susan'; son='Kyle '; daughter=' ';
output;
father='Fred '; mother='Sally'; son='Clint'; daughter=' ';
output;
father='Clint '; mother='Jane '; son=' '; daughter='Laura';
output;
father='Fred '; mother='Sally'; son='John '; daughter='Rose ';
output;
run;

%macro listfam(famcode);
data _null_;
set relation(where=(family="&famcode"));
* Create one macro var for each observation.
* Use RELATION to name the macro var and NAME for its value;
call symput(relation,name);
run;

proc print data=reunion(where=(mother="&mom" & father="&dad"));
title "Same Mother and Father as Family &famcode";
run;

%mend listfam;
%listfam(MC24)

```

The PROC PRINT will print all observations in REUNION where the variable MOTHER = 'Sally' and FATHER = 'Fred'.

This indirect technique is especially useful when the macro programmer does not know what the true lookup criteria will be when the program is written. In this example, only the family code is known and the names of the mother and father are indirectly referenced through the control file.

6.3 Macro Variable Forms Used in Dynamic Programs

The examples in Section 6.2 use a data set to control the processing of the macro. This type of control frees the macro programmer from hard coding data-specific information into the macro. In the %PLOTIT example (Section 6.2.1) the macro will work for any number of regions. With minor changes to the macro (see Chapter 6 questions 2 and 3) even further data dependencies can be removed from the macro itself.

Programs and macros that write themselves or adjust to the data are dynamic in nature, and it is a huge programming advantage to be able to write macros general enough to be data independent. Programming successfully at this level is much more difficult. The %PLOTIT macro mentioned above assumes that the name of the BY variable will be REGION and that it is a character variable. The macro would have been much more dynamic if it could have accepted a list of one or more variables to use and to be able to determine if these variables were character or numeric. The most general macro will have the fewest data dependencies.

It is not easy to begin writing this type of macro. Most macro programmers write a great many macros that are, to varying degrees, dynamic before this type of programming becomes second nature. This section illustrates those characteristics or elements that are common to many macros that dynamically build code. Learn to watch for and to program these elements and the entire process becomes easier.

MORE INFORMATION

Chapter 9, "Writing Dynamic Code," discusses examples of dynamic macros in more detail.

6.3.1 Elements of a dynamic program

While not always present, there are certain elements that tend to be in macros that dynamically build code. These elements include the following:

- Macro variable values are based on an information source like a SAS data set.
- The SYMPUT routine or the SQL INTO: is used to build the macro variables.
- Macro arrays of the form &&VAR&I are common.
- The number of array elements is saved in a macro variable.
- %DO loops are used to step through the macro arrays.

As you learn to write this type of macro, watch for these elements and notice how they fit together. The pattern formed by these elements is repeated over and over again in the examples in this book. Browse through the examples noted in the table in Section 6.3.3, notice the pattern, and look for the elements described above.

Building macro variables based on an information source

The DATA step is used to bring in the control information and to assign values to the macro variables. Usually you will need to count the individual values (rows on the incoming data set) so that you can use the count as a subscript when building a macro array. The DATA step will look something like the following:

```
data _null_; ❶
  set <control file>; ❷

  i+1; ❸
  chari = left(put(i,4.)); ❹

  call symput('root'||chari,<var_name>); ❺
  call symput('rootcnt',chari); ❻
run;
```

- ❶ Usually a new data set is not needed and a DATA _NULL_ step is used.
- ❷ The information source might be a SAS data set, SAS View, some other table, or even a flat file.
- ❸ A running count of the incoming row number from the information source ❷ is kept.
- ❹ The row number is converted to a left-justified character value.
- ❺ A series of macro variables in the form of &ROOT1, &ROOT2, &ROOT3, and so on is created. Each is based on the value contained in a DATA step variable (VAR_NAME). The index (subscript) for each macro variable is contained in the variable CHAR1 ❹.
- ❻ The total number of macro variables in the series is retained in a macro variable. This is the latest value of CHAR1 ❹.

Using the macro variables dynamically

Since you will usually have a list of macro variables named something like &ROOT1, &ROOT2, &ROOT3, and so on, and since you will need to step through this list one element at a time, a %DO loop is usually used. The index of the %DO becomes the subscript to the macro array. These loops may be used to iteratively execute a block of statements (see Section 6.2.1) or to build a specific statement from the inside (see Sections 5.3.2, 9.1.3, and 9.4.1).

If the macro variables contain data set names, a SET statement might be written as

```
set
  %do i = 1 %to &rootcnt; ❶
    &&root&i ❷
  %end;
; ❸
```

- ❶ &ROOTCNT is the total number of data sets to be read.
- ❷ This indirect macro variable reference is used to point to the individual data set names. When &I is 3, &&ROOT&I becomes &ROOT3, which refers to the third data set.
- ❸ The SET statement is completed with a semicolon. The semicolons associated with the %DO and %END statements are “used up” by the macro facility, while this one is left behind and becomes part of the SET statement.

The power of this technique is that the programmer who wrote the above SET statement had no idea of what the names of the data sets were to be or even how many data sets would be listed. This SET statement will change dynamically as the list of data sets changes.

6.3.2 Indirect macro variable references

Common to virtually every dynamic macro is the use of indirect macro variable references. These are almost always one of two forms: `&&VAR&I` and `&&&VAR`. As has been mentioned elsewhere in this book (see Sections 2.5.3, 9.3.1, 13.1.1, and 13.2), these are macro variable references that will take two passes of the macro facility to resolve. The first resolution pass results in a macro reference that itself must be resolved in a second pass. Both of these forms have three ampersands, at least two of which must be adjacent.

While SAS does not formally define arrays in the macro language, the `&&VAR&I` macro variable form functions in a way that mimics a macro array, where the `&I` serves as the array subscript or index. This form is useful when a series of values need to be stored individually. The name of the array, "VAR" in this case, can of course be any appropriate name.

When an indirect reference is needed but there is only one element to be stored, an array is not necessary and the macro variable takes the form of `&&&VAR`. Unlike in the macro array, where the name of the array is known and the indirect reference is established with an index, in this form the name of the macro variable is itself unknown and is used to establish the indirect reference.

Although it is of practical value less often, the `&&&VAR` form can be combined with the `&&VAR&I` form of macro array. The result would be a macro array of the form `&&&VAR&I`. This would enable the programmer to create a macro array with an unspecified array name (`&VAR`). One use of this form is to establish what is effectively a doubly subscripted macro variable array. This type of array has a series of macro variable arrays (vectors), and a specific value is accessed with the `&I` that points to a row and to the `&VAR` that indicates the name (not the number) of the column of the array.

MORE INFORMATION

Section 13.2 specifically addresses issues relating to doubly subscripted macro arrays. Several examples of the use of the `&&&VAR` macro variable form are shown in Section 13.1.

SEE ALSO

Indirect array references in the form of `&&&VAR&J` are used by Kunselman (2001) in a SAS/IntrNet example.

6.3.3 Sources of control information

The information that is used to control or guide the dynamic macro can be found in a very wide variety of forms and places. Remember that this information will be used to build one or more macro variables, so first visualize the information that is needed in the macro variable(s) and then visualize how you can get this information into a SAS data set or a similar source. Once in a SAS data set it is simply a matter of using the SYMPUT routine to build the needed macro variables.

A number of examples throughout this book show various ways of creating or using control information from which you can build the macro variables. The following table suggests some sources for control information as well as locations within this book with examples:

Type of information source	Example sections
Control data sets built strictly to control macro operations	6.2.1, 6.2.2, 9.4
The analysis data itself	6.2.1, 6.2.2, Q6.2, Q6.3, 10.2.2, 11.4.3
OUT= option with PROC CONTENTS	10.1.5, 11.1.1, 11.1.2
SASHELP and Dictionary views	6.5.4, 9.5, 10.1.1, 10.1.5, 11.4.1
Results of operating system operations	10.1.2, 10.1.5
SAS system option settings	10.2.1

6.4 Moving Text from Macro to DATA Step Variables

You can use several methods to convert information that is stored in macro variables to values that are stored in DATA step variables. Since macro references are resolved before the DATA step is executed, moving a value that has already been established on a macro symbol table to a data set can successfully involve the use of macro references, and, consequently, this process is more direct than when writing a DATA step value to a macro variable. As was shown in Section 6.1, when writing DATA step values to the symbol table, you cannot use macro references such as &VAR or %LET statements.

Because you will assign a value to a variable on the Program Data Vector, the easiest and most common methods are through the use of either the assignment or RETAIN statements. In other situations, you may need to use either the SYMGET or RESOLVE functions.

6.4.1 Assignment and RETAIN statements

You can use the DATA step assignment statement to create a DATA step variable or to assign a value to an existing variable. In the following example, the macro variable &DSN is created and assigned the value `clinics`. Later in the program, &DSN is used to define the character variable DSET in the data set SUBWT.

```
%let dsn = clinics;
...code not shown...
data subwt;
  set clinics (keep=fname wt);
  dset = "&dsn";
  where wt>80;
run;
```

Because the macro variable is resolved before the DATA step is compiled, DSET will be a constant (with a value of `clinics`), and the RETAIN statement could have been used to make this step somewhat more efficient. The previous DATA step becomes

```

data subwt;
  set clinics (keep=fname wt);
  retain dset "&dsn";
  where wt>80;
run;

```

The assignment and RETAIN statements are the simplest methods and usually will be your primary methods for creating and assigning DATA step variables using symbol table values. The next most likely method, the SYMGET function, is covered in the next section.

MORE INFORMATION

A date constant is established using an assignment statement and the automatic macro variables &SYSDATE and &SYSTIME in Section 2.6.1.

6.4.2 Using the SYMGET function

The SYMGET function is not a macro function. Like the SYMPUT routine, it is used in the DATA step and converts the current value of a macro variable to a character string so that it can be placed into a DATA step variable. This function is the functional opposite of the SYMPUT routine.

The argument is either a macro variable name, a DATA step variable that takes on a value that is a macro variable name, or a character expression that constructs a macro variable name.

Syntax

`variable = SYMGET(argument);`

When using the SYMGET function remember that

- it is used within a DATA step
- the default length for a returned character string is 200
- the *argument* is enclosed in quotes to directly specify the name of a macro variable
- when the *argument* is written without quotes, it is assumed to be either a DATA step variable whose value is a macro variable name or an expression that constructs a macro variable name.

Using an assignment statement as opposed to the SYMGET function will have similar but not necessarily exactly the same results. In the following example, the variables DATASET and DSET will have the same value but not the same LENGTH. DATASET ❶ will have a length of 200, while DSET ❷ will have a length of 7.

```
%let dsn = clinics;
```

```

data subwt;
  set &dsn(keep=fname wt);
  where wt>80;
  dataset=symget('dsn'); ❶
  dset = "&dsn"; ❷
run;

```

*Symput Data step value
→ macro variable*

*Symget macro variable value
→ Data step variable*

*dsn is a macro variable.
Within quotes dsn is macro variable value → Data step variable value*

Assignment statement

In most cases you will not need to use the SYMGET function because it is simply easier to assign a value using the assignment statement, as was done for DSET in the previous example.

SYMGET is most often used when the macro variable is to be resolved at the time of DATA step execution. If you are going to compile the DATA step or if you are using SAS Component Language (SCL), then SYMGET becomes much more helpful, if not essential.

In the following example, the data set WEIGHTS has a variable CODE that identifies a correction factor that is to be applied to the variable WT. The variable CODE can take on the values of 1, 2, or 3, and the three correction factors have already been loaded into three corresponding macro variables: &CORR1, &CORR2, and &CORR3. One way to apply the correction factors would be to use a series of IF statements:

```
data corrwt;
  set weights;
  if code=1 then wt = wt*&corr1;
  else if code=2 then wt = wt*&corr2;
  else if code=3 then wt = wt*&corr3;
run;
```

You can improve performance of the DATA step by eliminating the IF-THEN/ELSE processing and, instead, do a table look-up. The DATA step becomes

```
data corrwt;
  set weights;
  wt = wt*symget('corr'||left(put(code,2.)));
run;
```

an expression that constructs a macro variable
automatically transformed to a numeric value!

The SYMGET function always returns a character string. In the previous example, this character string must be converted to numeric (the conversion is done automatically by SAS) before the arithmetic operation of multiplication can be performed. This results in the following message in the LOG:

```
NOTE: Character values have been converted to numeric values at the
places given by:
(Line): (Column).
10:9
```

This warning can be eliminated by using the INPUT function to convert the SYMGET results to numeric prior to performing the multiplication.

```
wt = wt*input(symget('corr'||left(put(code,2.))),best12.);
```

MORE INFORMATION

A series of SYMGET examples appear as part of a comparison with the RESOLVE function in Section 6.4.3.

The SYMGET function is especially useful in SAS Component Language programs because those programs are compiled, and in compiled programs macro variables are not usually referred to using an ampersand. Chapter 8, "Using Macro References with SAS Component Language (SCL)," discusses various issues associated with macros and macro references in SCL.

SEE ALSO

John Piet (2000) shows a good example of an application that uses SYMGET and DATA values to determine the macro variable of interest.

6.4.3 Using the RESOLVE function

Your third choice for assigning values is the RESOLVE function. This function is very similar to the SYMGET function. Although the RESOLVE function tends to use more resources than SYMGET, it will accept a wider variety of arguments. RESOLVE will attempt to resolve some arguments that SYMGET will not attempt.

The argument to the RESOLVE function is passed to the macro processor for resolution and potentially even execution. When the argument is enclosed in single quotes, the string is not resolved when the DATA step is compiled, but instead is passed intact to the macro facility for resolution during DATA step execution. This means that the argument can contain direct macro references that use the & and % signs.

The arguments for RESOLVE are **not** interchangeable with those of SYMGET. The three types of arguments that RESOLVE will act on are

- DATA step variable names
- Text enclosed in single quotes
- Character expressions.

Unlike with the SYMGET function, if the argument is an unquoted DATA step variable, the variable's value from the Program Data Vector (PDV) will be returned. If the resolved value from the PDV contains macro references (% or &), then macro facility resolution will take place during DATA step execution.

Text enclosed in single quotes is passed directly to the macro processor for resolution during DATA step execution. If the text does not contain macro references the value is passed back. Macro references are resolved.

A character expression is also passed to the macro processor. This expression can contain macro variable references and macro calls that include & and % signs.

Generally, when using RESOLVE with macro variables, the argument will have an & or a % either directly or indirectly, since otherwise the SYMGET function could be used instead.

Since the argument in the RESOLVE function is passed to the macro facility for processing, any macro calls are executed before the assignment statement is completed.

The following example contains a DATA step with several calls to the SYMGET and RESOLVE functions. Notice how this excerpt from the LOG demonstrates some of the differences for these two functions:


```

1
2      %let dsn = clinics;
3      %let clinics = Bethesda;
4
5      data demo;
6          dname = 'clinics';
7          amp   = '&dsn';
8          * unquoted arguments;
9          a1 = symget(dname);
10         a2 = resolve(dname);
11         a3 = symget(dsn);
12         a4 = resolve(dsn);
13         a5 = symget(amp);
14         a6 = resolve(amp);
15         put / a1= a2= a3= a4= a5= a6=;
16
17         * Using a single quote;
18         b1 = symget('dname');
19         b2 = resolve('dname');
20         b3 = symget('dsn');
21         b4 = resolve('dsn');
22         b5 = symget('amp');
23         b6 = resolve('amp');
24         put / b1= b2= b3= b4= b5= b6=;
25
26         * Single quote with &;
27         c1 = symget('&dsn');
28         c2 = resolve('&dsn');
29         c3 = symget('&amp');
30         c4 = resolve('&amp');
31         put / c1= c2= c3= c4=;
32
33         * Double quote with &;
34         d1 = symget("&dsn");
35         d2 = resolve("&dsn");
36         d3 = symget("&amp");
WARNING: Apparent symbolic reference AMP not resolved.
37         d4 = resolve("&amp");
WARNING: Apparent symbolic reference AMP not resolved.
38         put / d1= d2= d3= d4=;
39
40         * Quoted triple ampersand;
41         e1 = resolve('&&&dsn');
42         put / e1= ;
43         run;

NOTE: Numeric values have been converted to character values at the
places given by:
      (Line):(Column).
      11:24    12:25
NOTE: Variable dsn is uninitialized.

```

NOTE: Invalid argument to function SYMGET at line 11 column 17.

NOTE: Invalid argument to function SYMGET at line 13 column 17.

a1=Bethesda a2=clinics a3= a4=. a5= a6=clinics

NOTE: Invalid argument to function SYMGET at line 18 column 17.

NOTE: Invalid argument to function SYMGET at line 22 column 17.

b1= b2=dname b3=clinics b4=dsn b5= b6=amp

NOTE: Invalid argument to function SYMGET at line 27 column 17.

NOTE: Invalid argument to function SYMGET at line 29 column 17.

WARNING: Apparent symbolic reference AMP not resolved.

c1= c2=clinics c3= c4=&

NOTE: Invalid argument to function SYMGET at line 36 column 17.

WARNING: Apparent symbolic reference AMP not resolved.

d1=Bethesda d2=clinics d3= d4=&

e1=Bethesda

dname=clinics amp=&dsn a1=Bethesda a2=clinics a3= dsn=. a4=. a5=
a6=clinics b1= b2=dname

b3=clinics b4=dsn b5= b6=amp c1= c2=clinics c3= c4=& d1=Bethesda
d2=clinics d3= d4=&

e1=Bethesda _ERROR_=1 _N_=1

Using a data step variable as the argument		
LOG line #	Function call	Resultant value
9	symget(dname)	A1='Bethesda'
	DNAME has the value 'clinics', which points to the macro variable of the same name.	
10	resolve(dname)	A2='clinics'
	The value of DNAME is retrieved from the PDV. Its value contains no macro references.	
11	symget(dsn)	A3=' '
	Variable DSN is uninitialized. DSN is set to missing.	
12	resolve(dsn)	A4='.'
	The current value of DSN on the PDV is used, and as we just saw, this is missing.	

Table (continued)

Using a data step variable as the argument		
LOG line #	Function call	Resultant value
13	symget(amp)	A5=' '
	amp has the value of '&DSN'. The & is not expected and causes an INVALID argument message.	
14	resolve(amp)	A6='clinics'
	The value of amp '&DSN' is retrieved from the PDV, and is passed to the macro facility where it is resolved.	

Quoting the function argument with a single quote		
LOG line #	Function call	Resultant value
18	symget('dname')	B1=' '
	The macro variable &DNAME does not exist.	
19	resolve('dname')	B2='dname'
	The string contains no macro references and is passed back to the assignment statement.	
20	symget('dsn')	B3='clinics'
	&DSN is a macro variable that has the value of 'clinics'.	
21	resolve('dsn')	B4='dsn'
	'dsn' has no macro or data set references; it is only a string, and is passed through.	
22	symget('amp')	B5=' '
	There is no macro variable named amp and this causes an INVALID argument message.	
23	resolve('amp')	B6='amp'
	'amp' has no macro or data set references; it is only a string, and is passed through.	

Including an & in the function argument and quoting it with a single quote		
LOG line #	Function call	Resultant value
27	symget('&dsn')	C1=' '
	&dsn is not a valid macro variable name (the & is not anticipated) and results in a run-time INVALID argument message.	
28	resolve('&dsn')	C2='clinics'
	'&dsn' is passed to the macro facility where it is correctly resolved.	

Table (continued)

Including an & in the function argument and quoting it with a single quote		
LOG line #	Function call	Resultant value
29	symget('&')	C3= ' '
	& is not a valid macro variable name (the & is not anticipated) and results in a run-time INVALID argument message.	
30	resolve('&')	C4='&'
	'&' is passed to the macro facility, but there is no macro variable with this name and the warning 'Apparent symbolic reference AMP is not resolved' is issued at run time. The unresolved value is returned.	

Including an & in the function argument and quoting it with a double quote			
LOG line #	Function call	Resolved call	Resultant value
34-37	In each statement the macro reference (&dsn and &) is resolved before the DATA step is compiled. It is the resolved call that is executed by the DATA step.		
34	symget("&dsn")	symget("clinics")	D1='Bethesda'
	The value of the macro variable clinics is retrieved.		
35	resolve("&dsn")	resolve("clinics")	D2='clinics'
	'Clinics' has no macro or data set references; it is only a string, and is passed through.		
36	symget("&")	symget("&")	D3= ' '
	& is not a valid macro variable name and results in a compile-time INVALID argument message. At execution this results in a missing value.		
37	resolve("&")	resolve("&")	D4='&'
	'&' is passed to the macro facility, but there is no macro variable with this name and the warning 'Apparent symbolic reference AMP is not resolved' is issued at run time. The unresolved value is returned.		

Resolving an &&& macro variable reference		
LOG line #	Function call	Resultant value
41	resolve('&&&dsn')	Bethesda
	&&&dsn is passed to the macro facility where it is resolved, first to &clinics and then to Bethesda.	

Since the RESOLVE function accesses the macro symbol table at DATA step execution, it can retrieve values that were just added with the SYMPUT routine. The following example concatenates a series of catalog names, one at a time, onto a single macro variable. Notice that the argument to the RESOLVE function is quoted with single quotes and contains an &. The single quotation marks are used to prevent the macro processor from resolving the argument before or while the DATA step is compiled.

```
%let sumplots=;
data _null_;
  set sashelp.vcatalog(where=(libname='WORK'
    and memname='GSEG'
    and objname =: "HR_KM"; ❶
  call symput('sumplots', ❷
    trim(resolve('&sumplots'))|| ❸
    ' '||trim(objname)); ❹
run;
```

- ❶ Select only the catalog members of interest, in this case graphs with a name starting with 'HR_KM'.
- ❷ Create a macro variable named SUMPLOTS.
- ❸ Use the existing value of SUMPLOTS.
- ❹ Append on the current name. Each observation that passes the WHERE clause filter ❶ will append another name to &SUMPLOTS.

MORE INFORMATION

A concatenated list of values can also be loaded into a macro variable through the use of a PROC SQL step; see Sections 2.7.1, 2.7.2, and 11.4.2.

SEE ALSO

SAS Technical Report P-222, *Changes and Enhancements to Base SAS Software, Release 6.07* provides some of the earliest written documentation for the RESOLVE function (pp. 313–315).

SAS Macro Language Reference, First Edition documents RESOLVE (pp. 210–212).

Whitlock (1998) walks the reader through the use of the RESOLVE function in detail.

The SYMGET and RESOLVE functions are discussed by Jaffee (1999).

6.5 Doing More with the SQL Step

The SQL step can be used to define and populate macro variables. The %LET statement cannot be used within an SQL step for the same reasons that it cannot be used within a DATA step (see the table in Section 1.3). Teaching the SQL step is outside the scope of this book; however, since it can be used to create and populate macro variables, the discussion of certain aspects of SQL are appropriate.

When defining a macro variable within an SQL step, the INTO operator is used with the colon (:) to designate the name of the macro variable(s) to be created.

MORE INFORMATION

Section 2.7 introduces the use of PROC SQL to define and populate macro variables. Additional examples can be found throughout the book (see Sections 11.2.1, 11.4, and 13.1).

SEE ALSO

An introduction and additional detail on the topic of using SQL to populate macro variables can be found in Zdeb (1999b), Shi and Roberge (2003), and Satchi (2000 and 2001).

6.5.1 Placing a single value into a single macro variable

The following SQL step counts the number of observations found in the SAS data set SASCLASS.CLINICS and places that number into the macro variable &NOBS:

```
%let dsn = sasclass.clinics;
proc sql noprint;
select count(*) ❶
    into :nobs ❷
    from &dsn; ❸
quit;
```

- ❶ The COUNT(*) function counts the number of observations.
- ❷ The number of observations is then written to the macro variable :NOBS using the INTO operator.
- ❸ Macro variables can be used just as they are used elsewhere within the base language.

MORE INFORMATION

Using SQL to count the number of observations in a data set is not a very efficient approach. In this example each observation is counted as it is read from &DSN. Other approaches such as those in Sections 11.2.2 and 11.5.1 will tend to be more efficient.

6.5.2 Creating more than one macro variable

It is possible to create more than one macro variable in the SELECT statement. In the following example two macro variables are created (&LASTNAMES and &DOBIRTHS), and each will contain a comma separated list of values.

```
proc sql noprint;
select lname, dob ❶
    into :lastnames separated by ',' ❷
        :dobirths separated by ',' ❸
    from sasclass.clinics (where=(lname='S')); ❹
%let numobs=&sqllobs; ❺
quit;
%put lastnames are &lastnames;
%put dobirths are &dobirths;
%put number of obs &numobs;
```

concatenate the values together

- ❶ The values of the variables LNAME and DOB will be written INTO macro variables.
- ❷ The list of values of LNAME will be written into &LASTNAMES with the values separated by a comma. Without the SEPARATED BY clause the individual values will replace previous values rather than be appended onto the list. The last comma is needed to separate the two macro variables (:LASTNAMES and :DOBIRTHS).
- ❸ The list of the dates of birth will be written to &DOBIRTHS.
- ❹ The WHERE clause is applied to the incoming data.
- ❺ The SQL step automatically loads the macro variable &SQLOBS. In this case it contains the number of rows in the incoming data set that meet the WHERE criteria (see Section 6.5.5).

The data set contains five observations with a last name, LNAME, which starts with an 'S'. The variable DOB has been assigned the format DATE7., and the format is automatically used when the values are written to the macro variable &DOBIRTHS. The LOG shows:

```

29  %put lastnames are &lastnames;
lastnames are Smith,Simpson,Simpson,Stubs,Saunders
30  %put dobirths are &dobirths;
dobirths are 18MAR52,18APR33,18APR33,11JUN47,01MAR49
31  %put number of obs &numobs;
number of obs 5

```

the value of the macro variable &lastnames
the value of the macro variable &dobirths

MORE INFORMATION

&SQLOBS is one of several automatic macro variables that are generated by each SQL step. Section 6.5.5 describes these macro variables in more detail.

6.5.3 Placing a list of values into a series of macro variables

Rather than placing a list of values into a specific macro variable as was done in Section 6.5.2, you can create a series of macro variables each with its own distinct value. In the following example PROC CONTENTS will create a table that will contain one row for each variable in the original data set (&DSN). PROC SQL is then used to write the variable names into a series of macro variables of the form &VARNAME1, &VARNAME2, and so on.

```

%macro varlist(dsn);
* Determine the list of variables in this
* base data set;
proc contents data= &dsn ❶
out= cont noprint;

run;

* Collect the variable names;
proc sql noprint;
select distinct name ❷
into :varname1-:varname999 ❸
from cont;
quit;

```

can store up to 999 unique variable names. but is way more than is needed! but SAS will create the number that is really required

```

%do i = 1 %to &sqlobs; ❹
    %put &i &&varname&i;
%end;
%mend varlist;

%varlist(sasclass.clinics)

```

- ❶ The OUT= data set (WORK.CONT) has one row per variable in &DSN. The names of these variables are stored in the variable NAME.
- ❷ Select each unique variable name from NAME (they should already be unique).
- ❸ The variable names are stored in a series of macro variables &VARNAME1, &VARNAME2, and so on. As it is currently coded, no more than 999 values can be stored. This is way more than is needed in this example, but SAS will only create the number that are actually required (not the full 999).

CAVEAT: When specifying the upper bound for the number of macro variables (999 in this case), you should be **very** sure that the number is large enough. If it is too small, values that do not fit will be lost and no error or warning will be issued.

- ❹ The number of distinct values of NAME is stored in &SQLOBS. This value can be used to cycle through the list of macro variables. The %DO loop is included here only as an illustration, and would not normally be used in a macro such as %VARLIST.

The data set SASCLASS.CLINICS has 20 variables, and the LOG shows the following:

```

1 ADMIT
2 CLINNAME
3 CLINNUM
4 DEATH
5 DIAG
6 DISCH
7 DOB
8 DT_DIAG
9 EDU
10 EXAM
11 FNAME
12 HT
13 LNAME
14 PROCED
15 RACE
16 REGION
17 SEX
18 SSN
19 SYMP
20 WT

```


Starting in SAS 9 you can specify the list of macro variables with leading 0s. The SELECT statement becomes

```
select distinct name
into :varname001~:varname999
```

Unlike the previous example, which created &VARNAME1, &VARNAME2, and so on, this revised statement would produce &VARNAME001, &VARNAME002, and so on. While specifying your list this way would make it easier to sort the macro variable names, it would make it more difficult to step through the variables using an incremented %DO loop.

The following PROC SQL step also builds a series of macro variables. In this case we would like to save the unique values of CLINNAME and to count the number of observations within each clinic. The names and the corresponding counts are each stored in macro variables.

```
proc sql noprint;
select distinct clinname, count(*) ❶
into :name1~:name999, ❷
:cnt1~:cnt999
from sasclass.clinics ❸
group by clinname; ❹
quit;
```

- ❶ First we will select the unique values of CLINNAME and use the COUNT function to count the observations.
- ❷ The names will be placed in the macro variables &NAME1... and the counts in &CNT1.... Although we have specified up to 999 macro variables of each type, only the number needed will be created. Since there are 27 clinics, &NAME28 will be unresolved. Again if the upper limit is too small, in this case if there were more than 999 clinics, information can be lost.
- ❸ The variable CLINNAME is drawn from SASCLASS.CLINICS.
- ❹ The GROUP BY clause causes the COUNT function to count within groups as opposed to across all groups.

The same set of macro variables could also be created by using a combination of PROC SUMMARY and the DATA step.

```
proc summary data=sasclass.clinics noprint nway;
class clinname;
var dob;
output out=cnt n=count;
run;
data _null_;
set cnt;
i+1;
ii=left(put(i,best12.));
call symput('name'||ii,clinname);
call symput('cnt'||ii,left(put(_freq_,best12.)));
call symput('namecnt',ii);
run;
```

SEE ALSO

Examples of SQL steps that create a series of macro variables can be found in Eddlestone (1997), Satchi (2000), and Casas (2002). First (2001b), as well as Rajecski and Kahle (2000), both create a list of macro variables with a single INTO:.

6.5.4 Using the SQL dictionary tables

In order for SQL to more easily access information about its environment, SAS has constructed a series of DICTIONARY tables. When accessed from within an SQL step they can be used to gather information about the SAS environment that might otherwise require additional steps.

The example in Section 6.5.3 uses PROC CONTENTS to build a list of variable names. This step could be eliminated by using the DICTIONARY.COLUMNS table.

```
%macro varlist2(lib,dsn);

* Collect the variable names;
proc sql noprint;
select distinct name into :varname1-:varname999
  from dictionary.columns ❶
   where (libname=upcase("&lib") & ❷
         memname=upcase("&DSN"));

quit;

%do i = 1 %to &sqlobs;
  %put &i &&varname&i;
%end;
%mend varlist2;

%varlist2(sasclass,clinics)
```

❶ DICTIONARY.COLUMNS is used as the source table for the list of variable attributes.

❷ The DATA table is subsetting for the library and data set of interest.

Because of the usefulness of these tables, additional tables might be made available in future releases of SAS. A partial list of currently available DICTIONARY tables includes

CATALOGS	Lists catalogs, members, entries, and entry types
COLUMNS	Data set attributes
EXTFILES	External file information
INDEXES	Lists existing indexes
MACROS	Symbol table attributes; macro variables and their scopes
MEMBERS	SAS controlled data sets, catalogs, views, and so on.
OPTIONS	Current settings of system options
TABLES	Attributes of SAS tables (data sets)
TITLES	Current title settings and values
VIEWS	Attributes of SAS views

To be able to use these tables you will need to know their column names. You list their attributes by using the SQL DESCRIBE statement. The following SQL step writes the attributes of the DICTONARY.COLUMNS table to the LOG:

```
proc sql ;
  describe table dictionary.columns;
quit ;
```

MORE INFORMATION

The SQL DICTONARY tables are similar to the SASHELP views discussed in Section 9.5.

SEE ALSO

Whitlock (2000b) provides a nice introduction to the SQL dictionary tables, and a brief overview of the DICTONARY tables is given by Bruchecker (1998b).

Tomb and Carter (2001) and LeBouton and Rice (2000) both use DICTONARY.TABLES to build a list of like-named data sets.

Lists of variables that are to be renamed are built by Ravi (2003) with the use of DICTONARY.TABLES and DICTONARY.COLUMNS.

Mao (2003) uses DICTONARY.TABLES to establish a list of data sets from a specified library as well as creating a list of macro variables using a SQL step.

DICTONARY.CATALOGS is used by Whitlock (2001b) to establish a list of formats.

DICTONARY.EXTFILES is used by Gunshenan (2003) to build a list of external files.

6.5.5 Automatic SQL generated macro variables

Whenever PROC SQL is executed a series of macro variables is generated and placed in the most local symbol table. The names of these macro variables all start with the letters SQL and include the following:

SQLQBS	Number of rows processed by the Select statement
SQLQPS	Number of iterations of the inner loop
SQLRC	Step Return Code; 0 when the step is successful
SQLXRC	Return code from a DBMS when a SQL pass-through is used
SQLXMSG	DBMS message generated by a SQL pass-through
SQLXQBS	Undocumented, but seems to be associated with the number of observations associated with the pass-through (it is created even if there is no pass-through).

The following SQL step is adapted from the one in Section 6.5.2:

```
proc sql noprint;
  select lname into :lastnames separated by ','
    from sasclass.clinics (where=(lname='S'));
  %let numobs=&sqlqbs;
quit;
%put _user_;
```

The LOG shows that there were five names that start with 'S'. The result of the %PUT is

```
GLOBAL SLOBS 5
GLOBAL NUMOBS 5
GLOBAL SLOOPS 38
GLOBAL SQLXOBS 0
GLOBAL SQLRC 0
GLOBAL LASTNAMES Smith,Simpson,Simpson,Stubs,Saunders
```

6.6 Execution of Macro Code Using CALL EXECUTE

"Like many powerful features in SAS, CALL EXECUTE is too good to ignore and yet may be dangerous for the unwary." Ian Whitlock (1997)

As was carefully pointed out in the diagram in Section 1.3, macro statements execute before the DATA step is compiled and certainly before it is executed. This is why you cannot use a %LET statement to assign a value of a DATA step variable to a macro variable. However there are times, while we are executing a DATA step, when we do need to be able to control macro statement execution. The CALL EXECUTE routine allows us to pass code (with or without macro references) directly to the macro facility during the execution of the DATA step.

Syntax

CALL EXECUTE(argument);

The argument to the CALL EXECUTE routine can contain constant text or macro instructions. The value of the argument is always passed to the macro facility. If it contains only text, without macro references, the code is placed in an input stack for execution immediately after the current step completes (step boundary). If the argument does include macro references, they are executed or resolved immediately (the execution of the DATA step is temporarily suspended).

Case 1

SEE ALSO

An introduction to the CALL EXECUTE routine can be found in Whitlock (1997). Dave Riba has a detailed explanation of CALL EXECUTE along with numerous examples in his article written for the Web version of the journal *Observations*. It can be viewed at <http://support.sas.com/documentation/periodicals/obs/obswww03/toc.html>.

Another explanation of the CALL EXECUTE can be found in Heaton-Wright (2003).

Jaffee (1999) has an example and explanation of the CALL EXECUTE routine, and Hamilton (2000) uses the CALL EXECUTE to display error conditions in the LOG.

A number of examples that use CALL EXECUTE are shown in Rook and Yeh (2001). These include the delayed execution of operating system commands.

Croonan and Theuwissen (2002) uses a series of CALL EXECUTEs to perform a merge, while Mounib and Satchi (2000) use CALL EXECUTE in an example on matched sampling.

Chow (1999); Jin, Jin, and Wang (2003); and Jiang (2003) each include a CALL EXECUTE example.

Xia and Birkenmaier (2001) and Tomb and Carter (2001) both build a CALL EXECUTE to call a macro.

Several approaches utilizing CALL EXECUTE to solve the problem of renaming variables are presented by Viergever (2003).

6.6.1 Executing non-macro code

During the execution of the DATA step, the value of the CALL EXECUTE argument is passed to the macro facility. When the argument contains no macro references, the code is placed in the program stack for execution at the end of the DATA step that called the CALL EXECUTE.

Case 2: SO-called non-macro code
Assume that you have a data set that contains the names of a series of data sets that you would like to print. In the following DATA step CALL EXECUTE will cause a PROC PRINT step to be executed for each data set listed in the variable DSN:

```
data _null_;
  set datamgt.dbdir;
  prc = 'proc print data='||dsn||';run;'; text value!
  call execute(prc);
run;
```

PR ← full statement
execute one

The LOG shows that after the DATA _NULL_ step executes, the following steps are generated:

will execute: 'proc print data = '||dsn||';run;'

NOTE	CALL EXECUTE generated line.
1	proc print data=DEMOG ;run;
2	proc print data=MEDHIS ;run;
3	proc print data=PHYSEXAM;run;
4	proc print data=VITALS ;run;

Rather than construct a DATA set variable (PRC), as was done above, the argument can be constructed directly. The CALL EXECUTE statement becomes

```
call execute('proc print data='||dsn||';run;');
```

6.6.2 Executing macro code

Macro references are executed as soon as they are received by the macro facility. Rather than building the PROC PRINT step within the CALL EXECUTE argument as was done above, the PROC step could be placed in a macro with the data step name passed as a macro parameter. The macro %PRTDSN contains the simple PROC PRINT shown above.

```
%macro prtdsn(dsn);
  proc print data=&dsn;
run;
```

```

%mend prtdsn;

data _null_;
  set datamgt.dbdir;
  call execute('%prtdsn('||dsn||')');
run;

```

Because the %PRTDSN is enclosed in single quotes the text is not seen as a macro reference when the DATA step is compiled. During execution of the DATA step, the argument of the CALL EXECUTE will take on a value such as, %prtdsn(DEMOG). When the value is passed to the macro facility, it will be correctly interpreted as a macro call and the macro will be immediately executed. However, since the macro generates non-macro text, this text will be added to the program stack for execution after the current DATA step terminates. The macro call %prtdsn(DEMOG) will add the following code to the program stack:

```

proc print data=DEMOG ;
run;

```

As in the previous example, when the argument contains macro references they are executed or resolved immediately. However, if no non-macro text is generated, nothing is added to the stack. This can be demonstrated with the following macro:

```

%macro putit(jj);
  %put in putit &jj;
%mend putit;

data a;
do i = 1 to 4;
  call execute('%putit('||i||')');
  put i= '*****';
end;
run;

```

The LOG shows that %PUTIT was executed four times (four %PUT statement executions), and each %PUT is executed immediately before the PUT statement was executed.

```

in putit 1
i=1 *****
in putit 2
i=2 *****
in putit 3
i=3 *****
in putit 4
i=4 *****

```

As you might anticipate when both macro and non-macro statements exist, the macro statements are executed immediately while the non-macro statements are stacked for execution later. In the following example the previous macro %PUTIT has been augmented with a small DATA step:

```

%macro putit(jj);
%put in putit &jj;
data a&jj;
  x=&jj;
  run;
%mend putit;

data a; ❷
  do i = 1 to 4;
    call execute('%putit('||i||')');
    put i= '*****';
  end;
run;

```

The LOG shows that

- ❶ The %PUT executes immediately.
- ❷ The DATA step (DATA A;) completes before the statements generated by the CALL EXECUTE are executed.
- ❸ The generated DATA steps (DATA A&JJ) are executed.

```

in putit 1 ❶
i=1 *****
in putit 2
i=2 *****
in putit 3
i=3 *****
in putit 4
i=4 *****
NOTE: The data set WORK.A has 1 observations and 1 variables.
NOTE: DATA statement used: ❷
      real time           0.00 seconds

NOTE: CALL EXECUTE generated line.
1  + data a1; x=1; run; ❸
NOTE: The data set WORK.A1 has 1 observations and 1 variables.
NOTE: DATA statement used:
      real time           0.05 seconds

2  + data a2; x=2; run;

...portions of the LOG not shown...

```

6.6.3 Timing issues

One of the things that makes CALL EXECUTE problematic for many of the folks that use it, is the timing of events—what takes place and when.

Remember that the CALL EXECUTE resides in a DATA step that will be compiled, and that any macro references will be resolved during this compilation. Consequently in the following CALL EXECUTE statement, the macro %DOIT will execute BEFORE the DATA step is compiled:

```
%macro doit;
  proc print data=a; run;
%mend doit;

data a;
  set datamgt.dbdir;
  call execute(%doit);
run;
```

After %DOIT executes the DATA step becomes

```
data a;
  set datamgt.dbdir;
  call execute( proc print data=a; run;);
run;
```

And, of course this does **not** work (the variables PROC, PRINT, and so on do not exist). As in the earlier examples, prevent macro resolution during **DATA step compilation** by using single quotes. Restating the statement with quotes will execute correctly.

```
call execute('proc print data=a;run;');
```

or

```
call execute('%doit');
```

You will also need to be careful when using CALL SYMPUT and the SYMGET function when accessing macro variables that are used in the same DATA step that uses CALL EXECUTE, and even in any code that is executed by CALL EXECUTE. As always the key will reside in the timing of the execution of the various statements.

The following CALL EXECUTE further illustrates this point when it calls the macro %TEST. A couple of things happen that might not be anticipated.


```

%macro test;
data _null_;
  put 'calling symput'; ❶
  call symput('x3',100); ❷
  run;

%put ready to compile NEW; ❸

data new;
  %put compiling NEW; ❹
  put 'executing NEW'; ❺
  y = &x3; ❻
  run;
proc print data=new;
  run;
%mend test;

data _null_;
  call execute('%test'); ❼
  run;

```

%TEST contains a CALL SYMPUT, macro statements, and a macro variable to be resolved. If we apply our non-CALL EXECUTE experience to this macro or if we execute %TEST without using CALL EXECUTE, we expect the following steps to occur:

- ❶ Write a note to the LOG to indicate that the CALL SYMPUT ❷ is being executed.
- ❷ Create the macro variable &X3.
- ❸ Use the %PUT statement to indicate that the next DATA step is to be compiled and executed.
- ❹ Use %PUT to indicate that the compilation of the DATA step has begun. The macro variable &X3 will be resolved.
- ❺ Indicate that the execution of the DATA NEW step has begun.
- ❻ Execute the assignment statement (y=100;).

When %TEST is executed by the CALL EXECUTE ❼, this is **not** the sequence of events and the results are not completely as we anticipated. The following LOG is generated. The numbered items in the LOG correspond to the numbered lines in the CALL EXECUTE.

```

...portion of LOG not shown...
19
20  data _null_;
21      call execute('%test');
22      run;

ready to compile NEW ③
compiling NEW ④
WARNING: Apparent symbolic reference X3 not resolved. ⑥
NOTE: DATA statement used:
      real time          0.51 seconds
      cpu time           0.03 seconds

NOTE: CALL EXECUTE generated line.
1  + data _null_;      put 'calling symput';      call symput('x3',100);
run;

NOTE: Numeric values have been converted to character values at the
places given by:
      (Line):(Column).
      1:60
calling symput ①
NOTE: DATA statement used:
      real time          0.05 seconds
      cpu time           0.01 seconds

1  +
data new;      put
'executing NEW';      y = &x3;      run;  proc print data=new; run;

executing NEW ⑤
NOTE: The data set WORK.NEW has 1 observations and 1 variables.
NOTE: DATA statement used:
      real time          0.20 seconds
      cpu time           0.03 seconds

NOTE: There were 1 observations read from the data set WORK.NEW.
NOTE: PROCEDURE PRINT used:
      real time          0.46 seconds
      cpu time           0.04 seconds

```

The non-macro code ('%TEST' is resolved) within a macro is invoked via CALL EXECUTE and placed on the input stack following the RUN in the DATA step with the CALL EXECUTE (⑦). But if there are any macro statements or macro variables, they will immediately execute or resolve or both (they are not placed in the stack) **even across step boundaries**. This is a departure from code that is otherwise executed (without using CALL EXECUTE). Normally we

would expect the first DATA step to be executed before the second is even compiled. We expect the macro variable &X3 to be created (❷) in the first step before it is needed in the second step (❸).

Instead the log shows that the two %PUT statements are executed immediately (❸, ❹), and that an attempt is made to resolve &X3 (❺) before it has been created (this gives us the unresolved macro reference warning).

Once the macro statements have been executed, the two DATA steps execute. The macro variable is created (❶, ❷) and then used (❸, ❹). This time, when &X3 is needed, it exists and the macro variable resolves as it should (we got the value of &X3 that we wanted, but not in the way that we anticipated).

MORE INFORMATION

Section 9.4.2 shows examples of the CALL EXECUTE in dynamic programming situations.

Timing issues associated with CALL EXECUTE are discussed in *SAS Macro Language: Reference, Version 8*, pp. 92–94.

6.7 Chapter Summary

Control of a macro can be achieved by reading a file (external or SAS data set) that contains the values intended for the macro variables and perhaps even the names of the macro variables as well.

Through the use of macro variables that are defined in a control file, it is possible to dynamically create SAS code that is based on the data itself.

The **SYMPUT** routine is used to assign a DATA step value to a macro variable. The routine has two arguments, either of which may be a variable, a constant (such as a character string), or a combination of the two. The first argument identifies the macro variable and the second identifies the value to be assigned.

Syntax: `CALL SYMPUT(argument1,argument2);`

Values are assigned during DATA step execution. However, macro variables are resolved during the compilation phase of the DATA step. Macro variables cannot be directly referenced in the same step in which they are created.

You can also create data set variables from macro variables. Usually this is done through both the assignment and the RETAIN statements, but at times you also may wish to use the SYMGET and RESOLVE functions.

PROC SQL can also be used to generate macro variables based on a data table. SQL offers a number of alternatives for generating macro variable lists.

The DATA step routine CALL EXECUTE can be used to generate macro calls based on the values in a data table. Although care must be taken with timing, this technique can sometimes be used to circumvent the need to generate macro variable arrays of the form &&VAR&I.

6.8 Chapter Exercises

1. In the example that uses macro %PLOTIT in Section 6.2.1, the variable REGION is a character variable. What changes to macro %PLOTIT would be required if REGION were numeric?
2. The macro %PLOTIT in Section 6.2.1 uses a WHERE statement to subset CLINICS when plotting. Rewrite the macro so that one data set is created for each region and then use PROC GPLOT on each individual data set (eliminate the use of the WHERE statement).

The most efficient program will read CLINICS just once or twice, creating ten data sets.

Hint: Write the program without macros first.

```
data reg1 reg2.....;
  set clinics;
  if region='1' then output reg1;
  else if region='2' then output reg2;
  else if.....
  ...
run;
```

3. Modify the code from Question 2 so that the data set names reflect the value of the variable REGION in that data set. Is PROC SORT still needed?
4. **EXTRA CREDIT**
The SAS data set CLASS.BIOMASS has both numeric and character variables. Create a macro that will convert all the numeric variables to character. If a variable is associated with a format (for example, DATE7.), use it in the conversion.

All character variables should be passed through to the new data set and the variable names in the new data set should be the same as in the old data set.

Create a macro that is general enough to operate on any SAS data set.

HINT 1: The following code will convert the variable AA from numeric to character and the bolded code is data set dependent:

```
data t2 (drop=__aa);
  length aa $8;
  set t1(rename=(aa=__aa));
  aa = left(put(__aa,best9.));
run;
```

HINT 2: The following code will create the data set CONT, which contains the names (NAME) and type (TYPE {numeric=1 and character=2}) of the variables in T1:

```
proc contents data=t1 out=cont noprint;
run;
```



Chapter 7

Using Macro Functions

- 7.1 Quoting Functions 145
 - 7.1.1 Using the %BQUOTE function 147
 - 7.1.2 %STR 149
 - 7.1.3 Considerations when quoting 150
 - 7.1.4 Basic types of quoting functions and why we care 155
 - 7.1.5 A bit about the %QUOTE and %NRQUOTE functions 156
 - 7.1.6 %UNQUOTE 156
 - 7.1.7 %SUPERQ 157
 - 7.1.8 Quoting function summary 158
 - 7.1.9 Marking and quoting mismatched symbols with the %STR and %QUOTE functions 159
- 7.2 Text Functions 160
 - 7.2.1 %INDEX 160
 - 7.2.2 %LENGTH 161
 - 7.2.3 %SCAN and %QSCAN 162
 - 7.2.4 %SUBSTR and %QSUBSTR 164
 - 7.2.5 %UPCASE and %QUPCASE 165
- 7.3 Evaluation Functions 166
 - 7.3.1 Explicit use of %EVAL 166
 - 7.3.2 Implicit use of %EVAL 167
 - 7.3.3 Using %SYSEVALF 168
- 7.4 Using DATA Step Functions and Routines 170
 - 7.4.1 Using %SYSCALL 171
 - 7.4.2 Using %SYSFUNC and %QSYSFUNC 172

7.5 Building Your Own Macro Functions	178
7.5.1 Introduction	178
7.5.2 Building the function	179
7.5.3 Using the function	182
7.6 Other Useful Macro Functions	183
7.6.1 One-liners	183
7.6.2 Functions for the DATA step	189
7.6.3 Macro functions with logic	191
7.7 Chapter Summary	197
7.8 Chapter Exercises	197

Macro functions are similar to DATA step functions except that they operate on text strings and macro variables rather than character strings and data set variables.

Macro functions either change or provide information about the text strings that are the arguments. The following are general categories of macro functions:

quoting functions	mask or remove meaning from special characters.
text functions	return information about text strings.
evaluation functions	perform arithmetic operations.
bridging functions	provide access to DATA step functions and routines.

The following sections cover these types of macro functions in more detail.

MORE INFORMATION

Section 7.6 discusses how to create your own macro functions. Several of the SAS autocall macros also behave like macro functions. See Chapter 12, “Building and Using Macro Libraries,” for details and examples.

SEE ALSO

Ian Whitlock has written a great deal about the macro language, and his paper on macro quoting (Whitlock, 2003a) and gives a very nice in-depth explanation of the quoting process.

A summary of some of the more commonly used macro functions can be found in Carpenter (2000a).

Once you become familiar with the macro language, it is fairly easy to write macros that behave like macro functions. See Whitaker (1989) and *SAS Macro Language: Reference, First Edition* (p. 159) for more information.

7.1 Quoting Functions

Why we need to quote

Quoting functions are used to mask the meaning of words or characters that would otherwise be misinterpreted during the compilation or execution of the macro statement. Sometimes it is obvious what needs to be quoted and other times it will be much less obvious. In the following example we want to create a macro variable &P that contains three SAS statements that make up a PROC PRINT step:

```
%LET DSN = CLINICS;
%LET P=PROC PRINT DATA=&DSN; VAR HT WT; RUN;;
```

Because the first semicolon (the semicolon following &DSN) terminates the %LET statement, the macro variable &P contains:

```
PROC PRINT DATA=CLINICS
```

Not only will &P not contain what the programmer expects, the VAR statement will almost certainly cause a syntax error (the VAR HT WT; RUN;; will be passed back to the base language for further processing). We need to let SAS know which semicolon to use to terminate the %LET. In other words, we need to mask the special nature of all of the semicolons except the last one. This masking is done with quoting functions.

What quoting does

The semicolon, of course, is not the only character that can cause problems. The table in Section 7.1.8 gives a more complete list of the characters and combinations of characters that might need quoting.

Generally speaking, these are characters or words that can have special meaning or receive special attention when the code is parsed. These are characters with “attitude.” In the example above, the semicolon is used to close or terminate SAS statements; the semicolon has attitude.

Quoting functions are used when we need to mask or delay the attitude of these characters. The quoting functions do this by physically placing invisible quotation marks (special non-printable characters) on either side of the character that is to be masked. You rarely see any indication that these characters even exist (Section 7.1.3 shows you how to see them if you need to do so), and you usually do not need to worry about them. Once they are placed around a character they remain unless removed (Section 7.1.6 on %UNQUOTE discusses how to remove the marks).

Simple examples of quoting functions

The following overview highlights some simple examples that use quoting functions.

For the following examples assume that the macro variable &DSN has been defined as

```
%LET DSN=CLINICS;
```

Without a quoting function

Example: %LET P=PROC PRINT DATA=&DSN; RUN;;
&P: PROC PRINT DATA=CLINICS
 Syntax errors are likely as &P contains the incorrect text.

%BQUOTE

Example: %LET P=%BQUOTE(PROC PRINT DATA=&DSN; RUN;;);
&P: PROC PRINT DATA=CLINICS; RUN;
 &P contains the correct text.

%NRSTR

Example: %LET P=%NRSTR(PROC PRINT DATA=&DSN; RUN;;);
&P: PROC PRINT DATA=&DSN; RUN;
 &DSN is not resolved even when the PROC PRINT statement is executed.

%UNQUOTE

Example: %LET P=%NRSTR(PROC PRINT DATA=&DSN; RUN;;);
&P: PROC PRINT DATA=&DSN; RUN;
Apply the %UNQUOTE function to &P:
 %UNQUOTE(&P)
Resolves to: PROC PRINT DATA=CLINICS; RUN;
 The macro variable &DSN is resolved and the resolved unquoted value of &P contains the correct text.

The history (or lineage) of quoting functions

In the beginning of quoting there was the %STR function. Since this function has been around the longest and perhaps since it has the shortest name, it is the most commonly used quoting function. It does not, however, provide all the quoting capabilities that we need. To enhance the capabilities of the %STR function, it was augmented with a special masking character (%) that enables it to quote some special characters (see Section 7.1.9) that it does not otherwise quote.

These additions to the %STR function were not enough, so the %QUOTE function was added to the macro language to handle those situations that required quoting during macro execution. However, %QUOTE lacks the capability of directly quoting the symbols that come in pairs, so the family of quoting functions was still not complete.

The shortcomings of the %QUOTE were made up for with the inclusion of the blind quoting function, %BQUOTE. This function is designed to quote a wide range of characters including those that can become mismatched. This function essentially eliminated the value of %QUOTE, which is now rarely used.

The NR versions of %STR, %QUOTE, and %BQUOTE were included to enable quoting in those situations when you want to control how the & and % signs in the quoted text are to be handled.

When the text contains macro references that you **do not** want resolved, the %SUPERQ function can be used. This is the strongest of the quoting functions; however, because of how it is applied (see Section 7.1.7), the advantage of %SUPERQ over %NRBQUOTE is minimal and rarely needed.

Quoting functions

The following list gives a brief overview of some of the functions and their behavior. More complete descriptions are given in the sections that follow.

<code>%BQUOTE</code>	removes meaning from unanticipated special characters (except <code>&</code> and <code>%</code>) that are resolved during execution.
<code>%NRBQUOTE</code>	removes meaning from unanticipated special characters (including <code>&</code> and <code>%</code>) during execution.
<code>%STR</code>	removes meaning from special characters (except <code>%</code> and <code>&</code>) during compilation.
<code>%NRSTR</code>	removes meaning from special characters (including <code>%</code> and <code>&</code>) at compilation and prevents resolution of macro references.
<code>%SUPERQ</code>	prevents any resolution of the value of a macro variable.
<code>%UNQUOTE</code>	undoes quoting.

It is not necessary to have a complete understanding of each of these functions, but rather a general understanding of what they do. You will use some of these functions much more often than others. It has been this author's experience that the `%BQUOTE` and `%NRSTR` functions are the most useful and that the others are more rarely used.

Also, with the availability of `%BQUOTE` and `%NRBQUOTE`, it is unlikely that you will find a use for the now dated `%QUOTE` and `%NRQUOTE` functions.

SEE ALSO

Bercov (1993) contains a easy-to-read summary of macro quoting, and First (2001a) gives a brief overview of the topic. Another overview with a number of examples is given by Burlew (1998, pp. 147–157).

A very detailed and understandable explanation of macro quoting functions is given by O'Connor (1999). O'Connor was the lead developer of the macro language at SAS for a number of years, and her insights are invaluable.

Whitlock (2003a) has written a great deal about the macro language and his paper on macro quoting gives a very nice in-depth explanation of the quoting process.

You can find other examples that contrast compilation versus execution time quoting functions in Chapter 10, "Macro Quoting," (pp. 229–230) in *SAS Guide to Macro Processing, Version 6, Second Edition* and in Chapter 7, "Macro Quoting," in both *SAS Macro Language: Reference, First Edition* (pp. 76–78) and *SAS Macro Language: Reference, Version 8* (pp. 71–89).

Carpenter (1999a) discusses various aspects of quoting functions and includes additional examples.

Chapter 7, "Macro Quoting," (pp. 75–93) in *SAS Macro Language: Reference, First Edition* provides detailed explanations of macro quoting.

7.1.1 Using the `%BQUOTE` Function

This probably ought to be the first quoting function that you reach for when a quoting need arises. Also called the blind quote, this function is generally used to remove meaning from unanticipated characters in resolved text during macro execution. It is especially useful if the text string was entered by a user through an application, and you may not have been able to trap all possible characters that might cause the macro code to fail.

Assume that the macro variable &METHOD has been assigned the following value:

```
THE DOCTOR'S NEW THERAPY
```

Because there is an unmatched single quote ('), it is very likely that when &METHOD is resolved SAS would produce an error. This is demonstrated in the following %LET statement. When it is executed the mismatched quote will mask the semicolon that closes the %LET ❶ and will almost certainly create syntax problems. Because the %LET statement will not be completed until sometime after the next single quote is found, the resulting syntax errors will point to mismatched quotes in statements other than the %LET.

```
%let method2 = &method; ❶
```

becomes

```
%let method2 = THE DOCTOR'S NEW THERAPY; ❶
```

To get around this problem, you could use %BQUOTE as follows:

```
%let method2=%BQUOTE(&method);
```

The special meaning will be removed from the single quote when it is resolved, and it will, therefore, not cause syntax problems. Special characters that are not produced by resolution will not be masked by %BQUOTE.

Sometimes the programmer is surprised by what may need to be quoted. It is important to consider the text as it stands at each step (compilation and execution). The following macro creates a DROP statement that can then be inserted into a DATA step. You must use a quoting function in case the user passes a string into the macro that would result in an invalid comparison. This might include multiple items, an item that could be confused with a logical operator (AND, OR, or NOT), or even a variable list that is seen as an arithmetic operation (X1-X4).

```
%MACRO DROP(DROPLIST);
  %IF %BQUOTE(&DROPLIST) NE %THEN
    %BQUOTE(DROP &DROPLIST);
%MEND DROP;
```

You could write the call to %DROP as

```
%DROP(X1-X3 LASTNAME FRSTNAME SSN)
```

This call would generate the following DROP statement:

```
DROP X1-X3 LASTNAME FRSTNAME SSN;
```

Without the %BQUOTE in macro %DROP, &DROPLIST would be resolved before the %IF expression was evaluated. If, as in these examples, the resulting string contains any of the special symbols mentioned above, the macro %IF expression could become invalid. For this reason the following version of %DROP will not work correctly. The dash (-) in the list has special meaning in the expression, and the resulting %IF expression will not be syntactically correct.

```
%MACRO DROP(DROPLIST);
  %IF &DROPLIST NE %THEN
    %BQUOTE(DROP &DROPLIST);
%MEND DROP;

%DROP(X1-X12)
```

The resulting %IF becomes

```
%IF X1-X12 NE %THEN ....
```

The dash is interpreted as a minus sign, and since there is an implied arithmetic operation, %EVAL is called. Of course since X1 and X12 are not integers, %EVAL cannot perform the subtraction, and an error stating, in part, that a “character operand was found in the %EVAL function” is written to the LOG. In the first version of %DROP the %BQUOTE is masking this dash and it is not misinterpreted as a subtraction symbol.

Summary

- The %BQUOTE function takes its name from blind quote.
- This function removes meaning from resolved special characters during macro execution, including mnemonic operators such as AND or OR.
- Mismatched quotes and parentheses can be masked directly with %BQUOTE, and do not require special masking characters, as does %STR, in order for them to be quoted (see Section 7.1.9).

MORE INFORMATION

The %BQUOTE function is used to mask single quotes in the macro %DB2DATE (Section 7.6.1) and in the fourth example in Section 10.1.5.

7.1.2 %STR

This is likely the most commonly used quoting function. Popularity, however, does not make it the best quoting function. Probably its popularity is due to force of habit, and because it has been around the longest. It removes meaning from most special characters (except % and &) at compilation and is most commonly used to remove meaning from commas and semicolons, and to preserve blanks and null spaces. When the percent sign (%) is used as a special masking symbol (see Section 7.1.9), the %STR function can also handle characters that normally come in pairs, such as parentheses and mismatched quotes. *see p157*

In Section 5.2.1 the following %IF statement is used to compare the value of the macro variable &B1 to a null value: *handle mismatched quotes*

```
%IF &B1 ^= %THEN %SORTIT(&d,&b1,&b2,&b3);
```

While this coding is both acceptable and common, it makes some programmers uncomfortable to not have anything on the right side of the equal sign. The %STR function can be used in this situation as a place holder.

```
%IF &B1 ^= %STR() %THEN %SORTIT(&d,&b1,&b2,&b3);
```

In this case, since the comparison is to be made against a null value, there is no space between the parentheses in the %STR function. Of course the %BQUOTE function could also have been used here.

Summary

- STR stands for string.
- The %STR function removes meaning from special characters during compilation.
- This function does not remove meaning from the percent sign (%) or the ampersand (&).

MORE INFORMATION

The %STR function is used in the example that appears in Section 7.2.3 to preserve both a blank and a null value.

7.1.3 Considerations When Quoting

Programming to avoid quoting

The following macro, %EXIST, is based on a macro of the same name in *SAS Guide to Macro Processing* (pp. 265–266). It creates the global macro variable &EXIST that takes on the values of YES or NO, depending on whether or not the stated data set (&DSN) exists. You can then query &EXIST to determine how subsequent steps should be executed.

- ❶ The DATA _NULL_ step is written so that an attempt will be made to open &DSN.
- ❷ The DATA step never tries to read any data because the logical expression in the IF statement will always be false.
- ❸ If &DSN does not exist, the DATA step terminates in an error that is saved in the automatic system macro variable &SYSERR.

A quoting function is used because multiple statements (with semicolons) are needed in the DATA _NULL_ step.

```
%macro exist(dsn);
%global exist;
%if &dsn ne %then %str(
  data _null_; ❶
    if 0 then set &dsn; ❷
  stop;
  run;
);
%if &syserr=0 %then %let exist=yes; ❸
%else %do;
  %let exist=no;
  %put PREVIOUS ERROR USED TO CHECK FOR PRESENCE ;
  %put OF DATASET & IS NOT A PROBLEM;
%end;
%mend exist;
```

The macro %EXIST is called in the following code, which checks for the existence of the data set VARLISTS in the library &PAC.DATA ❹. If it exists, it is subsetted with a WHERE ❺ and appended to another data set, VARSAVER ❻.

```

%macro dotask;
...code not shown...
* Check to see if the varlists data set already exists
* (&exist=yes);
%exist(&pac.data.varlists) ④

* Add this list of variables to the overall &PAC list;
* If the VARLISTS data set exists use it, otherwise
* create it from VARSAVER;
data &pac.data.varlists;
set
  %if &exist=yes %then
    &pac.data.varlists(where={tbl ne "&tbl"}) ⑤);
    varsaवर; ⑥
run;

...code not shown...
%mend dotask;

```

In the macro %EXIST, the %IF statement that utilizes the %STR function could have been rewritten using the %DO statement rather than the quoting function:

```

%if &dsn ne %then %do;
  data _null_;
  * No observations are actually read;
  if 0 then set &dsn;
  stop;
run;
%end;

```

By using the %DO, the quoting function is no longer needed and the coding is more intuitive.

MORE INFORMATION

More sophisticated versions of the %EXIST macro can be seen in Sections 7.5 and 11.3.

Quoting before or after passing values

There are times when the text that requires quoting is to be passed as a macro parameter. The macro variable &TTL below contains a single quote that can, depending on how the macro variable is used, cause a problem.

```

data _null_;
  call symput('ttl',"Tom's Truck");
run;

```

In the following PROC PRINT, the resolved value of &TTL will be fine in TITLE1, but will cause errors in TITLE2:

```
proc print data=sashelp.class;
  title1 "&t1l";
  title2 &t1l;
run;
```

We may want to protect our macros from unanticipated characters, in this case a single quote, by using quoting functions. In the following PROC PRINT both titles will work correctly and will give the same result because the single quote mark has been masked:

```
proc print data=sashelp.class;
  title1 "%bquote(&t1l)";
  title2 %bquote(&t1l);
run;
```

Of course using the %BQUOTE with every macro variable is not practical. When passing values into a macro, one might consider quoting the passed parameter. In the following macro the PROC PRINT step from above has been incorporated into the macro %PPRINTIT:

```
%macro pprintit(txt);
  proc print data=sashelp.class;
    title1 "&txt";
    title2 &txt;
  run;
%mend pprintit;
```

If the passed parameter (&TXT) contains an unmatched quote, the macro %PPRINTIT will have the same issues as the PROC PRINT shown above. The macro call shown below, however, will have its own problems:

```
%pprintit(&t1l)
```

Remember the macro variable &TTL is resolved before %PPRINTIT is called. The macro call becomes

```
%pprintit(Tom's Truck)
```

Again the single quote causes a problem, this time in the macro call (before it is even passed to the TITLE statements). Quoting the resolved value solves this problem as well those in the TITLE statements.

```
%pprintit(%bquote(&t1l))
```

Quoting the parameter in a macro call can also be useful when the resolved value contains commas. Consider the following value of &TTL and call to %PPRINTIT:

```
%let ttl = a, b, c;
%pprintit(&t1l)
```

We receive an ERROR message because the macro call with the resolved value becomes

```
%pprintit(a, b, c)
```

The resolved value will be interpreted by the macro facility as having too many positional parameters (three are seen, when only one is expected). Again %pprintit(%bquote(&t1l)) solves the problem.

Quoting the parameters prior to calling the macro is not always necessary; certainly the macro calls will be easier to construct if the quoting can be handled inside of the macro. The macro %COASTAL shown below is passed the postal abbreviation of a state on the west coast of the U.S. The three possible values are California (CA), Oregon (OR), and Washington (WA).

```
%macro coastal(state);
%if &state = CA %then %do;
...code not shown...
%end;
%mend coastal;
```

The macro works fine until we pass in a value for Oregon. The OR will cause an error because the resolved value of &STATE is seen as a boolean operator. We can mask the resolved value by using a quoting function. The quoting function could be applied in the macro call, as was done above, or it can be placed in the %IF statement where the problem is encountered:

```
%macro coastal(state);
%if %bquote(&state) = CA %then %do;
...code not shown...
%end;
%mend coastal;
```

QUIZ: What other state abbreviation(s), besides Oregon, will cause this type of problem?
 Answer: Nebraska (NE) and in SAS 9.1 Indiana (IN).

How to see the invisible marks

The marks added by the quoting functions are not only invisible; they also become a part of the text. Usually this is a good thing. In the following example we create a macro variable &OWNER:

```
%let owner = %bquote(Tom's Truck);
```

As was shown earlier, without marking the text the single quotation mark might be misinterpreted when it is used. However, once the text has been quoted, the invisible marks remain and whenever we use &owner, say, in another macro variable assignment, we can do so without a problem:

```
%let vehiclecolor = &owner is blue;
```

The invisible quotation marks remain to protect us and the resolved value.

Since the quotation marks are invisible, they are sometimes hard to see, and therefore it is sometimes hard to know if a macro variable contains any quoted elements. If you want to verify that the marks exist, or if you just need to determine if something has been quoted, you can often use the MLOGIC system option to make these symbols visible in the LOG.

With MLOGIC turned on, the macro call %COASTAL(OR) is shown in the LOG as follows:

```

164      %macro coastal(state);
165      %if %str(&state) = CA %then %do;
166          %put is CA;
167      %end;
168      %mend coastal;
169
170      %coastal(OR)
MLOGIC(COASTAL): Beginning execution.
MLOGIC(COASTAL): Parameter STATE has value OR
MLOGIC(COASTAL): %IF condition [%state] = CA is FALSE
MLOGIC(COASTAL): Ending execution.

```

The invisible masking marks are shown as boxes (□). The symbol might be different on your operating system, and on some operating systems the symbol might not show up at all—even when using MLOGIC!

Since MLOGIC only applies to values in an %IF expression, the macro %ISITQUOTED can be used to check an individual macro variable for quoting.

```

%macro isitquoted(var);
    %if &var ne %then %put &var;
%mend isitquoted;

```

Using the following macro variable definitions and a call to %ISITQUOTED

```

%let owner = %bquote(Tom's Truck);
%let vehiclecolor = &owner is blue;
%isitquoted(&vehiclecolor)

```

the LOG shows the following:

```

237 %isitquoted(&vehiclecolor)
MLOGIC(ISITQUOTED): Beginning execution.
MLOGIC(ISITQUOTED): Parameter VAR has value [Tom's Truck] is blue
MLOGIC(ISITQUOTED): %IF condition &var ne is TRUE
MLOGIC(ISITQUOTED): %PUT &var
Tom's Truck is blue

```

Again the boxes give some indication as to what is quoted.

Notice that when the %PUT statement is used to display &VAR, the invisible quote marks do not appear. It is possible to use the %PUT statement to display these characters, however, by using the %PUT statement options that list macro variables (_LOCAL_, _GLOBAL_, _USER_, and _ALL_). The following example uses %PUT with the _USER_ option to display the same two variables that were used in the previous examples. Now the marks are displayed.

```

251 %let vehiclecolor = &owner is blue;
252
253
254 %put _user_;
GLOBAL OWNER [Tom's Truck]
GLOBAL VEHICLECOLOR [Tom's Truck] is blue

```


7.1.4 Basic types of quoting functions and why we care

NR Functions (for example, %NRSTR and %NRBQUOTE)

Of special interest in the text strings to be quoted are those that contain the special macro symbols % and &. Normally, these symbols indicate references to macro calls and macro variables that must be resolved prior to the execution of the function. These references are resolved by rescanning the text as many times as it takes to resolve the usage of the % and &. Sometimes this is not how you want to have these types of references handled. Sometimes you do not want these references resolved. Sometimes you want to pass the macro call or macro variable without resolution. In each of these cases, you need a function designated as a no-rescan or no-resolve (NR) function. Most of the quoting functions come in pairs (the name either does or does not start with the letters NR, such as, %STR and %NRSTR).

These two forms of the quoting functions perform essentially the same operation except that the function that begins with the letters NR will handle % and & differently than those that do not start with NR. When a quoting function whose name starts with NR is used, the macro processor will not see the % and & as the special characters in exactly the same way that it normally does. It is important to note, however, that the NRquoting functions do not all remove the meaning in the same way for each of the quoting functions. Nor does the NR necessarily completely prevent the resolution of a macro reference. The topic is complicated, and some examples are included below to highlight some of these differences.

The %NRSTR function behaves in the same way as %STR except that meaning is also removed from the % and the & at compile time. In the following example, the macro variable &CITY is not resolved in the %PUT because the special meaning has been removed from the &:

```
%LET CITY = MIAMI;
%PUT %NRSTR(&CITY) IS ON THE WATER.;
```

The LOG would show

```
&CITY IS ON THE WATER.
```

The %NRBQUOTE function is the no-resolve equivalent of the %BQUOTE function. Like the %NRSTR function, the %NRBQUOTE masks & and %; however, unlike the %NRSTR, it will first attempt to resolve the macro references and will then quote the result. In the following example the macro variable &STORE is given the value of A&P. Assume that the macro variable &P has not been defined.

```
%let store = A&P;
%put The store name is &store;
```

When the %LET is executed, a warning (Apparent symbolic reference P not resolved.) will be issued, and the unresolvable result (A&P) is placed into &STORE. Whenever &STORE is used, the macro facility will again attempt to resolve &P and another warning will be issued. Therefore, the %PUT statement will result in a second warning. However, if the %NRBQUOTE is used when &STORE is created as is shown here

```
%let store = %nrblockquote(A&P);
%put The store name is &store;
```

only the first warning from the %LET will be issued and subsequent uses of &STORE will **not** result in additional warnings as the &P is **not rescanned**. When %NRSTR is used as it is below, neither statement generates a warning and no attempt is made to resolve the &P.

```
%let store = %nrstr(A&P);
%put The store name is &store;
```

MORE INFORMATION

The %NRQUOTE function is used in the %SYMCHK macro in Section 13.1.5. The %NRSTR is used to delay resolution of the %SYMDEL statement in an example in Section 13.1.6.

Function application at compilation versus execution

Because macro statements are compiled and then executed, you may, on occasion, need to control when the text string is to be quoted. With the exception of %STR and %NRSTR, each of the quoting functions performs their action during macro execution. When the function is applied during compilation (%STR and %NRSTR), only the resultant text is passed to the processor. All of the other quoting functions are applied during the execution of the macro. For these functions, the entire call to the function and its associated text is passed to the macro processor where the function is applied when the statement is executed.

This will rarely be an issue for most macro programmers. Usually you need to worry about compilation versus execution only if text that is resolved during compilation becomes syntactically incorrect or if it is misinterpreted during execution.

7.1.5 A bit about the %QUOTE and %NRQUOTE Functions

The %QUOTE and %NRQUOTE functions are very similar to %STR and %NRSTR in terms of what characters are masked. The primary difference is that they are applied during the execution of the macro statement rather than during macro compilation. With the availability of the stronger %BQUOTE and %NRBQUOTE functions, these two functions are now only rarely used.

Like the %STR and %NRSTR functions you can mark mismatched quotes and parentheses so that they will be masked (see Section 7.1.9) when using these two functions as well.

MORE INFORMATION

The %NRQUOTE function is used in the %SYMCHK macro in Section 13.1.5.

SEE ALSO

Whitlock (1999c) uses the %QUOTE function in an example that limits the number of observations in an output data set.

7.1.6 %UNQUOTE

Once a quoting function has been applied, its effects remain associated with the text, even in subsequent usages. If you need to remove or change the effects of any of the quoting functions, use the %UNQUOTE function.

Three macro variables are defined below:

- ❶ &OTH is defined using the %NRSTR function.
- ❷ &UNQ will contain the same value as &OTH. However, the %UNQUOTE is used to counter the effects of the %NRSTR function.

- ❸ Because of the use of %NRSTR, &CITY cannot be resolved when &OTH is resolved. However, when the %UNQUOTE function is applied to &OTH its value (&CITY) is seen as a macro variable that is also resolved.

```
%let city = miami;
%let oth = %nrstr(&city); ❶
%let unq = %unquote(&oth); ❷

%put &city &oth &unq;
```

The LOG shows

```
miami &city miami ❸
```

Although &OTH looks like any other macro variable in the %PUT statement, it will not be treated as such because it is quoted. This can cause programming problems if the programmer does not know that a macro reference or special character has been quoted.

Summary

- The %UNQUOTE function undoes (or turns off) the effect of the %BQUOTE, %NRBQUOTE, %NRQUOTE, %NRSTR, %QUOTE, and %STR functions, as well as functions that return quoted results, for example, %QSUBSTR.
- The change affected by this function occurs during macro execution.
- An argument can be any character string.

MORE INFORMATION

Section 7.1.3 includes a discussion on methods that you can use to determine if macro variables contain quoted text. The fourth example in Section 10.1.5 uses the %UNQUOTE when building a FILENAME statement. The %UNQUOTE is used in Section 13.1.6 to control the timing of the %SYMDEL statement.

SEE ALSO

Carpenter (1998b) includes an example of quoted strings that can become problematic.

7.1.7 %SUPERQ

The %SUPERQ function operates only on the values of macro variables. It masks all items that may require quoting at macro execution. The argument to the function is the name of a macro variable (without the ampersand) or text that resolves to the name of a macro variable.

%SUPERQ provides the ultimate in quoting protection and is often used with macro variables that may contain text that is supplied by the user, such as through the use of %INPUT and &SYSBUFFR. Several of the examples shown in *SAS Macro Language: Reference, First Edition* (pp. 85–88) and in *SAS Macro Language: Reference, Version 8* (pp. 82–84) refer to the use of %INPUT and &SYSBUFFR.

In the following example, the macro variable &HASCALL contains a call to the macro %DOIT. When the first %PUT ❶ executes and attempts to write the value of &HASCALL to the LOG, the macro %DOIT will be executed. You can prevent this by using the %SUPERQ function ❷.

```

data _null_;
call symput('hascall', 'Call macro %doit');
run;

%put &hascall; ❶
%put %superq(hascall); ❷

```

Notice that when you use the %SUPERQ function, the macro variable name is stated without the ampersand.

Summary

- The %SUPERQ function is particularly useful for masking the value of macro variables that may be generated through the use of the %INPUT and %WINDOW statements (see Section 5.4.5).
- %SUPERQ removes meaning during macro execution from special characters, such as & and %, and from mnemonic operators such as AND or OR.
- The argument is either a macro variable name without the ampersand or text that resolves to a macro variable name.

7.1.8 Quoting function summary

Items that may need quoting can be placed into one of three groups, A, B, or C.

A symbols + - * / < > = ^ ; , | and comparison operators such as AND, OR, NOT, EQ, NE, LE, LT, GE, and GT. In SAS 9.1 this group will also contain IN and #.

B macro symbols such as & or %.

C unmatched and unmarked symbols normally expected in pairs such as quotation marks or parenthesis. (See Section 7.1.9 for a discussion of a method of marking these symbols when using %STR and %QUOTE.)

The quoting functions deal with these groups differently, as is shown in this table:

Function	Groups Affected	Works at
%STR	A	macro compilation
%NRSTR	A, B	
%QUOTE	A	macro execution
%NRQUOTE	A, B	
%BQUOTE	A, C	macro execution
%NRBQUOTE	A, B, C	
%SUPERQ	A, B, C	macro execution (prevents resolution)

7.2 Text Functions

Macro text functions either change or provide information about the text string that is provided as one of their arguments. These functions are analogous to similar character functions in the DATA step.

The functions discussed in this section are shown in the following table:

Section	Macro function(s)	Analogous DATA step function	Task
7.2.1	%INDEX	index	Locate first occurrence of a text string
7.2.2	%LENGTH	length	Count characters
7.2.3	%SCAN %QSCAN	scan	Search for the <i>n</i> th word in a text string
7.2.4	%SUBSTR %QSUBSTR	substr	Select text based on position
7.2.5	%UPCASE	upcase	Convert to upper case

It is important to remember the differences between these macro functions and their DATA step counterparts. DATA step functions always work on character strings and DATA step variables. Macro functions are applied to text strings that **never** contain the values of DATA step variables.

Notice that several of these functions have two forms, such as %SCAN and %QSCAN. Functions whose names start with Q (quoting) remove the meaning from special characters such as the ampersand (&), percent sign (%), and mnemonic operators in values that are returned by the function.

7.2.1 %INDEX

The %INDEX function searches the first argument (*argument1*) for the first occurrence of the text string in the second argument (*argument2*). If the target string is found, the position of its first character is returned as the function's numeric response.

Syntax

`%INDEX(argument1, argument2)`

Value returned

Position of the string (0 if not found)

This example stores three words in the macro variable &X. The %INDEX function is then used to search in &X for the string TALL, and the result is then displayed using the %PUT statement.

```
%LET X=LONG TALL SALLEY;
%LET Y=%INDEX(&X, TALL);
%PUT TALL CAN BE FOUND AT POSITION &Y;
```

Notice that the second argument, TALL, is not in quotes. The %PUT statement results in the following text being written to the LOG:

```
TALL CAN BE FOUND AT POSITION 6
```

It is also common for both arguments to be macro variables. The previous example could easily be rewritten with two macro variable arguments and still return the same result:

```
%LET SRCH = TALL;
%LET X=LONG TALL SALLEY;
%LET Y=%INDEX(&X, &SRCH);
%PUT &SRCH CAN BE FOUND AT POSITION &Y;
```

It is also not unusual to use the %INDEX function as part of a logical comparison. The macro %CHECK below is a simplified version of a macro that monitors an ongoing process. Periodically, %CHECK is executed and &VALUE is examined:

```
%LET VALUE=WATCH ENGINEERING EMERGENCY;

%MACRO CHECK;
  %IF %INDEX(&VALUE, EMERGENCY) > 0 %THEN
    %PUT *** CRITICAL ***;
  %ELSE %PUT *** FINE ***;
%MEND CHECK;
```

The function returns a value greater than 0 when the text is found. This value is then used by the %IF to make its determination.

SEE ALSO

Aboutaleb (1997b) uses the %INDEX function in an example that deals with character variable strings. Whitlock (1999c) uses the %INDEX function to check for a two-level data set name.

7.2.2 %LENGTH

The %LENGTH function determines the length (number of characters) of its argument. The number of detected characters is then returned. Unlike the DATA step LENGTH function, which will always return a value of at least 1, when the argument for %LENGTH is a null string, the value 0 is returned.

Syntax

```
%LENGTH(argument)
```

Value returned

Number of characters in *argument*
0 if *argument* is a null string

Graphics catalog entries can be named by using the NAME= option. Even in Version 8, however, the name cannot exceed eight characters. In the following example, each of a series of variables is to be plotted against the variable stored in &DEPVAR. Some of these names may, however, be longer than the eight-character limit. The %LENGTH function ❶ is used to detect longer names.

```

...portions of the macro not shown...

filename cntyplt "&path\results\fsa\regsctr";

goptions device=webframe gsfname=cntyplt;
symbol1 c=blue v=dot r=45;
axis1 label=(a=90 f=simplex c=blue
             'Adjusted MVT Rate');
title2 'Data Plots';
proc gplot data=mvt.fsacnty;
  %do i = 1 %to &varlistcnt;
    %if %length(&&varlist&i) > 8 %then ❶
      %let v8name=%substr(&&varlist&i,1,8);
    %else %let v8name=&&varlist&i;
    %* One plot statement for each
    %* independent variable;
    plot (&depvar)*(&&varlist&i)/vaxis=axis1
                                     name="&v8name";
  %end;
run;
quit;

```

This solution assumes that the first eight characters of each dependent variables name (&&VARLIST&I) are unique.

MORE INFORMATION

It is possible for the %LENGTH function to count trailing blanks when the argument has been created using a quoting function. An example and discussion can be found in Section 12.3.5.

SEE ALSO

Aboutaleb (1997b) uses the %LENGTH function in an example that deals with character variable strings.

7.2.3 %SCAN and %QSCAN

The %SCAN and %QSCAN functions both search a text string (*argument1*) for the *n*th word (*argument2*) and return its value. If *argument3* is not otherwise specified, the same word delimiters are used as in the DATA step SCAN function. For an ASCII system, these include the following (for EBCDIC, the ¬ is substituted for the ^):

```
blank . < ( + | & ! $ * ) ; ^ - / , % > \
```

%QSCAN removes the significance of all special characters in the returned value.

Syntax

```
%SCAN(argument1, argument2 <,delimiters>)
%QSCAN(argument1, argument2 <,delimiters>)
```

Value returned

*n*th word, where *n* is *argument2*. A null value is returned if the *n*th word does not exist.

The macro variable &X in the following example can be broken up into words using the %SCAN function:

```
%LET X=XYZ:ABC/XYZ;
%LET WORD=%SCAN(&X,3); the 3rd word is XYZ
%LET PART=%SCAN(&X,1,Z);
%PUT WORD IS &WORD AND PART IS &PART;
```

The %PUT returns the following:

```
WORD IS XYZ AND PART IS XY
```

Notice that the third argument (delimiter list) is not enclosed in quotes as it would be in the DATA step SCAN function.

The %QSCAN function is usually needed when you want to return a value that contains an ampersand, percent sign, or other special character that may be misinterpreted. This is demonstrated in the following example:

```
%let dsn = clinics;
%let string = %nrstr(*&stuff*&dsn*&morestuff);

%let wscan = %scan(&string,2,*);
%let wqscan = %qscan(&string,2,*);

%put &wscan &wqscan;
```

The %PUT statement writes

```
clinics &dsn
```

Both functions return the value &DSN, but because the meaning of the & is not masked by %SCAN, &DSN is resolved to clinics before it is assigned as the value of &WSCAN.

The following example counts the number of BY variables that are contained in the macro variable &KEYFLD. In this example, each variable name is saved in a macro variable, such as &VAR2, as is the overall count of variables (&CNT).

```
%Macro cntvar;
  %global cnt;
  %let I = 1;
  %do %until(%scan(&keyfld,&I,%str( ) ❶)=%str() ❷);
    %global var&I;
    %let var&I = %scan(&keyfld,&I,%str( ) ❶);
    %let I = %eval(&I + 1);
  %end;
  %let cnt = %eval(&I-1);
%mend cntvar;
```

- ❶ The word delimiter in both of these %SCAN function calls is a blank character, and the %STR function is used to preserve the space.
- ❷ Notice that the second %STR function in the %DO %UNTIL statement does not encompass a blank because it is used to test for a null string.

After running the macro shown, you might need to know the name of the final variable in the list so that it can be used in FIRST. and LAST. processing. You could use the following statement in a subsequent DATA step to select for unique observations:

```
if first.&&var&cnt and last.&&var&cnt;
```

Although not documented, it is permissible to use negative numbers as the second argument in %SCAN. Negative values count words from the right rather than from the left, and the following would return the last word in &KEYFLD without knowing the number of words in &KEYFLD:

```
%let last = %scan(&keyfld,-1,%str( ));
```

The IF statement from above could be rewritten as the following:

```
if first.%scan(&keyfld,-1,%str( ))
    and last.%scan(&keyfld,-1,%str( ));
```

MORE INFORMATION

Section 11.4.3 expands on this example. The macro %WORDCNT in Section 7.3.2 uses a slightly different approach to count the number of words.

SEE ALSO

The %SCAN function is used in a similar manner in Roberts (1997). Izrael, Hoaglin, and Battaglia (2000) use the %SCAN to rank a list of numbers. Whitlock (1999c) uses the %QSCAN function in an example that limits the number of observations in an output data set. Lund (2003a) uses a -1 as the second argument to find the last BY variable in a list.

Q: just remove the special meaning of those special characters such as "&".

7.2.4 %SUBSTR and %QSUBSTR

Like the DATA step SUBSTR function, these macro functions return a portion of the string in the first *argument*. The substring starts at the *position* in the second argument and optionally has a *length* of the third argument.

Syntax

```
%SUBSTR(argument, position <length>)
%QSUBSTR(argument, position <length>)
```

Value returned

Substring of *argument*

As is the case with most other macro functions, each of the three arguments can be a text string, macro variable, expression, or a macro call. If you do not specify a value for *length*, a string that contains all of the characters from *position* to the end of the argument is produced.

```
%LET CLINIC=BETHESDA;
%IF %SUBSTR(&CLINIC,5,4) = ESDA %THEN %PUT *** MATCH ***;
%ELSE %PUT *** NOMATCH ***;
```

SAS would print *** MATCH *** because &CLINIC has the value ESDA in characters five through eight.

As shown in the following example, the %QSUBSTR function enables you to return unresolved references, as well as other special characters, to macros and macro variables:

```
%let dsn=clinics;
%let string = %nrstr(*&stuff*&dsn*&morestuff);

%let sub = %substr(&string,9,5);
%let qsub = %qsubstr(&string,9,5);

%put &sub &qsub;
```

The %PUT statement will write `clinics* &dsn*` in the LOG.

7.2.5 %UPCASE and %QUPCASE

The %UPCASE macro function converts all characters in the *argument* to upper case. This function is especially useful when you need to compare text strings that may have inconsistent case.

Syntax

```
%UPCASE(argument)
%QUPCASE(argument)
```

Value returned

ARGUMENT in all capital letters

The following code enables the user to differentially include a KEEP= option in the PROC PRINT statement. The %UPCASE function controls for variations in text that is supplied by the user in the macro call.

```
%macro printit(dsn);
  * use a KEEP for CLINICS;
  %if %upcase(&dsn)=CLINICS %then
    %let keep=(keep=lname fname ssn);
  %else %let keep=;
  proc print data=&dsn &keep;
    title "Listing of %upcase(&dsn)";
  run;
%mend printit;

%printit(cLinICs)
```

This macro call to %PRINTIT produces the following code:

```
proc print data=cLinICs (keep=lname fname ssn);
  title "Listing of CLINICS";
run;
```

7.3 Evaluation Functions

Because there are no numbers or numeric variables in the macro language (there is only text and macro variables that contain text), numeric operations such as arithmetic and logical comparisons become problematic. Evaluation functions are used to bridge the gap between text and numeric operations.

The evaluation functions are used

- to evaluate arithmetic and logical expressions.
- inside and outside of macros.
- during logical comparisons to specify TRUE or FALSE. Evaluation functions return a value of 1 for logical expressions if the condition is true, 0 if it is false.
- to perform integer and floating point arithmetic.

The requests for these functions are either *explicit* (called by the user by name) or *implicit* (used automatically during comparison operations without being directly called).

7.3.1 Explicit use of %EVAL

The %EVAL function **always** performs integer arithmetic. Even when the result of the arithmetic operation is not an integer (for example, 7/3), the value returned by the %EVAL function will still be an integer.

Syntax

%EVAL(argument)

In the following example, the %EVAL function is called to perform arithmetic operations. The code uses %EVAL to add the value of 1 to &X, which in this case contains 5.

```
%LET X=5;
%LET Y=&X+1;
%LET Z=%EVAL(&X+1);
%PUT      &X      &Y      &Z;
```

The %PUT statement writes the following to the LOG:

```
5      5+1      6
```

Noninteger arithmetic is not allowed. The statement

```
%LET Z=%EVAL(&X+1.8);
```

would result in the following message being printed in the LOG:

```
ERROR: A character operand was found in the %EVAL function or %IF
condition where a numeric operand is required. The condition was:
5+1.8
```

Although noninteger arithmetic is not allowed, integer arithmetic that results in a noninteger value is allowed. The following %LET statement:

```
%let d = %eval(7 / 3);
```

results in the value 2 being placed into &D.

Like most languages that distinguish between integer and noninteger values, the macro language has specific rules that define integers. Certainly we would not expect the number 1.8 to be an integer, but what about 1.0? It turns out that the presence of the decimal point is sufficient to cause a number to be seen as a noninteger. The numbers 1.0 and even 1. are both seen as nonintegers and could not be used in integer arithmetic.

MORE INFORMATION

Sections 5.3.3 and 5.3.4 provide other examples of the explicit use of the %EVAL function.

7.3.2 Implicit use of %EVAL

Sometimes the %EVAL function will be used even when it is not explicitly included in the code. This is an implicit use of %EVAL and there are a number of factors that will cause its insertion. Most common is the logical comparison of two integers (as is shown in the next example). When the comparison is between two integers, the %EVAL is used so that a numeric, rather than an alphabetic, comparison can be made.

```
%macro chkwt(wt1, wt2);
  %if &wt1 > &wt2 %then %let note = heavier;
  %else %let note = lighter;
  %put First weight is &note. &wt1 &wt2;
%mend chkwt;
```

```
%chkwt(1,2)
%chkwt(2,1)
%chkwt(2.1,2.2)
%chkwt(10.0,9.9)
```

compared in an integer way
compared alphabetically

Notice that the third and fourth calls to %CHKWT have noninteger parameters. %EVAL will not be used in noninteger comparisons, and these numbers will instead be compared alphabetically. Because 1 comes before 9 alphabetically, 10.0 is seen as **smaller** than 9.9 in the fourth comparison.

The LOG shows

```
21  %chkwt(1,2)
First weight is lighter  1 2
22  %chkwt(2,1)
First weight is heavier 2 1
23  %chkwt(2.1,2.2)
First weight is lighter 2.1 2.2
24  %chkwt(10.0,9.9)
First weight is lighter 10.0 9.9
```

The incorrect determination in the last comparison is not seen as a SAS mistake, but rather as a user error for working with nonintegers!

When determining whether a number is or is not an integer, the macro interpreter bases the decision on the presence or absence of any noninteger characters.

When both sides of the comparison are seen as integers the %EVAL function is used to make the numeric comparison. For integers it is as if the programmer had written the comparison as

```
%if %eval(&wt1 > &wt2) %then %let note = heavier;
```

An implicit use of the %EVAL also occurs within arguments of some functions. The following macro function counts the number of blank separated words in the parameter that is passed into the macro. The second argument of the %QSCAN function has a mathematical operation; however, the %EVAL has not been explicitly used. This works because SAS detects the plus sign (+) and assumes an arithmetic operation, which it performs using an implicit %EVAL.

```
%macro wordcnt(string);
%local string wcnt;
%* The word count is stored in wcnt;
%let wcnt=0; ❶
%do %until(%qscan(&string,&wcnt+1 ❷,%str( ))=%str());
%let wcnt=%eval(&wcnt+1);
%end;
&wcnt
%mend wordcnt;

%* example;;
%put count is %wordcnt(aa bb cc);
```

Unlike in the %CNTVAR macro shown in Section 7.2.3 the counter is initialized to zero. ❶ This simplifies the macro and prevents the need to “correct” the total at the end of the macro. Notice that the word number in the %SCAN function is incremented through the use of an implicit %EVAL function. ❷

Any of the macro functions that have implicitly numeric arguments (like the second argument of the %SCAN function) automatically invoke %EVAL. These include the following:

%DO	%IF-%THEN	%SUBSTR
%DO %UNTIL	%SCAN	%QSUBSTR
%DO %WHILE	%QSCAN	

MORE INFORMATION

Section 13.3.4 contains an example that uses %QSCAN incorrectly with some unanticipated results.

7.3.3 Using %SYSEVALF

You can get around the integer limitations of %EVAL by using the floating point evaluation function, %SYSEVALF. You can use %SYSEVALF to perform noninteger arithmetic as well as logical comparisons, and it will even return a noninteger result from an arithmetic operation.

Syntax

%SYSEVALF(*expression* <,*conversion-type*>)

The *expression* is any arithmetic or logical expression that is to be evaluated, and it may contain macro references.

The second argument, *conversion-type*, is an optional conversion to apply to the value that is returned by %SYSEVALF. Because this function can return noninteger values, problems could occur in other macro statements that use this function's result, but expect integers. Therefore when you need the result of this function to be an integer, use one of the conversion types. A specification of the *conversion-type* converts a value that is returned by %SYSEVALF to another form, such as an integer or Boolean value.

Conversion-type can be

BOOLEAN	0 if the result of the expression is 0 or missing. 1 if the result is any other value.
CEIL	round to the next largest whole integer.
FLOOR	round to the next smallest whole integer.
INTEGER	truncate the decimal fraction.

The CEIL, FLOOR, and INTEGER (which can be abbreviated as INT) conversion types act on the expression in the same way as the DATA step functions of the same (or similar) names, that is, the CEIL, FLOOR, and INT functions.

The following table shows a few results of the use of %SYSEVALF:

Example	Result (%put &x;)
%let x = %sysevalf(7/3);	2.3333333333
%let x = %sysevalf(7/3,boolean);	1
%let x = %sysevalf(7/3,ceil);	3
%let x = %sysevalf(7/3,floor);	2
%let x = %sysevalf(1/3);	0.3333333333
%let x = %sysevalf(1+.);	.
%let x = %sysevalf(1+.,boolean);	0

Although *SAS Macro Language: Reference, First Edition* (p. 237) states that the results will be written using BEST32. (Version 6 and later), this may not always be the case. The format seems to have changed with different versions of SAS.

The macro %FIGUREIT, shown in the next example, was taken from the SAS online help system for release 6.12. It demonstrates each type of conversion for values that are returned by the %SYSEVALF function.

```

%macro figureit(a,b);
  %let y=%sysevalf(&a+&b);
  %put The result with SYSEVALF is: &y;
  %put Type BOOLEAN is: %sysevalf(&a +&b, boolean);
  %put Type CEIL is: %sysevalf(&a +&b, ceil);
  %put Type FLOOR is: %sysevalf(&a +&b, floor);
  %put Type INTEGER is: %sysevalf(&a +&b, int);
%mend figureit;

%figureit(100,1.597)

```

Executing this program writes the following to the SAS LOG:

```

The result with SYSEVALF is: 101.597
Type BOOLEAN is: 1
Type CEIL is: 102
Type FLOOR is: 101
Type INTEGER is: 101

```

The %SYSEVALF function can also be used for the comparisons of noninteger values. In the example in Section 7.3.2, the comparison indicates that $10.0 < 9.9$. The following %IF statement uses the %SYSEVALF function and correctly indicates that $10.0 > 9.9$:

```
%if %sysevalf(&wt1 > &wt2) %then %let note = heavier;
```

%SYSEVALF can also be used to build date, time, and datetime constants in the macro language. The following call to %SYSEVALF converts the datetime constant to the number of elapsed seconds:

```

%let time = 18nov2003:12:25:10;
%put &time;
%put %sysevalf("&time"dt);

```

MORE INFORMATION

The %SLEEP macro in Section 7.6.1 uses nested calls to %SYSEVALF to work with datetime constants. Although %SYSEVALF does not support a ROUND conversion type, rounding can be achieved by using the %SYSFUNC function in conjunction with %SYSEVALF (see Section 7.4.2).

7.4 Using DATA Step Functions and Routines

Two macro functions enable you to execute a majority of the functions and routines that are available in the DATA step as if they were a part of the macro language. %SYSCALL calls DATA step routines, and %SYSFUNC executes DATA step functions.

A call routine alters variable values or performs other system function. "CALL routines" are similar to Chapter 7: Using Macro Functions 171 functions, but differ from functions in that you cannot use them in assignment statements.

7.4.1 Using %SYSCALL

You can use %SYSCALL to execute CALL routines that are ordinarily called only from within the DATA step. Because these routines normally operate on data set variables that are, of course, not available to the macro language, the arguments are instead generally macro variables.

Syntax

%SYSCALL call-routine <(routine-arguments)>;

Nearly all CALL routines, either user-written with SAS/TOOLKIT software, or supplied with SAS, can be used with %SYSCALL. The primary exceptions include: LABEL, VNAME, SYMPUT, and EXECUTE. This leaves the routines for random number generation and system command execution. See Chapter 12, "SAS CALL Routines," (pp. 149–160) in *SAS Language Reference, Version 6, First Edition* for more information on DATA step CALL routines.

When you use %SYSCALL, the arguments will be macro variables specified without the ampersand. In the following example, the SYSTEM routine is used to execute the operating system directory command at the DOS level:

```
%let cmd = dir;
%syscall system(cmd);
```

Notice that the argument is a macro variable without the ampersand. Inserting the directory command, DIR, directly instead of the macro variable &CMD would not have worked because DIR does not resolve to the name of a macro variable.

If you need to create a macro variable that contains a random number, you can use any of the numerous random number routines. The following example assigns a random number to the macro variable &RAND, which must be initialized prior to the call:

```
%let seed=9876;
%let rand=0 ;
%put seed is &seed pseudo random number is &rand;
%syscall ranuni(seed,rand);
%put seed is &seed pseudo random number is &rand;
```

After execution the LOG will show the following:

```
5  %let seed=9876;
6  %let rand=0 ;
7  %put seed is &seed pseudo random number is &rand;
   seed is 9876 pseudo random number is 0
8  %syscall ranuni(seed,rand);
9  %put seed is &seed pseudo random number is &rand;
   seed is 1482492922 pseudo random number is 0.6903395628
```

Note that the value of both macro variables, &SEED and &RAND, were changed by the call to RANUNI.

MORE INFORMATION

The macro %BUILDVARLIST in Section 11.4.1 uses %SYSCALL to execute the RXFRI routine.

(Comparison !!)
CALL routines on data step variables
%syscall routines on macro variables specified without &

SAS Macro

Function

A macro X will be passed and call ranuni(seed, X)
↓
a macro seed is returned each time after execution.

SEE ALSO

SAS Macro Language: Reference, First Edition (pp. 231–232) documents the syntax of %SYSCALL.

7.4.2 Using %SYSFUNC and %QSYSFUNC

These two macro functions greatly increase the list of functions that are available to the macro language by making available many of the DATA step functions. As a consequence, they can substantially reduce the need for single-observation DATA _NULL_ steps.

Syntax

```
%SYSFUNC(function-name(function-arguments)<format>)
%QSYSFUNC(function-name(function-arguments)<format>)
```

The first argument is the DATA step function that is to be executed and in the second optional argument you may supply a format that is to be applied to the result of the function. Not all of the DATA step functions may be used with %SYSFUNC; however, most are available. You will notice slight variations in how the arguments to the functions are specified when they are used with %SYSFUNC. For the most part the differences will be due to the differences between how the DATA step uses variable names and quoted strings, and the macro language's specification of macro variables and text (without quotes).

Converting dates

The format of the automatic macro variable &SYSDATE9 is converted using the WORDDATE. format in the following example:

```
%put &sysdate9;
%put %sysfunc(putn("&sysdate9"d,worddate18.));
```

The LOG shows

```
73 %put &sysdate9;
17SEP2003
74 %put %sysfunc(putn("&sysdate9"d,worddate18.));
September 17, 2003
```

Specifying dates in titles

The following example shows three ways to add the current date to a TITLE. Although the automatic macro variable &SYSDATE is easy to use, it usually has to be reformatted. More importantly, since &SYSDATE and &SYSDATE9 only indicate the date that the current SAS session started, they may not be current. Consequently, prior to the availability of %SYSFUNC, most users used a DATA _NULL_ step with an assignment statement and a CALL SYMPUT statement.

Handwritten notes:
 " &sysdate9 " = without it it will not be treated as a number!!
 must be a number!!
 would be a variable with a value assigned at runtime

create a formatted macro variable. The DATA step can now be avoided by using the %SYSFUNC macro function, as is shown in TITLE3 below:

```
data _null_;
    today = put(date(),worddate18.);
    call symput('dtnull',today);
run;

title1 "Using Automatic Macro Variable SYSDATE &sysdate";
title2 "Date from a DATA _NULL_ &dtnull";
title3 "Using SYSFUNC %sysfunc(date(),worddate18.)";
```

This code produces the following three titles:

```
Using Automatic Macro Variable SYSDATE 26MAR2003
Date from a DATA _NULL_      March 26, 2003
Using SYSFUNC      March 26, 2003
```

The leading spaces before the date in the second and third titles are caused by the date string being right-justified. You can use the LEFT and TRIM functions to remove the space. However, if you are not careful, you may encounter a couple of problems.

The first problem is that function calls cannot be nested within %SYSFUNC. Fortunately, this is rather easily handled because you can nest %SYSFUNC requests.

Secondly, the resolved values of interior calls to %SYSFUNC are used as arguments to the outer calls. When the resolved value contains special characters, especially commas, they can be misinterpreted. The following revised TITLE3 will not work because the interior %SYSFUNC results in a formatted value that contains a comma:

```
title3 "Using SYSFUNC %sysfunc(left(%sysfunc(date(),worddate18.)))";
```

After the inner %SYSFUNC is executed, the result is

```
title3 "Using SYSFUNC %sysfunc(left(March 26, 2003))";
```

Because of the comma, the LEFT function will see two arguments (it is expecting exactly one), and the message “too many arguments” is generated.

You can use the %QSYSFUNC function to mask special characters in the text string that is passed to the outer function. Rewriting the title as is shown next eliminates the problem with the comma:

```
title3 "Using SYSFUNC %sysfunc(left(%qsysfunc(date(),worddate18.)))";
```

The title becomes

```
Using SYSFUNC March 26, 2003
```

The following macro has been slightly adapted from an example that can be found on p. 139 of *SAS Macro Language: Reference, First Edition* (1997) and on p. 137 of *SAS Macro Language: Reference, Version 8* (1999). The macro %DELFILE can be used to delete a file (in this example F:\junk\biomass.raw). In its current form, which only passes in the name of the *fileref*, this macro is not terribly useful, but it does illustrate an important point—and probably not one intended by the original programmer.

if the operation of FILENAME returns 0 if successful, then ...

```

%macro delfile(filrf);
  ** Establish the fileref;
  let rc=%sysfunc(filename(filrf ①, f:\junk\biomass.raw ③));
  ** Delete the file if it exists;
  if &rc = 0 and %sysfunc(fexist(&filrf ②)) %then
    let rc=%sysfunc(fdelete(&filrf ②));
  ** else %put File Not Found;

  ** Clear the fileref;
  let rc=%sysfunc(filename(filrf ①));
%mend delfile;

%delfile(myfile)

```

no quotes are required here!
macro variable without an ampersand!!
if exist, returns 1 if not, returns 0
selection is successful then returns 0 if not, then 1
delete the external file
clear the fileref

- ① When used with %SYSFUNC the FILENAME function **expects** the first argument to be a macro variable; hence, you would get an error if you used an ampersand as in

```
let rc=%sysfunc(filename(&filrf, f:\junk\biomass.raw));
```

When the ampersand is used, as in the previous statement, &FILRF resolves to MYFILE, and SAS looks for a macro variable named MYFILE (which it will not find).

The documentation is not clear as to which functions expect macro variable arguments to be specified **without** the ampersand when they are used with %SYSFUNC. Fortunately, few functions do.

- ② The FEXIST and FDELETE functions **do not expect** the argument to be a macro variable and the ampersand is required.
- ③ Notice that, as we might expect in the macro language, quotes are not required for the second argument of the FILENAME function.

Using the PATHNAME function

The following macro uses the PATHNAME function to retrieve the path that is associated with a *libref*. This path is then used to build a new *libref* with a different engine. Since the LIBNAME statements include an ENGINE option, the copied data sets will be rewritten using the new engine. If the new engine is one associated with SAS/ACCESS or another product such as DBMS/Engines, the output data set might even be converted to a format other than SAS.

```

%macro engchgng(engine,dsn);
  * engine - output engine for this &dsn
  * dsn - name of data set to copy

```

```

data _null_;
  * libref for location of new file;
  aa = pathname("sasuser"); ①
  call symput('outpath',aa); ②

```

```

  * Create a libref for the stated Engine;
  libname dbmsout clear;
  libname dbmsout &engine "&outpath"; ③

```

disassociates one or more currently associated librefs.

Syntax
pathname (fileref)
returns the physical name of an external file or SPS library

```

* Copy the SAS data set using the alternate engine;
proc datasets;
  copy in=sasuser out=dbmsout; ❹
  select &dsn;
run;

%mend engchn;

*****;
%engchn(v6,classwt) ❺ * convert to alt. engine;

```

- ❶ The PATHNAME function is used to return the PATH, which is associated with an established *libref* (SASUSER).
- ❷ This PATH is then stored in the macro variable &OUTPATH.
- ❸ A new *libref* is established using the alternate engine (&ENGINE) and the original PATH (&OUTPATH).
- ❹ PROC DATASETS uses the new engine to copy the selected member (&DSN).
- ❺ This call to %ENGCHNG specifies that the new engine should be V6 when copying SASUSER.CLASSWT.

The previous version of macro %ENGCHNG does not take advantage of the %SYSFUNC function and relies instead on the DATA _NULL_ step. This macro is rewritten and substantially simplified using %SYSFUNC:

```

%macro engchn(engine,dsn);
  * engine - output engine for this &dsn
  * dsn     - name of data set to copy
  *;

  * Create a libref for the stated Engine;
  libname dbmsout clear;
  libname dbmsout &engine "%sysfunc(pathname(sasuser))";

  * Copy the SAS data set using the alternate engine;
proc datasets;
  copy in=sasuser out=dbmsout;
  select &dsn;
run;

%mend engchn;

*****;
%engchn(v6,classwt) * convert to alt. engine;

```

Notice that the *libref* (SASUSER) in the PATHNAME function call is **not** in quotes. Remember that arguments to macro functions are always text, so the quotes are not necessary. As you use DATA step functions with %SYSFUNC, you should expect the behavior of many of the arguments of the DATA step functions to vary slightly in ways such as this.

Generating PATTERN statements using PUTN

In the following example, the macro %PATTERN uses %SYSFUNC with the PUTN function to build a series of PATTERN statements that subdivide the spectrum of gray scales into equal divisions. The PATTERN statements are to be used with a PIE chart, and the macro determines the number of distinct slices that will be needed.

```
%macro pattern(dsn,pievar);
%local g i j nslice;

* Determine the number of unique values of the
* variable that will be used to determine the slices;
proc sql noprint;
select count(distinct &pievar) into :nslice ❶
  from &dsn;
run;

%do j = 1 %to &nslice;
  /* Create &nslice pattern statements;
  %let i=%sysvalf(255/(&nslice+1)*&j,floor); ❷
  %let g=%sysfunc(putn(&i,hex2.)); ❸
  pattern&j v=psolid c=gray&g r=1; ❹
%end;
%mend pattern;
```

- ❶ The number of unique values of the plotting variable (&PIEVAR) is saved in &NSLICE.
- ❷ There are 256 (this is 16^2) potential shades of gray. Since they gradually darken from white to black, they can be separated into &NSLICE equally spaced shadings.
- ❸ Each gray scale is identified using its hex number. The PUTN function, which is called by %SYSFUNC, performs the decimal to hex conversion.
- ❹ The HEX value, which is stored in &G, becomes a part of the name of the color.

The following code demonstrates the use of this macro.

```
data slices;
do slice = 1 to 10;
  output slices;
end;
run;

%pattern(slices,slice) * set up the pattern statements;

title 'Pie diagram of slices showing gray scale';
footnote;

goptions reset=all border htext=1.5 ftext=simplex;
goptions dev=win;

proc gchart data=slices;
  pie slice / type=percent
              slice=arrow
              noheading
              midpoints=1 to 10;
run;
quit;
```

The %PATTERN macro has the huge advantage of automatically separating the gray scale spectrum into the correct number of equally spaced shadings. The user does not need to know how many shades are needed or how to calculate the arcane values that the scale uses.

Rounding numbers

Although %SYSEVALF does not support a ROUND conversion type, rounding can be achieved by using the ROUND DATA step function with %SYSFUNC. The following statement will round the arithmetic result to the nearest tenth:

```
%let r = %sysfunc(round(%sysevalf(7/3),.1));
```

MORE INFORMATION

The view SASHELP.VSLIB also contains path information, and you can use it instead of the PATHNAME function. The SASHELP views are discussed in Section 9.5.1

The %SLEEP macro in Section 7.6.1 uses nested %SYSFUNC calls to work with datetime constants.

SEE ALSO

You can find a summary description and the syntax for DATA step functions that you can use with %SYSFUNC and %QSYSFUNC in Appendix 3, “Syntax for Selected SAS Functions Used with the %SYSFUNC Function,” (pp. 277–280) in *SAS Macro Language: Reference, First Edition*.

A short introduction to %SYSFUNC using functions such as FILENAME, DOPEN, DCLOSE, DNUM, DREAD, PUTN, and PUTC is provided by Kenney (1999).

Yindra (1998) includes numerous examples that use %SYSFUNC in a variety of interesting and very readable macro tools. Murphy (2003b) uses %SYSFUNC to gather information about the meta-data (number of observations, number of variables, variable names, variable types) of a data set.

Several examples are shown by Yu (1998) that use %SYSFUNC to read and write external files, while Chen (2003) discusses a number of ways to read external files. Lund (2003b) uses %SYSFUNC to return a number of external file attributes.

%SYSFUNC is used with the PUTN function to calculate hexadecimal to decimal conversions in Watts (2003a).

Burlew (1998, p.127), Stokke (2000); Izrael, Hoaglin, and Battaglia (2000), Mao (2003), Lund (2003b), and Hamilton (2000) each use %SYSFUNC with a GETOPTION function. Rook and Yeh (2001) use it with the PATHNAME, FEXIST, and FDELETE functions.

Teberg (2002) applies the %SYSFUNC function to the INTNX function to work with dates of files.

The date example used to introduce %SYSFUNC above is expanded in Carpenter (2000b). Palmer (2001) also uses %SYSFUNC to augment titles and footnotes with dates.

A variety of DATA step functions, such as VARTYPE, VARNAME, and FILEEXIST, are used with %SYSFUNC and %QSYSFUNC by Olaleye (1998), Borgerding and Chai (2000), Wang (2003), and Bramley (2001).

DBMS/Engines is a product of the DataFlux Corporation, a SAS Company, and enables a SAS program to read and write SAS data sets to and from formats other than SAS without SAS/ACCESS.

7.5 Building Your Own Macro Functions

The SAS macro language includes a number of macro functions, such as %UPCASE, %QSCAN, and %SUBSTR. In addition, a number of autocall macros (see Chapter 12), such as %LEFT, %TRIM, and %VERIFY, are also supplied that act like macro functions. Macro functions are especially useful as programming tools because the function call is replaced directly by the result of the call. Many macro programmers are unaware that they can also write useful macro functions.

Fortunately macro functions are not difficult to program—not difficult, that is, if you know the simple techniques required to turn a macro into a macro function. This section shows you how to create a macro function, the statements that you must avoid, and the technique used to “pass” the result of the function back to the calling program.

7.5.1 Introduction

Macros are often used as programming tools to answer specific questions about the programming environment. A part of this process is passing information back from the macro that determines the information to the macro that needs it. A typical macro solution is to have the macro create one or more macro variables that are added to the global macro symbol table. Since these variables are then available throughout the SAS session, they can later be accessed as required. The disadvantage of this approach is that the macro variables must be global and therefore the programmer risks collisions with other global macro variables that may be in use by the program or application. A further disadvantage is the need to use macro variables to pass information in the first place. These disadvantages can often be eliminated by converting the macro into a macro function.

The following macro, a variation of which was published in the *SAS Guide to Macro Processing, Version 6*, can be used to determine if a data set exists (see Section 7.1.3 also). It utilizes a DATA _NULL_ step, the automatic macro variable &SYSERR, and a global macro variable (&EXIST). When the macro is called, the DATA step is compiled including the SET statement. During compilation SAS determines if the data set (&DSN) exists, and if it does not exist, the macro variable &SYSERR will be loaded with a value greater than zero. Because of the STOP statement, no data are actually read.

```
%macro exist(dsn);
  %global exist;
  %if &dsn ne %then %do;
    * An unknown data set causes a
    * compile error that is reflected
    * in the SYSERR macro variable;
    data _null_;
      stop;
      set &dsn;
      run;
  %end;
```



```

    %if &syserr=0 %then %let exist=YES;
    %else %let exist=NO;
%mend exist;

```

After the macro has been called, branching and execution decisions can be made based on the value stored in the global macro variable &EXIST. For example, assume that in a DATA step a user wants to conditionally execute a block of code depending on the existence of the data set SASUSER.BIGDAT. The macro call and associated statements might be as follows:

```

%exist(sasuser.bigdat)

data .....;
  set.....;
  if "&exist"="YES" then do;
    ...more SAS statements...
  end;

```

The macro is first called (this creates and loads the global macro variable &EXIST) and then the global macro variable &EXIST is later tested. The user is responsible for not having another global macro variable named &EXIST; the user must know that the macro variable created by the macro is named &EXIST, and that &EXIST will take on the values of YES and NO. The macro works, and it is reasonably efficient, but it is NOT a macro function. The following discussion demonstrates how this macro differs from a macro function, and what needs to be done in order to convert it into a macro function.

7.5.2 Building the function

In order for a macro to be classified as a macro function, it must be usable within macro statements, and it must be able to build base language code directly. In the above example of %EXIST, the macro MUST be called outside of the DATA step. Since %EXIST generates its own DATA step, the following macro call would cause problems in the DATA step logic:

```

data .....;
  set.....;
  %exist(sasuser.bigdat)
  if "&exist"="YES" then do;
    ...more SAS statements...
  end;

```

Macro function attributes

The attributes required of a macro function include the following:

- All statements in the macro must be macro statements.
- The macro should create **no** macro variables other than those that are local to that macro.
- The macro should resolve to the value that is to be returned.

Use only macro statements

The %SYSFUNC macro function is a wonderful tool when building your own macro functions. It allows you to use DATA step functions without using the DATA step. In the previous version of the macro %EXIST the DATA step is used to ascertain whether or not the data set named in &DSN exists. Since the DATA step reads no data and is only used to build a macro variable, it

can be completely replaced with a call to %SYSFUNC. The following version of the macro %EXIST eliminates the use of the DATA step:

```
%macro exist(dsn);
%global exist;
/* Check if &DSN has been created;
%if %sysfunc(exist(&dsn)) %then %let exist=YES;
%else %let exist=NO;
%mend exist;
```

Since this version of %EXIST does not utilize a DATA step, the macro call could now be placed within the DATA step that uses &EXIST. Although the macro call from within the DATA step caused problems in the previous example, the following code will now work:

```
data .....;
set.....;
%exist(sasuser.bigdat)
if "&exist"="YES" then do;
...more SAS statements...
```

Notice that even the comments in the macro function should be macro comments. In this usage of %EXIST it really does not matter whether or not the comment is a macro comment, but as we continue to build the macro function, it becomes crucial that we use only macro-style comments.

Create only local macro variables

Since the previous version of %EXIST still creates a global macro variable (&EXIST), the macro is still not a macro function. The %LOCAL statement causes macro variables to be assigned to the local symbol table, and should be used for **all** of the macro variables that are defined within the macro. Use of the %LOCAL statement ensures that there will be no collisions between macro variables defined within the macro and those that might already exist on the global symbol table.

The problem, of course, is that these local macro variables cannot be accessed outside of the macro. How then does one pass values out of a macro without creating global macro variables? The answer to this question is at the heart of the solution of how to build macro functions.

The macro call resolves to the value to be returned

Rather than creating a macro variable that is made available later (often through the global symbol table), we can let the call to the macro resolve to the value of interest. This way the %EXIST(SASUSER.BIGDAT) macro call will be literally replaced in the code stream by whatever value is to be tested. In the following version of %EXIST the macro call will be replaced by either YES or NO:

```
%macro exist(dsn);
/* Check if &DSN has been created;
%if %sysfunc(exist(&dsn)) %then YES;
%else NO;
%mend exist;
```

Notice that the **action** for the %IF-%THEN statement (YES) and the %ELSE statement (NO) contains only text. Since a YES or NO is not a macro statement, it cannot be executed by the macro processor (as a %LET statement would be), and it is just “left behind.” When the macro expression is true (%SYSFUNC and the EXIST function find the data set named in &DSN), the

whole `%IF-%THEN/%ELSE` resolves to YES. Since this is not a macro statement, it is effectively passed on to the base language.

Now we can ask the IF-THEN question without referring to the macro variable `&EXIST` and without first calling the macro.

```
data .....;
  set.....;
  if "%exist(sasuser.bigdat)"="YES" then do;
  ...more SAS statements...
```

If the data set SASUSER.BIGDAT exists the DATA step becomes

```
data .....;
  set.....;
  if "YES"="YES" then do;
  ...more SAS statements...
```

When we can anticipate the behavior of the functions that are called from within the macro by `%SYSFUNC`, we can write more sophisticated macro functions. The DATA step function `EXIST`, for instance, returns a non-zero positive value when the data set is located. That means, of course, that we can test for that value directly, and this is what was done in the previous example—within the macro. The following example takes this one step further by performing the check outside of the macro rather than within it, and this eliminates the `%IF-%THEN/%ELSE`.

Like the above example, the following version of `%EXIST` contains only macro statements and it establishes no macro variables, neither global nor local. It further reduces what the user needs to know in order to use the macro by eliminating the YES/NO and, in the process, the need for the macro `%IF-%THEN/%ELSE` statement is also eliminated.

```
%macro exist(dsn);
  /* The following sysfunc call results
  /* in a non-zero value when the data
  /* set exists;
  %sysfunc(exist(&dsn))
%mend exist;
```

It is the last line of the above macro that establishes it as a macro function. The call to `%SYSFUNC` will execute the `EXIST` function. The result of that execution will be a number, that is, 0 or 1. Since this value is not a macro statement, it will be left behind, and this value can then be tested for directly by the calling program.

Writing your macro functions this way enables you to use the macro as part of either a DATA step statement or a macro statement. In a DATA step IF-THEN/ELSE we might use the following statement:

```
if %exist(sasuser.bigdat) > 0 then do;
```

Or better yet just

```
if %exist(sasuser.bigdat) then do;
```

If the data set exists the IF-THEN statement might become something like

```
if 1 then do;
```

And of course 1 is true.

When used in the DATA step IF-THEN statement shown above, the %EXIST macro is a bit silly since it could have been coded directly using the EXIST function and without the macro language at all.

```
if exist(sasuser.bigdat) then do;
```

This of course would miss the point of the discussion—that building macro functions need not be difficult and that following a few simple rules is all that is required to make functions that can simplify your programming tasks.

7.5.3 Using the function

The first definition of the %EXIST macro shown in the “Introduction” (Section 7.5.1) has several limitations that have already been discussed. In addition, this macro might not be usable in some macro environments. Again, this is a direct consequence of the fact that it must run a DATA step before it can build the macro variable &EXIST. This means that this macro could never be used in a macro that was itself being used as a macro function. Furthermore, the user would run the risk that the sequence of the execution of macro statements and DATA steps might cause problems (base language statements are executed after macro language statements and consequently the macro variable &EXIST might not have been created when it is needed).

Excluding the non-macro statements and building a macro function, such as the one shown in the final version of %EXIST, enables its use in both the macro language and the base language. As a macro function %EXIST can also be used in macro language statements such as

```
%if %exist(sasuser.bigdat) %then %do;
```

Because of this flexibility the macro function becomes a building block that can be used in the construction of other macro functions. For instance, a function designed to return the number of observations in a data set would first need to ascertain whether or not the data set exists.

Of course the final version of the %EXIST macro has become so simplified that we might as well have used the %SYSFUNC directly, and in the process created code that is a bit more efficient.

```
%if %sysfunc(exist(sasuser.bigdat)) %then %do;
```

While a macro programmer familiar with %SYSFUNC and the EXIST function would indeed probably not go to the trouble of creating the %EXIST macro, there may be other programmers in your group who are not that knowledgeable. By creating this and other macro tools, and by placing them in a macro library you can expand the capabilities of everyone in your group—especially those who might not yet understand the finer points of creating macro functions.

MORE INFORMATION

The macro %PATTERN in Section 7.4.2 is a macro function that generates a series of complete SAS statements. Section 7.6 has a number of macro function examples. Several of the AUTOCALL macros described in Section 12.2 are written as macro functions.

SEE ALSO

Much of the material in this section was taken from Carpenter (2002). Pete Lund (2000b, 2001a, and 2001b) has a series of clever tools written as macro functions. Larsen (2001) demonstrates a macro function to center text within a PUT statement. Hamilton (2001) discusses a macro function that returns the number of observations in a data set after first checking to see if the data set exists.

7.6 Other Useful Macro Functions

When learning how to perform a task it is sometimes helpful to see some simple or otherwise interesting examples. This section includes a number of macro functions that should provide some insight into how you can go about creating your own macro functions.

7.6.1 One liners

Does the data set exist?

%EXIST

The %EXIST function shown in Section 7.5.2 returns a non-zero value if the data set (&DSN) exists.

```
%macro exist(dsn);
    %sysfunc(exist(&dsn))
%mend exist;
```

Although it would be easier and more efficient to use the EXIST function directly in the DATA step, this macro function could be used as part of either a DATA step statement or a macro statement. In a %IF-%THEN/%ELSE we might use the following statement:

```
%if %exist(sasuser.bigdat) %then %do;
```

You might argue that it would be just as easy to use the %SYSFUNC directly as it would be to write and use this macro. In this simple case you might be correct; however, users who are less sophisticated programmers—especially in the macro language—might not agree. They might not understand the %SYSFUNC function or even know that the EXIST function exists. For these users it is often easier to have them “call the macro,” than it is to explain how to do it themselves. Besides, the %EXIST macro can now be placed in your library of available macro tools where everyone can get to it.

Calculating a factorial

%FACT

The factorial of a number is the number multiplied by each integer between itself and one. Five factorial (often symbolized as 5!) would be $5*4*3*2*1 = 120$. The factorial calculation is often needed when calculating combinations and permutations of groups.

Prior to version 7 there was no DATA step function that was designed to directly calculate a factorial. However, the GAMMA DATA step function can be used to calculate a factorial, where $\text{GAMMA}(6) = 5!$. The %FACT macro shown here returns the factorial of the argument:

```
%macro fact(n);
    %sysfunc(gamma(%eval(&n+1)))
%mend fact;
```

Factorials are calculated on positive integers, and since %EVAL does not accept noninteger arguments, this macro will fail if &N is not a positive integer. The next version of %FACT corrects for nonintegers by truncating fractional parts with %SYSEVALF, but it still does not accommodate negative numbers.

```
%macro fact(n);
  %sysfunc(gamma(%sysevalf(&n+1,int)))
%mend fact;
```

Three new functions were introduced in Version 7 that deal directly with the calculation of factorials:

FACT calculates factorials
 COMB calculates combinations
 PERM calculates permutations.

Each of these functions can now also be used in a macro function. The %FACT macro can be rewritten as follows:

```
%macro fact(n);
  %sysfunc(fact(&n))
%mend fact;
```

The related calculation of the number of combinations (n things taken r at a time) uses the COMB function.

```
%macro comb(n,r);
  %sysfunc(comb(&n,&r))
%mend comb;
```

MORE INFORMATION

The PERM function can also be used to calculate factorials as well as the number of permutations. A macro that uses the PERM function is shown in Section 7.6.3. Macros that will calculate the permutations for large numbers are shown in Section 13.1.4.

Working with dates

%CURRDATE

In Section 7.4.2 %SYSFUNC was used to add the current date to a title. In that section the suggested solution was

```
title3 "Using SYSFUNC %sysfunc(left(%qsysfunc(date()),worddate18.))";
```

The process is simplified by creating a macro to do the same job:

```
%macro currdate;
  %trim(%left(%qsysfunc(date()),worddate18.))
%mend currdate;
```

The macro can be used in the **TITLE** statement directly:

```
title3 "Current date is %currdate";
```

MORE INFORMATION

Notice that the **%CURRDATE** macro makes use of the **%LEFT** and **%TRIM** macro functions, rather than nesting calls to **%SYSFUNC**. These two macros are actually supplied by SAS as autocall macros, and they are discussed further in Chapter 12.

%DB2DATE

Like **%CURRDATE**, this macro builds a text string based on the current date; however, it is built in a form that can be utilized in an SQL pass-through to a DB2 Table.

The macro resolves to the **quoted** date string in the form of “YYYY-MM-DD-00.00.00”.

```
%macro db2date;
  %bquote(')%sysfunc(date(), yymmdd10.)%bquote(-00.00.00')
%mend db2date;
```

The following SQL step shows how this macro might be used in an SQL pass-through to DB2.

```
proc sql;
  connect to odbc (&db2j);
  create table sincemidnight
    as select * from connection to odbc (
      select *
        from mydba.hospital1
       where proddate > %db2date
      for fetch only);
  %put &sqlxmsg;
  disconnect from odbc;
quit;
```

When executed on December 21, 2003, the **WHERE** clause becomes

```
where proddate > '2003-12-21-00.00.00'
```

The macro **%DB2DATE** could also have been written using the **%STR** function; however, since **%STR** does not mask the **'**, special care must be taken.

```
%macro db2date;
  %str('%')%sysfunc(date(), yymmdd10.)%str(-00.00.00%)
%mend db2date;
```

SEE ALSO

Lund (2003b) has a macro that returns the current date.

Building a logical expression

%FUZZRNGE

This function can be used to create a logical expression. It is passed as a base value and a displacement value. An expression is then built that will check for values within the range of the base value \pm the displacement.

The statement

```
if %fuzzrng(age,7,2) then do;
```

becomes

```
if (5 le age & age le 9) then do;
```

The macro is passed three values: the variable name (&VAR), the base value (&BASE), and the displacement (&DISP).

```
%macro fuzzrng(var,base,disp);
  (%eval(&base-&disp) le &var & &var le %eval(&base+&disp))
%mend fuzzrng;
```

This function could also be used to build a macro language expression. Assuming that &VAL contains a number, a %IF statement could be

```
%if %fuzzrng(&val,9,6) %then %do
```

Because this macro uses %EVAL both explicitly and implicitly it will only work for integer values.

MORE INFORMATION

A macro language caveat associated with composite expressions of a range of values is discussed in Section 13.3.5.

Putting the computer to sleep

%SLEEP

This macro can be used to halt a SAS program for the number of seconds noted in the &TIME parameter. Because the execution of the SLEEP function itself is the desired result, this function does not pass any value back to the calling program.

```
%macro sleep(time=5);
  %local rc;
  %let rc = %sysfunc(sleep(&time));
%mend sleep;
```

The following macro call will suspend execution of SAS for about 1 minute:

```
%sleep(time=60)
```


The DATA step SLEEP function returns the number of seconds of sleep. Consequently, this value must be stored in the local macro variable &RC. You could make use of this returned value by passing it back to the calling program.

```
%macro sleep(time=5);
    %sysfunc(sleep(&time))
%mend sleep;
```

Now the macro might be called as follows:

```
%put WARNING: Computer slept for %sleep(time=5) seconds;
```

A slightly more sophisticated version of %SLEEP enables the user to specify the units of time to suspend the execution of SAS.

```
%macro sleep(time=5,unit=seconds);
    /* Units can be seconds, minutes, or hours;
    %local sleeptime rc;

    %if &unit eq %then unit=seconds;
    %if %upcase(&unit)=SECONDS %then %let sleeptime=&time;
    %else %if %upcase(&unit)=MINUTES %then
        %let sleeptime=%sysevalf(&time*60);
    %else %if %upcase(&unit)=HOURS %then
        %let sleeptime=%sysevalf(&time*60*60);

    %let rc = %sysfunc(sleep(&sleeptime));
%mend sleep;
```

The following macro call will suspend execution for 1 minute:

```
%sleep(time=1,unit=minutes)
```

%WAKEUPAT

This macro also uses the SLEEP function; however, rather than passing it the duration of inactivity, it accepts a SAS datetime value at which it will wake up. The parameter must be in a datetime18. form. The current time is subtracted from the wake-up time, which is converted to a SAS datetime value using the standard datetime constant conversion (dt). Notice the use of the nested calls to the %SYSEVALF function.

```
%macro wakeupat(time=);
    %local rc;
    %let rc = %sysfunc(sleep(
        %sysevalf(%sysevalf("&time"dt)-
            %sysfunc(datetime()))));
%mend wakeupat;
```

The following macro call causes SAS to “sleep” until the stated date and time:

```
%wakeupat(time=07nov2003:18:54:00)
```

Retrieve the last word in a list

%LISTLAST

The %LISTLAST macro returns the last word in the argument. The list is reversed making the last word the first. This is then retrieved using the %SCAN and the selected word is then put back into the original order.

```
%macro listlast(list);
    %sysfunc(reverse(%scan(%sysfunc(reverse(&list)),1,%str( ))))
%mend listlast;
```

When the second argument of the %SCAN function is negative, the %SCAN function works from right to left, rather than from left to right. %LISTLAST could therefore be specified as follows:

```
%macro listlast(list);
    %scan(&list,-1,%str( ))
%mend listlast;
```

MORE INFORMATION

The %SCAN function is discussed in more detail in Section 7.2.3.

SEE ALSO

Pete Lund (2003a) uses the %SCAN function with a negative word number to select the last word in a list.

Searching for words

%INDEXW

The macro function %INDEX searches for occurrences of the second argument. It does not, however, take notice of word boundaries. The %INDEXW macro function mimics the INDEXW function in the DATA step and returns the position of the word specified in the second argument.

```
%macro indexw(list, wrd);
    %sysfunc(indexw(&list, &wrd))
%mend indexw;
```

Assuming that the following list of variable names is passed to %INDEXW

```
%let mylist = aa bb cc a bb c;
%put the position is %indexw(&mylist,a);
```

The LOG will show

```
the position is 10
```

MORE INFORMATION

The %REVSCAN function in Section 7.6.3 returns the word number rather than the position of the word.

Converting number systems

%RGBHEX

Perry Watts

Although we typically use the decimal number system for most of our calculations, SAS and other languages often utilize others. Of these, the binary and hexadecimal system are very common. The following macro can be used to convert three RGB (Red/Green/Blue) color components ranging from 0 to 255 in decimal to a single hexadecimal character code needed for making color assignments in SAS:

```
%macro RGBHex(rr,gg,bb);
%sysfunc(compress(CX
    %sysfunc(putn(%sysfunc(round(&rr)),hex2.))
    %sysfunc(putn(%sysfunc(round(&gg)),hex2.))
    %sysfunc(putn(%sysfunc(round(&bb)),hex2.))))
%mend RGBHex;
```

This function uses %SYSFUNC along with the PUTN and ROUND functions to do the conversion.

MORE INFORMATION

The macro %PATTERN in Section 7.4.2 also uses %SYSFUNC with the PUTN function to perform the same conversion.

SEE ALSO

Additional conversions and other macros that work with color systems within SAS are presented by Perry Watts in 2003a and 2003b.

7.6.2 Functions for the DATA step

While not, strictly speaking, “macro functions,” the macros in this section mimic functions by building SAS DATA step code. These can be very useful when the macro is only going to be used within a DATA step. By using this approach you can pass back DATA step code; consequently, you are not required to use %SYSFUNC to handle all DATA step functions. While these macros are limited to the DATA step, they are still powerful tools.

Working with dates

%ADDYEARS

Pete Lund

Looking Glass Analytics

Have you ever needed to increment a date by a number of years? This macro returns a SAS date that is offset by a user-supplied number of years.

```
%macro AddYears(base,add);
    intnx('month',&base,&add*12) + (day(&base) - 1)
%mend;
```

In this macro the INTNX function is used to advance a date from &BASE by &ADD*12 months. Because the INTNX function always returns the start of an interval, the number of days within the current month is then added to the new calculated value. Then since this effectively adds the first of the month twice, one day is subtracted. In the following example some of the arithmetic operations are replaced by the use of an interval shift operator and the END option.

%EOW

Garth Helf

Hitachi Global Storage Technologies

This macro can be used to create a new variable with a week-ending date from a SAS date variable in a DATA step. The desired accounting week runs from Saturday to Friday, so it's very common to need to determine the accounting week for a given date. If the data set MAGDATA contained a date variable called MAGDATE, you could create a variable called MAGWEEK with this macro:

```
data magdata;
  set magdata;
  %eow(magdate, magweek)
run;
```

The %EOW macro is

```
/******
Macro EOW                      Garth Helf    24 October 2003
Create new variable in a DATA step containing week
ending date for a SAS date variable.  Argument we_day
is day the week ends:
    1=Saturday (default) to 7=Friday.
******/
%macro eow(dayvar ①, eowvar ②,
           eowfmt=date., we_day=1);
  &eowvar=intnx("week.&we_day" ③, &dayvar, 0,"end" ④);
  format &eowvar &eowfmt;
%mend eow;
```

- ① &DAYVAR holds the name of the incoming date variable.
- ② &EOWVAR becomes the new variable with the constant date. This date will always be a Friday and all the dates within a given Saturday-to-Friday range will have the same value for the variable named by &EOWVAR.
- ③ &WE_END indicates an interval shift operator. The interval 'week.2' will cause the week to start with a Monday rather than a Sunday.
- ④ The END alignment operator requests that INTNX return the date of the end of the week rather than the start of the week. When alignment and interval shift operators are both used, the shift operator is applied first. This means that a week starting on Monday ('week.2') will end on Sunday, and a Sunday date will be returned.

7.1.3 Macro functions with logic

Generally the macro functions that you write will require the use of logic, do-loop processing, and internal calculations. Remember to

- use only code that will be processed by the macro facility (no DATA steps)
- make all intermediate macro variables local
- return the value by “leaving it behind.”

Calculating permutations

%PERM

The PERM function returns the number of permutations within a group or set of objects. In the macro below the number of permutations is based on N things taken R at a time. If the second argument is not used, the function will return the factorial (see Section 7.6.1); however, if the comma separating the two arguments is included and no second argument is provided, then the function will return a missing value. The following macro, %PERM, uses %IF-%THEN/%ELSE processing to detect when the second parameter has not been supplied:

```
%macro perm(n,r);
  %if &r ne %then %sysfunc(perm(&n,&r));
  %else %sysfunc(perm(&n));
%mend perm;
```

The %PUT statement shown here

```
%put perm5 %perm(5);
```

returns the factorial of 5 ($5*4*3*2*1$)

```
perm5 120
```

A call with two arguments will calculate the number of permutations rather than the factorial. The following call to %PERM

```
%put perm5,2 %perm(5,2);
```

results in

```
perm5,2 20
```

The previous version of %PERM will generate errors if it is passed inappropriate arguments. If you are not going to be able to control for inappropriate values, the macro itself should be robust enough to do the checking for you. The following version performs these error checks.

```
%macro perm(n,r);
  /* Check for improper values;
  /*   N & R must be positive integers;
  /*   N must be > R when R is provided;
  /* Calculate the factorial when R is not provided;
  %if %sysevalf(%sysevalf(&n,int) ne &n) ❶ or
    %sysevalf(&n < 1) ❷ or
    (&r ne %str()) & %sysevalf(%sysevalf(&r,int) ne &r)) ❶ or
    (&r ne %str()) & %sysevalf(&r < 1)) ❷ or
```

```

        (&r ne %str() & %sysevalf(&r > &n)) ❸ %then .; ❹
    %else %if &r ne %then %sysfunc(perm(&n,&r));
    %else %sysfunc(perm(&n));
%mend perm;

```

Since we may have noninteger arguments %SYSEVALF is used for each of the comparisons. Failure to meet the criteria results in a missing value ❹.

- ❶ Check that the argument is an integer.
- ❷ Make sure that the integer is 1 or larger.
- ❸ N should be greater than or equal to R .

MORE INFORMATION

The various factorial functions fail if the number exceeds 170 or so. The permutations of larger numbers can be handled by the %SMARTPERM macro in Section 13.1.4.

Does the macro variable exist?

%SYMCHECK

Sometimes you need to know if a macro variable has been established. This macro makes this determination by checking the view SASHELP.VMACRO.

```

%macro symcheck(mvname);
    /* Determine if a specific macro variable
    /* has been defined.;
    %local yesno fetchrc dsnid rc mvname;
    %let yesno = NO;
    %let dsnid = %sysfunc(open(sashelp.vmacro ❶
                        {where=(name=%upcase("&mvname")) ❷},i));
    %let fetchrc = %sysfunc(fetch(&dsnid,noset)); ❸
    %if &fetchrc eq 0 ❹ %then %let yesno=YES;
    %let rc = %sysfunc(close(&dsnid)); ❺
    &yesno
%mend symcheck;

```

- ❶ The view SASHELP.VMACRO contains a list of all macro variables.
- ❷ The WHERE clause allows us to subset all possible macro variables. Notice the use of the quotes.
- ❸ The FETCH function attempts to read an observation from the view. We are not going to do anything with this observation (although we could); it is the success or failure of the read that we are interested in.
- ❹ If the FETCH is unable to return an observation &FETCHRC will take on a non-zero value. A successful fetch of the observation returns a value of 0 (zero). The fetch will be successful if the macro variable exists.
- ❺ The view is closed when we are done with it.

In the following call `&SILLY` exists on the symbol table:

```
%let silly=asdf;
%put check for SILLY %symcheck(silly);
```

As a result of the `%PUT` statement the LOG shows

```
check for SILLY YES
```

However, assuming `&SILLYX` does not exist, the following `%PUT` statement

```
%put check for SILLYX %symcheck(sillyx);
```

results in

```
check for SILLYX NO
```

The above version of `%SYMCHECK` passes back a value of YES or NO. The user must necessarily know this and must know how to check the returned value. That is, the user must always use upper case in the comparison. The following version builds the comparison into the result:

```
%macro symcheck(mvname);
  /* Determine if a specific macro variable
  /* has been defined.;
  %local fetchrc dsnid rc mvname;
  %let dsnid=%sysfunc(open(sashelp.vmacro
                        (where=(name=%upcase("&mvname"))),i));
  %let fetchrc = %sysfunc(fetch(&dsnid,noset));
  %let rc = %sysfunc(close(&dsnid));
  not(&fetchrc) ⑥
%mend symcheck;
```

A conditional check for the existence of `&SILLY` might now be written as

```
%if %symcheck(silly) %then
  %put Silly was found and contains &silly;
```

⑥ The return code is negated since a successful `FETCH` results in a 0 return code.

MORE INFORMATION

The `%SYMCHK` macro in Section 13.1.5 uses a very different approach to solve the same problem.

SEE ALSO

Troxell and Chen (2003) take a similar approach to determine if a macro variable exists.

Return the word number from a list of words

%REVSCAN

The `%SCAN` function returns the selected word from a list. The reverse-scan function (`%REVSCAN`) returns the word number given the word itself.

```

%macro revscan(list, word);
  %local wcnt wnum;
  %let wcnt=0;
  %let wnum=0;
  /* Determine the word number in a list of words;
  %do %while(%scan(&list,%eval(&wcnt+1),%str( )) ne %str( )); ❶
    %let wcnt = %eval(&wcnt+1);
    %if %upcase(%scan(&list,&wcnt,%str( )))=%upcase(&word) %then
      %let wnum=&wcnt;
  %end;
  &wnum ❷
%mend revscan;

```

The %DO %WHILE statement ❶ is used to step through the list (&LIST) using the %SCAN function. Once the word is found the word number is passed back ❷. This macro is demonstrated below for the following definition of &DOSECODE:

```

%let dosecode = a1 b1 b2 c;

/* qb2 is not found and returns a 0;
%put qb2 %revscan(&dosecode,qb2);

/* b2 is the third word in &DOSECODE and a 3 is returned;
%put b2 %revscan(&dosecode,b2);

```

It would be possible to simplify the %DO %WHILE statement by using the %LENGTH function. The %DO %WHILE statement becomes

```

%do %while(%length(%scan(&list,%eval(&wcnt+1),%str( ))));

```

Now a comparison to %STR() is no longer necessary as the %LENGTH function will return a 0 (false) when no word is returned by the %SCAN. If the word appears more than once %RESCAN returns the number of its last occurrence; Q7.6, question 6 at the end of this chapter, addresses this “feature.”

SEE ALSO

Mao (2003) uses the %LENGTH function with the %SCAN in a %DO %WHILE statement to determine when the last word has been found.

Repeat text

%REPEAT

Pete Lund

Looking Glass Analytics

This macro uses the REPEAT function to generate a series of repeated characters. Blanks can be specified as the repeat character by passing a null value as the first argument ❶. Unlike the DATA step REPEAT function, which places the original character and then repeats it *n* times, the %REPEAT macro places the character exactly the number of times indicated by the second argument ❷.


```
%macro repeat(char,times);
  %let char = %quote(&char);
  %if &char eq %then %let char = %str( ); ❶
  %sysfunc(repeat(&char,&times-1)❷)
%mend;
```

This function can also repeat groups of characters. The call %REPEAT(abc,3) produces

```
abcbabcbcb
```

SEE ALSO

The %REPEAT macro is used by Lund (2000a) to produce customized comments in the LOG.

Convert text to mixed case

%MIXEDCASE

We already have functions to convert text to upper case (%UPCASE) and lower case (%LOWCASE), but sometimes we would like to convert the first letter of each word to upper case and all other letters to lower case. The %MIXEDCASE macro function performs this conversion:

```
%macro mixedcase(string);
%local word cnt mixed;
%let string = %lowercase(&string); ❶
%let mixed =;
%let cnt = 1;
%do %while(%scan(&string,&cnt,%str( )) ne %str());
  %let word = %scan(&string,&cnt,%str( )); ❷
  %let mixed = ❸ &mixed ❹ %upcase(%substr(&word,1,1)); ❺
  %if %length(&word)> 1 %then
    %let mixed = &mixed%substr(&word,2); ❻
  %let cnt = %eval(&cnt + 1);
%end;
&mixed
%mend mixedcase;
```

- ❶ The entire string is converted to lower case.
- ❷ The next word is extracted from the incoming string.
- ❸ The %LOCAL macro variable &MIXED will contain the converted string.
- ❹ Each new converted word is appended in &MIXED to the words already converted. Note the space after the &MIXED. This space becomes the new word delimiter. Without it, the individual words would become a single word.
- ❺ The first letter of the word is converted to upper case and appended.
- ❻ The rest of the word is then also appended to &MIXED (which already has the first letter of the word).

The call to the macro is replaced by the converted string (&MIXED). This is shown in the following %PUT statement which includes a call to the %MIXEDCASE macro:

```
%put %mixedcase(ALFRED E. NEWMAN is a Magazine celebrity);
```

The %PUT statement writes the following to the LOG:

```
Alfred E. Newman Is A Magazine Celebrity
```

SEE ALSO

Lund (1998) contains a number of macro functions including one that converts text to mixed case.

Compare text strings of unequal length

%COLONCMPR

In the DATA step we can use the colon operator to modify how a comparison is to be made. This operator compares two strings using the length of the shortest. There is no equivalent operator in the macro language; however, the %COLONCMPR macro makes the same types of comparisons.

The parameters are the two text strings that are to be compared and the comparison operator to use. The macro determines the shorter of the two strings and uses the %QSUBSTR function to truncate the longer of the two. The comparison is made on the upper case values of the text strings.

```
%macro coloncmpr(left,op,right);
  %local width;

  %if &op = %str() %then %let op==; ❶

  %* determine shorter of left and right;
  %let width = %sysfunc(min(%length(&left),%length(&right))); ❷
  %upcase(%qsubstr(&left,1,&width) &op %qsubstr(&right,1,&width)) ❸
%mend coloncmpr;

%macro tryit;
  %let a = smith;
  %let b = s;
  %if %coloncmpr(&a,=,&b) ❹ %then %put comparison is true;
%mend tryit;

%tryit
```

- ❶ If &OP, the comparison operator, is not specified it is set to equal (=).
- ❷ The minimum length of the two sides is determined. This value is used to subset the longer string.
- ❸ Although %QSUBSTR is applied to both sides it will change only the longer of the two. The %UPCASE function is used to convert both sides to upper case.
- ❹ The macro is designed to be called within a %IF statement. For the above example, the %IF becomes

```
%if S = S %then %put comparison is true;
```

7.7 Chapter Summary

Macro functions create new text strings from existing text strings. There are four main categories of macro functions:

- text functions
- evaluation functions
- quoting functions.
- bridging functions.

Text functions perform actions such as searching for characters, determining the length of an argument, scanning an argument for specific words, creating a substring from a text string, and translating lowercase characters to uppercase characters.

Evaluation functions perform integer and noninteger numeric evaluation of arithmetic and logical expressions.

Quoting functions remove the meaning of specific characters either during macro compilation or at macro execution time.

The %SYSFUNC, %QSYSFUNC, and %SYSCALL functions provide a “bridge” between the macro language and the DATA step. They enable you to access most of the routines and functions that are part of the DATA step. These functions often provide the ability to eliminate DATA _NULL_ steps that are only used to format macro variables.

It is possible to write your own macro functions. The rules for the construction of macro functions are straightforward and easily applied.

7.8 Chapter Exercises

1. Check all answers that are TRUE. Macro character functions can be used to
 - A. perform character searches
 - B. determine the length of an argument
 - C. scan an argument for specific words
 - D. translate lowercase characters to uppercase characters
 - E. none of the above
 - F. all of the above.



Chapter 8

Using Macro References with the SAS Component Language (SCL)

8.1 The Problem Is...	200
8.2 Using Macro Variables	201
8.2.1 Defining macro variables	202
8.2.2 Macro variables in SUBMIT blocks	202
8.2.3 Using automatic macro variables	203
8.2.4 Passing macro values between SCL entries	204
8.2.5 Using &&VAR&I macro arrays in SCL programs	204
8.3 Calling Macros from within SCL Programs	205
8.3.1 Run-time macros	205
8.3.2 Compile-time macros	205
8.4 Chapter Summary	207

The macro facility is available to your SAS Component Language (SCL—formerly known as Screen Control Language), programs. The syntax and usage of macros, macro statements, and macro variables is essentially the same in SCL programs as in your other SAS programs. The primary difference, and the one that causes the most problems, relates to the fact that SCL programs are compiled. Because the compilation process resolves macro references, you may need to adjust your perception of the order and sequence of macro-related events.

During the compilation of the SCL program, all macro references are resolved to their current values. A macro call, for instance, will be executed during the compilation of the SCL program and **not** when the SCL program is executed.

This is a different way of thinking about macro references, and most of this chapter is designed to help you deal with these differences.

SEE ALSO

Beyond the Obvious with SAS Screen Control Language (Stanley, 1994) contains a summary of warnings and tips on the use of macros and macro variables in SCL programs (pp. 40–43).

Chapter 10, “Using Macro Variables,” in *SAS Screen Control Language: Reference, Version 6, Second Edition* (pp. 99–101) discusses macro variables, and the substitution of text in SUBMIT blocks is discussed on page 110.

Several examples of macro variables in SCL can be found in *SAS Screen Control Language: Usage, Version 6, First Edition*. Chapter 27, “Using Macro Variables,” (pp. 487–500) contains a number of examples of SCL programs that utilize macro variables. Chapter 28, “Submitting SAS and SQL Statements,” (pp. 501–530) covers various combinations of macro and SCL variable substitution in SUBMIT blocks.

Norton (1991) contrasts the use of the macro language with SCL and suggests that SCL be used as an alternative to the macro language in some situations.

Davis (1997) shows how you can use macros to provide consistency among SCL frame entries.

Ward (1999) uses macros with frame entries to document and manage programs.

8.1 The Problem Is . . .

SCL programs are compiled and external references such as macro variables and macro calls are resolved during compilation, not during execution. The writer of the following SCL section would like to open the data set named in the macro variable &DSN:

```
init:
  * open the data set &dsn;
  dsnid = open("&dsn",'i');
  if dsnid=0 then put "unable to open &dsn";
return;
```

This code will fail even to compile unless &DSN is currently defined. If &DSN is defined at compile time to be `clinics`, then the code that is compiled will be

```
init:
  * open the data set clinics;
  dsnid = open("clinics",'i');
  if dsnid=0 then put "unable to open clinics";
return;
```

Subsequent changes to the macro variable &DSN will have no effect on what data set will appear in the OPEN function.

A similar problem exists for macro calls inside of SCL programs. The following code is supposed to execute an error-check macro (%ERRCHK) when the file named by the SCL variable DSNAME cannot be opened:

```
init:
  * open the data set &dsn;
  dsnid = open(dsname, 'i');
  if dsnid=0 then %errchk(dsname);
return;
```

The macro %ERRCHK is supposed to write the name of the missing data set to the LOG. This macro has previously been defined as

```
%macro errchk(d);
  %put "data set not found &d";
%mend errchk;
```

Regardless of the name of the unknown data set, the LOG will contain the following:

```
"data set not found dsname"
```

The macro call is resolved when the SCL program containing the macro call is compiled. At compile time, DSNAME in the macro call is seen as text, not as an SCL variable. DSNAME is, therefore, passed unresolved to the macro. Here, &D will take on the value of dsname. This example highlights the major disadvantages of using macro calls in SCL programs:

- The macro must be defined before SCL compilation.
- SCL variable values cannot be passed into the macro through a macro call.
- Changes to the macro will not be reflected in the SCL program until the SCL program is recompiled.

Despite these caveats, there are very definite uses for macro variables and macro calls within SCL programs. The following sections cover these topics in more detail.

8.2 Using Macro Variables

Macro variables are not necessarily part of any particular application or data set, and they are not associated with a particular screen, method, or program. This makes them ideal for passing information through various portions of your program or application. Once defined and globalized, the values of the macro variables are available throughout SAS.

Macro variables are especially useful when passing information between entries in your application. Values that are set once and then repeated or held constant across screens or data sets are prime candidates for use with macro variables. Macro variables are often used to hold values for

- the name of a SAS data set to be opened or the file identifier of an open data set
- the external file name to be opened or its file identifier once it is opened

- constant text such as the current date
- values passed between applications
- values passed between programs within an application.

Because SCL programs are compiled, macro variables must be mentioned obliquely in SCL programs. Outside of the SUBMIT block, it is unlikely that you will use the ampersand much, if at all, in SCL programs.

8.2.1 Defining macro variables

You can assign values to macro variables in an SCL program in much the same way as you would in other SAS programs. However, you will probably depend more on the CALL SYMPUT routine and less on the %LET statement. Like %LET statements in non-SCL programs, the characters on the right of the equal sign are taken to be text. Because %LET is executed when the SCL program is compiled, SCL variables are unresolvable. The following %LET statement creates the macro variable DSN, which will contain the string `dsname`:

```
%let dsn = dsname;
```

Even if `DSNAME` is an SCL variable (screen or nonscreen), the value of `DSN` is the **text string** `dsname`.

The SYMPUT routine is used **at run time** to create macro variables with a value that is associated with an SCL variable. This routine is used the same in SCL as it is in a DATA step (see Section 6.1 for more information on the SYMPUT routine and its usage). If `DSNAME` is a character SCL variable that contains the string `class.clinics`, the following statement assigns to the macro variable `DSN` the value taken on by `DSNAME`, that is, `class.clinics`:

```
call symput('dsn', dsname);
```

As in the DATA step routine, the second argument of SYMPUT is expected to be a character string or the name of a character variable. Unlike the DATA step, however, SCL has a numeric counterpart, SYMPUTN. This routine is used to create macro variables from numeric SCL variables. In the following example, the numeric SCL variable `OBS` contains the observation number of interest:

```
call symputn('obsnum', obs);
```

Often the macro variables that are created within an application are globalized so that they will be available throughout the application. Unless the macro variables have been previously defined as local, the SCL SYMPUT and SYMPUTN functions always create globalized macro variables.

8.2.2 Macro variables in SUBMIT blocks

SUBMIT blocks are used in SCL programs to pass statements to SAS for execution. Statements that are contained in SUBMIT blocks are not SCL statements, but are standard SAS language statements. However, these statements can contain references to both SCL and macro variables.

Inside of a SUBMIT block, SCL variables are identified by preceding them with an ampersand. In the following section of SCL, a SUBMIT block is used to execute the PRINT procedure:

```
if modified(rpt) then submit;
  proc print data=&dsname(obs=10);
  title 'Data Listing for &dsname';
  run;
endsubmit;
```

A couple of things are worth noting in this code. At the execution of the SCL program and before the SUBMIT block is passed to the SAS processor for execution, the reference &DSNAME is first checked against the list of SCL variables in the SCL Data Vector (SDV) and resolved if found. If it is not found on the SDV, it is then passed unresolved (still containing the &) for execution, where it will be treated as a macro variable reference. Because of this behavior, when passing values into SUBMIT blocks it is generally not a good idea to use the same name for both macro variables and SCL variables.

You may also have noted in the previous example that the title is enclosed by single quotation marks. In SCL SUBMIT blocks, single quotes will **not** mask the meaning of the ampersand as they will in Base SAS language statements. However, if the &DSNAME is passed unresolved, in single quotes, to Base SAS for processing, the macro variable reference will remain quoted. In both the previous and in the following example, if DSNAME is not an SCL variable, it will remain unresolved in the title.

If you do need to specify a macro variable in a SUBMIT block and an SCL variable exists with the same name, you can use a double ampersand to prevent its resolution as an SCL variable. In the following example, the &&VARLST will not resolve to the SCL variable VARLST even if it exists on the SDV:

```
if modified(rpt) then submit;
  proc print data=&dsname(obs=10);
  var &&varlst;
  title 'Data Listing for &dsname';
  run;
endsubmit;
```

8.2.3 Using automatic macro variables

The automatic macro variables that are discussed in Section 2.6 are all available within SCL programs. As was noted in Section 8.2.1, however, you need to be careful how you use them. The following SCL code attempts to open the most recently modified data set:

```
dsname = "&syslast";
dsid = open(dsname);
```

The code will not do what the programmer wants because &SYSLAST will be resolved when the SCL program is compiled. Upon execution the program will always try to open the same data set regardless of the value of &SYSLAST at the time of execution. The problem is solved by using the SYMGET function. In the following revised SCL code, the value of &SYSLAST is not retrieved until the SCL program is executed:

```
dsname = symget('syslast');
dsid = open(dsname);
```


When you want to create a numeric SCL variable from a macro variable, you should use the SYMGETN function. SYMGETN returns a numeric value rather than the character value that is returned by SYMGET.

8.2.4 Passing macro values between SCL entries

Passing macro variable values between SCL entries is very straightforward. Macro variables are defined using the SYMPUT and SYMPUTN routines, and these macro variables are then available throughout the application. A subsequent SCL entry (or even code within a SUBMIT block) can retrieve the values by using SYMGET and SYMGETN.

The following INIT section both retrieves macro variables, and creates them for later use:

```
INIT:
  * Specify a macro var used for SCL in edit screens;
  call symput('scrntype','DE'); ❶

  * Create a libref for the log used
  * by this Data Entry userid;
  userid = symget('userid'); ❷
  tst = symget('tst'); ❸
  path = compress('h:\studyx\phase2\'
    ||tst||'datprep\d_entry\'
    ||userid);
  call libname('delog',path);

  control enter;
  cursor subject;
return;
```

❶ The macro variable SCRNTYPE is initialized to DE. The macro variables USERID ❷ and TST ❸ (both of which must have been created earlier in the application) are retrieved and placed in SCL variables of the same name. Remember, using the same name for macro and SCL variables is acceptable and causes problems only when used carelessly in SUBMIT blocks.

Macro variables often may not be the best method for passing values between entries within an application or even between applications. Built into SAS Component Language is the concept of an SCL LIST. Analogous to an ARRAY list, entries can be loaded from files, saved to disc, and passed from one entry to another. Like macro variables, LISTS can be global or local. But because LISTS are designed to be an integral part of SCL (macro variables just coexist with SCL), they work more smoothly and have additional support functions. It is likely that SCL LIST functions will be quicker than SYMGET and SYMPUT.

8.2.5 Using &&VAR&I macro arrays in SCL programs

Since it is not possible to use the &&VAR&I macro variable form within an SCL program, these macro variables are again accessed by using the SYMGET and SYMGETN functions to create SCL variables.

Like in the macro language, when you want to step through a series of macro variables that have been created with a subscript, a DO loop is used. This time, of course, it will be an SCL DO loop rather than a %DO loop, and the resulting SCL variable (I) will be used as the index to identify the specific macro variable.

In the following example a series of data set names have been stored in the macro variables &LIVEDB1, &LIVEDB2, ..., and we would now like to step through the list of data sets from within an SCL program.

```
* Step through the list of data sets;
cnt = symgetn('livecnt');
do i=1 to cnt; ❶
  ii = left(put(i,3.)); ❷
  * Get the data set name and open it.;
  dsn = 'datamgt.'||left(symget('livedb'||ii ❸)); ❹
  dsid = open(dsn);
```

- ❶ Specify the SCL DO loop (rather than a %DO loop).
- ❷ The SCL loop creates a numeric index variable, which is converted to character.
- ❸ This index is then appended to the root name of the macro variable series.
- ❹ The concatenated value is retrieved using SYMGET or SYMGETN.

8.3 Calling Macros from within SCL Programs

The brief example using %ERRCHK in Section 8.1 illustrates the problem associated with calling macros from within SCL programs. Remember, macros are executed when the SCL is compiled. This means that when the macro changes, programs that use that macro will need to be recompiled. Despite this problem macro calls can have a definite place in SCL programs.

Macros called in SCL programs fall into two classes, and these are determined by when the macro is to be executed. Macros in SCL programs can be executed either when the SCL program is compiled (compile-time) or when it is executed (run-time).

8.3.1 Run-time macros

Run-time macros are macros that execute when the SCL program executes. Unlike macros in the base language, which are all run-time, this concept has very little meaning in SCL programs. The exception is found in SUBMIT blocks. Because SAS does not resolve macro references in SUBMIT blocks until the block itself is executed, you can safely call macros here. Because the block of code is essentially set aside, the called macro does not need to exist until the SUBMIT block is actually executed.

Run-time macros avoid the SCL compilation issues that are noted in Section 8.1 and behave as other base system macros behave.

8.3.2 Compile-time macros

Compile-time macros are most often used to write SCL code. These macros will execute when the SCL program is compiled—long before values for the SCL variables are available. If the macro contains reusable code but is not actually generating SCL, as in the example shown later in this section, consider using a METHOD instead of a macro. There is an informative discussion

on the advantages and disadvantages of METHODS and compile-time macros in *Beyond the Obvious with SAS Screen Control Language* by Don Stanley (1994, pp. 42–43).

The example shown in this section illustrates the use of a macro to generate code. A series of over 20 FSEDIT screens were to have similar (but not quite the same) SCL that was used to initialize protected variables. Rather than develop a series of parallel programs that would be difficult to maintain, the entire SCL program for each screen was placed within a single generalized macro. The source screen for each FSEDIT consisted only of the call to the macro %DATASTMP:

```
%datastmp(subject,ptid)
```

The macro arguments are names of SCL variables that will apply to a specific data set and screen. Remember, the **names** of the variables are being passed to the macro not the variable values, which are not yet available.

```
%macro datastmp(var1,var2,var3,var4);

* determine the number of vars;
%do i = 1 %to 4;
    %if &&var&i ne %then %let varcnt = &i;
%end;

fseinit:
    scrntype=symget('scrntype');
    if scrntype in ('CLN', 'PED') then do;
        control enter;

        ...ordinary SCL not shown...
    return;

init:
    if scrntype='DE' or word(1)='ADD' then do;
        %do i = 1 %to &varcnt;
            unprotect &&var&i;
            &&var&i = symget("&&var&I");
            protect &&var&i;
        %end;
    end;
return;

...ordinary SCL not shown...

%mend datastmp;
```

When the compile process starts to compile the SCL that contains the macro call, the macro executes. First the macro generates SCL code, and then that code is compiled. The INIT section that follows is generated when the SCL that contains the call to %DATASTMP (as previously shown) is compiled (after the macro executes):

```
init:
  if scrntype='DE' or word(1)='ADD' then do;
    unprotect subject; ❶
    subject = symget("subject"); ❷
    protect subject; ❸
    unprotect ptid;
    ptid = symget("ptid");
    protect ptid;
  end;
return;
```

The %DATASTMP macro has been used to generate SCL code that has SCL variables with the same name as the macro variables. When this INIT section executes, the ❶ screen variable SUBJECT will be unprotected, assigned a value ❷ based on the value of the macro variable of the same name, and then ❸ reprotected.

As a general rule, you should have a compelling reason to generate SCL code this way, especially for FSEDIT screens. The problem is that the SCL source code is not integral to the SCREEN. If you lose the uncompiled macro, you will be unable to regenerate the SCL code. From a practical point of view, the process itself can be cumbersome. If you need to change the code you will need to

- make the change in the macro source.
- if running interactively, make sure that the current version of WORK.SASMACR contains the correct version. Either delete the catalog entry (this should work, but it is not a guaranteed approach), or better yet recompile the macro.
- start FSEDIT and use modify.
- compile the SCL. (For compile errors, go back to the start of this list.)

MORE INFORMATION

The previous example was taken from a program that used a macro %DO loop to simplify the process that included 20 screens. This loop is shown in Section 9.3.2.

SEE ALSO

Bryher (1997b) uses a macro to build SCL code.

8.4 Chapter Summary

The full power of the macro facility is available in SAS Component Language (SCL) programs. However, because SCL programs are compiled there are special considerations.

Macro variables are usually referenced using the SYMGET function and the SYMPUT routine. A macro variable will rarely be called using the ampersand.

Globalized macro variables have values that are available throughout an application. This makes an easy way to transfer information from one SCL program to another.

Macros called from within an SCL program are called and executed during the compilation of the SCL program.

Macro variables and macro calls in SUBMIT blocks behave as they do in non-SCL programs.

Examples of other macros and utilities are very common in the SAS literature. Proceedings from SAS Users Group International (SUGI) conferences are especially rich in these kinds of programs. The proceedings from each SUGI and a number of regional conferences have been cited in the bibliography. Most of these papers are available in electronic form, and they provide an excellent source of examples and explanations.

The book *SAS System for Statistical Graphics, First Edition* by Michael Friendly (1991) has a very nice collection of macros that are useful in both statistical and graphical applications.

Multiple-Plot Displays: Simplified with Macros by Perry Watts (2002) is a collection of macros that are designed to work in the SAS/GRAPH environment. She presents additional SAS/GRAPH macros in Watts (2003a and 2003b).

SAS maintains a Frequently Asked Questions Web page for the macro language. This page is currently located at

<http://support.sas.com/techsup/faq/macro.html>



Part 3

Advanced Macro Topics, Utilities, and Examples

Chapter 9	Writing Dynamic Code	211
Chapter 10	Controlling Your Environment	239
Chapter 11	Working with SAS Data Sets	279
Chapter 12	Building and Using Macro Libraries	327
Chapter 13	Miscellaneous Macro Topics	347

The macros and programs that are contained in Part III were collected from a number of sources. Although the programs have been checked when possible, they cannot be warranted to be error free, nor should you expect them to do just what you want them to do on your system. You should use these macros as examples of coding possibilities that you can adapt to your own situation.

A number of the macros that follow were written by SAS programmers other than the author. These macros are noted with the name and occasionally (with their approval) additional contact information for the macro's author. Sometimes macros such as these are passed from programmer to programmer and it is hard to identify the original author. In these cases, I have included the name of the most recent contributor. In order to control content and to stress certain points, some of the macros have been slightly altered from their original form.

As you look over these macros please remember that programming, as in many creative endeavors, is very individualistic. Many of the programs included here may not reflect your style, and some are more efficient than others. They all have been included to demonstrate both techniques and style. In each case, the authors should be complimented on their contributions to the SAS community.



Chapter 9

Writing Dynamic Code

- 9.1 Elements of Dynamic Programs 212
 - 9.1.1 Logical branches 213
 - 9.1.2 Iterative step execution 213
 - 9.1.3 Building statements 214
- 9.2 Writing Applications without Hard-coded Data Dependencies 216
 - 9.2.1 Generalized and controlled repeatability 217
 - 9.2.2 Setting up control files 218
 - 9.2.3 Building macro variables from control files 220
- 9.3 Using &&VAR&I Constructs as Macro Arrays 221
 - 9.3.1 Resolving &&VAR&I 222
 - 9.3.2 Stepping through a list of data sets 222
 - 9.3.3 Checking for observations with duplicate KEY values 222
 - 9.3.4 Coordinating two macro variable lists 224
- 9.4 Building SAS Statements Dynamically 227
 - 9.4.1 Using the DATA _NULL_ and %INCLUDE 228
 - 9.4.2 Using the CALL EXECUTE routine 231
 - 9.4.3 Using macro lists rather than macro arrays 233
- 9.5 Using SASHELP views 234
 - 9.5.1 Overview of the SASHELP views 235
 - 9.5.2 Using a view 236
- 9.6 Chapter Summary 237

A dynamic program is one that by necessity is not completely defined prior to execution. Many, if not most, SAS programs are static and the programmer determines through the use of DATA step and PROC step statements the order and logic of execution. In static programs, when the programmer *knows* about data exceptions and special cases, he or she must *hard code* logic to handle them. Dynamic programs, on the other hand, use the data itself to determine the path and logic of execution. The programmer who writes dynamic programs has the ability to create generalized programs that will execute correctly on a wider variety of data.

Because dynamic programming techniques are such an integral part of the macro language, many of the examples in other sections of this book also demonstrate the topics that are outlined in the following sections of this chapter. The topic of dynamic programming is first introduced in Section 5.2.2, where several basic issues relating to the topic are covered.

Three primary areas of coding that benefit from dynamic techniques include

- logical branches that conditionally execute sections of code
- iterative loops that execute sections of code multiple times
- the construction or writing of SAS statements at macro execution.

In dynamic programming, the control and flow of the program is driven by the data or some other external source of control information. One of the keys to this process is the use of the SYMPUT routine, which is used to store DATA step values in macro variables. Once stored in macro variables, these values can be used by the macro language in subsequent steps. Section 6.1 defines the use of the SYMPUT routine and Section 6.2 has examples that build macro variables based on the values in the data. When macro variables are used to form what is essentially a macro variable array, they usually take the form of &&VAR&i. The use of this type of macro variable is introduced in Sections 6.2 and 6.3.

Data set values may also be assigned to macro variables by using the SQL procedure. A detailed discussion of the use of SQL to build macro variables is located in Section 6.5 with additional examples in a number of sections including Sections 2.7, 11.2.1, 11.4.1, 11.4.2, 11.4.9, and 11.5.4.

SEE ALSO

Several examples of dynamic programming can be found in Blood (1992), Carpenter and Callahan (1988), and Carpenter (1997). Carpenter and Smith (2000c) discuss the design aspects of a dynamic application.

9.1 Elements of Dynamic Programs

As you become more proficient at writing dynamic programs, you will discover that there are certain patterns in your code. There are a number of “elements” that most dynamic programs will have in common. This section introduces some of those elements within the context of dynamic programming techniques.

9.1.1 Logical branches

Logical branching based on the value of a macro variable is probably the most common form of dynamic coding. Available only within a macro, the use of the macro %DO block is often combined with the %IF-%THEN/%ELSE statements. The examples in Section 5.3.1 demonstrate this type of branch.

In this example, the number of observations in a data set has been stored in the macro variable &NOBS. This value is then used to determine if the data is to be summarized or printed.

```
%macro showdat;
...code not shown...
%if &nobs ge 10 %then %do;
  proc means data=statdata mean n stderr;
  var ht wt;
  title "Analysis Data - &nobs Observations";
  run;
%end;
%else %if &nobs gt 0 %then %do;
  proc print data=statdata;
  var subject ht wt;
  title "Data NOT Summarized";
  run;
%end;
%else %put Data Set STATDATA is empty; fix
...code not shown...
%mend showdat;
```

Although this author does not generally recommend the technique, branching can also include the use of the %GOTO statement, as is shown in Section 5.4.2.

SEE ALSO

Yindra (1997) has an example of a macro that also branches based on the number of observations in a data set.

9.1.2 Iterative step execution

You can use the iterative %DO loop to cause the execution sequence to pass through PROC and DATA steps multiple times. In the following example, the data set CLINICS has a variable REGION that you would like to use to summarize the data. Using the BY statement in each of the PROC steps will order the output so that the summary for all regions is followed by the plots for all regions. However, you would like to have the output collated. For example, you might want to group together all of the output for each value of REGION. This, of course, is not how SAS generates output. In order to regroup the output (without shuffling papers at the end), you need to pass through each of the procedures once for each value of the BY statement.

Because you want the output grouped for each BY value across procedures, you need a way to loop through the procedures for each BY value. The resulting output will have the data summary and plot for REGION='1' followed by the summary and plot for REGION='2', and so on.

```
%macro onereg;
proc sort data=sasclass.clinics out=clinics;
  by region;
run;
data _null_;
```

ii is a counter that records the total number of the regions

```

set clinics;
by region;
if first.region then do;
  i+1;
  ii=left(put(i,best12.));
  call symput('reg'||ii,region); ❶
  call symput('total',ii); ❷
end;
run;

%do i=1 %to &total; ❸
  proc means data=clinics mean n stderr;
    where region="&&reg&i"; ❹
  var ht wt;
  title1 "summary for height and weight";
  title2 "region &&reg&i";
  run;
  proc gplot data=clinics;
    where region="&&reg&i"; ❹
  plot ht * wt;
  title1 "plot of height and weight";
  title2 "region &&reg&i";
  run;
%end;
%mend onereg;

```

- ❶ A macro variable for each unique value of REGION is created in the form of &®&I (that is, ®1, ®2, ...).
- ❷ The total number of regions is saved in &TOTAL.
- ❸ A macro %DO loop is set up and includes the procedures of interest.
- ❹ The WHERE statement is used with each procedure to exclude all regions, except the one of interest (&®&I).

This example is somewhat inefficient in that it reads the data set CLINICS several times. The solutions to Questions 2 and 3 in the Chapter 6 Exercises (that is, Questions 6.2 and 6.3) address a similar problem more efficiently. The last example in Section 9.1.3 is similar to those solutions.

SEE ALSO

Carpenter and Callahan (1988) also discuss two similar macros, %BREAKUP and %SPLITUP, that you can use to control program flow and output organization. Another example of this programming technique can be found in Wobus and Gober (1997).

9.1.3 Building statements

You can use the macro language in a number of ways to build individual SAS statements. Macro logic based on the %IF statement is often combined with the various forms of the %DO loop to create full statements or just portions of the statements. Examples in Sections 5.2.2, 5.3.1, and 5.3.2 introduce the techniques, while Section 9.5.2 directly addresses issues that are related to building statements.

In the following example, the SET statement is defined conditionally:

```
data new;
  %if &cond=YES %then %str(set cond);
  %else %str(set general);
run;
```

It is not necessary to create the entire statement within the macro text. In the previous example, the SET statement could also have been written without using the %STR function:

```
data new;
  set
    %if &cond=YES %then cond;
    %else general;
  ;
run;
```

As is shown in Section 9.5.2 and a number of examples in the following chapters, you can also use the %DO loop very effectively to build statements.

The DATA step shown in the following example (which is a portion of a macro) is a part of the solution to Question 6.3. Assume that in a previous step a series of macro variables (®1, ®2, and so on) had been created (see the example in Section 9.1.2) to hold the unique values of the character variable REGION (the values in this case are '1', '2', '3', and so on). This DATA step will be used to create a separate data set for each unique value of REGION. For example, all observations with REGION='1' will be written to the data set R1.

```
data %do i = 1 %to &total; r&&reg&i %end;; ❶
  set clinics;
  %* Build the &total output statements;
  %do i = 1 %to &total; ❷
    %if &I>1 %then else; ❸
    if region="&&reg&I" then output r&&reg&i;
  %end;
run;
```

- ❶ The %DO loop builds the DATA statement by naming all of the data sets to be created. The resulting statement will look something like this:

```
data r1 r2 r3;
```

- ❷ This loop creates the statements used to OUTPUT each observation to the appropriate data set and will be executed once for each unique value of REGION.
- ❸ For three regions, the resolved statements might be

```
if region="1" then output r1;
else
if region="2" then output r2;
else
if region="3" then output r3;
```

SEE ALSO

You can find an additional example of a dynamically built SET statement in Tassoni, Chen, and Chu (1997).

Smith (1997) includes several examples of dynamic code building. These include the use of %DO loops in PROC FORMATS and in AXIS statements.

9.2 Writing Applications without Hard-coded Data Dependencies

An application generally consists of a series of interrelated programs. They may or may not include a user interface, but almost certainly they are going to contain macro language elements. Since a well-written set of programs can have broader application than originally intended, by writing the programs to be as dynamic as possible (that is, by removing as many hard-coded data dependencies as is possible) the programs become more reusable.

A dynamic program should be able to gather as much of the information that it needs about the data on its own—without programmer intervention. Assume that you need to perform a PROC MEANS on all numeric variables that start with the letters WT. (There may be nonnumeric variables in the data set that have names that also start with WT, so we can't simply use VAR WT;.) You are given only the name of the data set and the variable prefix. A dynamic macro will be able to determine the names of the available numeric variables that meet the naming requirements and will then build the appropriate VAR statement. If the list of analysis variables changes in the future, a macro written using dynamic programming techniques will adjust without any recoding.

Let's say that you have written a series of interesting and perhaps even complex SAS programs that perform a variety of data entry operations, data checks, exception reporting, statistical analyses, and summary reporting. Since the next study or project is somewhat similar to the one you just completed, you may be planning to build another set of programs based on (cannibalized from) the ones that you just finished.

Wouldn't you rather build the programs once? If the macros are able to somehow gather the information that they need to execute successfully whenever they are run, and not only when you initially write the macros, then you will not need to modify the programs for each task or project. The macros are dynamic and can adjust to the task.

There are a number of ways that macros can search out and find the information they need at run time.

- The information may already exist in the analysis data itself (see %PLOTIT in Section 6.2).
- A number of views, which are predefined by SAS, can provide information on the SAS environment as well as the operating environment. In the SASHELP library look for views that start with the letter "V" (see Section 9.5). Those using PROC SQL have access to a number of tables in the DICTIONARY library (see Section 6.5.4).
- Quite a few DATA step functions can be used to gather information about data tables, environmental settings, and system options.
- Control data tables that contain project, data set, or variable-specific information can be designed by the programmer (see Section 9.22).

These data sets and information sources are in turn used to create a series of SAS macro variables that are available to the programs and macros of the application. All project, data set, and variable-specific information is stored in the macro variables and hence never in the programs themselves. Once implemented all the programs in your application become data independent.

Most dynamic programs will use one or more of these information sources. For complex dynamic applications it is often necessary to fully describe all the data tables, their names and attributes, as well as the variables in each of those data tables, including their names and attributes. Usually the only practical way of storing and maintaining this information is through the use of one or more data tables. These tables are usually referred to as control tables, and since their use is a bit involved they are discussed or used in most of the following examples in this chapter.

The great advantage of writing dynamic programs is that it is possible to write them without hard-coding things like data set and variable names. This means that when data set attributes change, you can make the corresponding changes in your control file, not your program. Let “them” redefine the project. Your code is ready.

SEE ALSO

Jim Sattler (2003) discusses the elements of programs that he calls “Data Driven.” Molter, Millard, and Paciocco (2003) use metadata to construct SQL queries, and Luo and Luo (2003) discuss some macros that can be used in metadata construction.

Shen (2003) uses Excel spreadsheets to build the metadata (see Section 9.2.2) used to drive the dynamic programs.

9.2.1 Generalized and controlled repeatability

At the completion of a successful project everyone should be happy with what you have done as a SAS programmer. If everything went fairly well, then you are pleased, the boss is pleased, and the client is pleased. Of course your job is not really completed. The problem is that you now need to document what you did, create the data dictionaries (even though ideally the dictionaries and the documentation should come first), and prepare for that next study or project.

Often this means that you will need to reconstruct what you have done and then modify the existing programs for the upcoming project by changing data set names, variable names, and variable attributes. Once modified these programs will of course then have to go through the validation process again as well.

Wouldn't you rather build the programs so that they will be ready to go for the next study regardless of the number of data sets, names of variables, and error-check specifications? Wouldn't you rather just validate your programs once and not over and over again? Imagine the savings in change-control management alone with only one version of each program.

Many studies, including most clinical trials, are very data intensive. Very often there is a large number of distinct SAS data sets, each with a diverse suite of variables, and the data sets themselves may range from a few to many observations. Just keeping track of these data sets becomes a chore for the database manager.

Management issues become even more intense when an application is developed that must operate against these data sets. As part of the data management, analysis, and reporting process, numerous SAS programs are usually written to support the application.

Very often names of data sets, variables, and other data set and project-specific information is embedded within these programs. This makes it difficult, time-consuming, and expensive to modify the application for another similar project. A dynamic and automated application will overcome these limitations by avoiding any project, data set, or variable dependencies.

How then do we create an application that will work for any number of data sets, with any number of observations, and with any combination of variables? How can the application be written so that it requires little or no recoding when being ported from one project to the next?

Fortunately, although there are major differences between studies, many of the tasks are similar. Most studies require data entry and data validation. Many of the exception and adverse-event captures and reports are similar. Since these major events are similar across projects, it ought to be possible to generalize our programs so that they need not be modified for each project.

Indeed, it is possible to build the SAS programs so that they are general enough to work for each of your studies. The answer lies in the creation of data dictionaries that can be used as control files in order to build a series of macro variables, which are in turn used dynamically by the application. The key is to build a structure into your programs that is based on those things that are common to all projects. Some of these commonalities include project identification, library and folder relationships, data sets with specifiable characteristics, variables within data sets with specifiable characteristics, and variable-specific value constraints.

SEE ALSO

In two papers presented by Carpenter and Smith (2000c and 2001b), a number of considerations regarding the building and maintaining of dynamic programs are discussed.

9.2.2 Setting up control files

The data dictionaries or control files become both the starting point and heart of the control process for a dynamic application. This means that relative to your programs the addition of a new data set into the study or a change in a data set's variables may be as simple as changing one control file. Done properly, all of the programs that depend on these control files will require **no** modification when the control files are changed. This also implies that implementing a new study is as simple as building a new set of data dictionaries, and, of course, this is something that you should be doing first anyway.

The control files can be SAS data sets, Excel tables, or even flat files. Ultimately it is usually easiest to build and maintain these control files as SAS data sets through the use of a SAS/AF, FSEDIT, or FRAME application, although with the use of PROC IMPORT a large number of other possibilities exist. The kinds and types of information contained within these control files will depend, of course, on how you use them.

The example application described in the examples in Sections 9.2.2 through 9.4.2 performs a variety of tasks including double data entry, data set comparisons, data validation, and data integrity checks. Three separate control files were developed to provide the type of data-dependent information necessary to enable the macros of the application to operate successfully.

The three control data sets shown below are the primary ones of the several that you may need (for demonstration purposes each has been simplified). They are the following:

DBDIR	Data set definitions
VRDIR	Variables within data sets
FLDDIR	Data field constraints.

Each column or variable in these data sets will become a vector of macro variables with one macro variable for each data value in the data set.

DBDIR

This data set will contain one observation for each data set in the project. In addition to the data set name, variables often include data sheet page number, key variables, data set label, other data-set-specific information such as might be required by the various analysis programs.

OBS	DSN	PAGE	KEYVAR
1	DEMOG	1	SUBJECT
2	MEDHIS	2	SUBJECT MEDHISNO SEQNO
3	PHYSEXAM	3	SUBJECT VISIT REPEATN SEQNO
4	VITALS	4	SUBJECT VISIT SEQNO REPEATN

VRDIR

This data set contains one observation for each variable in each data set. Variables that are in all data sets within the project have ALL as the data set name so that they do not need to be constantly repeated. Although not shown in this example, many other variable-specific items such as formats, labels, and variable length can also be included in this control file.

OBS	DSN	VAR	VARTYPE	LABEL
1	ALL	SUBJECT	\$8	Patient number
2	ALL	PTINIT	\$8	Patient initials
3	DEMOG	DOB	8	Date of birth
4	DEMOG	SEX	\$8	Sex
5	MEDHIS	MEDHISNO	8	Medical History Number
6	MEDHIS	MHDT	8	Date of medical history
7	PHYSEXAM	PHDT	8	Physical exam. date
8	PHYSEXAM	WT	\$8	Weight

The data sets DBDIR and VRDIR are used to build the data dictionary and to document the data sets and the variables that they contain.

FLDDIR

The data set FLDDIR identifies data constraints for each data entry field or variable. These constraints can be used to build data exception and error-trapping reports.

OBS	DSN	VAR	CHKTYPE	CHKTEXT
1	DEMOG	CENTRE	notmiss	
2	DEMOG	RACE	list	('1','2','3')
3	MEDHIS	MHDT	format	date7.

Several different types of checks are possible. Shown here are

- notmiss the variable may not contain missing values
- list the value must be in the list of values in CHKTEXT
- format the formatted value of the variable (using the format in CHKTEXT) must not be missing. User-defined formats are permitted.

9.2.3 Building macro variables from control files

Each of the control files is used to create a series of macro variables. The observations are counted and the observation number becomes a part of the macro variable name. This results in names such as &LIVEDB1, &LIVEDB2, &LIVEDB3, and so on.

Although the macro language does not have an array statement *per se* this series of macro variables can be used as a vector of values. This vector effectively becomes a macro array.

SEE ALSO

Yao (1997) discusses a macro that derives control values from a flat file.

Building the list of data sets

In the following example, the variable *i* ❶ counts the observation number. It is then converted to a left-justified character variable (*ii*) ❷, which is appended to the name of the macro variable ❸. The macro variables are created using the CALL SYMPUT routine ❹.

```
data _null_;
  set datamgt.dbdir end=eof;
  i+1; ❶
  ii=left(put(i,3.)); ❷
  call symput('livedb'|| ❸ ii,trim(dsn)); ❹
  call symput('keys'||ii,keyvar); ❹
  if eof then call symput('livecnt',ii); ❺
run;
```

Using this DATA step and the DBDIR data set from above, the second observation yields the macro variable &LIVEDB2 as MEDHIS and the associated key fields are stored in &KEYS2 as SUBJECT MEDHISNO SEQNO. Since there is one observation for each data set in the project, the total number of observations in the data set DBDIR will be the same as the number of data sets in the project and this number is stored in &LIVECNT ❺.

Building the lists of variable information

The information for each individual variable is also read in a similar manner. The data set name is included ❶ as is the name of each variable ❷. Other information that can be transferred into macro variables includes the DATA variable's label ❸, format, length, and variable type ❹. The total number of variables across data sets is also saved in &VARCNT ❺.

```

data _null_;
  set datamgt.vrdir end=eof;
  i+1;
  ii=left(put(i,3.));
  call symput('vdsn'||ii,dsn); ❶
  call symput('var'||ii,var); ❷
  call symput('label'||ii,label); ❸
  call symput('vtyp'||ii,vartype); ❹
  if eof then call symput('varcnt',ii); ❺
run;

```

In Section 6.5.3 an SQL step was used to build a list of macro variables. An SQL step could also be used here to build the lists of macro variables created in the DATA_NULL_ step above. The following PROC SQL builds the same macro variables as were created in the previous DATA_NULL_:

```

* Create a list of data set attributes;
proc sql noprint;
  select dsn,var,label,vartype
    into :vdsn1-vdsn999, ❶
         :var1-var999, ❷
         :label1-label999, ❸
         :vtyp1-vtyp999 ❹
    from datamgt.vrdir;
quit;
%let varcnt = &sqllobs; ❺

```

Building the field check list

A similar data step is applied to the data set FLDDIR. Since these macro variables are used less frequently, they are loaded into the symbol table only when they are needed. The step that reads the FLDDIR data set is shown below in the section that also discusses how these macro variables are utilized.

9.3 Using &&VAR&I Constructs as Macro Arrays

Although the SAS macro language does not formally support the concept of macro arrays, it is possible to create what is effectively an array by using macro variables of the form of &&VAR&I. Within a macro, a macro %DO loop can be used to step through the list of macro variables. Since the index for a %DO loop is also a macro variable, it can be used as the array subscript by making it a part of the name of the macro variable that is to be resolved.

As was shown in the previous sections, the macro variable name is formed by concatenating a number onto the name portion. In the following SYMPUT the value contained in the character variable ii is concatenated onto 'LIVEDB'. The subsequent macro variable will contain the value stored in the data step variable DSN.

```

call symput('livedb'||ii,trim(dsn));

```

SEE ALSO

Fehd (1997a, 1997b, and 1997c) and Blood (1992) include macros and discussions on the use of macro arrays in dynamic coding situations.

9.3.1 Resolving &&VAR&i

Crucial to the use of macro variables of the form &&VAR&i is an understanding of how these macro variables are resolved and used. Because there are adjacent ampersands (&&), the resolution process is necessarily a two-step process.

The following macro %DO loop will execute &LIVECNT times:

```
%do i = 1 %to &livecnt;
    &&livedb&i
%end;
```

When &LIVECNT is 4, and the data set DBDIR was used as in Section 9.2.3, the loop creates the macro variable list of

```
&LIVEDB1 &LIVEDB2 &LIVEDB3 &LIVEDB4
```

which further resolves to

```
DEMOG MEDHIS PHYSEXAM VITALS
```

The process of the resolution of macro variables can be viewed in the LOG by using the SYMBOLGEN system option.

9.3.2 Stepping through a list of data sets

We now have the ability to step through a list of data sets. The following macro loops through each data set in the study. The PROC FSEDIT is executed for each data set ❶, using the appropriate customized SCREEN (which also named using the associated data set name) ❷.

```
%do jj = 1 %to &livecnt;
    proc fsedit data=livedb.&&livedb&jj ❶
        screen=appls.descrn.&&livedb&jj...screen; ❷
    run;
%end;
```

Notice the use of the three decimal points ❷. More than one is required as the SAS interpreter will see them as part of the macro variable name when they immediately follow a macro variable.

9.3.3 Checking for observations with duplicate KEY values

In the macro %CHKDUP, which follows, each data set is to be checked for observations with duplicate values of the KEY variables. In other words, the BY variables do not form a unique key. Again we step through the data sets, this time using the variable &JJ as the subscript ❶. For a given data set (&&LIVEDB&JJ), the BY variable list will be stored in &&KEYS&JJ ❷. The macro %NW ❸ (see Section 10.4) counts the number of variables in the list (&KEYCNT), and separates and stores the names of the BY variables in &KEY1, &KEY2, ..., &&KEY&KEYCNT. Once the data is sorted using the key variables ❹, FIRST and LAST processing can be used ❺ to determine if there are observations with duplicate sets of BY values.

```

%macro chkdup;
%do jj = 1 %to &livedb;❶
  %nw(&&keys&jj ❷ ,wordvar=key,wordcnt=keycnt) ❸
  * Sort the data sets for the
  * key variables;
proc sort data=livedb.&livedb&jj out=base;
  by &&keys&jj; ❹
run;

  * Check for duplicate key values;
%let dupp = 0;
data dupp; set base;
  by &&keys&jj;
  * determine if this is a dup obs;
  if not (first.&&key&keycnt and last.&&key&keycnt); ❺
  call symput('dupp','1');
run;

%if &dupp %then %do;
  * Duplicate key variables were found;
proc print data=dupp;
  id &&keys&jj;
  title1 "&livedb&jj";
  title2 "DUPLICATE KEYFIELDS in LIVE Data set";
run;
%end;
%end; * end the DSN do loop;
%mend chkdup;

```

Using DBDIR from Section 9.2.2, when &JJ is 2 the &LIVEDB2 is MEDHIS and &KEYS contains the BY variables SUBJECT MEDHISNO SEQNO. Since there are three BY variables for this data set, &KEYCNT is 3. The IF statement ❺:

```
if not (first.&&key&keycnt and last.&&key&keycnt);
```

resolves to

```
if not (first.&key3 and last.&key3);
```

which becomes

```
if not (first.seqno and last.seqno);
```

In the above example, when the %NW macro is called, the list of BY variables is broken up and counted. If, as in this example, you really only needed to know the last variable in the list of variables contained in &&KEYS&JJ, the call to %NW ❸ for the second data set (&JJ=2) could be replaced with

```

%let lastkey = %sysfunc(reverse(
  %scan(%sysfunc(reverse(&keys2)),1,%str( ))));

```

The IF statement ❺ then becomes

```
if not (first.&lastkey and last.&lastkey);
```

It is also possible to retrieve the last word in a list directly with the %SCAN function by using a negative second argument. The statement becomes

```
%let lastkey = %scan(&keys2,-1,%str( ));
```

MORE INFORMATION

The double reverse is also used in the %LISTLAST function presented in Section 7.6.1.

9.3.4 Coordinating two macro variable lists

In the process of dealing with various control files you will probably create a series of lists of macro variables. As in the examples in these sections, these lists may not all have the same number of elements. Coordinating these lists therefore becomes important. In these examples the coordination is maintained through the use of a common value—the name of the data set.

Using nested %DO loops

Sometimes the loop through the list of data sets will also require a second loop to pass through the variable list appropriate for each data set. This necessitates a double %DO loop with coordination between the two. In the following example we build a series of zero observation data sets that will be used as prototypes for the analysis data sets. For each data set the list of variables in the KEEP= option, the LENGTH statement, and the LABEL statement is built dynamically. Notice that in the building of each of these statements, the outer loop (&JJ) increments once for each data set, while the inner loop (&KK) cycles through all possible combinations of data sets and variables. A macro %IF statement selects the appropriate variables for a given data set.

```
%macro bldlive;
  %do jj = 1 %to &livedb;
    * One data step for each data set;
    data livedb.&livedb&jj(keep=); ❶
      * Build the var list to keep for this DB;
      %do kk = 1 %to &varcnt; ❷
        %if &livedb&jj=&vdsn&kk or &vdsn&kk=ALL
          %then &var&kk; ❸
      %end;
    );
    * Use length to define variable attributes;
    length ❹
      %do kk = 1 %to &varcnt;
        %if &livedb&jj=&vdsn&kk or &vdsn&kk=ALL
          %then &var&kk &vtyp&kk;
      %end;
    ;
    * Define the variable labels;
    label ❺
      %do kk = 1 %to &varcnt;
        %if &livedb&jj=&vdsn&kk or &vdsn&kk=ALL
          %then &var&kk = "&label&kk";
      %end;
    ;
  stop;
run;
%end;
%mend bldlive;
```

Within the DATA step, %DO loops are used to build the following, each using a variable list appropriate to that data set:

- ❶ KEEP= data set option
- ❷ LENGTH statement
- ❸ LABEL statement.

Since the variable loop encompasses all variables in all data sets, the %IF ❸ is used to select the appropriate variables from the *J*/th data set. Two %DO loops are used to coordinate the two lists of macro variables. The outer loop (with index of &JJ) steps through the list of data sets. The inner loop (with index &KK) steps through all the variables for all the data sets. Since we are interested only in the variables for the data set identified by &&LIVEDB&JJ, that value is compared to the value of the data set in the inner loop (&&VDSN&KK) ❸.

Since the inner loop must pass through all variables in all data sets for each data set of interest, there is a built-in inefficiency. This inefficiency is avoided in the following example dealing with the field checks.

Building a list while within a loop

In the previous example the list of data set variables encompasses all data sets; consequently, when we want to build a list of variables for a given data set we need to coordinate the lists by matching the data set name. An alternative is to build the second (and longer) list only when it is needed. We can then use IF-THEN logic or WHERE processing to build a list that contains only the elements that are needed at that time. For example, the list could contain the variables within a given data set.

The values in the FLDDIR data set are not loaded into macro variables until they are needed. This means that if we are within a macro loop that spans data sets (&&LIVEDB&JJ), we can create the field check list appropriate only for that particular data set. Consequently, the symbol table will only include those macro variables that are needed at that time.

In the example below we want to perform field checks for those data sets that have already been created ❶ (some data sets may not yet exist, thereby making checks unnecessary). The checks are listed in FLDIR, which contains one observation per check, and there can be any number of checks per variable in any given data set. ❷

...code not shown...

```
%do jj = 1 %to &livecnt;
  %* Perform checks if dsn present;
  %if %exists(livedb.&&livedb&jj) ❶ %then %do;
    %put * Field error check for &&livedb&jj *****;
    * Build macro vars that will be used to;
    * construct the tests;
    %let fldcnt = 0;
    data _null_;
      set datamgt.flldir (where=(dsn="&&livedb&jj")) end=eof; ❷
      i+1;
      ii=left(put(i,3.));
      call symput('fvar'||ii,trim(var)); ❸
  %end;
```

```

        call symput('ftyp' || ii, trim(chktype));
        call symput('ftxt' || ii, trim(chktext));
        if eof then call symput('fldcnt', ii); ❹
run;

        %if &fldcnt gt 0 %then %do; ❺

...code shown below...

```

We can first filter for the data sets that already exist by using the %EXISTS macro ❶ (see Section 7.5.2). The DATA _NULL_ step, which creates the macro variables, reads only those observations from FLDDIR that match the name of the data set of interest (&&LIVEDB&JJ) ❷. The macro variables are then built ❸ for each observation that passes the WHERE criteria. It is entirely possible that there will be no observations for a particular data set. That is, no field checks have been specified for this data set. This results in &FLDCNT = 0 ❹. This is noted using a %IF ❺ prior to performing the checks.

For those data sets with field checks (&FLDCNT>0), the following creates the code that performs the checks. The data field checks are conducted in a DATA step (one DATA step for each &&LIVEDB&JJ). The field check statements are constructed based on the values in FLDDIR and are made using IF-THEN/ELSE processing and assignment statements. These statements are built inside of the DATA step by using the macro variables created in the previous step.

In the code below any detected field errors are stored in the data set TEMPERR ❶ (one observation per error). Each observation receives a date stamp with the date the error was first detected ❷. A macro %DO loop ❸ steps through each of the field checks for this data set (&&LIVEDB&JJ) and builds the appropriate statements.

```

* Perform field and intra-field checks;
data temperr(keep= status &&keys&jj dsn var
               count msg text value chkdate); ❶
    set datamgt.&&livedb&jj;
    by &&keys&jj;

    * Date these field check problems ;
    * were first detected);
    retain chkdate %sysfunc(today()); ❷
    format chkdate date9.;

    * Count the number of times this ;
    * problem has been detected;
    * Status will be controlled by the ;
    * manager - initialize to NEW;
    length status $12;
    retain count 1 status 'NEW';

    * Specify various lengths to the ;
    * data set variables;
    * VALUE is only given a length of 15;
    * this can cause some truncation (in display);
    length dsn var $8 value $15 text msg $100;

    * Place the name of the data set into;
    * a data variable;
    retain dsn "&&livedb&jj";

```

```

%* Build the Field and Intra-Observation;
%* Field error checks;
%do i = 1 %to &fldcnt; ❸
  %if %upcase(&&ftyp&i) = LIST %then %do; ❹
    if &&fvar&i not in&&ftxt&i then do; ❺
      var = "&&fvar&i"; ❻
      msg = 'Value is not on list';
      text = "&&ftxt&i";
      value = &&fvar&i;
      output temperr;
    end;
  %end;
%end;

```

There are several types of field checks, including LIST, which specifies a list of acceptable values. When `&&FTYP&I = LIST` ❹, an IF-THEN/DO block is defined that checks to see if the value stored in the variable named in `&&FVAR&I` is in the list stored in `&&FTEXT&I`. When the value is not in the list ❺, the field is in error and a series of assignment statements are executed ❻. For the second observation in FLDDIR

OBS	DSN	VAR	CHKTYPE	CHKTEXT
2	DEMOG	RACE	list	('1','2','3')

the DO block becomes

```

if RACE not in('1','2','3') then do; ❺
  var = "RACE"; ❻
  msg = 'Value is not on list';
  text = "('1','2','3)";
  value = RACE;
  output temperr;
end;

```

This DATA step automatically expands to accommodate new field checks. When the macro was written, the programmer had no specific information about the name of the data set, the name of the variable, the type of field check, or the acceptable values for the field check.

9.4 Building SAS Statements Dynamically

Throughout this book, and especially within this chapter, dynamic coding has been accomplished using vectors of macro variables. The pattern has been to place information into macro variables of the form of `&VAR1`, `&VAR2`, `&VAR3`, and so on. This list of macro variables is then later utilized within a %DO loop and referenced using the `&&VAR&I` construct. This approach, however, is not the only one available.

Since the control file is accessed at some point with a DATA step (to make use of the CALL SYMPUT if nothing else), the power of the DATA step can be utilized in other ways. The techniques demonstrated here do not create the list of macro variables and do not use the `&&VAR&I` macro-variable form. While these techniques do have their place, they also have some disadvantages, and it has been my experience (or bias?) that they can be less useful and potentially more difficult.

9.4.1 Using the DATA _NULL_ and %INCLUDE

In Section 9.3.4 the %BLDLIVE macro dynamically creates a series of empty data sets. If we only wanted to create the DEMOG data set we would not need the outer %DO loop and the macro could be simplified to the following:

```
%macro blddemog;
  * Create the KEEP for DEMOG;
  data livedb.demog(keep=
    * Build the var list to keep for this DB;
    %do kk = 1 %to &varcnt;
      %if &&vdsn&kk=DEMOG
    or &&vdsn&kk=ALL %then &&var&kk;
    %end;
  );
  * Use length to define variable attributes;
  length
    %do kk = 1 %to &varcnt;
      %if &&vdsn&kk=DEMOG or &&vdsn&kk=ALL
        %then &&var&kk &&vtyp&kk;
    %end;
  ;
  * Define the variable labels;
  label
    %do kk = 1 %to &varcnt;
      %if &&vdsn&kk=DEMOG or &&vdsn&kk=ALL
        %then &&var&kk = "&&label&kk";
    %end;
  ;
  stop;
  run;
%mend blddemog;
```

The resulting DATA step becomes

```
* Create the KEEP for DEMOG;
data livedb.demog(keep= SUBJECT PTINIT DOB SEX );
* Use length to define variable attributes;
length SUBJECT $8 PTINIT $8 DOB 8 SEX $8 ;
* Define the variable labels;
label SUBJECT = "Patient number"
      PTINIT = "Patient initials"
      DOB = "Date of birth"
      SEX = "Sex" ;

stop;
run;
```

We can build the same DATA step using PUT statements in a DATA _NULL_ step. Since we will be creating three statements using three different sets of information (variable name, variable type, and variable label), we will read in the information, and store it in DATA step arrays ②.

```

filename temptxt 'c:\temp\temptext.txt'; ❶

data _null_;
  set datamgt.vrdir end=eof;
  array nam {1000} $8 _temporary_; ❷
  array typ {1000} $2 _temporary_;
  array lbl {1000} $40 _temporary_;
  if dsn in('DEMOG', 'ALL') then do; ❸
    i+1;
    *Save this variables information;
    nam{i} = var;
    typ{i} = vartype;
    lbl{i} = label;
  end;
  if eof then do;
    * Write the DATA step statements;
    file temptxt; ❹

    * Write the DATA statement with the KEEP=;
    put 'data livedb.demog(keep='; ❺
    do j = 1 to i; ❻
      put '      ' nam{j};
    end;
    put '      ');';

    * Write the LENGTH statement;
    put 'Length ';
    do j = 1 to i;
      put '      ' nam{j} typ{j};
    end;
    put '      ';';

    * Write the LABEL statement;
    put 'Label ';
    do j=1 to i;
      put '      ' nam{j} ' = "' lbl{j} "''; ❼
    end;
    put '      ';';
    put 'run;';
  end;
run;

%include temptxt; ❽

```

- ❶ The FILENAME statement is used to identify a text file that will hold the code written by the DATA step.
- ❷ Since we will need to build three statements, the individual values are stored in an array. When only one statement is being built it is often unnecessary to use arrays.
- ❸ Values from the observations that meet the stated criteria (variables in the DEMOG data set) are stored in the appropriate arrays.
- ❹ The FILE statement directs the text generated by the PUT statements to the file designated in the FILENAME statement ❶.
- ❺ The DATA statement is written out as a text string.

- ⑥ The DO loop cycles through the variable names that are held in the NAM array.
- ⑦ Notice the use of the quoting. In this statement single quotes are used to write double quotes into the code.
- ⑧ Once the DATA step has been coded, it is brought in for execution via the %INCLUDE statement.

For the example above, the following SAS DATA step is written to 'TEMPTEXT.TXT':

```
data livedb.demog(keep= ⑤
  SUBJECT ⑥
  PTINIT
  DOB
  SEX
);
  Length
  SUBJECT $8
  PTINIT $8
  DOB 8
  SEX $8
;
  Label
  SUBJECT = "Patient number " ⑦
  PTINIT = "Patient initials "
  DOB = "Date of birth "
  SEX = "Sex "
;
run;
```

The use of the DATA _NULL_ avoids the use of macro arrays and the &&VAR&I macro variable constructs. For programmers who are already familiar with the use of the PUT for formatted report writing, it is not a difficult extension to write steps that generate code.

One of the difficulties associated with these techniques involves writing quoted strings ⑦. Since the PUT statement itself uses quotes to write text, you will need to be very careful to correctly nest the single and double quote marks in order for you to build statements that contain quoted strings.

Notice also that this example was for only one of the data sets. The code becomes more interesting when it is generalized to allow multiple data sets.

MORE INFORMATION

The first example in Section 11.4.1 uses this technique to build a %LET statement.

SEE ALSO

Graebner (1998) uses the DATA _NULL_ step to write a PROC REPORT step. Reading (2000) uses SAS data sets to construct a macro that is then brought in through the use of a %INCLUDE. Chow (1999) discusses both the %INCLUDE and the CALL EXECUTE. Johnson (2001) demonstrates the use of the %INCLUDE as part of a discussion of writing code that writes code. Morrill, Wiser, and Zhoo (2002) build an extensive macro and macro call that is included for execution.

9.4.2 Using the CALL EXECUTE routine

CALL EXECUTE (introduced in Section 6.6) is a DATA step routine. When the CALL EXECUTE is executed, its argument is immediately passed to the macro facility for evaluation. Arguments that contain macro statements (like a macro call) are immediately executed. Non-macro text or text generated by a macro is placed in a stack for execution after the completion of the current DATA step.

This routine is typically used when the values contained in a data set are to be applied as parameters to a macro. Because of the timing issues between the DATA step and the macro facility, as well as with the stack and how it is loaded, uses of the CALL EXECUTE can become complicated very quickly.

In Section 9.3.3 the macro %CHKDUP contains a %DO loop that executes once for each data set of interest. This loop then requires the use of &&KEYS&&JJ and &&LIVEDB&&JJ to identify data-set-specific information. Instead, the macro could have been rewritten to be called once for each data set. In %CHKDSN, shown below, the name of the data set and the list of BY variables are passed into the macro:

```
%macro chkdsn(dsn, keys); ❶
  %nw(&keys, wordvar=key, wordcnt=keycnt) ❷
  * Sort the data sets for the
  * key variables;
  proc sort data=livedb.&dsn out=base;
    by &keys;
    run;

    * Check for duplicate key values;
    %let dupp = 0;
    data dupp; set base;
    by &keys;
    * determine if this is a dup obs;
    if not (first.&key&keycnt and last.&key&keycnt);
    call symput('dupp', '1'); ❸
    run;

    %if &dupp %then %do;
    * Duplicate key variables were found;
    proc print data=dupp;
      id &keys;
      title1 "&dsn";
      title2 "DUPLICATE KEYFIELDS in LIVE Data set";
    run;
    %end;
%mend chkdsn;
```

- ❶ Parameters are included for the name of the data set to check and the list of the BY variables associated with that data set.
- ❷ %NW is called. This macro, described in Section 10.4.4, counts the number of variables in the list (&KEYCNT), and separates and stores the names of the BY variables in &KEY1, &KEY2, ..., &KEY&KEYCNT.
- ❸ If there are one or more observations that have duplicate keys, the macro variable &DUPP will be set to 1.

The following CALL EXECUTE calls the %CHKDSN macro once for each data set name (DSN) in the DBDIR data set. Notice that the macro call is built using **single** quotes. This is important as they prevent the immediate execution of the macro.

```
data _null_;
  set datamgt.dbdir;
  call execute('%chkdsn('||dsn||','||keyvar||')');
run;
```

This adds the following calls for %CHKDSN to the program stack. These calls are executed as soon as the DATA _NULL_ step has completed.

```
%chkdsn(DEMOG      ,SUBJECT
%chkdsn(MEDHIS     ,SUBJECT MEDHISNO SEQNO
%chkdsn(PHYSEXAM, SUBJECT VISIT REPEATN SEQNO
%chkdsn(VITALS    ,SUBJECT VISIT SEQNO REPEATN
```

In Section 9.3.4, the %BLDLIVE macro dynamically creates a series of empty data sets. The macro is called once and uses a %DO loop indexed with &JJ to cycle through a list of data sets stored in &&LIVEDB&&JJ. In the macro %BLDDSN, shown below, the %DO loop that cycles through the data sets is removed, and the name of the data set is passed directly into the macro ❶. Obviously, %BLDDSN is therefore called once for each data set. The macro calls themselves are built with a CALL EXECUTE.

```
%macro blddsn(dsn); ❶
  * One data step for each data set;
  data work.&dsn(keep=
    * Build the var list to keep for this DB;
    %do kk = 1 %to &varcnt;
      %if &dsn=&&vdsn&kk ❷
        or &&vdsn&kk=ALL %then &&var&kk;
    %end;
  );
  * Use length to define variable attributes;
  length
    %do kk = 1 %to &varcnt;
      %if &dsn=&&vdsn&kk
        or &&vdsn&kk=ALL %then &&var&kk &&vtyp&kk;
    %end;
  ;
  * Define the variable labels;
  label
    %do kk = 1 %to &varcnt;
      %if &dsn=&&vdsn&kk
        or &&vdsn&kk=ALL %then &&var&kk = "&&label&kk";
    %end;
  ;
  stop;
run;
%mend blddsn;
```

❶ The name of the data set to be created is passed into the macro as &DSN.

❷ The data set name is used to determine which variables and their attributes to use when building the various statements.

The CALL EXECUTE calls the %BLDDSN macro once for each data set name (DSN) in the DBDIR data set. This effectively replaces the outer %DO loop used in %BLDLIVE (Section 9.3.4).

```
data _null_;
  set datamt.dbdir;
  call execute('%blddsn('||dsn||')'); ❸
run;
```

- ❸ The argument to the CALL EXECUTE becomes a call to the %BLDDSN macro. Since there are four observations in DBDIR, there will be four calls to the macro %BLDDSN in the execution stack. The stack will contain the following:

```
%blddsn(DEMOG  )
%blddsn(MEDHIS  )
%blddsn(PHYSEXAM)
%blddsn(VITALS  )
```

MORE INFORMATION

The macro %USELIST in Section 9.4.3 is very similar to %BLDDSN: however, it does not utilize macro arrays.

SEE ALSO

Whitlock (1997) provides a good overview to the CALL EXECUTE routine.

A list of macro variables is cleared using SASHELP.VMACRO and CALL EXECUTE in an example by Rhoads and Letourneau (2002).

Jiang (2003) builds a macro call that is executed through a CALL EXECUTE.

9.4.3 Using macro lists rather than macro arrays

When working with lists of values it is possible to store all the values in a single macro variable rather than in a series of macro variables (with one value per macro variable as was done in the previous examples). This approach requires that you step through the “words” in the list rather than the list of macro variables, and we can use the %SCAN function to perform this operation.

In Section 9.4.2 the macro %BLDDSN is used to build the attributes of a stated data set. The attributes are stored in a SAS data set (DATAMGT.VRDIR—described in Section 9.2.2). In the following example the macro %USELIST accomplishes the same task; however, it completely avoids the use of macro arrays. Instead macro variables containing lists of values are built and the %SCAN function is used to select the individual values.

```
%macro uselist(dsn);
  * Create lists of data set attributes;
  proc sql noprint; ❶
    select var,label,vartype ❷
      into :varlist separated by ' ', ❸
           :lblist separated by ', ', ❹
           :typlist separated by ' '
    from datamt.vrdir
      (where=(dsn in('ALL',%upcase("&dsn")))); ❺
  quit;
```

```

* One data step for each data set;
data work.&dsn(keep= &varlist); ❸
  * Use length to define variable attributes;
  length
    %do kk = 1 %to &sqlobs;
      %scan(&varlist,&kk) %scan(&typlist,&kk) ❹
    %end;
  ;
  * Define the variable labels;
  label
    %do kk = 1 %to &sqlobs;
      %scan(&varlist,&kk)="&scan(&lbllist,&kk,str(,))" ❺
    %end;
  ;
stop;
run;
*/ *;
%mend uselist;

%uselist(demog)

```

- ❶ An SQL step is used to build the macro variables (&VARLIST, &LBLLIST, and &TYPLIST).
- ❷ The three variables whose values are to be read into the respective macro variables are noted in the SELECT statement.
- ❸ The receiving macro variable name is preceded by a colon. Since we are creating a list, the SEPARATED BY clause is used to append the individual values.
- ❹ The data set option WHERE clause in SQL does not support the UPCASE function when it is used within the IN operator, so the macro function %UPCASE is used instead.
- ❺ The full list is used to build the KEEP= option.
- ❻ Step through the list of “words” in these two macro variables. There are &SQLOBS items in each list, where &SQLOBS is automatically established by the SQL step.
- ❼ The values of the labels usually contain embedded blanks; therefore, blanks cannot be used as “word” delimiters. Commas are used instead, and this is noted in the %SCAN function for &LBLLIST. Commas would be needed as the word separators in the other lists if there were missing values.

SEE ALSO

Deprince and Li use macro variables to store disparate lists, which are then broken up into component parts.

9.5 Using SASHELP Views

A series of SAS views have been defined and stored under the SASHELP *libref*. These views contain valuable information about the current status of your system and operating environment. These views can be especially useful to the macro programmer because they can be used as a source for building macro variables.

Although the SASHELP views discussed below can also be used within an SQL step, a similar, though not as extensive, set of tables can be accessed, while in an SQL step, by using the *libref* DICTONARY. In terms of information content there is some overlap between the SQL DICTONARY tables and the SASHELP views. When you are using SQL and you have a choice between the two types of tables, the DICTONARY tables will often be a bit quicker.

MORE INFORMATION

SQL DICTONARY tables are introduced in Section 6.5.4 and a number of examples throughout this book make use of these tables.

SEE ALSO

The SASHELP views are documented in SAS Technical Report P-222 and in the Online Documentation. An introductory discussion of metadata as well as the use of SASHELP.VCOLUMN is presented by Spicer (2003).

Davis (2001) describes the SASHELP views and the corresponding DICTONARY tables used by SQL. Hamilton (1998) concentrates on the SQL DICTONARY tables. Casas (2002) uses DICTONARY.TABLES and DICTONARY.COLUMNS to build macro variables from within an SQL step.

A list of macro variables is cleared using SASHELP.VMACRO and CALL EXECUTE in an example by Rhoads and Letourneau (2002).

The basics, along with a couple of caveats, for using VCOLUMN, VCATALG, and VMEMBER are presented by Kelly (2000). VCOLUMN is also used by Goddard (2003) in an example that writes RTF files using ODS.

Troxell and Chen (2003) use SASHELP.VCATALG to determine if a macro has been compiled.

SASHELP.VMACRO is used by Yu and Huang (2003) in a macro that generates return codes.

9.5.1 Overview of the SASHELP views

Although they seem to be data sets in almost all aspects, views in fact contain no data. Instead they contain the instructions needed to build the table when the view is requested. The availability of these views is very important to the dynamic macro programmer as they provide a great deal of information about the SAS environment during macro execution. This information can be translated into macro variables, which in turn can be used to drive the dynamic application.

Following are some of the primary views available in SASHELP:

VCATALG	List of objects and entries within a catalog.
VCOLUMN	List of variables within data sets.
VEXTFL	<i>Filerefs</i> and their assigned paths.
VINDEX	Indexes including types.
VMACRO	Each macro variable, its scope, and its value.
VMEMBER	Names of all entities controlled by SAS (data sets and catalogs), their <i>librefs</i> , and their physical location.
VOPTION	Current setting of each system option.
VSCATLG	<i>Librefs</i> and names of all SAS catalogs.

VSLIB	List of all <i>librefs</i> and their associated physical location (path).
VSTABLE	List of all data sets and their associated <i>libref</i> .
VSTYLE	List of ODS styles and their location.
VSVIEW	List of all VIEWS and their associated <i>libref</i> , similar to VVIEW.
VTABLE	List of all data sets and views, including information like the number of observations and modification dates.
VTITLE	Definition and number of TITLES and FOOTNOTES.
VVIEW	List of defined views (includes the views in SASHELP).

The contents and attributes of these views are fairly straightforward. Try opening each one using the VIEWTABLE in the Display Manager. While it should be fairly obvious what each contains, it may not be immediately obvious as to how or when you will use any particular view in one of your macros. The important point is that you will now know what is available when you DO need it.

9.5.2 Using a view

The SASHELP views are used as you would any of the control data sets mentioned elsewhere in this chapter. Simply read the appropriate values into one or more macro variables and then use them in your program.

The following macro just does a PROC PRINT. The catch is that we want its title to be the next available title. Since we do not know what the last defined title was, we need to determine it dynamically. The view SASHELP.VTITLE is read using an SQL step and the maximum title number is saved in a macro variable that is used later.

```
%macro look(dsn);
%local maxn;
* Determine the next available title;
proc sql noprint; ❶
  select max(number) ❷
    into :maxn ❸
    from sashelp.vtitle ❹
    where type='T'; ❺
quit;

title%eval(&maxn+1) ❻ "Listing of &dsn";
proc print data=&dsn;
  run;
title%eval(&maxn+1); ❼

%mend look;
```

- ❶ An SQL step is used to build the macro variable containing the largest title number.
- ❷ The MAX function collects the largest value of the variable NUMBER.
- ❸ The maximum value of NUMBER is placed INTO the macro variable &MAXN.
- ❹ SASHELP.VTITLE is used as the information source. This view has one observation per title or footnote.

- ❺ Only lines for titles are included (footnotes have TYPE='F').
- ❻ The title is assigned the next title number.

Notice that this macro does not do any checking to make sure that the total number of titles does not exceed 10. Like a good child, it does, however, clean up after itself by resetting the new title after the PROC PRINT is complete ❼.

MORE INFORMATION

The next available title is also determined in the example in Section 10.2.3.

9.6 Chapter Summary

Control data sets can be used to store any project, data set, or variable-specific information that will be needed by the application programs. This information can include the names of data sets, the variables within those data sets, variable attributes, and field-check information. These control data sets are then used to create a series of macro variables. Application programs that require project specific information, such as the names of data sets and variables, use these macro variable lists to dynamically build the SAS code needed for the project, data set, or variable of interest.

Dynamic code building requires the use of SAS macros and a number of macro statements. The macro %DO loop is used extensively, as is the &&VAR&I macro variable form. The double ampersand macro variable (&&VAR&I) acts like a macro variable array with VAR as the array name and the &I macro variable as the index or subscript. The macro variables themselves are generally created from the control data sets by the use of the CALL SYMPUT routine.

Other tools used when creating dynamic applications include the CALL EXECUTE routine, %INCLUDE, and SASHELP views.

Dynamic programming is an advanced macro topic. Creating the SAS programs and macros that can take advantage of dynamic techniques, such as &&VAR&I macro variables, is not initially easy. You might need to practice and work with the techniques discussed in this chapter for a while before dynamic programming techniques become second nature for you.



Chapter 10

Controlling Your Environment

- 10.1 Operating System Operations 240**
 - 10.1.1 Copy members of a catalog 240**
 - 10.1.2 Write the first *N* lines of a series of flat files 242**
 - 10.1.3 Storing system clock values in macro variables 244**
 - 10.1.4 Checking for write access 245**
 - 10.1.5 Appending unknown data sets 246**
 - 10.1.6 Making a directory 251**
 - 10.1.7 Executing a series of SAS programs 252**
 - 10.1.8 Using %SYSGET to access system variables 255**
- 10.2 Controlling Your Output 255**
 - 10.2.1 Combining titles 256**
 - 10.2.2 Renumbering listing pages 260**
 - 10.2.3 Coordinating titles (or footnotes) 261**
- 10.3 Adapting Your SAS Environment 262**
 - 10.3.1 Maintaining system options 262**
 - 10.3.2 Building and maintaining formats 265**
 - 10.3.3 Working with libraries and directories 268**
- 10.4 Coordinating with the Output Delivery System (ODS) 269**
 - 10.4.1 Why we might need to automate with macros 270**
 - 10.4.2 Controlling ODS locations 270**
 - 10.4.3 Using ODS to control the destination 271**

10.4.4 Graphics devices	271
10.4.5 Using WEBFRAME with GRSEG catalog entries	272
10.4.6 Building an index with PROC PRINT	274
10.4.7 Creating drill-down graphs and charts	275
10.5 Chapter Summary	277

Through the use of macro examples, this chapter introduces several concepts that you can use to control your operating environment. This includes the routing of output, control of report appearance, and working with system files other than SAS system files.

10.1 Operating System Operations

As a general rule, the macros in this section perform operations on files, programs, catalogs, and members of catalogs rather than on lines of code. This means that in each case you need to be able to gather the appropriate information on the operating system (for example, file names) and create macro and data set variables. SAS gathers a great deal of the system information into a series of SAS views that are stored in the SASHELP library. You can use these views to generate lists of data sets, members of catalogs, variables within a data set, and more. The macro in Section 10.1.1 uses one of these views to gather information about the members of a SAS catalog, and in Section 10.2 SASHELP.VOPTION is used to read and store current OPTION settings.

MORE INFORMATION

Section 9.5 introduces and discusses the views in SASHELP.

SEE ALSO

Geary (1997) discusses a macro that produces summary information on a series of SAS data sets.

A disk space utilization macro is presented in Mast (1997).

A series of directories are built using a macro presented by Dynder, Cohen, and Cunningham (2000).

Under Windows operating systems, Dynamic Link Libraries (DLLs) are often used for operating system operations. Roper (2001) discusses the use of Win32APIs to access DLLs.

10.1.1 Copy members of a catalog

%CATCOPY

You can use this macro to copy selected members of a catalog in the TEST environment to a production area. You do not need to know the names of the members prior to execution. However, a filter is available to select members that begin with certain types of names.

The SASHELP.VSCATLG view ❶ is used to create a list of the members in the TEST library. A DATA _NULL_ step is used with a subsetting IF ❷ to select the members of interest. The macro

variables (&&CNAME&I) that contain the member names ❸ and the number of names (&CATCNT) ❹ are used within PROC DATASETS to build the SELECT statement ❺.

```
* Copy catalogs from the TEST to the PRODUCTION
* areas.;

options nomprint nomlogic nosymbolgen;

%macro catcopy(test,prod);
* test - libref for the test area
* prod - libref for the production area
*;

* Determine catalogs in TEST area;
data _null_;
set sashelp.vscatlg(where=(libname="%upcase(&test)")); ❶
length ii $2;

* Select only some of the catalog members;
if memname in: ('DE', 'ED', 'PH'); ❷

i+1;
ii=left(put(i,2.));
call symput('cname'||ii,memname); ❸
call symput('catcnt', ii); ❹
run;

proc datasets ;
copy in=&test out=&prod memtype=catalog;
select
  %do i = 1 %to &catcnt; ❺
    &&cname&i
  %end;
;
quit;
%mend catcopy;

%catcopy(appls,work)
```

SEE ALSO

The Technical Support section of *SAS Communications*, Volume xxii, 4th Qtr. 96 (p. 43) has a similar example that uses the BUILD procedure to build the new catalogs with modifications.

Rook and Yeh (2001) discuss several alternative methods for concatenating and copying members of catalogs.

10.1.2 Write the first *N* lines of a series of flat files

%DUMPIT

Clarence Wm. Jackson
CJAC

The %DUMPIT macro is used to list the first few lines of each of a series of flat files. The following example code dumps some QSAM files, PDS's, and so on, using a list. Notice that this was used on the mainframe, so the FILENAME statement ❹ has a DISP=SHR.

```
%MACRO DUMPIT (CNTOUT);
  %* Create a local counter;
  %LOCAL CWJ;

  %DO CWJ=1 %TO &NUMOBS;

    %* Fileref to identify the file to list;
    FILENAME DUMP&CWJ "&&INVAR&CWJ" ❺ DISP=SHR; ❹

    * Read and write the first &CNTOUT records;
    DATA _NULL_ ; ❻
      INFILE DUMP&CWJ END=DONE; ❼
      * Read the next record;
      INPUT;
      INCNT+1;
      IF INCNT LE &CNTOUT THEN LIST;
      IF DONE THEN DO;
        FILE PRINT;
        PUT @@10 "TOTAL RECORDS FOR &&INVAR&CWJ IS "
          +2 INCNT COMMA9. ;
      END;
    RUN;

    FILENAME DUMP&CWJ CLEAR;

  %END;
%MEND DUMPIT; * The Macro definition ends;

* Read the control file and establish macro variables;
DATA DUMPIT; ❶
  INFILE CARDS END=DONE;
  INPUT FILENAM $25.;
  CNT+1;
  NEWNAME=TRIM(FILENAM);
  * The macro variable INVARi contains the ith file name;
  CALL SYMPUT ('INVAR'!!TRIM(LEFT(PUT(CNT,3))),NEWNAME); ❸
  * Store the number of files to read;
  IF DONE THEN CALL SYMPUT('NUMOBS',CNT);
CARDS; ❷
PNB7.QSAM.BANK.RECON
PNB7.QSAM.CHECKS
PNB7.QSAM.CHKNMBR
PNB7.QSAM.CKTOHIST
PNB7.QSAM.DRAIN
PNB7.QSAM.RECON
```

```

PNB7.BDAM.BDAMCKNO
PNB7.BDAM.VCHRCKNO
PNB7.QSAM.CS2V3120.CARDIN
PNB7.QSAM.CASVCHCK
PNB7.QSAM.CASVOUCH
PNB7.QSAM.VCHR3120.CARDIN
PNB7.QSAM.VOUCHERS
TAX7.JACKSON
;;;

TITLE "CITY OF DALLAS - ECI (FINSYS), JOBNAME IS &SYSJOBID";
TITLE2 "LIST OF FILES TO DUMP &CNTOUT RECORDS";
PROC PRINT data=dumpit;
RUN;

* Pass the number of records to dump from each file;
%DUMPIT (25);

```

- ❶ The DATA step is used to read the names of the files ❷ that are to be dumped. These names are stored in NEWNAME.
- ❷ The list of files in this example is brought in using the CARDS statement. The list could just as easily have been built using CLIST or using other methods.
- ❸ Use the SYMPUT routine to load the name of the *i*th dump file into the macro variable &INVAR*i*.
- ❹ The *fileref* created in this FILENAME statement is used to identify the name of the file that is to be listed.
- ❺ The macro variable &&INVAR&CWJ in the FILENAME statement ❹ was created in the DATA step ❶ using a CALL SYMPUT ❸ before the macro %DUMPIT is called.
- ❻ A DATA _NULL_ step is used to read and list the lines in the flat file.
- ❼ The INFILE points to the *fileref* that was established in the preceding FILENAME statement ❹.

Note that although this macro lists only the first &CNTOUT lines of each flat file, the entire file is read in order to establish a line count.

SEE ALSO

Widawski (1997b) collects a list of dBase files that are to be converted to SAS files.

Several examples that are discussed by Yu (1998) use %SYSFUNC to read and write external files, and Chen (2003) discusses a number of methods that can be used to read external files.

10.1.3 Storing system clock values in macro variables

%UPDATE

Jørn Lodahl

The automatic macro variables &SYSDATE and &SYSTIME reflect the time that the SAS session was invoked, and during long sessions they may not accurately reflect the actual time that a particular step executed. The current date and time can be captured and stored using the macro %UPDATE.

%UPDATE can be used to grab either the current date values, the current time values, or both, and it refreshes or updates the macro variables &TIMESTR, &TODAYSTR, and &NOWSTR. You can specify any one of the three macro variables, or you can specify ALL to update each of the macro variables. Notice that these three macro variables reside in the global symbol table, so you may need to be selective about the names before using the macro in your system.

```

/*****
SYNTAX:
  %update(string_var)
EXAMPLES:
  %update(all)
  %update(timestr)
This macro updates some or all of the following date/time
string macro variables:
  &timestr
  &todaystr
  &nowstr
*****/

%macro update(string); ❶
%global timestr todaystr nowstr;
%if &string=all %then %do;
  %let string=nowstr;
%end;
data _null_;
  %if &string=todaystr or &string=nowstr %then %do;
    call symput('todaystr',put(today(),worddate.)); ❷
  %end;
  %if &string=timestr or &string=nowstr %then %do;
    call symput('timestr',put(time(),HHMM.));
  %end;
  %if &string=nowstr %then %do;
    %let nowstr=&timestr&todaystr; ❸
  %end;
run;
%mend update;

```

- ❶ The user passes in the name of the macro variable to update.
- ❷ CALL SYMPUT is used to place the formatted value of the current date into the appropriate macro variable.
- ❸ When %UPDATE is called with the parameter &STRING = all, the macro variable &NOWSTR will contain both the date and time strings.

Very often you can avoid the DATA _NULL_ step in situations like this. When data are not actually read, the %SYSFUNC macro function (see Section 7.4.2) can usually be used to access the same DATA step functions. The following macro does the same thing as %UPDATE, but does not require the use of the DATA _NULL_.

%UPDATE2

The DATA step has been removed in %UPDATE2 and the calls to the TODAY and TIME functions have been accessed with the %SYSFUNC macro function.

```
%macro update2(string);
  %global timestr todaystr nowstr;
  %let string = %lowercase(&string);
  %if &string=all %then %let string=nowstr;

  %if &string=todaystr or &string=nowstr %then
    %let todaystr = %left(%qsysfunc(today()),worddate.);
  %if &string=timestr or &string=nowstr %then
    %let timestr = %left(%sysfunc(time()),HHMM.);

  %if &string=nowstr %then %let nowstr=&timestr&todaystr;
%mend update2;
```

10.1.4 Checking for write access

%CHKCOPY

The following macro copies the data sets in one library (COMBINE) to another (COMBTEMP), and it uses &SYSERR to detect whether another user currently has write access to a data set in the library COMBINE. If a user has even a VIEWTABLE open for one of the data sets, the COPY will be incomplete. This macro was written so that if the complete copy does not take place, then the whole process will be aborted.

```
%macro chkcopy;
  * Copy the current version of the COMBINE files
  * to COMBTEMP;
proc datasets memtype=data; ❶
  copy in=combine out=combtemp;
  quit;
%if &syserr ge 5 %then %do; ❷
  data _null_;
    put '*****';
    put '*****';
    put '*****';
    put '*** combine copy failure ***';
    put 'One of the data sets may be in use.';
    put '*****';
    put '*****';
    put '*****';
    abort abend; ❸
  run;
```

```

* When aborted (inside this macro do block) nothing else
* should execute in this job including the following
* message;
%put JOB ABORTED!!!!
%put this message should never be written!!;
%end;
%mend chkcopy;

```

- ❶ Copy all members of the *libref* COMBINE to COMBTEMP. If another user has a write lock on one or more members, those members will not be copied.
- ❷ &SYSERR contains the success code for PROC DATASETS. Five or larger indicates an unacceptable lack of success.
- ❸ The ABORT statement causes the current SAS process to terminate. The behavior of the ABORT statement varies depending on the execution process (AF application, interactive submittal, or batch submittal), and the presence or absence of the ABEND option.

If you are using SAS 9 or later, you may wish to use the %ABORT statement. The %DO block would no longer be needed and the %IF becomes

```
%if &syserr ge 5 %then %abort abend;
```

10.1.5 Appending unknown data sets

%CMBNSUBJ

This macro is taken from an application where a series of separate *information* data sets are created (one for each subject in the study). Generally, this is an inefficient storage technique. However, because SAS/SHARE software was not available, these data sets were also used to control which users could access data for a particular subject at any given time. The data sets each contained a single observation that stored various data entry and edit status indicators.

In order to create a unified subject status report, it was necessary to combine these individual data sets by concatenating them into one. The problem, of course, is that the number of subjects and the associated code (used to name the data set) is constantly changing, and therefore a dynamic solution is required.

There are several ways that we can use to determine the names of the data sets that meet the selection criteria. These include the use of

- the view SASHELP.VTABLE
- PROC CONTENTS to create a data set
- operation system utilities through the X statement
- the PIPE device type on the FILENAME statement.

Using SASHELP.VTABLE

The following macro determines the list of current subjects from the names of the available files stored in the SASHELP.VTABLE view, and then places the subjects' numbers into a series of macro variables.

```

%macro cmbnsubj;

* Determine the subjects, make a macro var for each;
* DECCTRL.INxxxxxx data sets have one obs per subject;
data _null_;
  set sashelp.vtable(keep=libname memname); ❶
  where libname='DECCTRL' & memname='IN'; ❷
  length ii $3 subject $6;
  i+1;
  ii=left(put(i,3.));
  subject=substr(memname,3);
  call symput('subj'||ii,subject); ❸
  call symput('subjcnt',ii); ❹
run;

proc datasets library=work;
  delete subjstat;

  * Combine the subject control files;
  %do i = 1 %to &subjcnt;
    append base=subjstat data=decctrl.in&&subj&i; ❺
  %end;
quit;
%mend cmbnsubj;

```

- ❶ The SASHELP.VTABLE view is used to create a list of all of the data sets in the application.
- ❷ The WHERE statement subsets the entries to those data sets of interest, such as data sets that start with the letters IN in the DECCTRL library.
- ❸ A macro variable of the form &SUBJ1, &SUBJ2, and so on is created for each subject number (which is derived from the data set name, such as IN133032).
- ❹ The number of subjects is counted.
- ❺ The selected data sets are combined using the APPEND statement in PROC DATASETS. This is more efficient than dynamically building a SET statement, as was done in Section 6.3, and requires fewer data sets to be open at any given time.

There is a difference between SASHELP.VTABLE and SASHELP.VSTABLE. The latter contains only the member and library information, and either could be used in this macro.

Using PROC CONTENTS

Because the SASHELP.VTABLE is a view, it must be created each time it is called. Because it always builds a list of **all** members of **all** established *librefs* (the full list is created before the WHERE clause is applied), building the list can be a time-consuming process. A PROC CONTENTS with the OUT= and NOPRINT options might be a faster alternative to building this list as a view. The OUT= option for PROC CONTENTS will create a data set with one

observation per variable per data set. %CMBNSUBJ is modified here to use PROC CONTENTS with a SORT to eliminate the duplicate names:

```
%macro cmbnsubj;

* Determine the subjects, make a macro var for each;
* DECCTRL.INxxxxxx data sets have one obs per subject;

* Create a list of all data sets in the libref DECCTRL;
* ALLCONT will have one observation for each variable in
* each data set;
proc contents data=decntrl._all_
              out=allcont(keep=memname)
              noprint;

run;

* Eliminate duplicate observations;
proc sort data=allcont nodupkey;
  by memname;
run;

data _null_;
set allcont;
where memname=: 'IN';
length ii $3 subject $6;
i+1;
ii=left(put(i,3.));
subject=substr(memname,3);
call symput('subj'||ii,subject);
call symput('subjcnt',ii);
run;

proc datasets library=work;
delete subjstat;

* Combine the subject control files;
%do i = 1 %to &subjcnt;
  append base=subjstat data=decntrl.in&&subj&i;
%end;
quit;
%mend cmbnsubj;
```

Using PROC CONTENTS becomes faster than SASHELP.VTABLE as the number of data sets increases. However, even PROC CONTENTS might become too slow as the number of data sets increases even more. If the PROC CONTENTS approach becomes too slow, then you might consider using operating-system-level commands, which can be specified by using the X statement.

Using the X statement

Many directory-based operating systems have operating-system-level commands that can be used to construct a list of files in a directory. The following version of %CMBNSUBJ replaces PROC CONTENTS with an X statement that contains the operating system DIR command (for Windows based systems).

```
%macro cmbnsubj;

* Determine the subjects, make a macro var for each;
* DECNTL.INxxxxxx data sets have one obs per subject;

* Create a list of all data sets in the libref DECNTL;
%let depath = %sysfunc(pathname(decntl)); ❶

x "dir &depath\in*.sd2 /o:n /b > d:\junk\dirhold.txt"; ❷

data null;
infile 'd:\junk\dirhold.txt' length=len; ❸
input @; ❹
input memname $varying12. len; ❺
length ii $3 subject $6;
i+1;
ii=left(put(i,3.));
subject=substr(memname,3,len-6); ❻
call symput('subj'||ii,subject);
call symput('subjcnt',ii);
run;

proc datasets library=work;
delete subjstat;

* Combine the subject control files;
%do i = 1 %to &subjcnt;
    append base=subjstat data=decntl.in&&subj&i;
%end;
quit;
%mend cmbnsubj;
```

- ❶ Determine the path of the *libref* of interest and store it in &DEPATH.
- ❷ The X statement is used with the system's DIR command to build a list of files in the &DEPATH directory. This list is routed to a text file (D:\JUNK\DIRHOLD.TXT), where it is stored for later use.
- ❸ The text file is identified using the INFILE statement. Because the lengths of the data set names may not be constant, the LENGTH= option is used to store the length of each data set's name in the variable LEN.
- ❹ A dummy INPUT statement is used to load the next record into the input buffer. This statement also assigns a value to LEN.
- ❺ The \$VARYING12. format is used to read the name of the data set. The variable LEN is used by the \$VARYING format to specify the actual length of each incoming record (data set name).

- ⑥ The subject number is extracted from the data set name by using the SUBSTR function. In this example, each data set name (MEMNAME) will be in the form of INxxxxx.SD2, where xxxxx represents the subject number (the number of digits, x, varies from 1 to 6). You know that the subject number will start in the third column and will not include the last four columns (.SD2). Because the length of the data set is stored in LEN, the number of digits occupied by the subject number will be LEN-6.

Although it was not an issue in the previous example, the Windows operating system allows directory names to contain embedded spaces. When this occurs, most DOS commands will fail unless the path is enclosed in double quotes. This can cause a particular problem for macro programmers as both single and double quotes will be needed and macro variables will need to be resolved.

The following definition of &DEPATH and the X statement from the previous example

```
%let depath = c:\my documents\mydata;
x "dir &depath\in*.sd2 /o:n /b > d:\junk\dirhold.txt";
```

would **not** work correctly. Placing the path in double quotes still enables the macro variable to expand; however, the entire DOS command must also be quoted for the X statement. The statements become

```
%let depath = c:\my documents\mydata;
x 'dir "&depath\in*.sd2" /o:n /b > d:\junk\dirhold.txt';
```

This combination, however, will **not** work, because the outer single quotes prevent the resolution of &DEPATH. In order to get the X statement to work, we need to mask the outer single quotes until after &DEPATH has been resolved and for this we can use a quoting function. The X statement becomes (for display purposes it has been broken into two lines so that the statement will fit on this page):

```
x %bquote(')dir "&depath\in*.sd2" /o:n /b >
    d:\junk\dirhold.txt%bquote(');
```

Using the PIPE device type

The FILENAME statement can be used to point to more than just a file. It supports a *device type* option that enables the user to specify the access method to be used when reading from the specified file(s). One of these device types is PIPE, which enables the user to pass operating system commands directly through the FILENAME statement to the operating system. SAS can then, in effect, use the *fileref* to directly point to the result of the command.

The previous version of %CMBNSUBJ uses an X statement to build a file containing the list of data sets of interest. This file is then read as data using an INFILE statement. The PIPE device type on the FILENAME statement can be used to combine these two operations.

```
filename list pipe
    %unquote(%bquote(')dir "&depath\*.sd2" /o:n /b%bquote('));
```

The %BQUOTE function is needed to mask the single quotes so that the macro variable &DEPATH can be resolved. Unlike in the X statement, shown in the previous example, the macro quoting must be removed prior to the execution of the FILENAME statement. This is accomplished with the %UNQUOTE function.

SEE ALSO

Widawski (1997a) also uses the `X` statement to build a list of files. Chen and Gilbert (2002) build a list of SAS programs and place them in a batch file for later execution. Long (2003) builds and concatenates a list of Zip files.

The PIPE device type is used by Mao (2001), Rook and Yeh (2001), and LeBouton and Rice (2000) to build a list of files under UNIX, and both Kelley (2003) and Wang (2003) use it in Windows examples. Johnson (2001) uses it to execute various operating system commands, and Chakravarthy (2003) uses it in a non-macro example.

Heaton-Wright (2003) uses `SASHELP.VTABLE` to build a macro variable. Troxell and Chen (2003) show examples of the use of both `SASHELP.VSTABLE` as well as `DICTIONARY.TABLES`.

Under Windows based operating systems, Dynamic Link Libraries (DLLs) can also be used for operating system operations. Roper (2001) discusses the use of Win32APIs to access DLLs.

Murphy (2003a) builds the list of data sets using the Output Delivery System (ODS).

10.1.6 Making a directory

%MAKEDIR

The following macro can be used to verify that a directory exists, and, if it does not already exist, to create it. The only parameter used by `%MAKEDIR` is the directory path to be checked.

```
%macro makedir(newdir);
  * Make sure that the directory exists;
  %let rc = %sysfunc(fileexist(&newdir)); ❶
  %if &rc=0 %then %do;
    %* Make the directory;
    %sysexec md &newdir; ❷
  %end;
%mend makedir;

%makedir(c:\tempzzz)
```

❶ The `FILEEXIST` function is used to see if the “file,” which in this case is actually a directory, exists. The return code from this function is 0 if the file is not found.

❷ A return code of 0 indicates that the directory does not exist and should be created. `%SYSEXEC` is used to execute the `MD` (make directory) command.

MORE INFORMATION

The `%SYSEXEC` macro statement is introduced in Section 5.4.4.

SEE ALSO

Dynder, Cohen, and Cunningham (2000) use a `%WINDOW` interface to generate a series of directories. The `FILEEXIST` function is also used by Lund (2003a) to check and establish directories.

10.1.7 Executing a series of SAS programs

%MAKERUNBAT

Execution of SAS programs from batch mode can sometimes have advantages even for users that usually execute interactively. Depending on how you would like to approach the problem you might want to either execute the programs from within a single SAS job or by using a series of SAS executions—one per SAS program.

The following macro, %MAKERUNBAT, creates a batch file with a separate execution of SAS for each SAS program in the specified directory. This batch file can then be scheduled for execution at some later time.

```
%macro makerunbat;
  options noxwait; ❶
  x dir "&path\*.sas" /o:n /b > c:\temp\pgmlist.txt; ❷

  data _null_;
    length sasloc path $75 excmd $150;
    retain sasloc "&sasloc" path "&path";

    * Determine the names of the SAS programs;
    infile 'c:\temp\pgmlist.txt' length=len; ❸
    input @;
    input saspgm $varying35. len;

    * Write out the batchfile;
    file "&path\runbat.bat"; ❹

    * Build the executable command; ❺
    excmd = '''||trim(sasloc)||' -sysin "'
           ||trim(path)||'\'||saspgm||'";
    put excmd;
  run;

%mend makerunbat;

%let path = C:\My Documents\macro2; ❻
%let sasloc = C:\Program Files\SAS Institute\SAS\V8\sas.exe;
%makerunbat
```

- ❶ The NOXWAIT option causes the DOS window spawned by the X statement to be closed automatically.
- ❷ The list of SAS programs in the &PATH directory is routed to a TXT file, which will be read as data. A similar technique is used in the third version of %CMBNSUBJ in Section 10.1.5.
- ❸ The TXT file, which contains the list of SAS programs, is read as data.
- ❹ The name and location of the batch file is designated. This macro writes the batch file bat in the same directory (&PATH) as the SAS programs.
- ❺ A variable (EXCMD) is built that contains the command that will execute SAS for each program. Notice that the path for both the SAS.EXE file, as well as the path for the SAS program are both enclosed in double quotes. This may not be required, however, under

windows; if a directory name contains an embedded space many DOS commands will not work correctly unless they are enclosed in double quotes.

- ⑥ The locations are specified in the macro variables &PATH and &SASLOC. A more flexible version of %RUNBAT might use &PATH as a macro parameter.

The resulting batch file (RUNBAT.BAT) contains one line for each SAS program in the ..\macro2 directory:

```
"C:\Program Files\SAS Institute\SAS\V8\sas.exe"
-sysin "C:\My Documents\macro2\bigperm.sas "
"C:\Program Files\SAS Institute\SAS\V8\sas.exe"
-sysin "C:\My Documents\macro2\DelMacVar.sas"
"C:\Program Files\SAS Institute\SAS\V8\sas.exe"
-sysin "C:\My Documents\macro2\makedir.SAS "
"C:\Program Files\SAS Institute\SAS\V8\sas.exe"
-sysin "C:\My Documents\macro2\mixedcase.sas"
"C:\Program Files\SAS Institute\SAS\V8\sas.exe"
-sysin "C:\My Documents\macro2\sleep.sas "
"C:\Program Files\SAS Institute\SAS\V8\sas.exe"
-sysin "C:\My Documents\macro2\SYMCHK5.SAS "
"C:\Program Files\SAS Institute\SAS\V8\sas.exe"
-sysin "C:\My Documents\macro2\Timeit.sas "
"C:\Program Files\SAS Institute\SAS\V8\sas.exe"
-sysin "C:\My Documents\macro2\wakepat.sas "
```

Since there is a separate invocation of SAS for each program, there will also be a separate LOG and LISTING for each program. While this may be an advantage, overall this approach may also have a disadvantage. If there are eight programs to be executed (as there are in the previous example), SAS will be opened and closed eight times. It might be better to instead open SAS once and execute the eight programs (of course, if you use something like PROC PRINTTO, this results in only one LOG and LISTING). The following variation of %MAKERUNBAT will create a batch file that executes SAS only once while still executing all eight programs.

Rather than building a single batch file with a series of calls to SAS, two files are created. A temporary SAS program is written that uses the %INCLUDE to execute each program of interest. It is this program that is referenced in the batch program.

```
%macro makerunbat2;
  options noxwait;
  x dir "&path\*.sas" /o:n /b > c:\temp\pgmlist.txt;

  data _null_;
    length sasloc path $75 excmd $150;
    retain sasloc "&sasloc" path "&path";

    * Determine the names of the SAS programs;
    infile 'c:\temp\pgmlist.txt' length=len;
    input @;
    input saspgm $varying35. len;

    if _n_=1 then do; ①
      * Write out the batchfile name;
      file "&path\runbat.bat"; ②
```

```

        * Build the executable command;
        excmd = ' ' || trim(sasloc) ❸
              || ' -sysin "c:\temp\masterpgm.sas"';
        put excmd;
    end;

    * Write out the Master INC program;
    file "c:\temp\masterpgm.sas"; ❹

    * Build the executable command; ❺
    excmd = '%inc ' || trim(path) || '\ ' || saspgm || ' ';
    put excmd;
run;

%mend makerunbat2;

%let path = C:\My Documents\macro2;
%let sasloc = C:\Program Files\SAS Institute\SAS\V8\sas.exe;
%makerunbat2

```

- ❶ The batch program will have only one line, will contain no data dependent text, and will start the execution of the SAS program ❹ that contains the %INCLUDE statements.
- ❷ The batch program is named and placed in the directory of interest.
- ❸ A single DOS command, which will execute the temporary SAS program, is written to the batch program.

```

"C:\Program Files\SAS Institute\SAS\V8\sas.exe"
-sysin "c:\temp\masterpgm.sas"

```

- ❹ A temporary SAS program will be written that will contain a series of %INCLUDE statements.
- ❺ An %INCLUDE statement is written for each program name in the variable SASPGM. The temporary SAS program will contain the following statements:

```

%inc "C:\My Documents\macro2\bigperm.sas ";
%inc "C:\My Documents\macro2\DelMacVar.sas";
%inc "C:\My Documents\macro2\makedir.SAS ";
%inc "C:\My Documents\macro2\mixedcase.sas";
%inc "C:\My Documents\macro2\sleep.sas ";
%inc "C:\My Documents\macro2\SYMCHK5.SAS ";
%inc "C:\My Documents\macro2\Timeit.sas ";
%inc "C:\My Documents\macro2\wakeupat.sas ";

```

MORE INFORMATION

Section 10.1.8 uses the %SYSGET function to retrieve the location of the SAS executable file.

SEE ALSO

Chen and Gilbert (2002) show a macro that builds a batch file that executes all SAS programs from a set of directories using a series of executions of SAS.

10.1.8 Using %SYSGET to access system variables

The %SYSGET macro function can be used to access environment or host system variables. Environment variables can be set either through the operating system or by SAS, and since these variables can provide a link between SAS and the operating system, they can be a valuable interface tool when writing macros. One common use of these variables is to associate locations (paths or directories) with a name. Usually the LIBNAME or FILENAME statements are used to create this association from within SAS, but if the association is created outside of SAS, the programs can become more location independent and may require less maintenance when moved from machine to machine.

In the last two examples in Section 10.1.7 a macro variable has been created that contains the path to the SAS executable file.

```
%let sasloc = C:\Program Files\SAS Institute\SAS\V8\sas.exe;
```

Obviously, if the program was moved to another computer (or operating system), this path might change, and it becomes the responsibility of the user to know how to make that change. Fortunately this value is also stored in the !SASROOT environment variable, and the %SYSGET function can be used to determine this value directly. The %LET statement becomes

```
%let sasloc = %sysget(sasroot)\sas.exe;
```

A number of other environment variables are available to the user; however, they vary by operating system, and can be additionally tailored when SAS is invoked. You can see the SAS environment variables that are currently set by viewing the value of the SET system option in SASHELP.VOPTIONS. Your SAS Companion and SAS Online Doc go into some detail on setting environment variables, either through SAS or through the operating system.

SEE ALSO

Levin (2001) and Lund (2001a, 2001b) use the %SYSGET macro function.

10.2 Controlling Your Output

Macros are often used to control the appearance and order of reports and listings. The examples in this section represent these types of macros. Section 9.1.2 contains an example that causes a series of procedures to be executed for each level of the BY variable rather than by cycling through the BY groups for each procedure.

SEE ALSO

Aboutaleb (1997a) discusses a macro that you can use to control output lines when you work with proportional fonts.

Several tips and techniques are suggested by Hayden (2002) that can be used to control titles, data selection, and report presentation. A macro used to control the labels in a trend chart is presented by Bessler and Pierri (2002).

Spotts (2002) uses a short macro to document his output, and Lund (2002) demonstrates a macro to build data dictionary tables.

A series of macros that can be used to enhance reports generated with a DATA _NULL_ step are presented by Ewing (2002). Automated column assignment in a DATA _NULL_ is discussed by Squire (1999).

Cheng (2000) adds standardized titles using a macro.

Maximizing the size of the available page for a PROC REPORT is discussed by Allen (2002).

10.2.1 Combining titles

%TF

David Shannon

The %TF macro shown in this section enables the user to combine up to three text strings on a single title or footnote. These strings can then be individually justified. You can use the macro to prevent overwriting the page number, and the entire title can be underlined.

SASHELP.VOPTION is used to determine the current LINESIZE (width of the current output page) option. This value is then used to determine the position of the various strings.

```

/*
|
|                                     TF
|
|
| The TF macro enables Titles or Footnotes to be specified in any
| combination of LEFT, CENTERED or RIGHT aligned in one line.
| The user can optionally stop the output page number from being
| overwritten. Specifying SLINE in the LEFT parameter will
| print a solid line the width of the page.
|
|
| Parameter | Default | Description
|
| TF=       | TITLE   | OPTIONAL: Parameter which defines if texts
|           |         | are TITLES or FOOTNOTES.
|
| N=        | 1       | OPTIONAL: Title/Footnote number. SAS
|           |         | currently allows up to 10 titles/footnotes.
|
| Left=     |         | OPTIONAL: Text to be left aligned. If you
|           |         | specify SLINE then a solid line will be
|           |         | drawn the width of the linesize.
|
| Centre=   |         | OPTIONAL: Text to be centred on the page.
|
| Right=    |         | OPTIONAL: Text to be right aligned.
|

```



```

%* Determine FROM and TO positions of texts and spaces          *;

Data _null_;
Set SASHELP.VOPTION(Where=(optname='LINESIZE')); ❸
Length tol from2 to2 from3 gap1 gap2 settn 8.;
settn=(input(setting,??best.)-3);
If Ucase("&LEFT")="SLINE" Then
Do;
  %Let lwas=&left; ❺
  Call symput('LEFT',Left(Repeat('_',settn))); ❻
End;
Left=Trim(Symget(Resolve('Left')));
Centre=Trim(Symget('Centre'));
Right=Trim(Symget('Right'));

If Left^="" Then
Do;
  Tol=Length(Trim(left));
End;
Else Do;
  Tol=0;
End;

If Centre^="" Then
Do;
  From2=Floor( Floor(settn/2) - Length(Trim(centre))/2);
  To2=(from2 + Length(Trim(centre)))-1;
End;
Else Do;
  From2=Floor(settn/2);
  To2=Floor(settn/2)+1;
  Call Symput('Centre',repeat(byte(32),1));
End;

If Right^="" Then
Do;
  From3=( settn+1 - Length(Trim(right)))-1;
  If (%Eval(&N)=1 AND upcase(substr("&Pnum",1,1))='Y') then
    From3=from3-4;
End;
Else Do;
  From3=settn+1;
End;

gap1=( (from2) - (tol) -1 );
gap2=( (from3) - (to2) -1 );

If gap1 gt 0 then Call Symput('Gap1', repeat(byte(32),gap1));
Else %let gap1=;; ❹
If gap2 gt 0 then Call Symput('Gap2', repeat(byte(32),gap2));
Else %let gap2=;; ❹

%If %Ucase(&PNum)=YES %Then Call Symput('Gap3', repeat(byte(32),3));
%Else Call Symput('Gap3',repeat(byte(32),1));

```

```

If ((to1 ge from2) AND centre ne '') then
Do;
  PUT "ERROR: Centre aligned text will overwrite left-aligned text.";
  PUT "ERROR: Either shorten text or increase linesize";
  Call Symput ('ERRCOD2','1');
End;

If ((to2 ge from3) AND right ne '') then
Do;
  PUT "ERROR: Right aligned text will overwrite centre-aligned text.";
  PUT "ERROR: Either shorten text or increase linesize";
  Call Symput ('ERRCOD2','1');
End;

Run;

%*****;
%* Check for error status, if true jump to end of macro          *;

%If &ERRCOD2=1 %then %Goto EOM;

%*****;
%* Create Title/Footnote                                         *;

%If %Ucase(&LWAS)=SLINE %Then
%Do;
  &TF&N " &LEFT ";
%End;
%Else %Do;
  &TF&N " &LEFT&GAP1&CENTRE&GAP2&RIGHT&GAP3.";
%End;
%EOM:
Options Mprint Mlogic Symbolgen;
%Mend TF;

```

- ❶ Check for valid title numbers.
- ❷ Check to see that the page number option is valid.
- ❸ SASHELP.VOPTION view is used to determine system option settings.
- ❹ This is an example of code that was written with a style different than has been described elsewhere in this book. The intent of the code is to initialize the macro variables &GAP1 and &GAP2. This takes place, however, because the %LET is executed before the DATA step, the conditional logic does not apply. The %LET statements are always executed before the DATA step is executed. The second of the double semicolons closes what is effectively a null ELSE statement (Else;). If it had been necessary to conditionally assign a null value to &GAP1, the ELSE statement would have been followed not with a %LET but rather with the executable CALL SYMPUT:

```

If gap1 gt 0 then Call Symput('Gap1', repeat(byte(32),gap1));
Else Call Symput('Gap1','');

```

- ❺ As in ❹, an attempt is made to conditionally initialize the macro variable; however, the %LET is executed even before the DATA step starts to execute.
- ❻ The first argument in the REPEAT function is intended to be the symbol used to draw a straight line. This symbol may be different depending on your operating system and the type of line that you wish to draw.

SEE ALSO

Andresen (1997) presents a macro that will split and wrap a long text variable when creating a report using a DATA _NULL_ step. Chen (2001) uses the DATA step to form a Table of Contents. Larsen (2001) gives a quick description of a short macro that centers text in the DATA _NULL_ step through the use of the %LENGTH and %EVAL functions.

10.2.2 Renumbering listing pages

%REPORT**Paul Kairis****NIKH Corporation**

(Originally published in *The VALSUG News: Newsletter of the Valley of the Sun SAS Users Group*, 1996)

Typically, when running a series of procedures that produce output listings, the pages are numbered consecutively starting with one. You can use the system option PAGENO to reset the page numbers **between** PROC and DATA steps but not **within** a step, such as between BY levels. The following macro creates WHERE strings based on the BY variables. A separate PROC REPORT is then called for each combination of BY variables with the PAGENO reset between calls.

```
%macro report(dsn);
proc summary data =&dsn nway; ❶
  class tpcorder sysname bldgname grpname;
  output out=calllist;
run;

data _null_;
  set calllist end=eof;
  numcall+1; ❷
  * Build macro variable(s) to hold WHERE clause;
  call symput('call' || left(put(numcall,3.)), ❸
    "tpcorder=" || tpcorder ||
    "'and sysname=" || sysname ||
    "'and bldgname=" || bldgname ||
    "'and grpname=" || grpname || "'");
  if eof then call symput('numcalls',left(put(numcall,3.))); ❹
run;

%do i = 1 %to &numcalls; ❺
  options nobyline pageno=1;
  proc report data=&dsn nowindows headline;
    by tpcorder sysname bldgname grpname;
    where &call&i; ❻
  run;
%end;
%mend report;
```

❶ PROC SUMMARY is used to create a data set (CALLLIST) that contains the unique combinations of the BY variables.

❷ The number of unique combinations of the BY variables is stored in the variable NUMCALL

- ③ The WHERE clause is built by concatenating the names of the BY variables with their values, which in this example are all character. In Chapter 9, “Writing Dynamic Code,” several examples created individual lists of macro variables rather than constructing the final value and saving it into a single list (&&CALL&I).
- ④ Store the overall number of BY group combinations in the macro variable &NUMCALLS.
- ⑤ The macro %DO loop executes the PROC REPORT once for each BY group combination.
- ⑥ The WHERE statement uses the clause that is stored in &&CALL&i to subset the data prior to the execution of the PROC REPORT.

MORE INFORMATION

The example in Section 9.1.2 and the solutions to Exercise Questions 6.2 and 6.3 utilize a similar approach by creating data subsets.

SEE ALSO

Felty and Nicholson (1999) discuss a macro that produces page numbers in a variety of styles and report types (including DATA _NULL_). Another macro that places page numbers is discussed by Peterson (1999).

10.2.3 Coordinating titles (or footnotes)

At times you may need to write a macro that will use titles, but you might not know what the next available title number is. Of course using an arbitrary number could wipe out important titles, so we need to be able to determine which titles have already been defined. There are a couple of ways to do this. The SQL DICTIONARY.TITLES table and the SASHELP.VTITLE view both have one row for each title and footnote that are currently defined. Each has the column TYPE that takes on the values of ‘T’ and ‘F’ for titles and footnotes, respectively. Each also has a column for the title or footnote NUMBER.

The following SQL step uses the DICTIONARY.TITLES table to determine the largest title number that is currently in use. This number is stored in the macro variable &T.

```
proc sql;
  reset noprint;
  select max(number) into :t
    from dictionary.titles
     where type='T';
quit;
```

A DATA _NULL_ step can also be used to obtain the same information.

```
data _null_;
  set sashelp.vtitle(keep=type number
                    where=(type='T'));
  retain max 0;
  max = max(max,number);
  call symput('t',left(put(max,2.)));
run;
```

Once you have determined the largest defined title, you can begin generating additional titles in your macro. In the following code I want to be able to add two new titles. &T has been created using one of the two methods shown above. There is the possibility that more than eight titles have been defined; if this is the case I will be forced to replace those titles with new ones.

```
%if &t > 8 %then %let u=9;
%else %let u = %eval(&t + 1);
%let v=%eval(&u+1);

title&u 'First custom title';
title&v 'Second custom title';
```

MORE INFORMATION

A next available title is also determined in the example in Section 9.5.2.

10.3 Adapting Your SAS Environment

As we write macros that are more sophisticated, we often need for them to be able to detect and work with the environment in which they run. They might need to change options and then later restore the original values. SAS accesses locations on the operating system through *librefs* and *filerefs* and these points of reference are often maintained from within the macro language. When formats are stored permanently, the catalog that contains them must be made available. The process of creating formats, saving the formats, and making them available can also be controlled through macros.

10.3.1 Maintaining system options

It is not unusual for you to need to write a macro that either needs to determine the current value of a system option or needs to reset an option to a previous value. In either case your macro will need to be able to access the list of current OPTION settings. There are a couple of ways to do this.

%STOREOPT

The data view SASHELP.VOPTION can be used to determine the current option settings. This view takes the form of one observation per option with the columns of OPTNAME and SETTING holding the option name and value, respectively. The macro %TF in Section 10.2.1 uses this view to determine the setting for the LINESIZE option. The following macro, %STOREOPT, creates a list of global macro variables that will contain the current option settings of interest:

```
%macro storeopt(oplist);
%local len i tem;
%if &oplist ne %then %do;
  %let oplist = %cmpres(&oplist);
  %global &oplist; ①
  %let len = %length(&oplist); ②
  /* Establish a var with opt names quoted;
```

```

%let tem = ; ❸
%do i = 1 %to &len; ❹
    %if %substr(&oplist,&i,1) = %str( ) %then %do;
        /* Add quotes around the words in the list;
        %let tem = &tem%str(%', %'); ❺
    %end;
    %else %let tem = &tem%UPCASE(%substr(&oplist,&i,1)); ❻
%end;
%let tem = %bquote(%str(%')&tem%str(%')); ❼

* Retrieve the current option settings;
data _null_;
    set sashelp.voption;
    if optname in:(%UNQUOTE(&tem)) then do; ❽
        call symput(optname, left(trim(setting)));
    end;
run;

%end;
%mend storeopt;

%storeopt(linesize pagesize obs mlogic date) ❾

```

- ❶ Create a list of global macro variables with the same names as the system options whose value is to be stored.
- ❷ Determine the number of characters in the list of option names.
- ❸ A temporary variable will be used to hold the quoted option names. The quoted names will be used with an IN: operator while searching for options of interest ❽.
- ❹ Step through the list character by character searching for blanks.
- ❺ When blanks are found they are replaced with ' ', thus `linesize pagesize` becomes `linesize' 'pagesize`.
- ❻ Nonblank values are transferred to &TEM.
- ❼ Leading and trailing quotes are added to &TEM.
- ❽ Only the requested options are read from SASHELP.VOPTION, and their respective settings are stored in a macro variable of the same name.
- ❾ The macro is called with a list of one or more system options that are to be stored.

Once the option values have been collected they can be restored after they are subsequently modified. To restore the option settings one would use the following `OPTION` statement:

```
options ls=&linesize ps=&pagesize obs=&obs &mlogic &date;
```

In `%STOREOPT` the `SASHELP.VOPTION` view must be created and read each time the macro is called. Also it does not contain any information about the form of the option (keyword or on/off), nor does it contain information about the settings of the graphics options (controlled through the `GOPTIONS` statement). These shortcomings can be rectified through the use of the `GETOPTION` data step function, which is discussed next in the macro `%HOLDOPT`.

%HOLDOPT**Pete Lund****Looking Glass Analytics**

This macro also saves the current settings of system options; however, it takes a very different approach than %STOREOPT. Not only does each execution of %HOLDOPT save just one option's value, but the value is determined using the GETOPTION function.

```

/*****
/* Macro: HoldOpt */
/* Programmer: Pete Lund */
/* Date: September 2000 */
/* Purpose: Holds the value of a SAS option in a macro */
/* variable. The value can then be used to reset options */
/* to a current value if changed. */
/* */
/* Parameters: */
/* OptName - the name of the option to check */
/* OptValue - the name of the macro variable that will */
/* hold the current value of the option */
/* The default name is made up of the word */
/* "Hold" and the option name. For */
/* example, if OptName=Notes, OptValue */
/* would equal HoldNotes */
/* Display - Display current value to the log (Y/N) */
/* The default is N */
/* */
*****/
%macro HoldOpt(
    OptName=,      /* Option to hold value */ ❶
    OptValue=XX,   /* Macro var name to hold value*/ ❷
    Display=N);    /* Display value to the log */

    %if %substr(&sysver,1,1) eq 6 ❸
        and ((%length(&OptName) gt 4 and &OptValue=XX)
            or %length(&OptValue) gt 4) %then %do;
        %put WARNING: Default variable name of Hold&OptName is too long for
V&sysver..;
        %put WARNING: Please specify a shorter name with the OptValue=
macro parameter.;
        %goto Quit;
    %end;
    %if &OptValue eq XX %then %let OptValue = Hold&OptName; ❹
    %global &OptValue; ❺
    %let &OptValue = %sysfunc(getoption(&OptName)); ❻
    %if &Display eq Y %then
        %put The current value of &OptName is &&&OptValue;
    %Quit;
%mend;

```

❶ The option name of interest is stored in &OPTNAME.

❷ &OPTVALUE will hold the name of the macro variable that will contain the system option value.

- ❸ For Version 6 and before, which does not allow names of lengths greater than 8, the default naming of the macro variable may be limiting.
- ❹ If a name for the macro variable is not otherwise supplied, the name will be the option name preceded by HOLD.
- ❺ Make the macro variable global.
- ❻ The GETOPTION function is used to retrieve the system option whose name is stored in &OPTNAME.

The following three calls to %HOLDOPT will save the values of the DATE, OBS, and LINESIZE options:

```
%HoldOpt (OptName=date)
%HoldOpt (OptName=obs)
%HoldOpt (OptName=ls, OptValue=myls)
```

If these options are changed in the user's program, they can be reset to their original values with the following statement:

```
options &holddate obs=&holdobs ls=&myls;
```

Both of the previous macros require the user to know if a given option is a keyword-style option, such as `ls=96`, or a toggle option such as `DATE` or `NODATE`. When the second argument of the DATA step GETOPTION function is `KEYWORD`, the function returns the value in keyword format (as needed). In %HOLDOPT the code at ❸ could be rewritten as

```
%let &OptValue = %sysfunc(getoption(&OptName,keyword));
```

If this change is made in %HOLDOPT, one could write the following in order to reset the DATE, OBS, and LS options:

```
options &holddate &holdobs &myls;
```

MORE INFORMATION

The GETOPTION function is also used in %FMTSRCH in Section 10.3.2.

SEE ALSO

The %HOLDOPT macro is discussed further in Lund (2001a and 2003b).

10.3.2 Building and maintaining formats

Customized formats, which can take on a variety of forms, can be built with PROC FORMAT using several different approaches. They can be built by hand using procedure statements, such as the `VALUE` statement, or they can be based on relationships in a SAS data set. Once created, these formats can be stored temporarily or permanently, and when stored permanently, the `FMTSEARCH` option can be used to designate the various format catalogs that can be searched for the customized formats. The following macros work with this process.

Checking the format search path

%FMTSRCH

The list of libraries for SAS to search, when looking for catalogs that might hold user-defined format definitions, is set using the FMTSEARCH system option. The following macro checks whether or not the library specified in its first parameter is on the search path. If it is not it can optionally be added.

```
%macro fmtsrch(lib,add);
  /* Check to see if a libref is in the format search path
  /* lib   libref to check
  /* add   If &add is not null, then add the libref, if
  /*       it is not already in the fmtsearch value;

  /* Macro returns.
  /* &add is null
  /*      0   if &lib is not on path
  /*      >0  if &lib is on path
  /* &add is not null
  /*      Option statement to add &lib if it is not on path
  /*      null if &lib is on path;

  %local optval insrch;

  /* Determine if the library is on the fmt search path;
  %let optval = %sysfunc(getoption(fmtsearch)); ❶
  %let insrch = %index(&optval,%upcase(&lib)); ❷

  %if &add ne %then %do; ❸
    /* Add &lib to format search,
    /* if it is not already on the path;
    %if &insrch = 0 %then %do; ❹
      /* Remove the trailing close parenthesis;
      %let optval = ❺
        %substr(&optval,1,%eval(%length(&optval)-1));
      /* Add the library;
      options fmtsearch=&optval &lib); ❻
    %end;
  %end;
  %else &insrch; ❼
%mend fmtsrch;
```

- ❶ Retrieve the current value of the FMTSEARCH option. The returned value is enclosed in parentheses. The value may not include the default WORK and LIBRARY librefs.
- ❷ Check if &LIB is currently on the list of libraries to be searched (&INSRCH is > 0 if &LIB is found and 0 if it is not). This approach might not work if &LIB is a member of a concatenated library.
- ❸ When &ADD is not null, add &LIB to the search path if it is not already on the path.
- ❹ &LIB is not on the current format search path and it should be added.
- ❺ The closing parenthesis is removed from &OPTVAL in preparation for appending the new library in &LIB.

- ⑥ The OPTIONS statement that will rebuild the format search path is specified. &OPTVAL already has the open parenthesis.
- ⑦ When &ADD is null the macro returns the result of its check whether or not &LIB is in the format search path.

Assuming that the current path contains only the WORK library, for example:

```
options fmtsearch=(work);
```

then the following calls to %FMTSRCH return

```
%fmtsrch(work)           ➡➡  1
%fmtsrch(sasclass)       ➡➡  0
%fmtsrch(sasclass,1)     ➡➡  option fmtsearch=(WORK sasclass);
%fmtsrch(work,1)         ➡➡  <nothing is returned>
```

Building a format from a data set

%MKFMT

When a data set contains two or more paired columns it can be advantageous to use them to build a format. Preparation of the data set for use by PROC FORMAT is not difficult, but it requires that the user knows the various options and variable names used by the procedure. The following macro performs this conversion and builds the format:

```
%macro mkfmt(lib=, dsn=, fmtname=, from=, too=);

  /* When the fmt name is unspecified,
  /* use the incoming var name (FROM) as the format name;
  %if &fmtname= %then %let fmtname=&from; ①

  * Eliminate duplicate values of &from;
  proc sort data=&dsn(keep=&from &too)
            out=temp
            nodupkey;
    by &from;
  run;

  data control(keep=fmtname start label type); ②
    set temp(rename=(&from=start &too=label))
      length type $1 fmtname $8;
    retain fmtname "&fmtname" type ' ';

    * Determine the format type;
    If _n_=1 then type = vtype(start); ③
  run;

  proc format
    %if &lib ne %then library=&lib; ④
    cntlin=control;
  run;

  %if &lib ne %then %fmtsrch(&lib,1); ⑤
%mend mkfmt;
```

- ❶ When a format name is not provided by the user, the name of the START (&FROM) variable will be used.
- ❷ These variables (at a minimum) are needed by PROC FORMAT:

FMTNAME	specifies the name of the format to be created.
START	specifies the value that is to be converted.
LABEL	specifies the value that START will be converted to.
TYPE	specifies a numeric (N) or character (C) format.

While TYPE is not strictly required by PROC FORMAT, it is needed in this macro because character format names will not start with \$.

- ❸ The type of format (numeric or character) is determined by the type of the START variable.
- ❹ When &LIB is specified, the catalog &LIB.FORMATS is used to store the format.
- ❺ If this library is not already on the format search path it is added using the %FMTSRCH macro (also shown in Section 10.3.2).

The following call to %MKFMT creates a permanent format in SASCLASS.FORMATS using the variables CLINNUM and CLINNAME from the data set SASCLASS.CLINICS. The format name will be CLNAME and it will map clinic numbers to clinic names.

```
%mkfmt(lib=sasclass,dsn=sasclass.clinics,
       from=clinnum,too=clinname,
       fmtname=clname)
```

SEE ALSO

Lopez (1998) discusses some macro tools that can be used to dynamically create SAS formats. Pete Lund (2003c) discusses a more sophisticated macro for the management of the FMTSEARCH= option.

10.3.3 Working with libraries and directories

There are a number of DATA step functions that can be utilized within the macro language to establish, check, and clear libraries. These include the following:

pathname	determines the path or location pointed to by a given <i>libref</i> or <i>fileref</i> .
libname	establishes or clears a <i>libref</i> .
libref	determines if a given <i>libref</i> has been established.
filename	establishes or clears a <i>fileref</i> .
fileref	determines if a <i>fileref</i> has been established.

%MKLIB

This macro can be used to create a library with a supplied engine. It can be passed either the name of an existing library or a path. Sometimes you know an existing *libref*, but not its corresponding location, and you need to establish a second *libref* with a different engine pointing to the same location.


```

%macro mklib(lname=, engine=default, lloc=);
%* Establish a libref
%* lname      new libref name
%* engine     engine type (optional)
%* lloc       path or location
%*           may be an existing libref
%*;

%local rc;
%* Clear libref if it exists;
%if %sysfunc(libref(&lname))=0 ❶ %then
    %sysfunc(libname(&lname)); ❷

%* Determine if &lloc already is a libref;
%if %sysfunc(libref(&lloc))=0 %then ❸
    %let lloc = %sysfunc(pathname(&lloc)); ❹
%let rc = %sysfunc(libname(&lname,&lloc,&engine)); ❺
%put &rc %sysfunc(sysmsg());

%mend mklib;

```

- ❶ The LIBREF function returns a 0 when its argument is an established *libref*.
- ❷ The LIBNAME function clears a *libref* when no other arguments are included.
- ❸ Determine if &LLOC contains an existing *libref*.
- ❹ For an existing *libref*, &LLOC contains its physical location (path), which is retrieved using the PATHNAME function.
- ❺ &LLOC now contains a physical path, and the new library (&LNAME) is established.

Assuming that the following LIBNAME statement is successful, either of the following calls to %MKLIB will establish the *libref* of TEMP with the V6 engine.

```

libname test 'c:\temp';

%mklib(lname=temp, engine=v6, lloc=test)
%mklib(lname=temp, engine=v6, lloc=c:\temp)

```

MORE INFORMATION

The FILEEXIST function is used in the %MAKEDIR macro in Section 10.1.6. The PATHNAME and FILEEXIST functions are also used in the examples of %SYSFUNC in Section 7.4.2.

10.4 Coordinating with the Output Delivery System (ODS)

The Output Delivery System, ODS, can be used to produce and control graphs, charts, and reports. These can be redisplayed through a number of means, including the use of standard internet browsers. In the production environment this often means the generation of a large number of graphs and reports. Of course, when the numbers are large, it becomes problematic to

locate, name, point to, and redisplay all of these reports, graphs, and tables in such a way as to be fairly easy for the user and fairly automatic for the programmer.

This is of course the forté of the macro language, which when used with ODS can be used not only to automate, but to manage the process as well.

Most of the code that is shown in Section 10.4 is taken from longer macros and programs, and is included here to demonstrate technique rather than complete examples.

SEE ALSO

Several macros that are used to generate styles and templates are presented by Bessler (2003).

10.4.1 Why we might need to automate with macros

The graphs and reports must be named and placed in an accessible location. When programs are rerun and graphs are updated or regenerated, the naming conventions must be well enough defined so that the correct graphs are replaced. The longer naming conventions of Version 8 make this easier to do; however, constraints such as limitations within the naming of graphical catalog entries must be dealt with.

Once created, the graphs and reports must then be placed in a location that is automatically determined within the production process, and if the location does not already exist, it must be created. This automated process requires a standardized and well thought out naming convention.

In some of the examples below, a series of survival analyses have been conducted for a variety of different models. Because each of over 150 models will generate as many as 30 graphs and tables, the naming conventions and locations were determined to a large extent by a model designation code.

SEE ALSO

A macro used to control the labels in a trend chart is presented by Bessler and Pierri (2002).

Lund (2002) coordinates data dictionaries in a drill-down HTML report. Carpenter and Smith (2002) discuss the use of macro variables and macro %DO loops to name, coordinate, and link listings and graphs.

The design of directory structures used to facilitate large and/or dynamic applications is presented in Carpenter and Smith(2001b).

Leprince and Li (2003) use macros to coordinate tabular output with SAS/GRAPH plots. These are then written to PDF files using ODS.

VCOLUMN is used by Goddard (2003) in an example that writes RTF files using ODS.

10.4.2 Controlling ODS locations

The physical location for the various files must be controlled as part of the automated process. You can determine if a specific location (in this case a directory) exists through the use of the FILEEXIST function. Since for discussion purposes we are operating in the macro environment, %SYSPF is used to call FILEEXIST, and if needed %SYSEXEC is used to create the new directory.

```

...code not shown...
* Make sure that the hazardratios
* directory exists;
%let rc = %sysfunc(fileexist("&drive\&project\hazardratios"));
%if &rc=0 %then %do;
    %* Make the directory;
    %sysexec md &drive\&project\hazardratios;
%end;
...code not shown...

```

Notice that the location of the directory and portions of the path are controlled by macro variables (&DRIVE and &PROJECT). These global macro variables are set up by the application to make sure that all files and directories can be located by the application. These same macro variables are used whenever it is necessary to point to the primary or upper portion of any path within the application.

10.4.3 Using ODS to control the destination

HTML files will be generated through a variety of methods within the application. Depending on how they are built, these files can be used to point to other HTML files, GIF files, or other tables and graphs. Tables stored as HTML files can be used as indexes, which can be used to point to another level of graphs, tables, reports or even to another level of indexes. HTML files are also built by SAS/GRAPH drivers (primarily HTML and WEBFRAME device drivers) to “wrap up” graphs in the form of GIF files.

The ODS HTML statement is used to name and point to these HTML files. The PATH= option designates the directory (notice the use of the same macro variables). The URL=NONE is used to build relative links within the HTML file. This enables you to move the completed HTML files to another location such as a server. The BODY= option specifies the name of the file itself.

```

ods html path = "&drive\&project\hazardratios" (url=none)
        body = "HR_&rptgrp&i._&hrggrp&i...html";

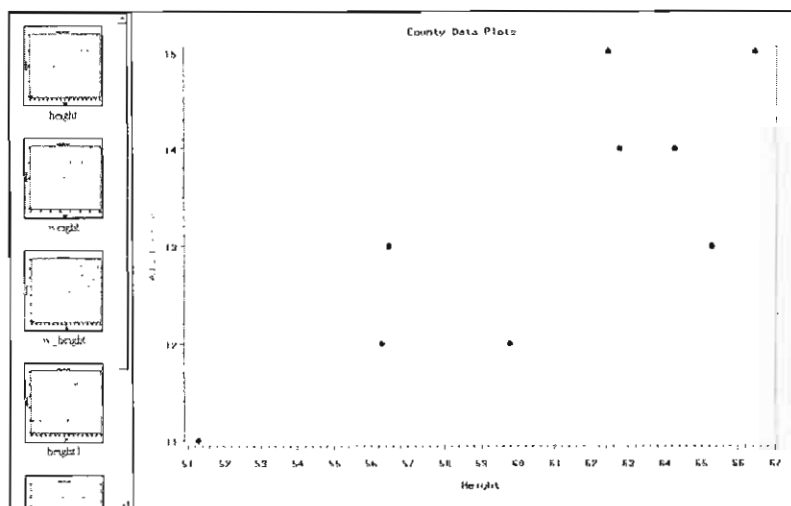
```

In this example, macro variables in the general form of &&VAR&I have been built based on values within the data. These macro variables in turn are used to determine the name of the HTML document. In this particular case, the ODS statement above resides inside of a macro %DO loop. After macro variable resolution, the name will resolve to something like ‘HR_7_3.html’. By basing the names of the files on data values, the program will not need modification when new report or data combinations are added.

10.4.4 Graphics devices

Several of the newer graphics device drivers are especially appropriate when you are publishing to the Web. Most of these produce GIF files, and indeed there is even a GIF device. The problem with building a series of individual GIF graphics files is that the user must browse them individually. The graphics devices HTML and WEBFRAME solve this problem by wrapping up a series of GIF graphs in a single HTML document. Like one-stop shopping, the reader now needs to browse only one file to see all the graphs.

The following low-resolution screen capture shows a portion of a display generated with the WEBFRAME device:



To use this device, use a FILENAME statement to point to a directory (not to a file!). The device will then create a series of HTML and GIF files in this directory. One of these will be named INDEX.HTML, and this is the one that your application should have your user browse.

```
filename regplt "&drive\&project\results\regscrt";

goptions device=webframe gsfname=regplt;
symbol1 c=blue v=dot r=45;
axis1 label=(a=90 f=simplex c=blue 'Adjusted Rate');
title2 'County Data Plots';
proc gplot data=sasclass.cnty;
...code not shown...
```

Obviously the WEBFRAME device is most useful when a series of graphs are to be displayed. One of its limitations, however, is that it cannot be used across PROC step boundaries. This means that you would be unable to combine the results of a PROC GPLOT with a PROC GCHART within the same display.

Fortunately we can get around this problem by utilizing the NODISPLAY graphics option and saving all the graphs of interest as graphics entries (entry type of GRSEG) in a catalog. These graphs can then all be redisplayed using PROC GREPLAY in a single step. This process is described in detail in the following section.

10.4.5 Using WEBFRAME with GRSEG catalog entries

Whenever a SAS/GRAPH procedure creates a graph, an entry is also created in a catalog. Unless otherwise specified, the catalog is WORK.GSEG and the entry type is GRSEG. The name of the entry can be specified by the user, but otherwise it takes on the value of the name of the procedure that generated the graph. A PROC GPLOT, therefore, would create a catalog entry named WORK.GSEG.GPLOT.GRSEG. If a second plot is generated by GPLOT, the entry name becomes GPLOT1 and so on.

When the objective is to display a number of graphs from different procedures using a single WEBFRAME, each graph needs to be stored as a catalog entry. Since the storage is temporary,

WORK catalog can be used. If each graph in the work catalog is to be redisplayed, the catalog should be cleared first. The following PROC DATASETS will delete the graphics catalog (GSEG) so that it can be re-created anew:

```
*clear the default graph catalog;
proc datasets library=work mt=cat nolist;
  delete gseg;
quit;
```

The WEBFRAME device utilizes the name of the graphics entry as the label for the thumbnail, so the default naming behavior is less than desirable, such as gplot, gplot1, gplot2, and so on. Fortunately the entry name can be specified by the user through the use of the NAME and DESCRIPTION options. In the following example, one plot is generated for each call to GPLOT. The macro variables used in the NAME= option provide a unique name for the graph.

```
goptions nodisplay;
proc gplot data=pltdat ;
  plot hrplt*Q=pltvvar/
    skipmiss nolegend
    vaxis=axis1
    name = "H&&cmod&i&&regim&i"
    des = "&&model&i &&regim&i";
run;
quit;
```

Since there is no reason to display the graph at this time, the graphics option NODISPLAY is used, and the graph is saved in a catalog (in this case WORK.GSEG) for redisplay later when the WEBFRAME device can be used. Also notice that although Version 8 allows longer variable names, the catalog entry can still have no more than eight characters, so care must be taken when constructing a value for the NAME= option. In the following step, all of the graphics entries with names starting with "H&&cmod&I" are selected for plotting by placing a list of their names into the macro variable &SUMPLOTS.

```
* create macro variable containing a list of the plots;
data _null_;
  set sashelp.vcatalog;
  where libname = 'WORK'
    and memname = 'GSEG'
    and objname =: "H&&cmod&i";
  length sumplots $20000;
  retain sumplots ' ';
  sumplots = trim(sumplots)||' '||trim(objname);
  call symput('sumplots', trim(left(sumplots)));
run;
```

If the eight characters in the name are not enough to create a retrievable name, the DESCRIPTION= option can also be used to store information. This information can also be used to determine subsets in much the same way as the name was used in the previous DATA _NULL_ step.

The procedure GREPLAY is used to redisplay the selected graphs. The REPLAY statement specifically names the plots to be redisplayed by using the list of names stored in the macro variable &SUMPLOTS.

```

filename webloc "&drive\&project\hazardratios\HR&&cmod&i";
* Wrap the various graphs with html;
goptions dev=webframe display gsfname=webloc;
proc greplay igout=gseg nofs;
  replay &sumplots;
run;
quit;

```

Remember that among all the files generated within the designated directory, there will always be one named INDEX.HTML. Have your user browse this file to see all of the individual graphs and thumbnails.

10.4.6 Building an index with PROC PRINT

In the previous example, a series of graphs were built using the WEBFRAME device. Each execution of the GREPLAY procedure wrote another series of graphs to another directory. The user can now display all of the graphs in a given subdirectory simply by going to that directory, finding the INDEX.HTML file, and browsing it. This is still quite a challenge if there are a lot of directories and/or if there are many graphs within any given directory. What we next need is a single file that, when browsed, will direct the user to the correct index file and directory combination by a simple click of the mouse. We will create this file with a simple PROC PRINT.

The data set to be printed must contain a variable whose value is a valid HTML anchor tag that references the name of another file. The form of this link (GRPREF ❶ in the following DATA step) includes the value to be displayed (the report group in this case, which is stored in CRPTGRP ❷), and the name or location of the file to branch to (the link is stored in the variable MYLINK ❸).

```

data prep2;
  set prepl;
  length grpref mylink $150 crptgrp $3;
  crptgrp= trim(left(put(rptgrp,3.)));
  mylink = "\\&project\hazardratios\hr" ||
    trim(crptgrp) || '\index.html';
  grpref❶= "<a HREF=" || trim(mylink)❸ ||
    '>' || trim(crptgrp)❷ || '</a>';
  label grpref = 'Report Group';
run;

* Define location and index name for the new index;
ods html path="&drive\&project\&outloc" (url=none)
  body="&indexname..html";
ods listing close;

proc print data=prep2 label noobs;
  var grpref groupdesc;
  title1 "&title Control File: &indsn";
run;
ods html close;

```

When the file generated by the PROC PRINT is displayed, only the portion of the variable GRPREF that came from the variable CRPTGRP will be displayed. When the user selects one of these values, the corresponding INDEX.HTML file, which was built by WEBFRAME, will be displayed.

In a more complex study or analysis, presenting more than one level of PROC PRINT indexes may be needed. Perhaps the first level selects general types of reports or general areas of interest. A selection here then points to a secondary index that enables the reader to further select the set of graphs of interest. This process effectively creates a hierarchical Table of Contents that eventually passes the reader to the graphs of interest.

10.4.7 Creating drill-down graphs and charts

Not only do we want to be able to use the tools discussed above to locate and display a specific graph, we might also want to use the graph itself to point to more information. As we browse the graph created using the techniques shown above, we want to be able to create “hot zones” that allow us to drill down through a portion of a graph by clicking on a bar, line, or symbol, so that we can then display and browse another related graph or table.

The examples in this section use data from an air pollution study conducted in California in 1988. Data were collected at each of three stations throughout the year, and this data set contains the monthly averages.

In the example below, both a histogram and a scatter plot are generated as GIF files. Each of these graphs contains information that is STATION specific. When the user is browsing the graph and the cursor is moved over a station specific portion of the graph, a *hot zone*, the cursor changes to a pointer. The pointer indicates that clicking on that hot zone will open another file or graph. In this case the newly opened file will be to a table of that station’s data created by PROC PRINT.

First we create a PROC PRINT for each value of the variable STATION. Notice that the name of the HTML file, the TITLE, and the value of the variable STATION in the WHERE clause all contain the name of the station (AZU in this example).

```
* List the AZUSA data;
ods html path="&drive\&project\figures" (url=none)
      body='azu.html';
proc print data=sasclass.ca88air (where=(station='AZU'))
      noobs;
  var month co o3 no3 tem;
  title1 'AZU Pollution';
  footnote;
run;
ods html close;
```

In the production environment it is likely that the list of stations will be stored in a macro array. The PROC PRINT can then be placed in a %DO loop. This enables the automated generation of the secondary files as shown below:

```
%do i = 1 %to &stacnt;
* List the &&sta&i data;
ods html path="&drive\&project\figures" (url=none)
      body="&&sta&i...html";
proc print data=sasclass.ca88air
      (where=(station="&&sta&i")) noobs;
  var month co o3 no3 tem;
  title1 "&&sta&i Pollution";
  footnote;
run;
ods html close;
%end;
```

The key to pointing to the secondary files (created by PROC PRINT above) is a character variable that contains a reference file pointer. The form of the value of this variable will be "href=xxxxxx.html", where xxxxxx is data dependent (in this case the station name).

In the data set to be plotted, we create a variable, DRILLSTA, which contains a file pointer in the form of HREF=xxxxxx.html, and the name of the station as part of the file pointer.

```
data ca88air;
  set sasclass.ca88air;
  length drillsta $15;
  drillsta = 'href='||trim(left(station))||'.html';
run;
```

This variable is utilized by the PROC GCHART in the VBAR statement through the HTML= option. Notice also that we are using the GIF device, although other devices such as ACTIVEEX, JAVA, and WEBFRAME, could also be used.

```
goptions device=gif;
ods html path="&drive\&project\figures" (url=none)
      body='chart.html';
PROC GCHART DATA=ca88air;
  VBAR station / type=mean sumvar=o3
                patternid=midpoint
                subgroup=station
                html=drillsta
                raxis=axis1 maxis=axis2;

run;
quit;
ods html close;
```

It is also possible to create hot zones in a scatter plot. In the following example, the same data that was used above in a histogram is also plotted. In this case there is one line per station. Because the DRILLSTA variable is a constant for each station, each line becomes a separate hot zone.

```
goptions device=gif;
ods html path="&drive\&project\figures" (url=none)
      body='gplot.html';
proc gplot DATA=ca88air;
  plot o3*month=station /
        html=drillsta
        htmllegend=drillsta
        vaxis=axis1;

run;
quit;
ods html close;
```

The HTMLLEGEND= option creates hot zones in the legend as well. In this example, the variable DRILLSTA is constant for each station. If instead it had been unique for each plotted point, then each point could have been a separate hot zone.

SEE ALSO

More information about creating drill-down graphics can be found in Smith (2003) and in Carpenter and Smith (2002). Hadden (2003) creates drill-down maps using SAS/GRAPH.

10.5 Chapter Summary

The macro language makes an excellent interface between SAS and the operating environment. SASHELP views and SQL DICTIONARY tables enable the user to gather information about the environment in which SAS is operating. These tables in turn contain information that can be placed into macro variables where it can be used by the program itself.

Knowing the current settings enables the user to write programs that can change and then reset system options and operating system conditions.

The Output Delivery System (ODS) utilizes a number of options, such as file names, which can require constant updates in a production environment. Using the macro language to control these options enables the programmer to write code that does not require manual intervention or coding updates.



Chapter 11

Working with SAS Data Sets

- 11.1 Creating Flat Files 280
 - 11.1.1 Column-specified flat file 280
 - 11.1.2 Creating comma-delimited files 284
- 11.2 Subsetting a SAS Data Set 286
 - 11.2.1 Selection of top percentage using SQL 287
 - 11.2.2 Selection of top percentage using the POINT option 288
 - 11.2.3 Random selection of observations 288
 - 11.2.4 Building a WHERE clause dynamically 292
- 11.3 Checking the Existence of SAS Data Sets 295
- 11.4 Working with Data Set Variables 296
 - 11.4.1 Create a list of variable names from the PDV 296
 - 11.4.2 Create a list of variable names from an ID variable 304
 - 11.4.3 Creating individual macro variables from an existing list 304
 - 11.4.4 Counting words within a macro variable 306
 - 11.4.5 Placing commas between words 308
 - 11.4.6 Quoting words in a list 311
 - 11.4.7 Check for existence of variables 313
 - 11.4.8 Remove repeated words from a list 314
 - 11.4.9 Working with a list of class variables 316
- 11.5 Counting Observations in a Data Set 319
 - 11.5.1 Using %SYSFUNC and ATTRN 320
 - 11.5.2 Controlling observations in PRINT listings 321
 - 11.5.3 Using a DATA _NULL_ step 323
 - 11.5.4 Using SQL 324
- 11.6 Chapter Summary 326

One of the things that SAS does best is to work with all types of data sets. You can use the macro language to form powerful tools that can be applied in a variety of situations that interface with these data sets. This chapter includes examples of a number of common problems that are associated with the maintenance, construction, and use of flat files and SAS data sets.

SEE ALSO

Although no examples of reading flat files have been included here, Kowitz (2000) has written a macro that standardizes the process.

Gerlach and Misra (2002) demonstrate a macro that will split a large data set into N subsets.

Zirbel (2002) compares two data sets to locate differences.

Mahnken (2002) discusses a macro that performs a many-to-many merge.

When the limitations of PROC IMPORT affect your ability to bring EXCEL tables into SAS, consider some of the techniques offered by Zhou (2002).

11.1 Creating Flat Files

Although flat files are not as commonly used to transfer data as they once were, they can still be very handy as they are almost universally accepted. A number of tools have been written to convert a SAS data set into a flat file. Most of these use PROC CONTENTS to determine the data structure (variable names and type). You can also employ DATA step functions, the view SASHELP.VCOLUMN, and the SQL table DICTIONARY.COLUMNS.

MORE INFORMATION

The SQL DICTIONARY tables are discussed in Section 6.5.4 and the SASHELP views are discussed in Section 9.5.

SEE ALSO

Kretzman (1992) includes a macro that dumps a SAS data set to a flat file. A description of the macro %FLATFILE (available from SAS), is provided by Buchecker (1998a).

11.1.1 Column-specified flat file

%SAS2RAW

This macro can be used to write the values within a SAS data set to a flat file with fixed columns. The header section of the resulting file contains sufficient information to reread the data. Although building this header information requires knowledge of the attributes of the data and the variables, this macro gathers this information automatically.

You can use PROC CONTENTS to determine the attributes of each of the variables in the data set that is to be converted. This information, including formatting, is used to build the PUT statement that is used to write the flat file. The syntax of the PUT statement is also written to the file as a header record to document the data set variable names and column information. Because

formats may be used in the SAS data set, variables with associated formats will be written in formatted form.

```

* sas2raw.sas
*
* Convert a SAS data set to a RAW or flat text file. Include
* SAS statements on the flat file as documentation.
*;

* LIB LIBREF OF THE DATA BASE (data base name) e.g. BENTHIC;
* This argument can be used to control the path.
* MEM NAME OF DATA SET AND RAW FILE (member name)
* e.g. FULLBEN;
* The raw file will have the same name as the data set.
*;
%MACRO SAS2RAW(lib, mem);

* The libref for incoming data is &lib ;
libname &lib "d:\training\sas\&lib"; ❶
* New text file written to the fileref ddout;
filename ddout "d:\junk\&mem..raw "; ❶

* DETERMINE LENGTHS AND FORMATS OF THE VARIABLES;
PROC CONTENTS DATA=&lib..&mem
              OUT=A1 NOPRINT;

  RUN;

PROC SORT DATA=A1; BY NPOS;
  RUN;

* MANY NUMERIC VARIABLES DO NOT HAVE FORMATS AND THE RAW FILE;
* WILL BE TOO WIDE IF WE JUST USE A LENGTH OF 8;
* Count the number of numeric variables;
DATA _NULL_; SET A1 END=EOF;
  IF TYPE=1 THEN NNUM + 1;
  IF EOF THEN CALL SYMPUT('NNUM',LEFT(PUT(NNUM,3.)));
  RUN;

%if &nnum > 0 %then %do; ❷
* DETERMINE HOW MANY DIGITS ARE NEEDED FOR EACH NUMERIC VARIABLE;
* D STORES THE MAXIMUM NUMBER OF DIGITS NEEDED FOR EACH;
DATA M2; SET &lib..&mem (KEEP=_NUMERIC_) END=EOF;
ARRAY _D DIGIT1 - DIGIT&NNUM; ❸
ARRAY _N NUMERIC_; ❹
KEEP DIGIT1 - DIGIT&NNUM;
RETAIN DIGIT1 - DIGIT&NNUM;
IF _N_ = 1 THEN DO OVER _D; _D=1; END;
DO OVER _D; ❺
  _NUMBER = _N;
  _D1 = LENGTH(LEFT(PUT(_NUMBER,BEST16.)));
  _D2 = _D;
  * NUMBER OF DIGITS NEEDED;
  _D = MAX(_D1, _D2);
END;
IF EOF THEN OUTPUT;
RUN;
%end;

```

```

*** THIS SECTION DOES NOT WRITE DATA, ONLY THE PUT STATEMENT;
* MAKE THE PUT STATEMENT AND SET IT ASIDE.;
* It will serve as documentation as well as the PUT statement;
DATA _NULL_; SET A1 END=EOF;
RETAIN _TOT 0 _COL 1;
FILE DDOUT NOPRINT lrecl=250;
IF _N_ = 1 THEN DO;
    %if &num > 0 %then SET M2;;
    TOT = NPOS;
END;
%if &num > 0 %then %do;
ARRAY _D (NNUM) DIGIT1 - DIGIT&NNUM; ❸
* TYPE=1 FOR NUMERIC VARS;
IF TYPE=1 THEN DO;
    NNUM + 1;
    DIGIT = _D; ❹
    * TAKE THE FORMATTED LENGTH INTO CONSIDERATION;
    LENGTH = MAX(FORMATL, FORMATD, DIGIT);
END;
%end;
TOT = _TOT + LENGTH + 1;
CHAR = '          ';
* SPECIAL HANDLING IS REQUIRED WHEN FORMATS ARE USED.
* CHAR IS USED TO STORE THE FORMAT;
IF FORMAT ^= ' ' | FORMATL>0 | FORMATD >0 THEN DO;
    * BUILD THE FORMAT FOR THIS VARIABLE;
    CHAR = TRIM(FORMAT);
    IF FORMATL>0 THEN

CHAR=TRIM(CHAR)||TRIM(LEFT(PUT(FORMATL,3.)));
    CHAR= TRIM(CHAR)||'.';
    IF FORMATD>0 THEN
        CHAR=TRIM(CHAR)||TRIM(LEFT(PUT(FORMATD,3.)));
END;
IF TYPE = 2 & FORMAT = ' ' THEN CHAR = '$';
* _COL IS THE STARTING COLUMN;
IF _N_ = 1 THEN _COL = 1;
IF _N_ = 1 THEN PUT '/* *** */ PUT@' _COL NAME CHAR;
ELSE          PUT '/* *** */      @' _COL NAME CHAR;
COL = _COL + LENGTH + 1;
IF EOF THEN DO;
    PUT '/* *** */ ;' ;
    CALL SYMPUT('LRECL',_TOT);
END;
RUN;

* Write out the flat file using the PUT statement in DDOUT;
DATA _NULL_; SET &dsn..&mem;
FILE DDOUT NOPRINT MOD lrecl=250; ❺
%INCLUDE DDOUT; ❻
run;
%MEND sas2raw;

*****;

%SAS2RAW(sasclass,ca88air)    run; ❼

```

- ❶ The path information is system dependent. SASHELP.VSLIB and the PATHNAME function can both be used to determine path information when you are given a *libref*.
- ❷ The macro variable &NNUM contains the number of numeric variables in this data set. Several macro %DO blocks are used to provide special processing for numeric variables.
- ❸ The DDOUT *fileref* points to both the location of the PUT statement and the file where the data will be written. This causes the data to be appended to the PUT statement, which then serves as file documentation.
- ❹ The %INCLUDE statement brings the PUT statement into the DATA step, where it can be used to write the flat file.
- ❺ The macro call contains the arguments for the *libref* and the member that is to be converted to a flat file.
- ❻ Notice that this macro represents some fairly old (Version 5 or before) code. The array definitions, which are implicit rather than explicit, are an indication of legacy code. While they work just fine, implicit arrays are generally not recommended. As a macro programmer, you may have to work with legacy code from time to time and it is generally a good idea to bring the code up to date whenever possible.

A PROC PRINT of the first ten observations of the data set SASCLASS.BIOMASS produces the following output:

Listing of SASCLASS.BIOMASS							
OBS	STATIO	DATE	BMPOLY	BMCRUS	BMMOL	BMOTHR	BMTOTL
1	DL-25	18JUN85	0.40	0.03	0.17	0.02	0.62
2	DL-60	17JUN85	0.51	0.09	0.14	0.08	0.82
3	D1100-25	18JUN85	0.28	0.02	0.01	4.61	4.92
4	D1100-60	17JUN85	0.36	0.05	0.32	0.47	1.20
5	D1900-25	18JUN85	0.03	0.02	0.11	1.06	1.22
6	D1900-60	17JUN85	0.54	0.11	0.03	4.18	4.86
7	D3200-60	17JUN85	0.52	0.14	0.04	0.05	0.75
8	D3350-25	18JUN85	0.18	0.02	0.11	0.00	0.31
9	D6700-25	18JUN85	0.51	0.06	0.03	0.01	0.61
10	D6700-60	17JUN85	0.32	0.14	0.04	0.22	0.72

The macro call %SAS2RAW(sasclass, biomass) produces the text file BIOMASS.RAW, a portion of which is shown below:

```
/* *** */ PUT @1 STATION $
/* *** */      @10 DATE DATE7.
/* *** */      @18 BMPOLY
/* *** */      @23 BMCRUS
/* *** */      @29 BMMOL
/* *** */      @34 BMOTHR
/* *** */      @39 BMTOTL
/* *** */ ;
DL-25      18JUN85 0.4  0.03 0.17 0.02 0.62
DL-60      17JUN85 0.51 0.09 0.14 0.08 0.82
D1100-25   18JUN85 0.28 0.02 0.01 4.61 4.92
D1100-60   17JUN85 0.36 0.05 0.32 0.47 1.2
D1900-25   18JUN85 0.03 0.02 0.11 1.06 1.22
D1900-60   17JUN85 0.54 0.11 0.03 4.18 4.86
D3200-60   17JUN85 0.52 0.14 0.04 0.05 0.75
D3350-25   18JUN85 0.18 0.02 0.11 0      0.31
D6700-25   18JUN85 0.51 0.06 0.03 0.01 0.61
D6700-60   17JUN85 0.32 0.14 0.04 0.22 0.72
```

MORE INFORMATION

Flat files are also created in Section 6.2.1.

SEE ALSO

Whitlock (1993) and Buchecker (1998a) provide a description and code for the macro %FLATFILE that operates in a similar manner to %SAS2RAW (except with more options).

11.1.2 Creating comma-delimited files

%DELIM

Susan Haviar

You can use comma-delimited files as import data for a number of applications, including spreadsheets and Microsoft Word tables. The following macro creates a comma-delimited flat file that uses PROC CONTENTS and a DATA _NULL_:

```
%delim(vitals,log)
* delim.sas
*
* Convert a SAS data set to a comma-delimited flat file.;
*
```

```
* Presented at PharmaSUG April, 1997
* by Susan Haviar
*;
```

```
data vitals;
input value $10. target $8. nums err mini maxi;
cards;
Diastolic Baseline 8 64.5 59 74
Diastolic 0.25 hrs 8 66.6 57 72
Diastolic 0.50 hrs 8 62.9 51 70
Diastolic 1 hrs      8 69.5 57 88
Diastolic 2 hrs      8 69.8 53 83
run;
```

```
%macro delim(dsn, out);
```

```
  filename &out "d:\junk\&dsn.txt"; ❶
```

```
  proc contents data=&dsn ❷
```

```
    out=_temp_ (keep=name npos)
    noprint;
```

```
  run;
```

```
  proc sort data=_temp_; ❸
```

```
    by npos;
```

```
  run;
```

```
  data _null_;
```

```
  set _temp_ end=eof;
```

```
  call symput('var'||left(put(_n_,5.)),name); ❹
```

```
  if eof then call symput('total',left(put(_n_,8.)));
```

```
  run;
```

```
  data _null_;
```

```
  file &out noprint; ❺
```

```
  set &dsn;
```

```
  put
```

```
    %do i=1 %to &total; ❻
```

```
      &&var&i +(-1)', ' ❼
```

```
    %end;
```

```
    +(-1)' '; ❽
```

```
  run;
```

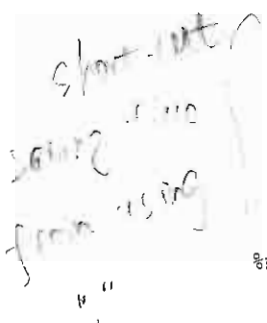
```
%mend delim;
```

```
%delim(vitals,outfile)
```

- ❶ Name of the flat file that is to be created. In this example, it will be vitals.txt.
- ❷ Use PROC CONTENTS to create a data set (_TEMP_) that contains the variables and their position in the data set that is named in &DSN.
- ❸ Sort the variable names according to their position on the PDV.
- ❹ Create a series of macro variables (&VAR1, &VAR2, and so on) that contain the variable names in the data set &DSN. The total number of variables is stored in &TOTAL.
- ❺ Name the output file in a FILE statement.

- ⑥ Dynamically create the PUT statement that will contain each variable in the data set.
- ⑦ The +(-1) moves the pointer back one space and prevents a blank from being written between the value and the comma.
- ⑧ Use a blank to cover up the comma that follows the last variable.

With current versions of SAS it is also possible to create delimiter-separated files by taking advantage of the DELIMITER or DLM option on the FILE statement. The DATA _NULL_ step from above becomes



```
data _null_;
file &out noprint dlm=',';
set &dsn;
put
    %do i=1 %to &total;
        &&var&i
    %end;
    ;
run;
%mend delim;
```

MORE INFORMATION

The %MAKECSV in Section 11.2.4 uses the DLM= option to build a CSV file.

SEE ALSO

Hahl and Shelton (1995) demonstrates a similar example that places quotes around character variables. First (2001b) briefly describes a general utility macro to create CSV files. Comma separated files specifically generated to be imported into Microsoft Excel are created by Rajecki and Kahle (2000).

11.2 Subsetting a SAS Data Set

It is often of interest to create subsets of a data set based on some predefined criteria. This might include randomly selected observations or a certain percentage of the observations with the largest values of a particular variable. This section discusses some of these selection techniques.

SEE ALSO

Gerlach and Misra (2002) demonstrate a macro that will split a large data set into N subsets.

2.1 Selection of top percentage using SQL

%TOPPCNT

Diane Goldschmidt

This macro creates a data set that is a subset of an original data set, and the subsetting criterion is based on a percentage of the largest values of a particular variable. You might use this macro if you want to regularly run a report on the top 10 percent of a data set that is constantly changing size, and you don't want to manually calculate and then edit the number of observations that you want to look at.

SQL is used to count the number of observations with distinct values of the ID variable (&IDVAR) ①. The requested fraction of this number (&PCNT) ② is then loaded into a macro variable &IDPCNT ③. The first &IDPCNT observations are then written to the new data set TOPITEMS ④ using the OBS= data set option.

```
%macro toppcnt(dsn,idvar,pcnt);
*****
* CREATE TABLE PCNT FOR INDICATING &PCNT OF Ids *;
*****

PROC SQL NOPRINT;
  SELECT
    COUNT(DISTINCT &IDVAR) ① times * &PCNT ② INTO :IDPCNT ③
  FROM
    &dsn;      ****<-- Number of obs in &dsn is unknown;

*****
*
*      SORT ON DESCENDING &IDVAR
*
*****

PROC SORT DATA= &dsn OUT=ITEMS;
  BY DESCENDING &IDVAR;
  RUN;

*****
*   KEEP TOP % USING GLOBAL MACRO VARIABLE
*
*****

DATA TOPITEMS;
  SET ITEMS(OBS=%sysevalf(&IDPCNT,ceil)); ④ **<-- Reflects the % ;
  RUN;
%mend toppcnt;

%toppcnt(sasclass.biomass,bmtotl,.25);
```

The macro %TOPPCNT counts the distinct values of the variable named by &IDVAR. If each observation does not contain a unique value for that variable, the percentage of observations in TOPITEMS may not be accurate.

④ Because of the way that &IDPCNT is calculated, it may result in a noninteger value. Since this would cause an error, the %SYSEVALF function with the CEIL option is used to return the next largest integer value.

MORE INFORMATION

The %SYSEVALF function is discussed in Section 7.3.3.

11.2.2 Selection of top percentage using the POINT option**%SELPCNT**

You can also use the POINT and NOBS options in the SET statement to select data subsets. In the following macro, which uses many of the same naming conventions as the macro in Section 11.2.1, the data are sorted first ❶ and then subsetted in the following DATA step. The NOBS= option ❷ creates a variable on the PDV (NOBS) that is equal to the number of observations in the data set. This enables you to calculate the number of observations to read (IDPCNT) ❸. Unlike the macro %TOPPCNT in Section 11.2.1, the count is based on total observations, not total number of distinct values of the ID variable (&IDVAR).

```
%macro selpcnt(dsn,idvar,pcnt);

* Sort the incoming data set in descending order;
proc sort data=&dsn ❶
    out=items;
    by descending &idvar;
run;

* Read the first IDPCNT observations from ITEMS;
data topitems;
    idpcnt = nobs*&pcnt; ❸
    do point = 1 to idpcnt;
        set items point=point noobs=noobs ❷;
        output;
    end;
stop;
run;

%mend selpcnt;
```

Creates a temporary constant with the value of the number of observations in the dataset being read.

names a variable where the variable is "point" which indicates which observation from the existing data set is to be read as input to the new data set

```
%selpcnt(sasclass.biomass,bmtotl,.25);
```

Remember that, since the value for NOBS is established when the DATA step is compiled, the variable NOBS can be used in the assignment statement ❸, which is **before** the SET statement, where NOBS is declared ❷.

11.2.3 Random selection of observations

A variety of routines have been written to create a data subset that is based on the random selection of observations. The use of the macro language is actually secondary to the process, which is fairly simple.

Random selection routines select either with or without replacement. **WITH** replacement selection means that a given observation is eligible for selection more than once. Each observation can be selected one time, at most, when using a **WITHOUT** replacement criteria.

The two macros presented here represent these two selection methods.

Selection without replacement

%RAND_WO

In this macro, the user selects the data set from which the observations are to be selected, and the fraction of observations to select. This routine will result in an approximate subset. Other routines that are based on conditional probabilities have also been written and these routines can result in a more precise number of observations in the subset.

```
%macro rand_wo(dsn,pcnt=0);

* Randomly select an approximate percentage of
* observations from a data set.
*
* Sample WITHOUT replacement;
*     any given observation can be selected only once
*     all observations have equal probability of selection.
*;

* Randomly select observations from &DSN;
data rand_wo;
  set &dsn;
  if ranuni(0) le &pcnt then output; ❶
run;
%mend rand_wo;
```

The RANUNI function ❶ is used to return a random number between 0 and 1. The observation is only written to the new data set RAND_WO if the returned value is less than the requested fraction (&PCNT). A call to %RAND_WO requesting a 25-percent subset could be written as

```
%rand_wo(study03,pcnt=.25);
```

Using this method allows any given observation to be selected only once; however, the number of observations selected is only approximately correct. The following version of this macro selects the correct number of observations by using an ARRAY to store the observation numbers that have been selected:

```
%macro rand_wo(dsn=,pcnt=0);
  %local obscnt;
  %let obscnt = %obsct(&dsn); ❶
  %put obs count is &obsct;

  * Randomly select observations from &DSN;
data rand_wo(drop=cnt totl);
  * Calculate the number of obs to read;
  totl = ceil(&pcnt*&obsct); ❷
  array obsno (&obsct) _temporary_; ❸
```

* Temporary array is only needed for the duration of the Data Step

* Parameter to temporary data elements by the array name and dimension

* temporary array elements do not have names, and they do not appear in the output data set.

* temporary array elements are automatically retained instead of being reset to missing at the beginning of the next iteration of the data step.

array
"numeric"
name { }
numeric
one string
and the
are numerical
values which
are a kind of thing

```

do until(cnt=totl);
  point = ceil(ranuni(0)*&obsent); ④ - random number
  if obsno(point) ne 1 then do; ⑤
    * This obs has not been selected before;
    set &dsn point=point; ⑥
    output;
    obsno(point)=1; ⑦
    cnt+1;
  end;
end;
stop;
run;
%mend rand_wo;

```

> counter for total read observations.

- ① Determine the total number of observations in the data set (&DSN) using the %OBSCNT macro described in Section 11.5.1.
- ② Calculate the number of observations to read (TOTL).
- ③ A temporary array, with the observation number as the index, will be used to mark those observations that have been selected.
- ④ Randomly generate an observation number between 1 and &OBSCNT.
- ⑤ This observation has not already been read.
- ⑥ Read the selected observation using the POINT= option on the SET statement.
- ⑦ Mark this observation as having been used.

A request to read 50 percent of the observations in the data set SASHELP.CLASS would use the following call to the macro.

```
%rand_wo(dsn=sashelp.class, pcnt=.50)
```

Selection with replacement

%RAND_W

Sampling with replacement allows for more flexibility and a more interesting macro. When sampling with replacement, it is possible for the resulting data set to have more observations than the original. This can be very useful when you use statistical techniques, such as bootstrapping.

The following macro, %RAND_W, enables the user to select either the number of desired observations or a percentage. Both may result in numbers that are larger than the number of original observations. The POINT and NOBS options in the SET statement are used to randomly select the observations that are to be included in the sample.

```

%macro rand_w(dsn,numobs=0,pcnt=0); ❶

* Randomly select either a specified number of
* observations or a percentage from a data set.
*
* Sample WITH replacement;
*     any observation can be selected any number of
*     times.

* When NUMOBS is specified create a subset of exactly
* that many observations (ignore PCNT).
* When PCNT is specified (and NUMOBS is not)
* calculate NUMOBS using PCNT*total number in DSN.
*;

* Randomly select &NUMOBS observations from &DSN;
data rand_w;
retain numobs .;
drop numobs i;

* Create a variable (NUMOBS) to hold number of obs
* to write to RAND_W;
%if &pcnt ne 0 and &numobs=0 %then %do;
    * Use the percent to calculate a number of obs;
    numobs = round(nobs*&pcnt); ❷
%end;
%else %do;
    numobs = &numobs; ❸
%end;

* Loop through the SET statement NUMOBS times;
do i = 1 to numobs;
    * Determine the next observation to read;
    point = ceil(ranuni(0)*nobs); ❹

    * Read and output the selected observation;
    set &dsn point=point nobs=nobs ; ❺
    output; ❻
end;
stop;
run;
%mend rand_w;

```

- ❶ The user selects one of two named parameters:

NUMOBS= is used when a specific number of observations is desired.

PCNT= specifies a fraction (&PCNT can be greater than 1) of the observations to be selected.

- ❷ The variable NUMOBS will be used in a DO loop that controls the number of observations in the output data set. Here, NUMOBS is calculated based on the number of observations in the data set (NOBS) and the desired percent (&PCNT).
- ❸ When a specific number of randomly selected observations has been requested, that value is transferred to the variable NUMOBS.

- ④ POINT will be a random integer that can range from 1 to NOBS. It is important that the CEIL function be used to create the integer. Functions such as INT, FLOOR, or ROUND will assign the incorrect probabilities to the first and last observations.
- ⑤ Read the observation indicated by the POINT variable.
- ⑥ Output the observation to the new data set and continue the loop.

Typical calls to %RAND_W might include

```
%rand_w(study03,numobs=100);
```

```
%rand_w(study03,pcnt=.25);
```

SEE ALSO

Additional subsampling techniques, including the use of PROC SURVEYSELECT, are discussed by Chapman (2001).

11.2.4 Building a WHERE clause dynamically

Often we need to build a WHERE clause based on information contained within the macro variables. The process is fairly straightforward in that we are simply building code; however, there are a couple things that are worth a special note.

%MAKECSV

This macro was written to dump the contents of a SAS data set to a comma-separated flat file. Since not all of the original data set is desired, a WHERE clause is built based on the values of the macro parameters. This macro was written to work against a specific data set; however, the concepts could be generalized as long as the macro parameters contain the kind of building blocks that you will need to build your own WHERE clause. These building blocks include

- ① a list of values
 - ② a specific value
 - ③ a logic flag.
- ```
%macro makescv(dsn,list,reg,miss,no=no);
options &no.mprint &no.mlogic &no.symbolgen;
%* ① LIST one or more blank separated clinic numbers;
%* Blank to get all companies;
%* ② REG Region of interest
%* ③ MISS are observations with missing weights allowed?
%* ok missing ok
%* <other> nonmissing only;
```

```
%local qlist wclause; ④
```

```
%* Quote the words in the list and separate
%* them with commas;
```

```
%let qlist = ⑤
```

```
%str(%)%sysfunc(tranwrd(&list,%str(,),%str(','))%str(%));
```

```
%str(%)
```

should have spaces!  
or else string message  
will say the word  
is already being processed  
has reached the  
maximum length of  
65535

```

%* Build the WHERE clause;
%if &miss=ok %then %let wclause = wt ge ⑥;
%else wt ne .;
%if ® ne %then %let wclause = &wclause & region="®"; ⑦
%if %bquote(&list) ne %then
 %let wclause = &wclause & clinnum in(&qlist); ⑧

data _null_;
 set &dsn(wher=(%unquote(&wclause))); ⑨
 file "c:\temp\makecsv.csv" dlm=',';
 if _n_=1 then put 'clinicnumber,clinicname,region,weight';
 put clinnum clinname region wt;
run;
%mend makecsv;

```

- ④ Macro variables are declared as local. &WCLAUSE is created and then used in the WHERE statement.
- ⑤ The list of values will be used as character values in an IN operator; therefore, they need to be quoted and comma separated. The TRANWRD function is used to convert embedded blanks to a ' '.
- ⑥ &MISS is used to determine if missing values of the variable WT are to be included. In this macro the WHERE clause will **always** have this clause; of course wt ge . will eliminate nothing.
- ⑦ If a value for the variable REGION is supplied, & region="&reg" is added to the clause.
- ⑧ When one or more values are supplied in &LIST the quoted values (&QLIST) are used with the IN operator, and & clinnum in(&qlist) is appended to the clause.
- ⑨ Since the %STR function was used to mask the single quotes, the %UNQUOTE function is applied so that the clause can be applied correctly.

Notice that the WHERE clause was built in a macro variable, which was then used in the WHERE= option. Because of timing issues between the macro facility and how the WHERE clause is applied, %IF-%THEN/%ELSE statements can cause problems when applied directly inside of what will become the WHERE clause ⑨.

The following macro call:

```

%makecsv(sasclass.clinics,
 051345 057312,
 5, ok,no=no) * write the transmet data set;

```

results in the following WHERE= option:

```
(where=(wt ge . & region="5" & clinnum in('051345','057312')));
```



**MORE INFORMATION**

Sections 11.4.5 and 11.4.6 both contain examples where commas and quotes are inserted into lists. A WHERE clause is also created in Section 10.2.2.

**%FINDOUTLIERS**

The macro %FINDOUTLIERS is used to find observations that meet a criteria, which has been passed into the macro through the macro parameters. Unlike %MAKECSV this macro is more general in that the user can select the variable names and the criteria for selection.

Macro parameters include

|         |                                                                                           |
|---------|-------------------------------------------------------------------------------------------|
| dsn     | name of the data set to check.                                                            |
| prefix  | name or prefix letters of the variable(s) to use to determine the outliers.               |
| value   | value of the variable to use as the determining criteria.                                 |
| op      | comparison operator, for example "eq" or "=" (either the mnemonic or symbol may be used). |
| logicop | logic operator joining comparisons (may be either AND or OR).                             |

The following call to %FINDOUTLIERS selects observations where any variable whose name starts with "BM" has a value greater than or equal to 6:

```
%findoutliers(dsn=sasclass.biomass,prefix=bm,
 value=6, op=ge, logicop=or)
```

The resulting WHERE clause becomes

```
(where=(BMCRUS ge 6 or BMMOL ge 6 or BMOTHR ge 6 or BMPOLY ge 6
or BMTOTL ge 6))
```

All of the macro's parameters are named parameters; however, only &OP and &LOGICOP have default values.

```
%macro findoutliers(dsn=,prefix=,value=,op=ge, logicop=or);
%local i wclause;
proc contents data=&dsn noprint
```

```
out=contdsn;
run;
data _null_;
set contdsn;
if name =: %upcase("&prefix");
cnt+1;
charcnt= left(put(cnt,4.));
call symput('var'||charcnt,trim(name));
call symput('varcnt',charcnt);
run;
%if varcnt ge 1 %then %do;
%let wclause= &var1 &op &value;
%if varcnt gt 1 %then %do i = 2 %to &varcnt;
%* Build the where clause;
%let wclause = &wclause &logicop &&var&i &op &value;
%end;
```

You can compare only a specified prefix of a character string by using a colon after the comparison operator. The following example shows the colon modifier after the prefix in the SAS code. Look at only the first character of values of the variable. Last time we select the observations with names beginning with the letter 'S'.

if lastName == 'S' :

Only here the macro variable 'varcnt' turns to a numeric and implicitly change from numeric type to character type.

```

data outliers;
 set &dsn(where=(&wclause)); ❹
 run;
%end;
%mend findoutliers;

```

- ❶ PROC CONTENTS is used to create a list of the names of the variables in &DSN.
- ❷ The variable names that meet the criteria in &PREFIX are stored in the macro variables of the form &&VAR&i.
- ❸ The number of variables of interest is saved.
- ❹ The first condition is assigned to the WHERE clause.
- ❺ Additional conditions are attached to the WHERE clause.
- ❻ The WHERE clause is applied.

## SEE ALSO

Curtis Smith (1999) builds a WHERE clause with a LIKE operator.

---

## 11.3 Checking the Existence of SAS Data Sets

At times, you will want to be able to determine if a data set exists before you use it with a procedure such as PROC PRINT. When creating systems dynamically, some data sets may not exist under certain circumstances, and you need to be able to determine their status at execution time.

The macro %EXIST, described in Section 7.1.3, uses a DATA step to determine if a data set exists. In the following example, the EXIST function is used in conjunction with the %SYSFUNC macro function to eliminate the DATA step and the annoying error message in the LOG when the data set is not found:

```

%macro exist(dsn);
 %global exist;

 %if %sysfunc(exist(&dsn)) ❶ %then %let exist=YES;
 %else %let exist=NO; ❷
%mend exist;

```

- ❶ The EXIST function returns a value that is equal to 1 when the data set exists. Otherwise, it returns a 0, causing the %IF to be false, which results in &EXIST being set to NO ❷.

## MORE INFORMATION

This example is expanded in Section 7.5, and is described as a macro function in Section 7.6.1.

**SEE ALSO**

The EXIST function is combined with the %SYSFUNC macro function in a similar example in *SAS Macro Language, Reference, First Edition* (p. 242).

Whitlock (1997) uses a DATA step to determine if a data set contains any observations.

---

## 11.4 Working with Data Set Variables

The use of macros and macro variables is often important when dealing with data set variables. This is especially true when the data sets and variable lists are generated dynamically in such a way that the developer does not necessarily know what the program will be operating on at the time of execution. The following examples create or work with macro variables that contain data set variable names or values.

**SEE ALSO**

The documentation for %SYSFUNC (*SAS Macro Language, Reference, First Edition*, p. 242) contains an example that counts the number of both observations and variables in a SAS data set.

Hahl (1996) describes the macro %DROPVAR that can be used to DROP any variable that only takes on MISSING values.

Whitaker (1989) provides several utilities to work with lists, including

- counting and extracting words
- eliminating duplicate words
- building lists with SYMPUT
- building lists with %INCLUDE.

---

### 11.4.1 Create a list of variable names from the PDV

**Kim Kubasek**  
Consultant

Sometimes it becomes necessary to create a macro variable that contains a list of variable names. When you deal with programs that were written dynamically, you might not know what those names will be, especially when the data sets are created by SAS procedures.

One procedure that commonly creates new variables is PROC TRANSPOSE. When you use the ID statement, the names of the new variables will be the values taken on by the ID variable (or, more importantly some variation of those values). The following example creates a list of variable names and places them in a macro variable.

In this code, the data set WORK.TEMP4 was created by PROC TRANSPOSE ❶, and the new (unknown) variables will be the values of the ID variable GROUP1 ❷. PROC SQL is used to access the SASHELP.VCOLUMN view ❸ that contains the names of the variables (PROC CONTENTS could also be used—see Section 11.1.1). A DATA \_NULL\_ step ❹ is then used to build a %LET statement ❺ that is then included back into the program ❻.

```

...code not shown...
*** create the transposed data set;
proc transpose data=temp3 out=temp4 ; ❶
by group2 year &spvar ;
var mean ;
id group1 ; ❷
run ;

*** what variable names did proc transpose create?
 the table VARLIST will contain just the variable
 names of interest;
proc sql noprint ;
create table varlist as
select name
from sashelp.vcolumn ❸
where libname='WORK'
and memtype='DATA'
and memname='TEMP4'
and not(name in ('GROUP2' 'YEAR' "%upcase(&spvar)"
'_NAME_' '_LABEL_')) ;

quit ;

*** write short files of code to be included later that will
 create a space-delimited variable list ;
data _null_ ; ❹
set varlist end=e ;
file 'temp2.sas' lrecl=70 ;
if _n_=1 then put '%let varlist= ' @ ; ❺
put name @ ;
if e then put ';' ;
run ;

%include 'temp2.sas' / source ; ❻

```

The resulting file TEMP2.SAS will contain a single %LET statement, which will be executed as soon as the file is included ❻. A sample %LET statement might be

```
%let varlist=station date depth;
```

The SQL and DATA\_NULL\_ steps could have been combined into a single SQL step by using the technique shown in Section 11.4.2.

If the VNAME routine is used, you can replace the PROC SQL and %INCLUDE portions of the previous example with a single DATA \_NULL\_. The following DATA step is designed to fit into the previous example right after the PROC TRANSPOSE. This solution uses the variable STR to accumulate the list of variable names; however, since its length is fixed, there is an implied upper limit as to the number of names that can be processed. This would not be a limitation of the previous method.

```

data _null_;
length name $32 str $500;
set temp4;
array allc {*} _character_; ❶
array alln {*} _numeric_; ❶
if dim(allc) then do i=1 to dim(allc); ❷
 call vname(allc(i),name); ❸
 * Exclude vars we know we do not want;
 if name not in('_NAME_' '_LABEL_' 'NAME' 'STR'

```

```

 'GROUP2' 'YEAR' "%upcase(&spvar)") then
 str = trim(str)||' '||trim(name); ❹
end;
if dim(alln) then do i=1 to dim(alln);
 call vname(alln{i},name);
 str = trim(str)||' '||trim(name);
end;
call symput('varlist',str); ❺
stop; ❻
run;

```

- ❶ Separate arrays are set up for the numeric and character variables.
- ❷ A DO loop is used to walk through each of the variables in the respective array.
- ❸ The CALL VNAME routine returns the name of the variable on the PDV (identified by the array ALLC{i}) and places it in the variable NAME.
- ❹ Because the value of the variable NAME is the name of the unknown variable in the array, it is then appended to the variable STR, which accumulates all the variable names.
- ❺ Once all variable names have been collected, the variable STR is written to the macro variable VARLIST using the SYMPUT routine. Notice that in this example the variable STR is assigned an arbitrary length of \$500, and this may be limiting if the list becomes long. The first example in this section does not have this limitation.
- ❻ Because you are interested in only the names of the variables and not in the data, the STOP statement is used to prevent further passes of the data.

The following macro also builds a list of variable names; however, it does it without using a DATA step.

## %GETVARS

**Michael Bramley**  
**Kendle International, Inc.**

Because we are not reading any data and because we only need to create a macro variable, we should be able to avoid the use of the DATA and SQL steps altogether. The macro %GETVARS acts like a macro function and returns the list of variables in a data set (&DSET). This macro makes use of the VARNAME function to return the variable name.

```

%Macro GetVars(Dset) ;
 %Local VarList ;
 /* open dataset */
 %Let FID = %SysFunc(Open(&Dset)) ; ❶
 /* If accessible, process contents of dataset */
 %If &FID %Then %Do ;
 %Do I=1 %To %SysFunc(ATTRN(&FID,NVARS)) ; ❷
 %Let VarList=&VarList %SysFunc(VarName(&FID,&I)); ❸
 %End ;
 /* Close dataset when complete */
 %Let FID = %SysFunc(Close(&FID)) ; ❹
 %End ;
 &VarList ❺
%Mend ;

```

- ❶ The data set of interest is opened for inquiry.
- ❷ The ATTRN function with the NVAR argument returns the number of variables in the data set.
- ❸ The VARNAME function returns the name of the &Ith variable. This variable is appended to the growing list of variables stored in &VARLIST.
- ❹ Once the data set is no longer needed, it is closed.
- ❺ The list of variables is “left behind” and effectively returned by the macro.

The two %PUT statements

```
%put biomass vars are;;
%put %getvars(sasclass.biomass);
```

will write the following lines to the LOG:

```
biomass vars are:
STATION DATE BMPOLY BMCRUS BMMOL BMOTHR BMTOTL
```

%GETVARS returns a list of all of the variables in a data set. A more sophisticated version of the macro, %BUILDVARLIST, which enables the user to select only those variable names that match one or more patterns is shown next. This macro takes advantage of the group of RXxxxxxx pattern matching functions and routines.

## %BUILDVARLIST

**Michael Bramley**  
**Kendle International, Inc.**

```
/*-----
NOTICE:
 This program is copyrighted by Michael Bramley, 2003.
 All rights to this source code are retained. You may use
 and/or distribute this program as long as it
 remains unaltered in any fashion or in any media.

SYNOPSIS:
 This program contains a SAS macro function that will
 return a variable list from the specified dataset, where
 each variable name matches the "pattern", as described below.
 Note that within this macro and the following article
 "pattern" is defined to be a regular expression, as per
 SAS documentation. Moreover, each variable name is
 returned only once (even if it matches more than one pattern).

 For a thorough exposition of this macro, please refer to my
 SUGI 27 (37-27) article: Combining Pattern-Matching And File
 I/O Functions: A SAS Macro To Generate A Unique Variable
 Name List.
```

Report any bugs, unexpected results, inconsistent behaviour or desired new behaviour to Michael Bramley (mbramley@cinci.rr.com) with the subject line of BUILDVARLIST SAS MACRO.

FILE: BuildVarList.sas

AUTHOR: Michael Bramley  
 WRITTEN: January 4, 2001  
 UPDATED: March 8, 2003

MACRO CALL:  
 %BuildVarList( Dset, Type, Patterns )

#### WHERE:

Dset = SAS data set to obtain variable information from,  
 Type = Type of variables to include in list  
 (N=numeric only, C=character only,  
 or blank for all types),  
 Patterns = |-separated list of zero or more patterns to use  
 for selecting variable names (see below).

#### NOTES:

This is a Macro Function, which means that it has a return value of blank or a list of variable names matching the pattern(s). As such, it can be used anywhere you would place a list of variable names, with the caveat that said usage includes blanks (in the case of no variable names being returned).

Specifying Type=N or C and blank for Patterns will result in a list of all numeric or character variables, respectively, similar to the `_Numeric_` or `_Character` keywords in a DATA step or procedure.

Leaving the Type and Patterns blank will cause the macro to return a list of all variable names in the data set.

Case is only respected if the pattern is quoted, for example, 'VAR' matches all variable names beginning with uppercase VAR.

If more than one pattern is desired, then Patterns should be of the form:

pattern-1|pattern-2|...|pattern-n.

The macro will process each pattern separately. Each pattern may be an expression containing (optionally) a prefix, asterisk, and/or suffix. The following case insensitive examples should help:

Ex.1: DK\*A will return all variable names that begin with DK and end with A,

Ex.2: DK (or DK\*) will return all variable names that begin with DK,

Ex.3: \*A will return all variable names that end with A--the asterisk is required for this example.

## EXAMPLE:

```
%Let MyVars = %BuildVar(TestData,,VAR) ;
```

This assigns to the macro variable MyVar the value of a Space-delimited list of all variable names that match the pattern "VAR\*", regardless of type, where \* may be expanded into 0 to n characters.

```
-----*/

%Macro BuildVarList(Dset, Type, Patterns) ;
/*
 Define local macro variables:
 VarList = list of variables matching pattern(s)
 VarName = name of current variable
 CurPat = current pattern being used
 FID = File Identifier (for FILE I/O functions)
 RX = Regular expression ID - SAS says "do not PUT..."
*/
%Local VarList VarName CurPat FID RX I J ;

/*
 Check Access to SAS data set.
*/
%Let FID = %SysFunc(Open(&Dset)) ;
%If Not &FID %Then %Do ;
 %Put ERROR: BuildVarList could not access &Dset.. ;
 %Go To Exit ;
%End ;

/*
 Check Type and Patterns. Default Type=NC to allow
 all variable types if none
 specified, otherwise validate Type.
*/
%Let Type = %UpCase(&Type) ; ❶

%If "&Type" EQ "" %Then
 %Let Type = NC ;
%Else
 %If Not %Index(NC, &Type) %Then %Do ;
 %Put ERROR: BuildVarList found invalid variable type of
 &Type.. Must be blank, N, or C. ;
 %Go To Exit ;
 %End ;

/*
 If Version 6.xx engine, ensure that the Patterns parameter
 is uppercase.
*/
%If %Index(%SysFunc(AttrC(&FID, ENGINE)), V6) %Then
 %Let Patterns = %UpCase(&Patterns) ; ❷
```



```

/*
 If no Patterns specified, default to : to match all
 variables in data set.
*/
%If "&Patterns" EQ "" %Then
 %Let Patterns = : ;

/*
 BuildVarList Macro Main Processing Block: process
 patterns (separated by | <an or bar>) to generate
 variable list by calling SAS regular expression routines.

 NOTE: As some regular expressions may begin with SAS
 arithmetic operators, for example, + or *,
 it is imperative to quote them so that the SAS %Eval
 function is not invoked in the %Do %While Loop.
*/
%Let I = 1 ;
%Let CurPat = %Scan(&Patterns, &I, |) ;

%Do %While("&CurPat" NE "") ;
 %Let RX = %SysFunc(RXParse(&CurPat)) ; ❸
 /*
 If pattern accepted by parser, then process variables
 in SAS data set.
 */
 %If &RX %Then %Do ;
 %Do J = 1 %To %SysFunc(AttrN(&FID, NVAR)) ;
 /*
 If Type matches, check variable name against
 pattern and add the variable name to the list
 (if it is not already included).
 */
 %If %Index(&Type, %SysFunc(VarType(&FID, &J)))
%Then %Do ;
 %Let VarName = %SysFunc(VarName(&FID, &J)) ;
 %If %SysFunc(RXMatch(&RX, &VarName)) And ❹
 Not %Index(&VarList, &VarName) %Then
 %Let VarList = &VarList &VarName ;
 %End ;
 %End ;
 %SysCall RXFree(RX) ; ❺
 %End ;
%Else
 %Put WARNING: BuildVarList detected an invalid pattern
 ("&CurPat")...ignoring. ;

 %Let I = %Eval(&I + 1) ;
 %Let CurPat = %Scan(&Patterns, &I, |) ;
%End ;

/*
 Did any variable names match any of the pattern(s) ?
*/
%If Not %Length(&VarList) %Then
 %Put WARNING: BuildVarList found no variable names in &Dset
 matching specified pattern(s). ;

```

```

%Exit:
 %If &FID %Then
 %Let RX = %SysFunc(Close(&FID)) ;

 &VarList
 %Mend BuildVarList ;

/*
 Include a test data set...(ignore the warnings...)
*/
Data Test1 ;
 Length Var1-Var6 4 ;
 Length Var1RT Var2RT Var3RT Var4RT T1 - T10 4 ;
 Length Var1A Var1b Var1C VarRT T1R T2R T3R $10 ;
Run ;

/*
 Test the BuildVarList macro function...
*/
%Put Character Only = %BuildVarList(Test1, C) ;
%Put Numeric Only = %BuildVarList(Test1, N) ;
%Put All Variables = %BuildVarList(Test1) ;
%Put Var* Variables = %BuildVarList(Test1, , Var) ;
%Put VAR* Variables = %BuildVarList(Test1, , 'VAR') ;

*** End of BuildVarList Macro ;

```

- ❶ The user can specify the variable type (character or numeric or both).
- ❷ The user specifies one or more patterns that the variable names must match. In Version 6 and before the names are always upper case.
- ❸ The RXPARSE function sets up a memory location for the pattern or patterns for which the comparisons are to be made.
- ❹ The comparison is made by the RXMATCH function using the patterns stored by RXPARSE.
- ❺ After the comparisons have been made the RXFREE routine is used to clear the memory. Notice that the macro variable &RX is coded without the ampersand when RXFREE is used with %SYSCALL.

The RXMATCH function is quite sophisticated and if you do not understand it well, it might return results that you do not anticipate. Taking care when using the RXMATCH function includes the use of wild cards like the \* and pattern separators – both of which are used in this macro.

## MORE INFORMATION

Section 2.7.2 uses PROC SQL to build a similar list of variables, as does the example in Section 11.4.9.

## SEE ALSO

Burroughs (2001) also uses the VNAME routine to capture the variable name from an array. The macro %GETVARS is discussed in detail in Bramley (2001) and similar code can be found in the SAS OnlineDoc under the VARNAME function. Lund (2003b) discusses a macro function that returns the list of variables in a data set.

## 11.4.2 Create a list of variable names from an ID variable

Jørn Lodahl

PROC SQL can also be used directly to create a macro variable that contains a list of data set variable values. If your data set already contains a variable whose values are to be loaded into a macro variable list, then you can use the following code (which was suggested by Jørn Lodahl and is used in %INDVAR in Section 11.4.9). The examples in Sections 6.2.1 and 9.2 also use a data set to build macro variables, but the results are placed in individual macro variables rather than into a single list.

```
proc sql noprint;
 select distinct loc ❶
 into :loclist separated by ' ' ❷
 from samdat ❸
 order by loc; ❹
quit;
```

All unique values of the variable LOC ❶ are written to the macro variable &LOCLIST ❷, where they are separated by blanks. The variables are read from the data set SAMDAT ❸, and the values are placed in ascending order by using the ORDER clause ❹.

### SEE ALSO

Widawski (1997a) creates a list of filenames, which is written to a macro variable.

## 11.4.3 Creating individual macro variables from an existing list

This example assumes that you have created a macro variable that contains a list of data set variables (see Sections 11.4.1 and 11.4.2). In this case, the macro variable (&KEYFLD) contains the variables that form the key fields (variables used in a BY statement). Section 9.3.3 also discusses the use of &KEYFLD.

A sample definition of the macro variable &KEYFLD might be

```
%let keyfld = investid subject treatid;
```

In the program that uses this list, we might expect to see a BY statement such as

```
by &keyfld;
```

In order to use FIRST. and LAST. processing, as in the following example, you need to know the names of the individual variables in the BY list. This enables you to write statements such as this:

```
by investid subject treatid;
if last.treatid then do;
```

To do this dynamically using &KEYFLD, you need to know its component parts (variable names in the list) and the number of names.

There are several ways to solve this type of task. The one that involves the least use of the macro language uses a DATA step to break up the list. The following DATA step can be used to create a series of macro variables (&KEY1, &KEY2, and so on), one for each name in &KEYFLD:

```
*determine the list of key vars;
data _null_;
* count the number of keyvars
* save each for later;
str="&keyfld"; ❶
do I = 1 to 6;
 key = scan(str,i,' '); ❷
 if key ne ' ' then do;
 ii=left(put(i,1.));
 call symput('key'||ii,
 trim(left(key))); ❸
 call symput('keycnt',ii); ❹
 end;
end;
run;
```

- ❶ An assignment statement creates the variable STR to hold the list. You also could have used SYMGET.
- ❷ This string is then broken into words using the SCAN function.
- ❸ Macro variables are then created using the SYMPUT routine.
- ❹ The number of key variables is also counted and assigned to &KEYCNT.

Once the macro variables have been established, statements that require FIRST. or LAST. processing can be rewritten as

```
by &keyfld;
if last.&&key&keycnt then do;
```

Because &KEYCNT is the number of key variables, &&KEY&KEYCNT will resolve to the name of the last variable in the list that is stored in &KEYFLD.

Using the DATA step in the previous example is not very efficient. The same result can be accomplished using only macro language statements. In the following example, the %DO %UNTIL loop is used to step through and count the elements in &KEYFLD:

```
%Macro keylist(keyfld);
%local I;
%global keycnt;
%let I = 1; ❶
%do %until (%scan(&keyfld,&I,%str()) = %str()); ❷
 %global key&I;
 %let key&I = %scan(&keyfld,&I,%str()); ❸
 %let I = %eval(&I + 1); ❹
%end;
%let keycnt = %eval(&I-1); ❺
%mend keylist;
```

- ❶ Initialize &I, which will be used as the counter in the %DO %UNTIL loop.
- ❷ Scan &KEYFLD for the *i*th word, using a blank as the separator, and check to see if the function results in a null string. Notice that the first %STR contains a blank space (the word separator), while the second has no space (a null text string).
- ❸ If the previous line ❷ determined that the *i*th word exists, retrieve it and store it in the global macro variable &KEYi.
- ❹ Increment the counter by 1 in preparation for the next scan.
- ❺ The last word has been found, and the counter was incremented one time too many. Reduce the value by 1 and save the count in &KEYCNT.

### MORE INFORMATION

The %KEYLIST macro shown here is very similar to the %CNTVAR macro discussed in Section 7.2.3. Question 4 in the Chapter 7 exercises suggests an improved solution that uses the %DO %WHILE. As is true for most exercises in this book, a solution for this exercise can be found in Appendix 1.

### SEE ALSO

Roberts (1997) presents several macros that work with variable lists. One of his examples is very similar to the above code but uses a %DO %WHILE loop.

## 11.4.4 Counting words within a macro variable

### %NW

#### Anonymous

The %NW macro can be used to count the number of words (usually a list of variable names) in a list. The count is then stored in the global macro variable &NW. Although not its primary purpose, this macro also stores the individual words in a series of global variables of the form &&WORD&i.

```
* nw.sas
*
* Counts the number of words in a string.
*
* Called by the macros CSTR and QSTR.
*
*****;

%macro nw(string);

%local count word;
%global nw word1;

%let count=1; ❶

%let word=%qscan(&string,&count,%str()); ❷
%let word1=&word;
```

`%do %while(&word ne);
 %let count=%sysevalf(&count + 1); ❸
 %let word=%qscan(&string,&count,%str( )); ❹
 %global word&count;
 %let word&count=&word; ❺
%end;

%* nw is the number of words;
%let nw=%eval(&count-1); ❻
%put %str(&nw);
%mend nw;`

*Handwritten notes:* In each {nw} + the rest of the list

- ❶ The word count is initialized to 1.
- ❷ %QSCAN is used to determine if this word exists.
- ❸ A word exists for the current value of &COUNT so the count is incremented.
- ❹ Again %QSCAN is used to determine if the word exists for the current value of &COUNT.
- ❺ The current word is stored in the macro variable &&WORD&COUNT.
- ❻ The count will have been incremented one too many times; consequently, the value is adjusted and assigned to the macro variable &NW.

This macro is not a macro function and requires the use of the global symbol table to return not only the word count (&NW), but also the individual words. The following macro overcomes these limitations, but does not save the individual words.

## %WORDCOUNT

The macro %NW does more than just count the words in a list. Counting is all that is done by %WORDCOUNT. This macro returns the count and does not create any global macro variables.

`%macro wordcount(list);
 %* Count the number of words in &LIST;
 %local count;
 %let count=0; ❶
 %do %while(%qscan(&list,&count+1,%str( )) ne %str() ❷);
 %let count = %eval(&count+1); ❸
 %end;
 %return &count ❹
%mend wordcount;`

*Handwritten notes:* return this value &count ❸ works fine!

- ❶ The count is initialized to 0.
- ❷ Unlike the %NW macro, %QSCAN looks ahead to see if the next word (&COUNT + 1) exists.
- ❸ Check to see if there is a next word. If there is then enter the loop so that &COUNT can be incremented❹.
- ❹ Since the &COUNT+1 word was found, increment the value of &COUNT.
- ❺ Return the word count by passing it back.

The following use of %WORDCOUNT:

```
%let five = a s d f g;
%put five is %wordcount(&five);
```

writes the following to the LOG:

```
five is 5
```

## 11.4.5 Placing commas between words

### %CSTR

#### Anonymous

This macro can be used to separate the words of an incoming string with commas. Since most usages of lists of variable names in SAS are separated by spaces, this macro may prove to be more useful with values rather than with names of variables.

This macro uses the %NW macro, which is described in Section 11.4.4, to count the number of words in the incoming list. The new comma-separated string is stored in the macro variable &WRDx, where x is optionally passed as the second argument.

```
* cstr.sas
*
* Puts commas into a character string
* e.g aa bb will become
* aa,bb
*
*****;

%macro cstr(instring= , numx=2);

 /* Count the number of words in the string;
 /* Save the value in the macro variable NW;
 %nw(&instring); ❶

 %global wrd&numx; ❷

 %if &nw>0 %then %do;
 %let front=%str(); ❸
 %let back=%str(,); ❹
 %do i=1 %to &nw;
 /* MID is the ith word;
 %let mid=%scan(&instring,&i); ❺

 /* The new string is stored in WRD and is built here;
 %if &i=1 %then %do;
 %let wrd=%str(&front&mid&back); ❻
 %end;
```

```

 %else %if &i>1 %then %do;
 %let wrd=%str(&wrd&front&mid&back); ⑦
 %end;
 %end;

 /* Subtract 1 to eliminate the trailing comma;
 %let long=%eval(%length(&wrd)-1); ⑧

 /* Save the new string in the macro var WRD&NUMX;
 %let wrd&numx=%substr(&wrd,1,&long); ⑨
%end;

%*put %str(wrd&numx is &&wrd&numx);

%mend cstr;

```

the length you want to stop

returns a portion of a string

SYNTAX of %substr( , , - )

the position the string you start from searching for

- ① The %NW macro is used to count the number of words in &INSTRING.
- ② The new string will be stored in this global macro variable.
- ③ This macro variable can be used to insert a space between the words. In this version all spaces between words will be removed.
- ④ &BACK contains the character that will be used as the word separator, in this case a comma.
- ⑤ &MID contains the next word in the list.
- ⑥ The first word with its trailing comma (&BACK) is stored in &WRD.
- ⑦ The next word and a trailing comma are added to &WRD.
- ⑧ Determine the total length of the new comma-separated string and subtract one.
- ⑨ Since an extra comma was appended to the last word, %SUBSTR is used to exclude it. The final new string is ready to be stored in the global macro variable.

The macro call %cstr(instring=&list, numx=comma) would place the comma-separated string in the global macro variable &WRDCOMMA.

## %CSTR2

The %CSTR2 macro takes better advantage of the capabilities of the %NW macro and changes some of the basic logic used in %CSTR.

```
%macro cstr2(instring= , numx=2);
```

```

/* Count the number of words in the string;
/* Save the value in the macro variable NW;
/* Save the individual words in WORDi;
%nw(&instring)

```

these are global macro variables.

```

%global wrd&numx;
%local wrd front back i; ①

```

```
%let wrd&numx=; ②
```



```

%if &nw>0 %then %do;
 %let front=%str();
 %let back=%str();
 %do i=1 %to &nw;
 %* From the NW macro &&word&i ③ is the ith word;

 %* The new string is stored in WRD&numx and is
 %* built here;
 %let wrd&numx=&&wrd&numx&front&&word&i; ④
 %* If this is not the last word add the separator;
 %if &i < &nw %then %let wrd&numx=&&wrd&numx&back; ⑤
 %end;
%end;
%mend cstr2;

```

iteratively appending  
to the "wrd&numx"

there should be a space, or else "error" !!

- ① I like to declare all my "extra" macro variables as local. This helps to protect me from macro variable collisions (see Section 5.4.2).
- ② Initialize the macro variable. Since this is a global macro variable, an earlier execution of %CSTR or %CSTR2 might have already populated this variable and left something behind.
- ③ The individual words were stored by the %NW macro.
- ④ This word is appended to those that came before
 

|            |                                                            |
|------------|------------------------------------------------------------|
| WRD&NUMX   | The name of the macro variable holding the separated words |
| &&WRD&NUMX | The value of the macro variable WRD&NUMX                   |
| &FRONT     | A word spacer (when used)                                  |
| &&WORD&I   | The ith word in the list of words built by %NW             |
- ⑤ The separator (&BACK) is appended for each word except the last (&I = &NW).

### %CSTR3

It is also possible to separate the words without using the %NW macro or the global symbol table. The %WORDCOUNT macro in Section 11.4.4 counts the words without saving them. %CSTR3 is a macro function that expands %WORDCOUNT to not just count the words, but to separate and save them as well.

```

%macro cstr3(list);
 %* Return a list of words as a comma-separated list;
 %local clist count;
 %let count=0; ①
 %let clist=; ②
 %do %while(%qscan(&list,&count+1,%str()) ne %str());
 %let count = %eval(&count+1);
 %if &count > 1 %then %let clist = &clist,; ③
 %let clist =&clist%qscan(&list,&count,%str()); ④
 %end;
 %let clist ⑤
%mend cstr3;

```

- ❶ Initialize the count.
- ❷ Initialize the list—this should not be necessary.
- ❸ Since this is not the first word, a comma is appended prior to adding the current word ❹.
- ❹ The current word is retrieved from the list using the %QSCAN function and &COUNT. This word is then appended to the list.
- ❺ The comma-separated list is returned.

The list of words passed into the %CSTR3 macro through the macro call is replaced by the comma-separated string. Using the %LET and %PUT statements shown here:

```
%let str = aa cc g d ss;
%put |%cstr3(&str)|;
```

writes the following to the LOG:

```
|aa,cc,g,d,ss|
```

If you can assume that the word separators are always a single blank space, the blanks could be replaced with a comma with the TRANWRD function. The following statement creates a comma-separated list of words:

```
%let clist = %sysfunc(tranwrd(&list,%str(),%str(,)));
```

Using this technique will cause each blank to become a comma. Multiple blanks become multiple commas. If your string might have multiple blanks between words, you can first eliminate them by using the %CMPRES macro (see Section 12.2.3). The code becomes

```
%let clist = %sysfunc(tranwrd(%cmpres(&list),%str(),%str(,)));
```

## 11.4.6 Quoting words in a list

### %QSTR

#### Anonymous

The %QSTR macro is very similar to %CSTR (see Section 11.4.5). Both macros process a list of words one word at a time. %QSTR is used to separate words in an incoming string with commas as well as to quote the individual words. The new string is stored in the macro variable &WRDX, where x is optionally passed as the second argument. A typical macro call is shown below with &LIST containing the list of words to be quoted and separated by commas:

```
%qstr(instring=&list, numx=1)
```

In the following use of this macro, a list of values (subjects) has been stored in the macro variable &SELECT. This list is to be used with an IN operator in a WHERE clause and must therefore have its values enclosed with quotes.

```

%let clause=;
%if &select ne %then %do;
 * Call macro QSTR to put quotes around the list of
 * subjects (assumes subject numbers are character);
 %qstr(instring=&select)

 %let clause = and subject in(&wrd1);
%end;

```

The %QSTR macro calls %NW (see Section 11.4.4) to count the number of values (words) in the list that is passed into &INSTRING. The individual words are then processed one at a time to build the new string with quotes and commas added.

```

* qstr.sas
*
* Put quotes around words in a character string
* and then separate them with commas.
*
* e.g aa bb will become
* 'aa','bb'
*
*****;

%macro qstr(instring= , numx=1);

 %* Count the number of words in the string;
 %nw(&instring); ❶

 %global wrd&numx; ❷

 %if &nw>0 %then %do;
 %let front=%str('%'); ❸
 %let back=%str(','); ❹
 %do i=1 %to &nw;
 %* Select the ith word;
 %let mid=%scan(&instring,&i);
 %if &i=1 %then %do;
 %* initialize the var used to hold the new string;
 %let wrd=%str(&front&mid&back); ❺
 %end;
 %else %if &i>1 %then %do;
 %* Build the new string;
 %let wrd=%str(&wrd&front&mid&back); ❻
 %end;
 %end;

 %* Subtract 1 to eliminate the trailing comma;
 %let long=%eval(%length(&wrd)-1); ❼

 %* Save the new sting in the macro variable WRD&NUMX;
 %let wrd&numx=%substr(&wrd,1,&long); ❸

 %end;

 %*put %str(wrd&numx is &&wrd&numx);

%mend qstr;

```

- ❶ The number of words in &INSTRING are counted using the %NW macro.
- ❷ The new list will be stored in this global macro variable.
- ❸ The value of &FRONT (a single quote in this case) will be pre-appended to the front of each word.
- ❹ The value of &BACK will be appended to the back of each word.
- ❺ &WRD temporarily holds the list and is initialized for the first word.
- ❻ Each subsequent word is appended, with its quotes and commas, to the previous words.
- ❼ There is an extra comma at the end. Determine the length of the desired portion of the list.
- ❽ %SUBSTR is used to eliminate the last character (the extra comma).

If you know that all the words in the list are separated by exactly one space, the following statement could be used instead of the %QSTR macro:

```
%let qlist =
 %str('%')%sysfunc(tranwrd(&list,%str(' '),%str(','))%str('%');
```

## 11.4.7 Check for existence of variables

### %VAREXIST

A process might generate a list of variable names that are to be used in a VAR statement or in a KEEP= option. If one or more of the variables are not on the data set of interest, errors will usually be generated; therefore, the %VAREXIST macro can be used to verify that each of the variables in the list is on the data set of interest.

In this macro the VARNUM DATA step function is used with %SYSFUNC to verify each variable's existence, and the result is passed back.

```
%macro varexist(dsn,varlist);
 %local dsid i ok var num rc;
 %let dsid=%sysfunc(open(&dsn,i)); ❶
 %let i=1;
 %let ok=1; ❷
 %let var = %scan(&varlist,&i); ❸
 %do %while(&var ne);
 %let num = %sysfunc(varnum(&dsid,&var)); ❹
 %if &num=0 %then %let ok = 0; ❺
 %let i = %eval(&i+1); ❻
 %let var = %scan(&varlist,&i); ❼
 %end;
 %let rc = %sysfunc(close(&dsid)); ❽
 %ok ❾
%mend varexist;
```

- ❶ The data set of interest is opened for access.
- ❷ &OK is initialized to 1 (all variables exist on the data set).
- ❸ %SCAN is used to retrieve the first variable name.

- ④ VARNUM is used to retrieve the number of the variable, given its name (&VAR). If the variable is not found a 0 is returned.
- ⑤ If the variable is not found, then change &OK to 0.
- ⑥ Move on to the next word in the list.
- ⑦ Retrieve the next variable in the list and repeat the process until no more variable names are found in the list.
- ⑧ Close the data set.
- ⑨ The value of &OK is returned. 0 indicates that one or more of the variable names are not present.

This macro works as a macro function so it can be used in a %PUT or %IF statement. The call is used in an %IF below.

```
%macro doit;
 %let list= name age sex gg;
 %if %varexist(sashelp.class,&list) %then
 %put all are on data set;
 %else %put missing var;
%mend doit;
%doit
```

The variable GG does not exist on the data set SASHELP.CLASS, so %DOIT returns

|             |
|-------------|
| missing var |
|-------------|

## 11.4.8 Remove repeated words from a list

### %DISTINCTLIST

When a list of variable names has been created from a source such as PROC CONTENTS or SASHELP.VCOLUMNS, you can be confident that each variable will occur in the list exactly once. Sometimes, however, the list will be constructed in such a way that a given variable name might occur more than once. If this list is used in a KEEP or VAR statement, or another statement that accepts a list of variables, the repeated names may cause errors.

The following DATA step shows a simple example where this type of problem can show up. It is likely that the lists of incoming variables from each of the three incoming data sets are similar and it is possible that they are not exactly the same. The KEEP= option will have some (if not most) variables listed more than once.

```
data allyears(keep=&list00 &list01 &list02);
 set year2000(keep=&list00)
 year2001(keep=&list01)
 year2002(keep=&list02);
 ...code not shown...
run;
```

While the overlapping variables will not cause an ERROR in this usage, it would be more convenient to have a single unified list of variables. Using the %DISTINCTLIST macro, the code becomes

```
%let alllist = %distinctlist(&list00 &list01 &list02);
data allyears(keep=&alllist);
 set year2000(keep=&list00)
 year2001(keep=&list01)
 year2002(keep=&list02);
 ...code not shown...
run;
```

The macro %DISTINCTLIST will remove any repeated word within a list. Each word is selected from the incoming list and added to a new list only if it has not already been found. The %INDEX function is used to see if any given word has already been added to the new list.

```
%macro DistinctList(list);
 %local dlist i;
 /* Select the first word;
 %let dlist = %scan(&list,1,%str()); ❶
 %let i=2; ❷
 %do %while(%scan(&list,&i,%str()) ne %str()); ❸
 /* There is another word.
 /* Has it already been selected?;
 %if %index(&dlist,%scan(&list,&i,%str()))=0 ❹ %then %do;
 /* First occurrence of this word add it to the list;
 %let dlist = &dlist %scan(&list,&i,%str()); ❺
 %end;
 /* Increment counter to get the next word;
 %let i = %eval(&i+1); ❻
 %end;
 &dlist ❼
%mend distinctlist;
```

- ❶ The first variable name is automatically added to the new list.
- ❷ Set the word counter for the next possible word.
- ❸ Determine if there is another word to check.
- ❹ The %INDEX function is used to see if the current word has already been added to the new list (&DLIST).
- ❺ This is the first occurrence of this word so add it to the new list.
- ❻ Increment the word counter.
- ❼ Return the new list.

This macro has a logic bug and can return an incorrect result under some circumstances. Consider the following list:

```
%let mylist = aa bb cc a bb c;
```

The %DISTINCTLIST macro will create a list without the variables A or C. This is because the %INDEX function will see the variable A as present when it evaluates the variable AA. The DATA step function INDEXW avoids this difficulty by only searching for full words. Since there is no %INDEXW, the %SYSFUNC macro function can be used to make use of INDEXW. The %IF statement ❹ becomes

```
%if %sysfunc(indexw(&dlist,%scan(&list,&i,%str())))=0 %then %do;
```

## MORE INFORMATION

%INDEXW is presented as a macro function in Section 7.6.1.

---

## 11.4.9 Working with a list of class variables

### %INDVAR

**Jørn Lodahl**

Often in statistical analysis it is desirable to create a matrix of indicator variables where each variable takes on the value of 0 or 1 depending on the value of a classification variable. This macro creates these variables and assigns each the value of 0 or 1. This is similar to the design matrix created by PROC GLMMOD.

A quick example of this macro uses the following simple data set in which the variable I takes on the values of ., 0, 1, 2, 3, and 5:

```
data simple;
 i=.; output;
 i=0; output;
 do i= 1 to 3,5;
 output;
 end;
run;

%indvar(simple,i,ind,basevar= 2)

proc print data=simple;
 title 'A Simple data set when basevar is 2';
run;
```

The LOG displays the value taken on by the macro variable &IND:

```
macro indvar created macro variable ind = ind0 ind1 ind3 ind5
```

The following PROC PRINT of the modified SIMPLE data set shows the new indicator variables. Notice that although IND2 is not on the list of variables stored in &IND, it is in the data set (&BASEVAR=2).

A Simple data set when basevar is 2

| Obs | i | ind0 | ind1 | ind2 | ind3 | ind5 |
|-----|---|------|------|------|------|------|
| 1   | . | 0    | 0    | 0    | 0    | 0    |
| 2   | 0 | 1    | 0    | 0    | 0    | 0    |
| 3   | 1 | 0    | 1    | 0    | 0    | 0    |
| 4   | 2 | 0    | 0    | 1    | 0    | 0    |
| 5   | 3 | 0    | 0    | 0    | 1    | 0    |
| 6   | 5 | 0    | 0    | 0    | 0    | 1    |

PROC SQL is used to build the list of the names of the new indicator variables. These variables are then constructed with assignment statements in a DATA step. The assignment statements are created dynamically within a %DO %WHILE loop.

```
/******
```

```
SYNTAX:
```

```
%indvar(datasetname,classvar,prefix_for_ind_var,basevar=)
```

This macro creates a series of indicator variables from a class-variable and adds them to the dataset. The class-variable MUST be restricted to positive integers or zero. The prefix\_for\_ind\_var is used as the prefix for names of the indicator variables.

A macro variable named prefix\_for\_ind\_var is also created as a string. This string contains the names of all of the newly created indicator variables. The names are separated by a space. By default (basevar=min), the indicator variable associated with the minimum value of the class-variable is excluded from this list (the indicator variable is still included on the data set). Setting basevar=none excludes no names from the list. A specific name can be excluded by setting basevar=x, where x is a value taken on by the class-variable.

EXAMPLE:

Consider a data set where the variable class takes on the values 1, 2 and 4.

```
%indvar(&syslast,class,c)
```

Variables c1 c2 and c4 with values 0 or 1 are added to the data set. (c3 is always 0 since class is never 3 and therefore c3 is not created).

Moreover, the macro variable c equals "c2 c4" (c1 is omitted since by default basevar=min and 1 is the minimum value of class). If instead %indvar(&syslast,class,c,basevar=2) was called, c2 would be omitted from the macro variable. Basevar=none includes all.

```
*****/
```



```

%macro indvar(dataset,byvar,prefix,basevar=min);
%local item list value min;
%global &prefix;

/* generate list */
PROC SQL NOPRINT;
 SELECT DISTINCT &byvar INTO :list SEPARATED BY ' ' ❶
 FROM &dataset
 ORDER BY &byvar;
 SELECT min(&byvar) INTO :min ❷
 FROM &dataset;
QUIT;

/* create variables */
%LET item=1;
%LET value = %scan(&list, &item) ; ❸
/* in other words: element no. &item in &list = &value */
%LET &prefix= ; /* turns into a list of "all" ind variables */❹

data &dataset;
 set &dataset;
 %IF &basevar=min %THEN %LET basevar=&min;; ❺
 %do %while(%quote(&value) ^=) ;
 IF &byvar=&value THEN &prefix&value=1; ELSE &prefix&value=0;❻
 %IF %quote(&value)^=%quote(&basevar) %THEN ❼
 %LET &prefix=&&prefix &prefix&value;; ❽
 %LET item = %eval(&item +1) ; ❾
 %let value = %scan(&list, &item);
 %end;
run;
%put macro indvar created macro variable &prefix = &&prefix; ❿
%mend indvar;

```

- ❶ The macro variable &LIST contains the unique values that are taken on by the classification variable (&BYVAR). The elements of the list are separated by blanks.
- ❷ The minimum value taken on by the classification variable is stored in &MIN. This is used if &BASEVAR=min ❺.
- ❸ The %SCAN macro function is used to return the *i*th (&ITEM) word of the classification variable (&LIST), and this word is temporarily stored in &VALUE. At this point in the program, &ITEM=1 so the first word is selected.
- ❹ &PREFIX is used to store the text portion of the name of the indicator variables. This same text (in this example IND) is used to name the macro variable that holds the list of indicator variable names. When &PREFIX appears in the %LET statement, IND becomes the name of a macro variable.
- ❺ When &BASEVAR=min, then substitute the minimum value taken on by the classification variable. There are a couple of places that the author uses a double semicolon (❺ and ❽) to close the %IF statement. Although unnecessary, this does not cause a problem in these instances, since there is no %ELSE statement following the %IF statements.

- ⑥ This DATA step IF statement sets up the assignment statements that create and load the indicator variables. A resolved statement might be

```
IF i=2 THEN ind2=1; ELSE ind2=0;
```

- ⑦ Exclude the &BASEVAR value from the list (not the data set).
- ⑧ This %LET statement appends the new variable name (&PREFIX&VALUE) onto the existing list (&&&PREFIX) and stores the result back into &IND, which is pointed to using &PREFIX. Here, &&&PREFIX resolves to &IND, which resolves to IND1 IND2....
- ⑨ Increment the item number to step through the list of values taken on by the classification variable.
- ⑩ This %PUT statement displays both the name of the macro variable (&PREFIX) that contains the list of variables as well as the value of that variable (&&&PREFIX).

---

## 11.5 Counting Observations in a Data Set

A variety of methods can be used to count the number of observations in a SAS data set. In Sections 2.7.2, 11.2.1, and 11.5.4, PROC SQL is used. The SET statement options (POINT and NOBS) can also be used, as in Sections 11.2.2 and 11.2.3. More efficient tools are also available. These use the %SYSFUNC function in conjunction with functions such as OPEN, ATTRN, and CLOSE.

### SEE ALSO

Yindra (1997) has an example of a macro that makes a decision branch based on the number of observations in the data set.

*SAS Screen Control Language, Reference, Second Edition* describes the ATTRN SCL function in detail (pp. 238–239).

%OBSCNT (Section 11.5.1) was originally adapted from a similar macro in *SAS Macro Language, Reference, First Edition* (p. 242). That example uses ATTRN to retrieve both the number of variables as well as the number of observations. The same example was used with slight modification in %TESTPRT (Section 11.5.2).

Kretzman (1992) discusses a similar macro. The related functions, ATTRC and VARNAME, are used by Bramley (2001). Conley (2000) uses the ATTRN, VARNAME, and VARLABEL functions to retrieve and use variable labels.

The ATTRN function is used to get several data set attributes including the number of observations in Allen (2003).

## 11.5.1 Using %SYSFUNC and ATTRN

### %OBSCNT

In the following example, the macro %OBSCNT acts like a macro function in that the macro call resolves to a value that is the number of observations in the stated data set:

```
%macro obscnt(dsn); ❶
%local nobs;
%let nobs=.;

 /* Open the data set of interest;
 %let dsnid = %sysfunc(open(&dsn)); ❷
 /* If the open was successful get the;
 /* number of observations and CLOSE &dsn;
 %if &dsnid %then %do; ❸
 %let nobs=%sysfunc(attrn(&dsnid,nlobs)); ❹
 %let rc =%sysfunc(close(&dsnid)); ❺
 %end;
 %else %do; ❻
 %put Unable to open &dsn - %sysfunc(sysmsg());
 %end;

 /* Return the number of observations; ❼
 &nobs ❽
%mend obscnt;
```

- ❶ The user passes the name of the data set into the macro using &DSN.
- ❷ The selected data set is opened using the %SYSFUNC and OPEN functions. When opened, it is assigned an identification number, which is stored in &DSNID.
- ❸ If the data set was found and was opened successfully, the macro variable &DSNID will be greater than 0. The %IF expression will, therefore, be true.
- ❹ The ATTRN function is used to determine the number of observations in the data set. In this case, the NLOBS option returns the number of observations, excluding those marked for deletion. The ATTRN function can be used to make a number of queries on the data set once it is opened. These include password and indexing information, as well as the number of variables and the status of active WHERE clauses.
- ❺ The data set should be closed after retrieving the desired information.
- ❻ When the OPEN is unsuccessful, you might want to write a message to the LOG. The SYMSG( ) function returns the reason the OPEN failed.
- ❼ This must be a macro comment. An asterisk-style comment would result in the comment itself being written as text along with the number of observations. This would result in syntax errors if the macro is used as it is in the following %PUT statement. See Section 5.4.1 for more on macro comments.
- ❽ Because this is the only text in the macro that is not part of a macro programming statement, the resolved value of &NOBS will, in effect, be returned to the calling program and its value will be a period (.) if the data set was not opened successfully.

*Attrn function  
Returns the value  
of a numeric attribute  
for a specified data  
set.  
Syntax:  
attrn(data set -  
attr name)*

The following program creates the data set A and then calls %OBSCNT to write the number of observations to the LOG:

```
data a;
 do i = 1 to 10;
 x=i**i;
 output;
 end;
run;

%put number of obs is %obsent(a);
```

The following line is written to the LOG:

|                     |
|---------------------|
| number of obs is 10 |
|---------------------|

### MORE INFORMATION

The example in the Section 11.5.2 also uses the ATTRN function.

### SEE ALSO

Hamilton (2001) discusses various problems, including errors returned by the ATTRN function, associated with capturing the number of observations in a data set. He also introduces and discusses the macro %MTANYOBS in detail.

The ATTRN function is also used by Zirbel (2002) in a macro that compares two data sets.

The ATTRN function is used to get several data set attributes, including the number of observations, in Allen (2003).

---

## 11.5.2 Controlling observations in PRINT listings

### %TESTPRT

**Jerry Kagan**  
Kagan Associates, Inc.

In several of the earlier sections of this book, the macro %LOOK is developed to check the contents of a SAS data set. %LOOK is improved in the macro shown in this section.

The macro %TESTPRT is used during program testing and will print out the first *n* observations from the last data set created. The macro is designed to be called immediately after each DATA step so that the first *n* observations will be displayed from the data set that is created or modified in that step. Knowing what is in a data set, of course, helps you to know where things are going wrong.

Additional utility is gained from the macro printing the name of the data set and the number of observations and number of variables in a title line. It is possible to turn the macro on or off by setting a globalized macro variable (&TEST) to 0 or 1. This enables you to leave the macro calls in your production code after testing is complete.

```

/*****
* Program: TestPrt.SAS
* Language:SAS 6.12/MACRO
*
* Purpose:This macro prints samples of the last dataset created for
* program testing
*
* Protocol:%let test = 1; *** Turn macro on;
* %let obs = 10; *** Print first 10 obs from dataset;
*
* %testPrt(&obs) *** First &obs is printed;
*
* Author: Jerry Kagan
* Kagan Associates, Inc.
*
* jerrykagan@msn.com
*
* Date: 29Jun1993
*
* Revisions:
* 21Feb97 JBK Modified to use new sysfunc for simplicity and
* efficiency
*****/

```

```
%macro TestPrt(obs);
```

```
 %if &test %then
```

```
 %do;
```

```
 %let _dsid = %sysfunc(open(&syslast)); ①
```

```
 %if &_dsid %then %do;
```

```
 %let _nobs = %sysfunc(attrn(&_dsid,NOBS)); ②
```

```
 %let _nvar = %sysfunc(attrn(&_dsid,NVARS)); ③
```

```
 %let _rc = %sysfunc(close(&_dsid)); ④
```

```
 %end;
```

```
 proc print data=&syslast (obs=&obs) uniform;
```

```
 %if &obs < &_nobs %then %do;
```

```
 title1 "&syslast (Created with &_nobs observation(s) "
 "&_nvar variable(s), first &obs printed)";
```

```
 %end;
```

```
 %else %do;
```

```
 title1 "&syslast (Created with &_nobs observation(s) "
 "&_nvar variable(s), all printed)";
```

```
 %end;
```

```
 run;
```

```
 %end;
```

```
%mend TestPrt;
```

- ① The most recently modified data set (&SYSLAST) is opened.
- ② The number of observations that are contained in the opened data set (&\_DSID) is returned using the ATTRN function with the NOBS option.
- ③ The ATTRN function can also be used to return the number of variables in the data set.
- ④ The data set is closed using the CLOSE function.

As it stands, %TESTPRT will wipe out any titles that have been set prior to calling the macro. This might or might not be a problem. A potential modification to avoid this situation could be made by determining the next available title number (see Sections 9.5.2 and 10.2.3).

### 11.5.3 Using a DATA \_NULL\_ step

#### %NUMOBS

When you do not need to create a macro function, and the DATA step can be used, the NOBS= option on the SET statement can be used to determine the number of observations in the SAS data set.

In the following example, a DATA \_NULL\_ step is used with a SET statement to place the number of observations into a macro variable through the use of the NOBS= option.

```
%macro numobs(dsn);
%global numobs; ❶
data _null_;
 call symput('numobs',left(put(dsnobs,5.))); ❷
 stop; ❸
 set &dsn nobs=dsnobs; ❹
run;
%mend numobs;
```

- ❶ The macro variable that will hold the number of observations is added to the global symbol table.
- ❷ CALL SYMPUT is used to create the macro variable NUMOBS. The value of the variable DSNNOBS is assigned during the compilation of the DATA step.
- ❸ The STOP statement prevents execution of the SET statement; consequently, no observations are read from &DSN.
- ❹ The NOBS= option is used to load the number of observations into the variable DSNNOBS.

**CAVEAT:** This technique, which is widely used, generally works correctly. If, however, the data set contains deleted observations, the NOBS= option will return the incorrect number. Since deleted observations are usually the result of deleting observations while in an interactive session (with, for example, PROC FSEDIT), this is generally not a major issue.

#### SEE ALSO

The %SPLIT macro demonstrated by Gerlach and Misra (2002) also uses the NOBS= option on the SET statement.

## 11.5.4 Using SQL

### %COUNTER

David Shannon

This macro uses a PROC SQL step (see Section 2.7) to count the number of observations in a SAS data set. The macro parameters are the name of the data set and the name of the macro variable that will be created and globalized. In the macro, &INTO resolves to the name of the macro variable that will hold the counter, and &&&INTO will resolve to the number of observations.

Consider the macro call %counter(mydata,cnt). During the execution of %COUNTER, the following occurs:

| Macro variable | Resolves to              |
|----------------|--------------------------|
| &INTO          | cnt                      |
| &&&INTO        | && &INTO → &CNT          |
| &CNT           | {number of observations} |

```

/*-----
COUNTER
The number of observations contained in the input dataset,
after completion of the DATA step, is stored in the macro
variable.

Parameter

DATA
INTO

Usage : %COUNTER(NEWDAT,NOOBS);
^ ^
- Dataset name
Note(s) : Datasets may be empty, but must have at least one
variable, and therefore must exist.
Written by: David Shannon, V1, 14/4/97, V6.10+
Bugs to : David@schweiz.demon.co.uk
References: Getting Started With the SQL Procedure, S.I., 1994.
-----*/

```

```

%MACRO Counter(data,into);

%*****;
%* Define local macro variables(those only referred to locally)*;

%Local DATA INTO; ❶

%*****;
%* Set defaults if not provided in positional macro call. *;

%If &DATA EQ %Then %Let DATA=_last_;
%If &INTO EQ %Then %Let INTO=COUNT;

%*****;
%* Make output variable global, *;
%* hence can be referred to outside this macro. *;

%Global &INTO; ❷

%*****;
%* Count observations, *;
%* store result in the global macro variable *;

Proc Sql noprint;
 Select count(*) into:&INTO ❸
 From &DATA;
Quit;

%PUT Counted &&&INTO observations.;
%PUT Stored in macro variable &INTO..; ❹

%MEND Counter ;

```

The following discussion assumes that this macro call has been made:

```
%counter(mydata,cnt)
```

- ❶ The macro variables &DATA and &INTO are declared local. In this example, these macro variables are macro parameters. They will be LOCAL by default, and the statement is not really necessary. However, when you use a macro within an application, it might be wise to get in the habit of using the %LOCAL statement to protect the application. This prevents your macro from inadvertently changing the value of a globalized macro variable of the same name that you might not know about and that has been defined elsewhere in the application.
- ❷ The macro variable &INTO resolves to CNT, which is then used as the name of a globalized macro variable.
- ❸ The number of observations is loaded into the macro variable named by &INTO (CNT).
- ❹ &&&INTO resolves to the number of observations. The macro variable &INTO. resolves to the name of the macro variable that will be available outside of the %COUNTER macro.



If MYDATA has 30 observations, the macro call %COUNTER (MYDATA, CNT) will result in the following lines being written to the LOG:

```
137 %counter(mydata,cnt)
NOTE: The PROCEDURE SQL used 0.11sec

Counted 30 observations.
Stored in macro variable cnt.
```

The macro variable &CNT has now been globalized, is available outside of the macro %COUNTER, and has a value of 30.

---

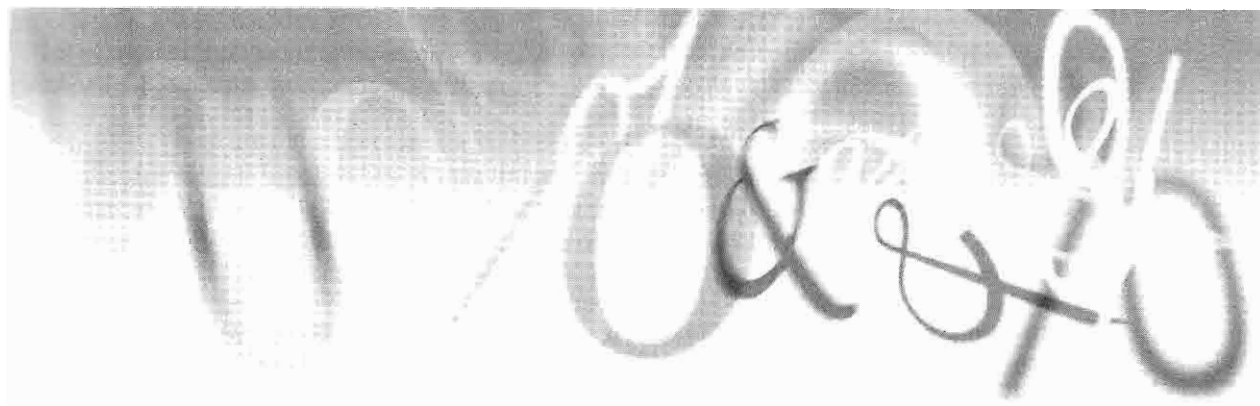
## 11.6 Chapter Summary

It is very common to use the macro language to interface with SAS data tables. The tables can serve as both a source of information, which can drive the macros, or even as the result of the macro's execution.

Through the use of the macro language it is possible to create self-documenting flat files from SAS data tables.

Macros and macro language statements can be used to control the subsetting process when selecting observations from data tables. While working with these data tables, the macro can detect and work with the columns, and build and work with lists of variables.

When using applications to create data tables, it is sometimes advantageous to be able to detect whether or not a data table has been created. The first step in this process is to determine if the data table even exists by using a DATA step function. The second step is to see if the data set contains any observations. There are several tools available to the macro language that can be used to determine the number of rows in a table.



# Chapter 12

---

## Building and Using Macro Libraries

- 12.1 Library Overview 328
  - 12.1.1 Using %INCLUDE as a macro library 329
  - 12.1.2 Using compiled stored macros 330
  - 12.1.3 Using the autocall facility 331
- 12.2 Macro Library Essentials 332
  - 12.2.1 The macro library search order 332
  - 12.2.2 Establishing a macro library structure and strategy 333
  - 12.2.3 Interactive macro development 334
  - 12.2.4 Modifying the SASAUTOS system variable 335
- 12.3 SAS Autocall Macros 335
  - 12.3.1 %VERIFY 337
  - 12.3.2 %LEFT 338
  - 12.3.3 %CMPRES 339
  - 12.3.4 %LOWCASE 340
  - 12.3.5 %TRIM 342
  - 12.3.6 %DATATYP 343
  - 12.3.7 %COMPSTOR 344
- 12.4 Chapter Summary 345

The management of large numbers of macros can be problematic. Names and locations must be remembered, changes must be managed, duplication and variations of individual macros must be monitored, and efficiency issues must be taken into consideration. These problems come even more to the forefront when macros are shared among programmers—especially in a networked environment.

Macro libraries enable you to control and manage the proliferation of your macros. Libraries can consist of a formal use of the %INCLUDE statement, a group of programs defining macros that can be automatically called, or even collections of macros that were previously compiled and then stored.

Macro libraries enable the macro developer to store, maintain, and manage large numbers of macros. The two types of macro libraries (autocall and stored compiled macros), were introduced in Version 6. These libraries enable the user to avoid the use of the %INCLUDE statement to bring in macro definitions, while efficiently accessing and utilizing the power of the macro.

Using macro libraries is not difficult nor is it overly complicated. Both styles of libraries are established through the use of the OPTIONS statement, although the autocall macro facility is usually available by default.

Macro libraries foster an environment that promotes good macro programming practices by making it easier to avoid the use of duplicate macro definitions. As a result of using these libraries, you will find that it is easier to track and maintain your macros, and macro programming will become even more fun.

### MORE INFORMATION

Sections 3.3.3 and 3.3.4 introduce the system options used to establish and control the use of macro libraries.

### SEE ALSO

First (2001a) discusses various aspects of macro libraries. Additional detail on macro libraries can be found in Carpenter (2001a), and a more general discussion on strategies for building automated systems, including the use of macro libraries, can be found in Carpenter and Smith (2001b).

Burlew (1998, Chapter 8) discusses various aspects of macro libraries. A good summary of the use of macro libraries can also be found in *SAS Macro Language: Reference, Version 8*, pp. 101–105.

## 12.1 Library Overview

The use of macros is essential to an automated and flexible system. This implies that the control of the macro code is very important to the maintenance of the application. All too often macro definitions become buried within the programs that use them. The result is usually a proliferation of multiple versions of similar macros in multiple programs, each with its own definition of the macro. Parallel code, two programs or macros that do essentially the same thing, is an especially difficult problem in large applications that are maintained by multiple programmers. Even when there is only one programmer, there is a tendency to clone a program ("with slight modifications"). Will each version be updated each time a bug is found? Who makes sure that the update happens? Are the various versions of the macro documented? These problems are minimized by placing the definition in a macro library.

Macro libraries are used to avoid the problem of macro cloning by providing a single location for all macro definitions. Rather than cloning the macro, it is adapted (generalized) to fit each of its calling programs and the new generalized macro is stored in one of the libraries. Once in a library, there will only be one macro definition to maintain, and it can be used by as many

programs as is needed. Obviously this requires documentation as well as diligence. Part of the solution is to place **all** macro definitions in a common library, which is accessible to all programs in the application. This way no macro definition will be buried within a program.

There are three types of macro libraries: %INCLUDE, compiled stored macros, and autocall macros. Each has its advantages and disadvantages, and best of all they can be used in conjunction with each other!

It is important to understand the behavior of these libraries, how they are set up, and how they interact with each other.

---

### 12.1.1 Using %INCLUDE as a macro library

The least sophisticated approach to setting up a macro library is obtained through the use of %INCLUDE files that store macro definitions. This was often the only viable type of macro library in SAS before Version 6, and not a few SAS programmers have failed to make the transition to true macro libraries.

Strictly speaking, the use of %INCLUDE does not actually set up a macro library. The %INCLUDE statement points to a file, and when the statement is executed, the indicated file (be it a full program, macro definition, or a statement fragment) is inserted into the calling program at the location of the call. When using the %INCLUDE to build a macro library, the included file will usually contain one or more macro definitions.

Although you can identify the file to be included directly from within the %INCLUDE statement, it is usually done indirectly through the use of a *fileref*.

```
filename macdef1 'c:\mymacros\macdef1.sas';
.....
%include macdef1;
.....
%macdef
.....
```

One way to build a library or collection of files that are to be included, is to place them in one central location. This way they will be easier to find and maintain. Some users of include libraries will indicate that these files are to be included, and therefore might not be complete programs, by using an extension of INC instead of SAS. While using an extension of INC in no way hampers the file's use by SAS, Microsoft Windows users should be aware that the file will not necessarily be marked by a SAS registered icon.

As was mentioned above, the included file can contain any snippet of SAS code. Within the context of this section, however, we are interested in building macro libraries, and to do this, the included file would contain a macro definition—for example, %MACRO and %MEND statements. There are a couple of efficiency issues that the user of %INCLUDE libraries should remember.

Generally a given file should not contain more than one macro definition, and the macro being called should not be called from within the included code. The first is important because if the file contains multiple macro definitions, then all the definitions must be loaded and compiled just to use one of the macros. Of course if all the macros will eventually be used within that SAS session, then this does not really matter. Second, if the macro call is placed within the included code, the code will need to be re-included (and the macro recompiled) each time that particular macro call is to be used.

The greatest disadvantage of using %INCLUDE in a large application is tracking and maintaining the *filerefs* that point to the individual files. While this is less of a problem in static systems, it can still be non-trivial. This issue virtually goes away with the use of the other macro library alternatives that are discussed next.

### 12.1.2 Using compiled stored macros

Macros are always compiled before they are executed, and it is possible to store the compiled code for future use. Compiled macros are stored in a catalog named SASMACR, and by default this catalog is stored in the WORK library. Each compiled macro is stored with the entry type of MACRO and with an entry name corresponding to the name of the macro. When a macro is called, SAS automatically searches this catalog for the compiled version of the macro that was called. Permanently compiled stored macros are also written to a SASMACR catalog, but this catalog is stored in a different (permanent) library.

The ability to store and make use of compiled stored macros is by default turned off. The user turns on the ability to make use of compiled stored macros with the system option MSTORED (NOMSTORED turns it off). The library that is to be used to store the permanent SASMACR catalog is specified by using the SASMSTORE= option.

The following OPTIONS statement turns on the use of compiled stored macros and designates PROJSTOR *libref* as the location for the catalog PROJSTOR.SASMACR:

```
options mstored sasmstore=projstor;
```

You cannot use more than one location reference with the SASMSTORE option. Therefore, if you do want to search multiple catalogs, then you need to use either a concatenated *libref* or a concatenated catalog.

The following creates a concatenated location (MULTI) from the two *librefs* PROJSTOR and ALLMSTOR. Now both locations will be searched when SAS looks for the SASMACR catalog. If the compiled macro is in more than one catalog, the definition found first will be used (read left to right).

```
libname projstor 'c:\myprojA\sasmacros';
libname allmstor 'f:\GroupProjA\saspgms\macros';
libname multi (projstor, allmstor);

options mstored sasmstore=multi;
```

By default the compiled macro is stored in WORK.SASMACR. You redirect this location at compile time through the use of the /STORE option on the %MACRO statement. For the SASMSTORE definition above, the macro AERPT shown below, will be stored in the PROJSTOR.SASMACR catalog:

```
%macro aerpt(dsn, stdate, aelist) / store;
```

The use of compiled stored macros is not without problems. The developer must make sure that the macro does not exist in more than one catalog, or if it does (on purpose, of course) that the search order of the catalogs is correct. Also, since it is not always possible to reconstruct the code used to create the compiled macro, the developer must make sure that the code is correctly maintained independently from the SASMACR catalog. New options and statements discussed below are designed to assist with the recovery of the code used to compile macro definitions.

This highlights a second problem for these libraries. The compiled macros are all in one location, in the SASMACR catalog; however, the code itself could be anywhere. Usually developers that use compiled stored macro libraries will also have a central location to store the source code. A logical location is in the same directory as the SASMACR catalog.

The problem of retrieval of the source code is solved to some extent in SAS 9.1 with the addition of a new option that can be used with /STORE on the %MACRO statement. When the SOURCE or SRC option is included, SAS will save the source code as part of the macro entry in the SASMACR catalog. This entry will contain all the code including the %MACRO and %MEND statements. To store the source code for %AERPT the %MACRO statement becomes

```
%macro aerpt(dsn, stdate, aelist) / store source;
```

You cannot store the source code when there are nested macro definitions; however, as was discussed in Section 5.1.3, it is rarely a good idea to nest macro definitions anyway. Although the source code is stored in the catalog, you will not see a catalog SOURCE entry.

Once you have source code stored in the SASMACR catalog, the %COPY statement, also new to SAS 9.1, can be used to copy it from the catalog to an external file or *fileref*.

### Syntax

```
%COPY macro_name </options>;
```

Options include the following:

**LIB=<libref>**

library (other than WORK) that contains the catalog with the SOURCE code.

**OUTfile=<fileref | external\_file>**

output destination of the %COPY statement (defaults to LOG).

**SOURCE or SRC**

specifies that the code is to be copied.

To copy the source code for %AERPT from the SASMACR catalog to an external file, the following might be used:

```
%copy aerpt / lib=multi
 out='c:\temp\ aerpt.sas'
 src;
```

---

## 12.1.3 Using the autocall facility

When a requested macro is not found in a SASMACR catalog, the autocall library is searched. Much like an %INCLUDE file, this library contains the macro definitions in the form of SAS code. When a macro is called, SAS searches for a file in the specified autocall location with the **same** name as the name of the macro. The code in the corresponding file is then submitted for processing. Since this file contains the macro definition, the macro is then compiled and made available for execution. Under Windows this location is a folder and each macro definition is an individual file. Under MVS a partitioned data set (PDS) is used with each member corresponding to a macro.

By default the ability to make use of the autocall facility is turned on. The following code makes sure that the autocall facility is available (MAUTOSOURCE), and it specifies the *filerefs* of the locations (SASAUTOS=) that contain the SAS programs with the macro definitions:

```
options mautesource sasautos=(projauto allauto);
```

The previous search path for the autocall library fails to point to the autocall macros supplied by SAS. This could be a major problem and it is discussed further in Section 12.3.

Be sure that you use *filerefs*, **not** *librefs*, to specify the locations of the autocall programs (Burlew, 1998, correctly specifies the locations with the FILENAME statement). You can also replace the *fileref* in the SASAUTOS= option with a direct reference; however, this approach is less flexible and is not as "clean."

When you have a series of autocall libraries it is sometimes difficult to determine from which location a given macro was drawn. The SAS 9 system option MAUTOLOCDISPLAY can be used to write the location to the LOG. If you call a macro and SAS must retrieve it from an autocall library, the location of that library is noted.

### MORE INFORMATION

Section 12.3 shows how to create concatenated autocall libraries that will not cancel SAS software's ability to locate and utilize the autocall macros supplied by SAS.

### SEE ALSO

A short discussion of the use of the autocall facility can be found in Jensen and Greathouse (2000).

---

## 12.2 Macro Library Essentials

Although establishing and using macro libraries is fairly straightforward, there are a number of issues that arise that tend to make the process more difficult. Issues that you might want to know more about before creating or using macro libraries include

- how SAS searches for a macro definition
- how to establish a macro library structure
- how to develop macros in an interactive environment
- how to modify the SASAUTOS system variable.

---

### 12.2.1 The macro library search order

As these libraries of macros are established it is important for both the developer and the user to understand the relationship between them. One of the more important aspects of this relationship is the order of locations that SAS searches for a macro definition.

When a macro is called, the macro facility must first find the macro definition before it can be compiled and executed. Since the macro definition can be stored in any of several locations, the developer needs to be able to control the search. The search for the macro definition uses the following order:

1. WORK.SASMACR
2. stored compiled macros
3. autocall macros

The first time that a macro is called it will not be found in any of the catalogs of compiled macros, but once it is called, it will be compiled and the compiled code will be stored (in WORK.SASMACR unless otherwise specified). On successive calls to that macro, its compiled definition will be found and the search will not extend to the autocall library a second time. This process minimizes the compilation of the macro.

This is a major advantage over the use of %INCLUDE as a macro library. When a macro definition is brought into the program through %INCLUDE, it will be compiled for each %INCLUDE. Of course if the programmer is careful, this is not such a bad thing. If the %INCLUDE appears only once, the macro will be compiled at that time and added to the WORK.SASMACR catalog, and the compiled macro definition will be used from then on (as long as it is not included another time).

---

### 12.2.2 Establishing a macro library structure and strategy

Placing your macros in a library will help to organize your programs. However, in larger groups, projects, or in organizations with multiple programmers sharing macros, it becomes necessary to organize the libraries themselves. In the designations of the locations of both the autocall library (SASAUTOS=) and the compiled stored macros (SASMSTORE=) shown above, there are two locations. By specifying multiple locations for the libraries it becomes possible to organize them into collections of macros. Typically, for the work that I do, I like to arrange the collections according to how the macros are to be used. Macros that are specific to a task or project will be placed in the location that will be searched first. Then the macros that are more generalized or are useful to multiple projects are placed in a location that will be available to users of all projects. This arrangement enables the developer to create general tools that are available to everyone as well as specific macros that apply to only one project or task.

The question then is not **if** you should set up a library, but which macro library setup to use.

The %INCLUDE library and the autocall library can be set up in a very similar fashion. The primary advantage of the autocall library is that the developer does not need to manage or work with the individual *filerefs*, as these are controlled automatically through the SASAUTOS= system option.

Unless a macro is unusually large or complex, it generally takes very little time to locate and compile a macro stored in the autocall library. This suggests that there is not a substantial time savings in using the compiled stored macro library. Since both libraries require the developer to store and maintain the source code, there does not seem to be a compelling reason to adopt one library type over the other. For this reason and because almost all SAS sites already use the autocall facility (if for no other reason than to get access to the autocall macros supplied by SAS—see Section 12.3), the autocall library seems to be used much more frequently than the compiled stored macros library.



One strategy that has worked well in limited situations combines these two types of libraries. This combined library approach specifies an autocall library ( `SASAUTOS=`). It also turns on the compiled stored macro library ( `MSTORED`) and then points the location ( `SASMSTORE=`) to the **same** location as the autocall library. Since the `SASAUTOS=` *fileref* and the `SASMSTORE=` *libref* both point to the same directory, the `SASMACR` catalog will reside in the same location as the source code that defines the macro. Once this is done all the macros in the autocall directory will **also** have the `/STORE` option. Now we have the best of both worlds: one source program and a compiled macro, both stored in one easy-to-find location.

### 12.2.3 Interactive macro development

When developing macros in the interactive environment, special care must be taken to make sure that the correct macro definitions are compiled and stored. If the developer is not careful it is possible to call a macro without using the latest, most up-to-date version of the macro definition. This is not a good thing.

As was discussed earlier, after a macro definition is submitted, the compiled code is stored in the appropriate `SASMACR` catalog. This catalog will be in the `WORK` library unless compiled stored macros are being used and the `/STORE` option is present on the `%MACRO` statement. When the macro is called, SAS looks for the macro definition in one or more of these catalogs, and only if the definition is **not** found is the autocall library searched.

Suppose the developer is debugging a macro whose definition resides in the autocall library. She calls the macro for execution, but is not satisfied with the results. If she edits the program, saves it back into the autocall library, and then re-executes it, the results will be the same! Her changes will be ignored! Her changes are not implemented because the macro has not been recompiled! She has updated the code, but because the macro has already been compiled and it resides in the `SASMACR` catalog, SAS will never look for the new definition in the autocall library. After making the change in the macro definition, she needs to do one of the following:

- Submit the macro definition (`%MACRO` to `%MEND`). This places the latest definition in the `SASMACR` catalog. This can be done directly through the display manager or by naming the macro definition in a `%INCLUDE` statement.
- Delete the compiled macro entry from the appropriate `SASMACR` catalog. The next execution of the macro will cause SAS to seek out the autocall definition, which will then be compiled and re-stored in the `SASMACR` catalog.
- Exit and re-enter SAS.

Each of these solutions solves the same problem. When your SAS environment includes macro libraries, you must remember that as you edit your macro definitions, you must also update the appropriate `SASMACR` catalogs as well. It is not enough to just change the macro definition.

The above example illustrates a situation that is usually not an issue when you use an `%INCLUDE` library to hold macro definitions. When the `%INCLUDE` statement brings in a macro definition, the definition itself is usually inserted directly into the code. It is then submitted and compiled—all in the same operation. Although this may initially sound like an advantage, it is not because the macro must be recompiled each time the `%INCLUDE` is executed. This method does not take full advantage of the ability to save the compiled macro in the `SASMACR` catalog.

**CAVEAT:** It has been this author's experience that deleting the compiled version of the macro from the SASMACR catalog (the second method in the list above) successfully forces the recompilation of the macro. However, this technique is **not** recommended by SAS as this approach and its various ramifications have not been fully tested and could, under certain unforeseen circumstances, cause problems.

## 12.2.4 Modifying the SASAUTOS system variable

The default value of SASAUTOS is established as a system variable in the configuration file (the name and location of the configuration file varies by operating system and version—under Windows and SAS Version 8 this file is by default named `!sasroot\sasv8.cfg`). Since this file is automatically executed when SAS is initiated, it becomes a useful way to establish customized library locations without using system options.

To modify the SASAUTOS path simply edit your configuration file. Look for the `-SET SASAUTOS` statement and add one or more paths to the list. In the code below, the directory `"f:\GroupProjA\saspgms\macros"` has been added as the first directory in the concatenated set of autocall directories.

```
/* Setup the SAS System autocall library definition */
-SET SASAUTOS ("f:\GroupProjA\saspgms\macros"
 "!sasroot\core\sasmacro"
 "!sasext0\assist\sasmacro"
 "!sasext0\eis\sasmacro"
 "!sasext0\ets\sasmacro"
 "!sasext0\graph\sasmacro"
 "!sasext0\iml\sasmacro"
 "!sasext0\intranet\sasmacro"
 "!sasext0\or\sasmacro"
 "!sasext0\qc\sasmacro"
 "!sasext0\share\sasmacro"
 "!sasext0\stat\sasmacro"
)
```

Once edited, the configuration file can be saved in another location. The appropriate configuration file is selected at SAS startup with the `-config` initialization option.

The SASAUTOS environmental variable is not available under all operating systems, and its specification varies. Consult the SAS Companion appropriate for your version of SAS and your operating system.

## 12.3 SAS Autocall Macros

In addition to any macros that you might write and add to your own autocall library, SAS comes supplied with a number of macros in its own autocall library. These can be found in your default SASAUTOS location; under Windows this might be `!sasroot\core\sasmacro`. Depending on your operating environment and what SAS products are leased, additional autocall macros may be available. Consult the SAS Companion for your operating environment for more information on the location of the autocall libraries.

Not only are these macros useful in and of themselves, but because the code is available you can modify these macros for your own purposes or use them as patterns for new macros. One of the really big advantages of looking at these programs is that you will get to see some macro code that was written by the developers of portions of SAS. Very often this macro code includes interesting techniques and can be instructive for someone who learns by example.

By default the automatic *fileref* SASAUTOS is available (under UNIX, SASAUTOS is **not** automatically created in versions prior to SAS 9.1) to point to the location or locations of the SAS autocall libraries. An options statement setting the autocall options to the defaults would be as follows:

```
options mautosource sasautos=sasautos;
```

If you specify locations for AUTOCALL libraries, you need to make sure that the SAS autocall library is also specified. Failure to do this will make **all of the SAS autocall macros unavailable**. A sample OPTIONS statement could be rewritten as

```
options mautosource
 sasautos=(projauto allauto sasautos);
```

Notice that the SASAUTOS *fileref* is listed last. This enables the user to create macros that will override the default definitions of the macros supplied with SAS.

Several autocall macros that are supplied by SAS behave as macro functions, and looking at their code provides insight into writing your own macro functions. These macros are noted briefly here and several are discussed in detail in the sections that follow.

### **%CMPRES and %QCMPRES**

compresses a text string by removing multiple blanks as well as leading and trailing blanks.

### **%DATATYP**

determines whether or not the argument is numeric.

### **%LEFT and %QLEFT**

left-justifies the text argument by removing leading blanks.

### **%LOWCASE and %QLOWCASE**

converts the text string to lower case (this is the functional opposite of the %UPCASE macro function).

### **%TRIM and %QTRIM**

trims trailing blanks from text.

### **%VERIFY**

locates the first character not in a string—the opposite of the %INDEX macro function.

For the most part, these macros closely mimic DATA step functions in name, syntax, and result. None are difficult to use. Indeed the hardest part about using them is to know of their existence.

Many of these macros also have a “Q” version (for example, %QLEFT and %QTRIM) for use when the returned value contains symbols or characters whose meaning should remain hidden. See Section 7.1 for more on macro quoting and when it may be needed.

Because these are macros (although they sometimes act like functions), the text string or argument cannot contain imbedded commas. When the argument does contain a comma, the macro parser interprets the comma as a parameter delimiter and syntax errors can result. Often when this happens, the argument itself can be quoted to hide the imbedded comma.

To make matters a bit more complicated, some of these macros (such as %LOWCASE) also call macro functions that use commas to delimit arguments. For these macros, commas in the argument can also cause problems in the interior macro or macro function. Quoting may be of less utility in these cases, although using the Q version of the macro may help.

## SEE ALSO

*SAS Macro Language: Reference, First Edition* briefly describes these macros (pp. 158–160) and includes their description with the other macro language elements in Chapter 13, “Macro Language Dictionary” (pp. 161–270).

Chapter 7, “The Autocall Facility,” in the *SAS Guide to Macro Processing, Version 6* lists many of the standard SAS autocall macros in one place (pp. 185–187).

### 12.3.1 %VERIFY

Functionally the opposite of the INDEXC function, this macro enables you to search a text string for the first character that is not in the target list. %VERIFY then returns the position of that character. In the following example, the macro variable &YRCODE contains a string that starts with some unknown number of numbers. You want to strip off the numbers and create a new macro variable (&CODE) that starts with the first character that is not a number.

```
%let yrcode = 97CAdwz;
%let code = %substr(&yrcode,%verify(&yrcode,1234567890));

%put &yrcode &code;
```

The first argument is the string to search, and the second contains characters that we want to avoid (in this case all the numbers). The LOG is shown here:

```
162 %let yrcode = 97CAdwz;
163
164 %let code = %substr(&yrcode,%verify(&yrcode,1234567890));
165
166 %put &yrcode &code;
97CAdwz CAdwz
```

The %SUBSTR function was used to select the string starting in the position that was occupied by the first nonnumber.

## SEE ALSO

The %VERIFY macro is utilized by Hirabayashi (2001) in a macro that generalizes the output from PROC FREQ.

### 12.3.2 %LEFT

Like the DATA step LEFT function, this macro has a single argument that it returns without any leading blanks. The example shown here uses %LEFT to left-justify the value of the macro variable &LFIVE:

```
data _null_;
x=5;
call symput('lfive',x);
run;

%put *&lfive*;
%let lfive = %left(&lfive);
%put *&lfive*;
```

This produces the LOG entry shown here:

```
171
172 %put *&lfive*;
* 5*
173 %let lfive = %left(&lfive);
174 %put *&lfive*;
5
```

The macro %LEFT uses the %VERIFY macro ❶ to check for nonblank characters. When a nonblank character is found, its location is stored in &I, which is used as the column index ❷ for the %SUBSTR function ❸.

```
%macro left(text);
%*****;
%* *;
%* MACRO: LEFT *;
%* *;
%* USAGE: 1) %left(argument) *;
%* *;
%* DESCRIPTION: *;
%* This macro returns the argument passed to it *;
%* without any leading blanks in an unquoted *;
%* form. The syntax for its use is similar to *;
%* that of native macro functions. *;
%* *;
%* Eg. %let macvar=%left(&argtext) *;
%* *;
%* NOTES: *;
%* The %VERIFY macro is used to determine the *;
%* first nonblank character position. *;
%*****;
%local i;
%if %length(&text)=0 %then %let text=%str();
%let i=%verify(&text,%str());❶
%if &i❷ %then %substr(&text,&i);❸
%mend;
```

Notice that the %SUBSTR function resolves to the amended text ❸ so that the macro call is replaced with the left-justified string.

### 12.3.3 %CMPRES

The %CMPRES macro mimics the COMPBL DATA step function by removing multiple blanks from the text string. In effect, it also calls %LEFT and %TRIM because it removes all leading and trailing blanks.

In the examples in Section 6.2.1, a numeric variable is converted to character prior to using the SYMPUT routine. The following example creates the macro variable directly from the numeric variable and then uses %CMPRES to remove the leading blanks. In this example the asterisks are used to help show the position of the blanks.

```
data _null_;
x=5;
call symput('five',x);
run;

%put *&five*;
%let five = %cmpres(&five)*;
%put &five;
```

The resulting LOG shows the following:

```
138
139 %put *&five*;
* 5*
140 %let five = %cmpres(&five)*;
141 %put &five;
5
```

The code for %CMPRES is shown here:

```
%macro cmpres(text);

%*****;
%* *;
%* MACRO: CMPRES *;
%* *;
%* USAGE: 1) %cmpres(argument) *;
%* *;
%* DESCRIPTION: *;
%* This macro returns the argument passed to it in an *;
%* unquoted form with multiple blanks compressed to *;
%* single blanks and also with leading and trailing *;
%* blanks removed. *;
%* *;
%* Eg. %let macvar=%cmpres(&argtext) *;
%* *;
```

```

%* NOTES:
%* The %LEFT and %TRIM macros in the autocall library
%* are used in this macro.
%*
%*
%*****
%local i;
%let i=%index(&text,%str()); ❶
%do %while(&i^=0);
%let text=%qsubstr(&text,1,&i)%qleft(%qsubstr(&text,&i+1)); ❷
%let i=%index(&text,%str());
%end;
%left(%qtrim(&text)) ❸
%mend;

```

- ❶ The %INDEX function is used to find any occurrences of a double blank %STR( ). If found, the starting position is noted in &I.
- ❷ The %LET is used to rewrite the incoming text string &TEXT. Notice that two macro functions are used in the same statement. The %QSUBSTR function returns the first &I characters in the string (this is where the first of the two blanks is retained). The %QLEFT autocall macro then left-justifies the remainder of the string. Because the function and macro are resolved and executed before the %LET, the two resulting text strings are effectively concatenated into a new string, which is placed in &TEXT. Using %QLEFT eliminates the need to step iteratively through a long string of blanks.
- ❸ The %LEFT macro will be called last and will resolve to a text string. Effectively, this is the text that is **passed back** from the call to %CMPRES. Actually, nothing is really passed back. Rather, this macro resolves to a potentially modified version of &TEXT, and that string is what is seen as the resolved text in the code that called %CMPRES. Because this is a macro call, a semicolon is not wanted.

You could easily modify this macro to remove any combination of characters, not just double blanks.

### 12.3.4 %LOWCASE

The %LOWCASE macro can be used to convert all uppercase characters to lowercase characters. The macro accepts a single argument, which is translated using the %INDEX and %SUBSTR functions. This macro is demonstrated in the following example:

```

%let mixed = SAS Macro Language;
%let lower = %lowercase(&mixed);
%put &lower;

```

The resulting LOG shows the macro variable &LOWER to be all lowercase characters:

```

175 %let mixed = SAS Macro Language;
176 %let lower = %lowercase(&mixed);
177 %put &lower;
sas macro language

```

The code for this macro is interesting because of the way it finds uppercase characters and then translates them.

```
%macro lowercase(string);
%*****;
%* *;
%* MACRO: LOWCASE *;
%* *;
%* USAGE: 1) %lowercase(argument) *;
%* *;
%* DESCRIPTION: *;
%* This macro returns the argument passed to it unchanged *;
%* except that all uppercase alphabetic characters are changed *;
%* to their lowercase equivalents. *;
%* *;
%* E.g.: %let macvar=%lowercase(SAS Institute Inc.); *;
%* The variable macvar gets the value "sas institute inc." *;
%* *;
%* NOTES: *;
%* Although the argument to the %UPCASE macro function may *;
%* contain commas, the argument to %LOWCASE may not, unless *;
%* they are quoted. Because %LOWCASE is a macro, not a function, *;
%* it interprets a comma as the end of a parameter. *;
%* *;
%*****;
%local i length c index result;
%let length = %length(&string);
%do i = 1 %to &length;
 %let c = %substr(&string,&i,1); ❶
 %if &c eq %then %let c = %str();
 %else %do;
 %let index = %index(ABCDEFGHIJKLMNOPQRSTUVWXYZ,&c); ❷
 %if &index gt 0 %then ❸
 %let c = %substr(abcdefghijklmnopqrstuvwxyz,&index,1); ❹
 %end;
 %let result = &result.&c; ❺
 %end;
%result ❻
%mend;
```

When %LOWCASE is called, the value of &MIXED was passed into the macro, where it was assigned to &STRING; for example, &STRING = SAS Macro Language.

- ❶ The *i*th character in the string of interest (&STRING) is temporarily stored in the macro variable &C. When %LOWCASE is applied to &MIXED, the fifth letter in the string (&I=5) is M (&C=M).
- ❷ The %INDEX function is used to see if &C contains an uppercase letter. If an uppercase letter is found, its position is stored in &INDEX. For &C=M, the %INDEX function returns a 13 (since M is the thirteenth letter of the alphabet).
- ❸ If an uppercase letter is found (&INDEX > 0), it is converted using the %SUBSTR function.
- ❹ In the %SUBSTR function, &INDEX becomes the column indicator for the matching lowercase letter. For &INDEX=13, the lowercase m is selected and assigned to &C. The macro variable &C now has been converted to lowercase.



- ⑤ &C is then appended (converted or not) onto &RESULT, which continues to grow until the entire string has been checked.
- ⑥ The macro variable &RESULT contains the converted string, and this will be the final resolved value of the macro call.

### SEE ALSO

Sylvia Tze (2000) discusses a macro that, although not a macro function, will convert all except the first character to lower case.

## 12.3.5 %TRIM

The %TRIM macro removes trailing blanks from the argument and returns the result.

```
%macro trim(value);
%*****;
%*
%* MACRO: TRIM
%*
%* USAGE: 1) %trim(argument)
%*
%* DESCRIPTION:
%* This macro returns the argument passed to it
%* without any trailing blanks in an unquoted form.
%* The syntax for its use is similar to that of
%* native macro functions.
%*
%* Eg. %let macvar=%trim(&argtext)
%*
%* NOTES:
%* None.
%*
%*****;
%local i;
%do i=%length(&value) %to 1 %by -1; ①
 %if %qsubstr(&value,&i,1) ^=%str() %then %goto trimmed; ②
%end;
%trimmed: %if &i>0 %then %substr(&value,1,&i); ③
%mend;
```

- ① The macro uses a %DO loop to step through the argument from the last character forward until the first nonblank character is encountered. Notice that the %LENGTH function is used to determine the last character. Usually the value returned by the %LENGTH function will be the last nonblank character, and the %DO would only iterate once.

❷ %QSUBSTR is used to examine the *i*th character and if it is not a blank then our search is over.

❸ The %SUBSTR function is then used to select all the characters in the argument up to and including the last character (the *i*th character) examined by the %QSUBSTR ❷.

It is possible for the %LENGTH function to count trailing blanks. This is demonstrated in the following example. &B2 is created using a quoting function ❹, which preserves the nine trailing blanks. These trailing blanks (which would be removed by %TRIM) are counted by the %LENGTH function. ❺ The process is shown in the LOG:

```
1 %let b1 = x x;
2 %let lenb1 = %length(&b1);
3 %let b2 = %qsubstr(&b1,1,10); ❹
4 %let lenb2 = %length(&b2); ❺
5 %put b1 |&b1| &lenb1;
b1 |x x| 12
6 %put b2 |&b2| &lenb2;
b2 |x | 10 ❺
```

### MORE INFORMATION

The %LENGTH function is introduced in Section 7.2.2.

## 12.3.6 %DATATYP

Since macro variables are not designated as numeric or character, it may become necessary to determine whether or not a given macro variable contains a number. This can be important if arithmetic operations (including numeric comparisons) are to be performed.

The macro %DATATYP returns whether the parameter is numeric or character. The macro is not limited to integers and will even detect numbers written in scientific notation, although it does not detect numbers written in other bases such as hexadecimal. The single parameter is the value that you want to have checked and the macro returns either NUMERIC or CHAR. A sample LOG showing its usage contains the following:

```
12 %let d1 = 2.54;
13 %let typd1=%datatyp(&d1);
14 %put &d1 &typd1;
2.54 NUMERIC
15
16 %let d2 = 4.3e3;
17 %let typd2=%datatyp(&d2);
18 %put &d2 &typd2;
4.3e3 NUMERIC
19
20 %let d3 = abcd3;
21 %let typd3=%datatyp(&d3);
22 %put &d3 &typd3;
abcd3 CHAR
```

### 12.3.7 %COMPSTOR

For those sites that use a number of compiled stored macros, or that have an existing catalog of compiled macros (see Sections 3.3.4 and 12.1.2), or that want to optimize the access to the autocall macros supplied by SAS, the %COMPSTOR macro can be used to compile and permanently store the macros in the SAS autocall library. Once these macros have been compiled it is no longer necessary to include the SASAUTOS *fileref* in the SASAUTOS system option.

#### Syntax

```
%COMPSTOR(pathname=unquoted_path)
```

When the %COMPSTOR macro is executed, it creates a *libref* named SASMACR that points to the location (pathname=) passed into the macro. A SASMACR catalog is then created in that directory, the compiled macros are stored in the catalog, and a SASMSTORE= system option is executed that points to this directory (*libref*= SASMACR).

You may not quote the path that is specified in PATHNAME=. The following code establishes

- the *libref* SASMACR pointing to the C:\PERMMAC directory
- the SASMACR catalog in this library
- the compiled versions of all the autocall macros supplied by SAS.

```
%compstor (pathname=c:\temp)
```

There are a couple of caveats and even a couple of side benefits that you should be aware of before using %COMPSTOR.

%COMPSTOR is itself an autocall macro so SAS will need to have access to its own autocall library when %COMPSTOR is executed.

%COMSTOR issues its own SASMSTORE= option. This macro **will wipe out** any existing SASMSTORE definition. Be sure to reestablish your own option settings after running %COMPSTOR (see the %STOREOPT and %HOLDOPT macros in Section 10.3.1). Of course when you do reestablish the options be sure to include the location of the new SASMACR catalog.

You cannot use a *libref* as the argument, so you must know the physical path to the directory of interest. Assuming that the *libref* MYMACS points to C:\PERMMAC, this limitation can be overcome by using the following code:

```
%compstor (pathname=%sysfunc (pathname (mymacs)))
```

You will now have **two** *librefs* pointing to the same location (MYMACS and SASMACR).

A side benefit of using %COMPSTOR is that there are a couple of undocumented macros in %COMPSTOR that will be added to the SASMACR catalog.

Another benefit is that once these macros have been compiled, you will no longer need to include the SASAUTOS *fileref* in the SASAUTOS= system option.

If you already have a catalog containing compiled macros of your own, you can use %COMPSTOR to add the autocall macros to your existing SASMACR catalog. In the following example, the *libref* MYMACS points to a SASMACR catalog in which you have a number of your own macros. %COMPSTOR **adds** the autocall macros.

```
options mstored sasmstore=mymacs;
%compstor(pathname=%sysfunc(pathname(mymacs)))
```

You should be aware that if your catalog already contains a macro with the same name as one of the autocall macros, when %COMPSTOR is executed your version will be overwritten.

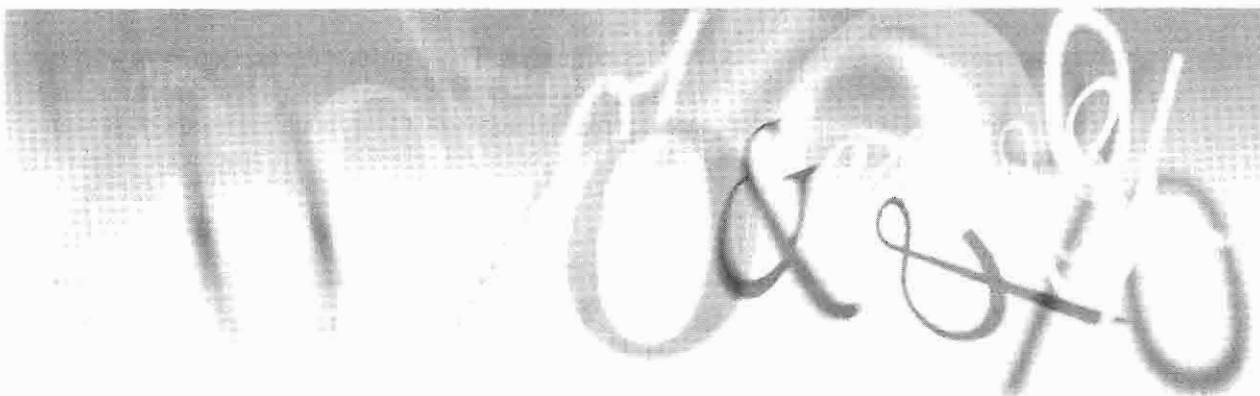
---

## 12.4 Chapter Summary

Macro libraries play a very important role in the development and maintenance of macros. This is especially true as the applications and systems that we develop become more complex or as we need to make our macro tools available to our colleagues.

Macro libraries are generally either set up as autocall or stored compiled macro libraries. Both methods have advantages and can even be used together. Although not a true macro library, the %INCLUDE statement can also be used to mimic a number of the capabilities of an autocall library.

SAS ships with a number of predefined macros that are made available as autocall macros. By including the automatic *fileref* SASAUTOS in the SASAUTOS= system option, these macros appear to be a part of the macro language. The macro definitions themselves can be examined by looking directly at the SAS code in the macro libraries.



# **Chapter 13**

---

## **Miscellaneous Macro Topics**

- 13.1 Other Specialized Tasks 348**
  - 13.1.1 &&&VAR – Using triple-ampersand macro variables 348**
  - 13.1.2 Totals based on a list of macro variables 349**
  - 13.1.3 Selecting elements from macro arrays 351**
  - 13.1.4 Calculating permutations 352**
  - 13.1.5 Checking if a macro variable exists 354**
  - 13.1.6 Extending the use of %SYMDEL 355**
- 13.2 Doubly Subscripted Macro Arrays 357**
  - 13.2.1 Subscript resolution issues 358**
  - 13.2.2 Naming row and column indicators 358**
  - 13.2.3 Using the &&&VAR&I variable form 361**
  - 13.2.4 Using the %SCAN function to identify array elements 363**
- 13.3 Programming Smarter 368**
  - 13.3.1 Efficiency issues 368**
  - 13.3.2 Programming with style 371**
  - 13.3.3 Debugging your macros 373**
  - 13.3.4 The DATA step versus the macro language 375**
  - 13.3.5 Little things with a big bite 380**
- 13.4 Working With Macro Parameter Buffers 389**
  - 13.4.1 Calling a macro using /PARMBUFF 390**
  - 13.4.2 Using the PARMBUFF macro switch 391**
- 13.5 Understanding Recursion in the Macro Language 393**
- 13.6 Determining Macro Variable Scopes 395**
- 13.7 Chapter Summary 397**

This chapter contains several eclectic topics of interest. These include the use of arrays with more than one index, more on the use of triple ampersands, the use of recursion in the macro language, and a number of issues regarding programming techniques that you should be aware of as you begin to write more sophisticated macros.

## 13.1 Other Specialized Tasks

There are a number of techniques that, although not used every day, are important to the macro programmer. Some of these techniques are specialized enough that when they are known to the programmer they can save hours of programming effort.

### 13.1.1 &&&VAR – Using triple-ampersand macro variables

As a general rule, it is usually not necessary to resort to the use of triple-ampersand macro variables. There are times, however, when it becomes either necessary or just expedient. The most common use is when you want to store the name of a macro variable in another macro variable. The examples in this section show, with increasing complexity, the use of the triple ampersand.

We have already seen in earlier sections of this book, numerous examples of the use of the &&VAR&J construct. Although the ampersands are not all adjacent, this is effectively still a triple-ampersand macro variable. Whether the ampersands are all adjacent as in the examples in this section, or if they are indexed, we still have a macro variable that resolves to another macro variable that resolves to the value of ultimate interest. *is the form of the triple ampersand*

If you are already comfortable with the use of the &&VAR&J form of the triple ampersand, remember that it references a list of numbered macro variables, and as such it mimics a macro array. If that array had only one element you would not need the index variable, and the macro variable could be rewritten as &&&VAR. Essentially then the &&&VAR form of a macro variable is nothing more than an array of one. It is a placeholder that contains the name of the macro variable of interest.

#### SEE ALSO

Yindra (1997) has an example that uses a &&& macro variable in a TITLE statement. He then expands the topic with more examples in Yindra (1998).

The macro presented by Widawski (1997a) to create a list of files is generalized by using &&& macro variables.

Yarbrough (2000) combines the &&& construct with the &&VAR&J form with an example that uses the macro variable form of &&&VAR&J.

An example using six ampersands can be found in Gerlach (1997).

*SAS Macro Language: Reference, First Edition* gives an efficiency tip that utilizes triple ampersands (pp. 136–137).

### 13.1.2 Totals based on a list of macro variables

#### %SUMS

Justina M. Flavin

Pfizer Global Research & Development, LaJolla Laboratories

This macro is used in a program that generates an adverse event incidence table. In many non-crossover studies, the client wants the rightmost column of the table to be a Total column, which contains a simple sum of all of the counts in all of the treatment groups (for a given adverse event or body system). The macro %SUMS calculates these total counts. When %SUMS is called, the parameters TOTAL and NUMTREAT are actually the names of the macro variables that contain the number of treatment or dosing groups.

The data set FREQPREF has variables for the *adverse event name* (preferred term), the body system, and a series of variables (PREF1, PREF2,...,PREF $n$ ) that contain the number of unique patients that have that adverse event for that particular treatment or dosing group. That is, PREF1 is the number of patients in dose group 1; PREF2 is the number in dose group2, and so on.

In the same manner, the data set FREQBODY has variables for the body system, and a series of variables (BODY1, BODY2,...,BODY $n$ ) that contain the number of unique patients in the body system for that particular treatment or dosing group. (A patient can have multiple adverse events in a body system, but on the body system line in the table, the patient should be counted only once.)

The data set EVENT contains a series of variables (EVENT1–EVENT $n$ ) that contain the number of unique adverse events within a body system. These variables are used to determine the proper sorting order for printing the table.

Because the program is completely data-driven, the number of treatment or dosing groups is determined early on and is assigned to the macro variable &NUMTREAT. Another macro variable, &TOTAL, is &NUMTREAT+1.

The following three DATA steps create synthetic data in the same form as might be expected in an actual application of this macro. Each data set has the key variable BODY and a list of variables (that in this case contain random numbers).

```
%let total=4;
%let numtreat=3;

data freqpref;
drop i;
array lst {&numtreat} pref1 - pref&numtreat;
do body = 1 to 5;
 do i = 1 to &numtreat;
 lst{i} = int(ranuni(9999)*10);
 end;
 output;
end;
run;

data freqbody;
drop i;
array lst {&numtreat} body1 - body&numtreat;
```

```

do body = 1 to 5;
 do i = 1 to &numtreat;
 lst{i} = int(ranuni(9999)*5);
 end;
 output;
end;
run;

data event;
drop i;
array lst (&numtreat) event1-event&numtreat;
do body = 1 to 5;
 do i = 1 to &numtreat;
 lst{i} = int(ranuni(9999)*3);
 end;
 output;
end;
run;

```

The macro %SUMS is used to dynamically generate a single assignment statement. The following call to %SUMS:

```

%let total=4;
%let numtreat=3;
%sums(pref, total, numtreat)

```

will generate the following assignment statement:

```
pref4=sum(of pref1-pref3);
```

The macro %SUMS expects to be passed the root portion of the name of the variable to be created (&PREFIX), the name of the macro variable that contains the number of the new variable (&SUFFIX), and the name of the macro variable that contains the number of variables in the list (&STOPNUM).

```

%macro sums(prefix, suffix, stopnum);
 &&prefix&&&suffix=sum(of &prefix.1-&&prefix&&&stopnum);
%mend sums;

```

The macro is called from within a DATA step. In the following step it is called three times, resulting in the creation of three assignment statements:

```

data counts(drop=j);
 merge freqpref freqbody event;
 by body;
 array zeroes{*} _numeric_;
 do j=1 to dim(zeroes);
 if zeroes{j}= . then zeroes{j}=0;
 end;
 %sums(pref, total, numtreat)
 %sums(body, total, numtreat)
 %sums(event, total, numtreat)
run;

proc print data=counts;
id body;
title1 'Counts and Totals';
run;

```



The PROC PRINT generates the following output, which contains the totals in PREF4, BODY4, and EVENT4:

| Counts and Totals |       |       |       |       |       |       |        |        |        |       |       |        |
|-------------------|-------|-------|-------|-------|-------|-------|--------|--------|--------|-------|-------|--------|
| BODY              | PREF1 | PREF2 | PREF3 | BODY1 | BODY2 | BODY3 | EVENT1 | EVENT2 | EVENT3 | PREF4 | BODY4 | EVENT4 |
| 1                 | 4     | 9     | 8     | 2     | 4     | 4     | 1      | 2      | 2      | 21    | 10    | 5      |
| 2                 | 7     | 1     | 8     | 3     | 0     | 4     | 2      | 0      | 2      | 16    | 7     | 4      |
| 3                 | 2     | 3     | 8     | 1     | 1     | 4     | 0      | 1      | 2      | 13    | 6     | 3      |
| 4                 | 5     | 3     | 1     | 2     | 1     | 0     | 1      | 1      | 0      | 9     | 3     | 2      |
| 5                 | 0     | 7     | 2     | 0     | 3     | 1     | 0      | 2      | 0      | 9     | 4     | 2      |

For the macro call %sums (body, total, numtreat), the triple ampersands are resolved as shown here:

| Pass number | String            | Resolves to   |
|-------------|-------------------|---------------|
| 1           | &&prefix&&&suffix | &prefix&total |
| 2           | &prefix&total     | body4         |

The version of %SUMS that is presented in this section is very much simplified from the version that was used in the actual application. For this particular version, you can simplify it further and eliminate triple ampersands. %SUMS can be rewritten as

```
%macro sums(prefix, suffix, stopnum);
 &prefix&suffix=sum(of &prefix.1-&prefix&stopnum);
%mend sums;
```

The macro call now must pass the resolved values of the macro variables (&TOTAL and &NUMTREAT) rather than the macro variable names. You can do that in this example because you really need only the value of the two macro variables inside of %SUMS. The macro call becomes

```
%sums (body, &total, &numtreat)
```

If %SUMS also needed the name of the macro variable (perhaps to put it in a title or to use it as a label in a %GOTO), then the original version with the triple ampersands would be required.

### 13.1.3 Selecting elements from macro arrays

#### %GETKEYS

**Richard O. Smith**  
Science Explorations

This macro is taken from an application that contains a series of data sets that exist with slight variations in different libraries. The data sets are similar enough so that many of the same processing programs can be used across libraries. Primarily, they differ in the names of the key

(BY) variables. Globalized macro variable arrays are created that store the names of the data sets and the associated key variables for each *libref*. Typical arrays might be

| <i>libref</i> | Number of data sets | Data set names          | Key variables       |
|---------------|---------------------|-------------------------|---------------------|
| live ❶        | &livecnt ❸          | &livedb1, &livedb2, ... | &keys1, &keys2, ... |

Given the data set name (for example, LIVE.AE) you need to be able to retrieve the associated key variables. The macro %GETKEYS uses the *libref* and data set name to look up the associated key variable list, which is then stored in the macro variable &KEYVARS.

```
%macro getkeys(inlib❶,indsn❷);
* getkeys.sas
* 25Jun97 - ROSmith
* Macro to get the key variables for selected library & member.
* Assumes that the global macros for the databases & keys
* are created.
* Outputs a macro variable KEYVARS.
*;
%global keyvars;
%do k = 1 %to &&inlib.cnt; ❸
 %if %upcase(&indsn❷) = %upcase(&&inlib.db&k❹) %then
 %let KEYVARS = &&KEYS&k; ❺
%end;

%mend getkeys;
```

- ❶ The *libref* (LIVE) is also used as part of the name of the macro variable that counts the number of data sets in this library ❸.
- ❷ &INDSN is used to store the name of the data set.
- ❸ The list of data set names in this *libref* is stepped through one at a time. The number of data sets (elements in this pseudo array) is stored in a macro variable whose name is made up in part with the name of the *libref*. &&inlib.cnt ==> &livecnt
- ❹ The list of data set names is stored in macro variables that are formed using the *libref* and followed by DB and then a number, such as &LIVEDB3. This array is indirectly referenced by using &&inlib.db&k.
- ❺ The key variables for this data set are assigned to the macro variable &KEYVARS.

### 13.1.4 Calculating permutations

The calculation of permutations and combinations for  $N$  things taken  $R$  at a time can be tricky when the values of  $N$  and  $R$  are large. This is because the formulas involve the use of factorials. The number of permutations is  $(N!)/(N-R)!$ , and the DATA step functions that perform these calculations (PERM and COMB) calculate the numerator and the denominator separately and then do the division. This can result in very large intermediate numbers even if the final result is small. The largest factorial that can be calculated by these functions is 170!.

It turns out, however, that by doing the division first, and the multiplication implicit in the factorials second, can often greatly reduce the size of the intermediate numbers. For example the number of permutations for 5 things taken 2 at a time is

$$5!/(5-2)! = 5!/3! = (5*4*3*2*1)/(3*2*1) = 120/6 = 20$$

or it could instead be calculated as

$$(5*4*3*2*1)/(3*2*1) = (5*4) * (3*2*1)/(3*2*1) = 20*1 = 20$$

Obviously, the second approach results in smaller intermediate numbers. The macros shown below use the second approach to calculate permutations.

### %SMARTPERM and %SMARTPERM2

These macros take advantage of the fact that the denominator will always be contained within the numerator. The numerator will therefore be the product of the largest  $R$  numbers. In both cases a simple %DO loop is used to multiply the numbers one after the other.

```
%macro smartperm(n,r);
 %local perm val; ❶
 %let perm=1; ❷
 %let val= &n; ❸
 %do %while(&val > %eval(&n - &r)); ❹
 %let perm = %eval(&perm*&val); ❺
 %let val = %eval(&val-1); ❻
 %end;
 &perm ❼
%mend smartperm;
```

- ❶ This macro acts as a function and returns the number of permutations by passing them back ❸: therefore, the macro variables are all local.
- ❷ Initialize the number of permutations to 1. This is the result when  $N = R$  and the loop is not entered.
- ❸ Initialize the variable to be decremented.
- ❹ Once  $\&VAL = N - R$ , there will be values in the denominator to cancel those in the numerator so we can stop.
- ❺ Multiply the current value against the growing product.
- ❻ Decrement the multiplier.

The macro can be rewritten and slightly simplified by using a %DO rather than a %DO %WHILE:

```
%macro smartperm2(n,r);
 %local perm val;
 %let perm=1;
 %do val = &n %to %eval(&n - &r + 1) %by -1; ❶
 %let perm = %eval(&perm*&val);
 %end;
 &perm ❷
%mend smartperm2;
```

- ⑦ The %DO starts high (at &N) and is decremented by 1 until the proper value is met.
- ⑧ The number of permutations is passed back.

### MORE INFORMATION

Examples showing the calculation of permutations and combinations using Base SAS Language functions are shown in Sections 7.6.1 and 7.6.3.

## 13.1.5 Checking if a macro variable exists

### %SYMCHK

#### Author unknown

When the macro facility attempts to resolve a macro variable that does not exist on any of the available symbol tables, a warning is written to the LOG. For instance, if the macro variable &TMP does not exist and it is used in a program, then you will receive the following warning in the LOG:

WARNING: Apparent symbolic reference TMP not resolved.

To prevent this warning, the macro %SYMCHK can be used to determine if a macro variable exists. The macro itself returns a YES or NO depending on whether or not the passed parameter is a macro variable.

```
%macro symchk(name);

%if %nrquote(&&&name)=%nrstr(&&name) %then ①
 %let yesno=NO;

%else %let yesno=YES;

&yesno ②
%mend symchk;

%put 'Does &tmp exist? ' %symchk(tmp); ③
```

- ① The macro variable &NAME contains the name of the macro variable that needs to be checked. When three ampersands are used, as in &&&NAME, the first two ampersands resolve to &. For %SYMCHK(TMP) this means that &&&NAME resolves to &TMP. If &TMP has been defined, it will be further resolved to its value; otherwise, if &TMP is not defined a warning is issued and &TMP remains unresolved.

%NRSTR(&&&NAME) will resolve to &TMP and because of the %NRSTR function it will not be further resolved. When &TMP is not defined, the unresolvable &TMP on the left side of the equal sign will equal the &TMP on the right and the expression will be true. When the expression is true, the macro variable does not exist and &YESNO is set to NO.

- ② The macro will return either a YES or a NO.
- ③ The %PUT is used to demonstrate a call to %SYMCHK.

It is interesting to look at the use of the %NRQUOTE function in this example. This macro has been around for a while and may predate %NRBQUOTE, which, in this instance, would also have worked just as well.

Resolution of the macro variable reference &&&NAME is not prevented by either %NRQUOTE or %NRBQUOTE; however, errors that would result from an unknown macro variable reference (say when &TMP is undefined as it was in the above example) are prevented. If the %NRQUOTE function was not used the %IF becomes

```
%if &&&name = %nrstr(&)&name %then ...
```

Let's assume, as in the above example, that &TMP does not exist. After variable resolution the %IF becomes

```
%if &tmp = %nrstr(&)tmp %then ...
```

We still get the "Unresolved macro variable" reference, but now, because of the &TMP on the left of the = sign, the %IF will be unable to evaluate the expression during macro execution (we get an error message that includes: "A character operand was found in the %EVAL function or %IF condition"). This is what the %NRQUOTE is masking, and this is why we use the %NRQUOTE. It does not prevent the resolution of &&&NAME, but rather masks any resulting special characters (the & in &TMP) during execution, which is when the resolved values are to be evaluated as part of the expression.

The macro %SYMCHK is posted on the SAS support page at

<http://support.sas.com/techsup/faq/macro.html>

## MORE INFORMATION

If this macro is used when a macro variable does not exist, a warning is written to the LOG. Another approach to detecting macro variables that avoids this message can be seen in the discussion of %SYMCHK in Section 7.6.3.

## SEE ALSO

Rhoads and Letourneau (2002) use %SYMCHK. This and additional variations on %SYMCHK are presented by Troxell and Chen (2003).

diTommaso (2003) presents a non-macro solution for detecting a macro variable.

## 13.1.6 Extending the use of %SYMDEL

%SYMDEL is intended to be used to delete macro variables from the GLOBAL symbol table. The statement accepts a list of macro variables that are referenced directly. The following statement removes the macro variables &NADA and &DSN from the GLOBAL symbol table:

```
%symdel nada dsn;
```

The %SYMDEL statement was not designed to accept indirect references to macro variables such as a macro variable that contains the name of one or more macro variables. The following usage of %SYMDEL could produce errors:

```
%let macvarlist = nada dsn;
%symdel &macvarlist;
```

```

%macro delvars;
 data vars;
 set sashelp.vmacro;
 run;
 data _null_;
 set vars;
 if scope='GLOBAL' then
 call execute('%symdel '||trim(left(name))||';');
 end;
 run;
%mend delvars;

```

Notice that since SASHELP.VMACRO is a VIEW that points back to the symbol table(s), it cannot be used in the same DATA step as the CALL EXECUTE. Again, this is a timing issue—a CALL EXECUTE timing issue this time.

If you try to delete macro variables that are not on the global table (perhaps because the variables do not exist or they only exist on a local table), you will get an error indicating that the macro variable was not found.

### MORE INFORMATION

The %SYMDEL statement was introduced in Section 2.8.

### SEE ALSO

Watts (2003a) has an example of the PROC SQL step that prepares a list of GLOBAL macro variables for deletion. Similar examples and discussions have appeared on SAS-L by several authors. Discussion of the use of %SYMDEL and variations of the macro %DELVARS can also be found in the SAS technical support area under the FAQ section relating to macros.

diTommaso (2003) also discusses the use of %SYMDEL with a CALL EXECUTE.

---

## 13.2 Doubly Subscripted Macro Arrays

In numerous examples elsewhere in this book, a series of macro variables have been used as a macro array. Because the macro language has no true array, you can fake it by creating macro variables in the form of &&VAR&i (see Sections 6.2.1 and 6.4 for introductory examples). These macro variables create a vector or what is essentially an array with a single subscript.

A *doubly subscripted array* describes a matrix of values or the rows and columns of a SAS data set or table. This enables you to store all of the values that are contained within a table in a single set of macro variables. You can approach the subscripting problem in several ways. Initially, one might try creating a logical extension of &&VAR&i as &&&&VAR&i&j, where &i and &j indicate the desired row and column. Life, however, is never quite this simple, and there are complications with this approach. Fortunately, there are solutions (or this section would not have appeared in the book).

### SEE ALSO

Noda, Kraemer, and Periyakoil (2000) use macro indexes to generate a pseudo matrix in the DATA step.

### 13.2.1 Subscript resolution issues

Assume that you want the macro variable &VAR34 to resolve to 5.62. That is, the fourth variable (column) in the third observation (row) has the value of 5.62. Assign the macro variables as follows:

```
%let i=3;
%let j=4;
%let var34 = 5.62;
```

The sequence of resolution is shown here:

| Pass number | String       | Resolves to |
|-------------|--------------|-------------|
| 1           | &&&&var&&i&j | &&var&i4    |
| 2           | &&var&i4     | &var&i4     |

Because &i4 is undefined, a WARNING is issued. In this example, you need to separate the &i from the 4. To do this, you can use a period to terminate the variable &i. The macro variable becomes &&&&var&&i.&j.

Now the sequence of resolution becomes

| Pass number | String        | Resolves to |
|-------------|---------------|-------------|
| 1           | &&&&var&&i.&j | &&var&i.4   |
| 2           | &&var&i.4     | &var34      |
| 3           | &var34        | 5.62        |

In this simple case, it would have been easier to write &&&&var&&i.&j as &&var&i&j, which resolves in only two passes.

This approach works fine as long as the number of rows and columns is fewer than ten. Larger numbers can result in naming conflicts. Does &var345 refer to row 34 and column 5 or row 3 and column 45? For small tables, this is not a problem. Fortunately, a more robust method exists for larger tables.

#### SEE ALSO

Bryher (1997a) and Geary (1997) both use more sophisticated examples of doubly subscripted macro arrays. Geary uses the form &&&&VAR&&J&K, while Bryher uses &&&VAR&J&K.

### 13.2.2 Naming row and column indicators

Rather than combining the row and column indicators into a single number, they can be kept separate. The macro variable &R3C4 completely specifies the location that the value came from without conflict. Further, it can be referenced without using quadruple ampersands, for example, &&R&i.C&j. Notice that the dot (.) is still needed following the &i.

```
%let i=3;
%let j=4;
%let r3c4 = 5.62;
```

The sequence of resolution is shown here:

| Pass number | String    | Resolves to |
|-------------|-----------|-------------|
| 1           | &&r&i.c&j | &r3c4       |
| 2           | &r3c4     | 5.62        |

The following example reads a SAS data set (table) and builds a macro variable for each value. The data set is taken directly from the one used in the example in Section 13.1.2. There are four variables that you would like to place into the array (PREF1, PREF2, BODY1, and BODY2) for each observation in the data set.

```
data _null_;
set counts (keep=body pref1 pref2 body1 body2);
array vlist {4} pref1 pref2 body1 body2; ❶
length name $8 ii jj $2;
i+1;
ii = trim(left(put(i,2.)));
call symput('rowcnt',ii); ❷
if i=1 then call symput('colcnt','4');

*** Store values for this row;
* Build the base for the macro vars for this row;
rowname = 'r'||ii||'c'; ❸

* Step through the values for this observation;
do j = 1 to 4;
 jj= left(put(j,2.));

 * Save the value for this row and column;
 call symput(trim(compress(rowname))||jj,vlist{j}); ❹

 * Save the variable name;
 if i=1 then do;
 call vname(vlist{j},name); ❺
 call symput('vname' || jj,name);
 end;
end;
run;
```

- ❶ The variables of interest are loaded into the array VLIST.
- ❷ &ROWCNT stores the number of observations, and &COLCNT stores the number of columns. These macro variables are often useful when stepping through the array in a %DO loop.
- ❸ The variable ROWNAME is used to store the first portion of the macro variable name, such as r3c.



- 4 The SYMPUT routine is used to load the value into the appropriate macro variable from within a DO loop that steps through the variables in the VLIST array.
- 5 The VNAME routine is used to capture the variable name that is associated with this column. This is not usually needed, but might be handy.

In the following statements, %PUT is used to access one of the values in the table (row 4 and column 3):

```
%let rr = 4;
%let cc = 3;
%put Row &rr and col &cc (%cmpres(&&vname&cc)) is %left(&&r&rr.c&cc);
```

The %PUT statement causes the following line to be written to the LOG. Notice that the two autocall macros %CMPRES and %LEFT are used to control spacing (see, "SAS System Autocall Macros" in Chapter 12, "Building and Using Macro Libraries," for more on the autocall macros).

|                              |
|------------------------------|
| Row 4 and col 3 (BODY1) is 2 |
|------------------------------|

Usually, some type of loop is used to step through the macro array. The macro %DBVAL displays each value in the table.

```
%macro dbval(maxrow,maxcol);
 %put row col value;
 %do row = 1 %to &maxrow;
 %do col = 1 %to &maxcol;
 %put &row &col %left(&&r&row.c&col);
 %end;
 %end;
%mend dbval;

%dbval(&rowcnt, &colcnt);
```

The macro call passes in the maximum number of rows and columns. These maximums are then used inside of two %DO loops with a %PUT statement. The following are the first few lines that are written to the LOG:

| row | col | value |
|-----|-----|-------|
| 1   | 1   | 4     |
| 1   | 2   | 9     |
| 1   | 3   | 2     |
| 1   | 4   | 4     |
| 2   | 1   | 7     |
| 2   | 2   | 1     |
| 2   | 3   | 3     |

The DATA \_NULL\_ used in this example is fairly simple. In an actual application, it is more likely that the number of variables and their names will be unknown and the names will themselves be stored in a macro variable list. This type of processing would be a direct extension of the example in Section 13.1.2.

**SEE ALSO**

Sun (1998) uses the form `&&VAR&I&J` to control two dimensional arrays, and Thorton (1999) uses the macro variable form `&A&B&C&I&J`.

**13.2.3 Using the `&&&VAR&I` variable form**

The `&&&VAR&I` form can be used to create an indirect reference to an array of macro variables. In the examples in Sections 13.2.1 and 13.2.2 two subscripts are used to identify the row and column associated with a value within a matrix. The approach shown here replaces either the row or column counter (usually the column) with a name.

In Section 9.2.2 a control file was built that contained the name of each data set and its associated BY variables. The data set DBDIR contained:

| OBS | DSN      | PAGE | KEYVAR                      |
|-----|----------|------|-----------------------------|
| 1   | DEMOG    | 1    | SUBJECT                     |
| 2   | MEDHIS   | 2    | SUBJECT MEDHISNO SEQNO      |
| 3   | PHYSEXAM | 3    | SUBJECT VISIT REPEATN SEQNO |
| 4   | VITALS   | 4    | SUBJECT VISIT SEQNO REPEATN |

The examples of that section built a series of macro variables of the form `&&VAR&I` for the data set variables DSN and KEYVER.

```
data _null_;
 set datamgt.dbdir end=eof;
 i+1;
 ii=left(put(i,3.));
 call symput('dsn'||ii,trim(dsn));
 call symput('keyvar'||ii,keyvar);
 if eof then call symput('dsncnt',ii);
run;
```

A loop to perform a PROC PRINT for each data set might look something like this:

```
%do i = 1 %to &dsncnt;
 proc print data=&&dsn&i;
 by &&keyvar&i;
 run;
%end;
```

In the above example the names of the variables in the control file, as well as how many variables there are, is known to the programmer. The variable names are needed as arguments in the CALL SYMPUT and are also used to name the macro variable arrays.

Sometimes these variable names are not known and the programmer must use an indirect reference to the array itself. In the following example the PROC PRINT is again executed; however, in this example the programmer does not know the names of the data set variables that are to be used to generate the macro variables.

The sample control data set DBDIR was introduced in Section 9.2.2 and might have been created using the following:

```
DATA dbdir;
 length dsn $8 page $1 keyvar $30 RPTTYPE $1;
 input @4 DSN $ @15 PAGE $ KEYVAR $20-47 @50 RPTTYPE $;
 datalines;
DEMOG 1 SUBJECT D
MEDHIS 2 SUBJECT MEDHISNO SEQNO D
PHYSEXAM 3 SUBJECT VISIT REPEATN SEQNO E
VITALS 4 SUBJECT VISIT SEQNO REPEATN R
run;
```

The macro %MATRIXPRINT is passed the name of the control file and the list of its variables that are to be read into macro variables. The order of the variables in the list (&VARLIST) is independent of the order of the variables in the control file. This particular implementation does assume that the first variable in the list (&VARLIST) will be a data set name and the second variable will contain a list of BY variables.

```
* Print a series of data sets using a BY list;
%macro matrixprint(control,varlist);
 %local vcnt i;
 %let vcnt=0;
 /* Count and save the variable names in VARLIST; ❶
 %do %while(%scan(&varlist,%eval(&vcnt+1),%str()) ne %str());
 %let vcnt = %eval(&vcnt+1); ❷
 %local var&vcnt; ❸
 %let var&vcnt = %trim(%left(%qscan(&varlist,&vcnt,%str())));
 %end;

 * Save all the selected data set values in macro;
 * vars
 data _null_; ❹
 set &control(keep=&varlist);
 rownum=left(put(_n_,6.));
 %do i = 1 %to &vcnt;
 call symput("&&var&i"||rownum,&&var&i); ❺
 %end;
 call symput('rowcnt',rownum); ❻
 run;
 %do i = 1 %to &rowcnt;
 proc print data=&&var1&i ❸ (where=(&var4="&&var4&i")); ❹
 by &&var2&i; ❷
 run;
 %end;

%mend matrixprint;

%matrixprint(dbdir,dsn keyvar page rpttype)
```

- ❶ The parameter &VARLIST contains the list of variables of interest in the control file. The %SCAN function is used to break the list up into individual variable names.
- ❷ The number of variables in the list is counted and saved for use later.

- ③ The names of the variables are declared to be %LOCAL.
- ④ The names of the variables are stored individually in the macro variables &VAR1, &VAR2, and so on.
- ⑤ The DATA \_NULL\_ step is used to build the lists of macro variables.
- ⑥ A %DO loop cycles through the list of variables that are to be used to generate the macro variable arrays. Each array will have a root name corresponding to the name of the variable on which it is based.
- ⑦ The number of elements in each macro variable array is stored for use later.
- ⑧ The data set name is stored in the first macro array. For &i = 2 (which refers to row 2 in the control file), &&var1&i resolves to &dsn2, which further resolves to MEDHIS.
- ⑨ One of the real advantages of this technique is that the programmer can code statements that need indirect references to both the value of a data set variable and its name. Here the WHERE clause is built using the fourth variable in the list. &var4 resolves to the variable name (RPTTYPE), while "&&var4&i" resolves to the value of RPTTYPE on which to base the subset. For &i=2 the report type will be "D".
- ⑩ This example assumes that the second variable contains a list of BY variables.

The macro %MATRIXPRINT can be used with any control file as long as the control file contains the name of the data set to be printed, a list of BY variables, and a variable that can be used in subsetting. There are no preconditions on the order or names of the variables in the control file.

## MORE INFORMATION

Section 9.2 goes into more detail on the use of control files to drive applications. The macro %GETKEYS in Section 13.1.3 uses the indirect array reference.

## SEE ALSO

Indirect array references in the form of &&&VAR&J are used by Kunselman (2001) in a SAS/IntrNet example. Burlew (1998, p.64–65) shows a brief example of this macro variable form.

## 13.2.4 Using the %SCAN function to identify array elements

In the previous section the %SCAN function was used to identify the names of the variables of interest. Each of these words was then saved as a macro variable. Rather than save the individual words of a macro variable as individual macro variables you can instead continue to use the %SCAN function directly when you need the individual values.

As was described in Section 13.2.3, to create individual macro variables from a list you might use the following:

```
%do %while(%scan(&varlist,%eval(&vcnt+1),%str()) ne %str());
 %let vcnt = %eval(&vcnt+1);
 %local var&vcnt;
 %let var&vcnt = %trim(%left(%qscan(&varlist,&vcnt,%str())));
%end;
```

As in previous examples, you could then refer to the third element by using `&var3`. Instead you could reference the third element as `%scan(&varlist,3,%str( ))`. Obviously this is more typing, but it does avoid generating the list of macro variables. This second approach is used in the following macro.

## **%SYMBOLSYNC**

**Justina Flavin**

**Pfizer Global Research & Development, LaJolla Laboratories**

The motivation for this macro was to create a series of graphs such that a given level of the variable DOSE always receives the same plot symbol, line, and so on. When all levels of the variable are present in the data, a standard set of SYMBOL statements will create a fixed association between the levels and the symbols. However, since there is nothing in the code that directly links a specific level of the plot variable to the SYMBOL, when all levels are not present the incorrect SYMBOL statement may be used.

One solution is to add dummy data such that at least one observation (even if it cannot be plotted) exists for each level of the plot variable. The other is to rewrite the SYMBOL statements based on what data is actually available for plotting. The second approach is tackled in the macro shown here. For a given level of the variable DOSE we would like to use the following plot characteristics:

| Dose level  | 1   | 2      | 3      | 4       |
|-------------|-----|--------|--------|---------|
| Plot symbol | Dot | Circle | Square | Diamond |
| Line type   | 1   | 2      | 33     | 6       |

This association of levels and symbols is placed into macro variables.

```
%let symbols = dot circle square diamond;
%let linetype = 1 2 33 6;
```

The macro %SYMBOLSYNC builds the necessary SYMBOL statements.

```
%macro symbolsync(dsn,dosevar);
%local dose j doselist numdose;
%local symbols linetype;

%let symbols = dot circle square diamond; ❶
%let linetype = 1 2 33 6; ❷

%options reset=symbol;

proc sql noprint;
 select count (distinct &dosevar) into: numdose, ❸
 distinct &dosevar into: doselist ❹
 separated by ' '
 from &dsn;
quit;

%do j=1 %to &numdose; ❺
 %let dose=%scan(&doselist, &j); ❻
```

```

symbol&j v=%scan(&symbols, &dose) ⑦
 l=%scan(&linetype,&dose) ⑧
 i=j c=black f= ;
%end;
%mend symbolsync;

```

- ❶ The plot symbols are designated as a list in a macro variable. The first symbol, dot, will be associated with dose level 1.
- ❷ The line types are designated as a list in a macro variable.
- ❸ The number of dose levels is stored in &NUMDOSE.
- ❹ Since not all dosage levels will necessarily exist in the data, a list of dose values that actually appear in the data is saved in &DOSELIST.
- ❺ A %DO loop is used to create the &NUMDOSE SYMBOL statements.
- ❻ The %SCAN function is used to determine the *J*th dose level. If the data contains only the levels 2, 3, and 4, then when &J=2 the %SCAN function returns a 3 and this value is placed in &DOSE.
- ❼ The value of &DOSE is used in another %SCAN function to select the appropriate symbol for use in the plot. If &DOSE is 3, then the selected symbol will be a SQUARE.
- ❽ &DOSE is also used with the %SCAN function to determine the line type.

A program that builds the appropriate SYMBOL statements might look something like the following:

```

data drug;
 input dose time result @@;
datalines;
 3 1 10 3 2 20 2 1 23 3 3 30 3 4 40 3 5 45
 2 2 31 2 3 35 2 4 17 4 1 40 4 2 32 4 3 18
 4 4 24 4 5 16 1 1 30 1 2 40 1 3 20 1 4 30
 1 5 35 2 5 13
run;

* Demonstrate that dot is not used when dose
* level 1 is not present;
data drug;
 set drug;
 where dose in (2,3,4);
run;

%symbolsync(drug,dose)

proc gplot data=drug;
 plot1 result*time=dose;
run;

```

In the version of %SYMBOSYNC shown above, the %SCAN function is used first to create an intermediate macro variable (&DOSE), and then again to determine the plot symbol and line type. Since, like DATA step functions, macro functions can be nested, we can avoid creating &DOSE. The SYMBOL statement could be rewritten with nested %SCAN functions as

```
symbol&j v=%scan(&symbols, %scan(&dodelist, &j))
 i=j c=black f=
 l=%scan(&linetype, %scan(&dodelist, &j));
```

The approach used by %SYMBOLSYNC only works because the levels of the dose variable are 1, 2, 3, 4, and so on. When the dose levels are not sequential, or perhaps not even numeric, the level number must be determined. In the following macro, %SYMOLSYNC2, the %REVSCAN function is used to retrieve the word number.

In this example the dose levels are not 1, 2, 3, 4, but are instead a1, b1, b2, and c.

| Dose level  | 1   | b1     | b2     | c       |
|-------------|-----|--------|--------|---------|
| Plot symbol | dot | circle | square | diamond |
| Line type   | 1   | 2      | 33     | 6       |

The %REVSCAN function (see Section 7.6.3 for more information on the reverse scan function) is used to return a sequential number for the given dose level.

```
%macro symbolsync2(dsn,dosevar);
%local dose j dodelist numdose;
%local allcodes symbols linetype;

%let allcodes = a1 b1 b2 c; ❶
%let symbols = dot circle square diamond;
%let linetype = 1 2 33 6;

goptions reset=symbol;

proc sql noprint;
 select count (distinct &dosevar) into: numdose,
 distinct &dosevar into: dodelist
 separated by ' '
 from &dsn;
quit;

%do j=1 %to &numdose;

 %let dose=❷%revscan(&allcodes,%scan(&dodelist,&j)❸);

 symbol&j v=%scan(&symbols, &dose)
 l=%scan(&linetype,&dose)
 i=j c=black f= ;

%end;
%mend symbolsync2;
```

- ❶ The list of all possible dose level codes is specified.
- ❷ The *J*th dose level in the list of observed dose levels (&DOSELIST) is determined using the %SCAN.
- ❸ The reverse SCAN function %REVSCAN returns the word number for a specified word. This number (&DOSE) is then used to select the symbols and line types.

A sample use of %SYMBOLSYNC2 could be

```
data drug;
 input dose $ time result @@;
datalines;
 b2 1 10 b2 2 20 b1 1 23 b2 3 30 b2 4 40 b2 5 45
 b1 2 31 b1 3 35 b1 4 17 c 1 40 c 2 32 c 3 18
 C 4 24 c 5 16 a1 1 30 a1 2 40 a1 3 20 a1 4 30
 A1 5 35 b1 5 13
run;

* Demonstrate that dot is not used when dose
* level 1 (dose='A1') is not present;
data drug;
 set drug;
 where dose in ('b1','b2','c');
run;

%symbolsync2(drug,dose)

proc gplot data=drug;
 plot1 result*time=dose;
run;
```

The concept of using the %SCAN function to replace a macro array can be applied to many of the examples where a macro array is formed (&&VAR&I). In %MATRIXPRINT in Section 13.2.3 a list of macro variables of the form &&VAR&I are created using a %LET statement, as follows:

```
%let var&vcnt = %qscan(&varlist,&vcnt,%str());
```

These macro variables were used inside a %DO loop.

```
%do i = 1 %to &rowcnt;
 proc print data=&&var1&i;
 by &&var2&i;
 run;
%end;
```

The explicit calls to &VAR1 and &VAR2 in the %DO loop could be replaced with the %QSCAN statement that was used to create the variable in the first place. The %DO loop becomes

```
%do i = 1 %to &rowcnt;
 proc print data=&&%qscan(&varlist,1,%str())&i;
 by &&%qscan(&varlist,2,%str())&i;
 run;
%end;
```

Assuming that the first word in &VARLIST resolves to DSN and &I is 2,

```
&&%qscan(&varlist,1,%str())&i
```



resolves to

&DSN2

### MORE INFORMATION

The reverse scan function, %REVSCAN, is discussed in Section 7.6.3.

### SEE ALSO

The SYMBOLSYNC example is discussed in more detail in Flavin and Carpenter (2003).

---

## 13.3 Programming Smarter

As complex as the macro language is, there are any number of ways to solve most problems. Of course some of the solutions are better than others. Fortunately there are techniques and strategies that you can use to maximize the results of your programming efforts. These range from the style that you use to program to your understanding of how the macro language interacts with the rest of SAS. Understanding these techniques will enable you to program smarter.

---

### 13.3.1 Efficiency issues

When you use the macro language, it is easy to get carried away and to overuse it. This is especially a problem for SAS users who are new to macros. They tend to use macro language elements at the least excuse and often when they are not at all needed. Some SAS sites have restricted the use of the macro language for this reason. Restrictions should not be necessary if you program smarter and more efficiently. A number of papers have been written about various efficiency issues and programming techniques (Culp 1991, Norton 1991; O'Connor 1992; Tindall 1991, and Westerlund 1991).

Information in Chapter 11, "Writing Efficient Macros," in *SAS Macro Language: Reference, First Edition* and *SAS Macro Language: Reference, Version 8* directly addresses these issues. This well-written chapter covers all of the major issues regarding macro efficiency and should be consulted for details not covered in the following summary.

Remember that during the discussion in this section that there is more than one kind of efficiency. Usually, one thinks of efficient code in terms of its execution by SAS, but you should also consider programmer efficiency (time to code and time to maintain code). Macro code can be very difficult to maintain (especially if you are maintaining someone else's code). A program that is **down** a week for maintenance can completely undo any savings realized by fast execution.

The following are some thoughts and tips on macro efficiency.

#### Macro language: to use or not to use

The macro language is very useful and powerful for situations that require

- conditional execution of blocks of code or procedures
- the storage of reusable or generalized code
- a block of constant code to be re-executed multiple times

- code to be written dynamically with parameters that depend on data values
- a series of repetitive tasks.

The use of macros may be inappropriate or inefficient when

- macro code is used just because the language is fun to program.
- macros are used to store constant text (see the macro `%COMMENT` in Section 3.1.2).
- macros that include the `%WINDOW` and `%DISPLAY` macro statements are used to interact with the user (SAS/AF screens are much more flexible and are easier to develop).

### **%MACRO versus %INCLUDE**

Reusable code can be stored either within a macro or in a file that is retrieved using the `%INCLUDE` statement. In addition, the macros themselves can be stored within macro libraries. There are several considerations when choosing the best techniques for your particular needs.

When macro level statements are required and macro parameters are not required, the retrieval of the code will probably be quicker with the use of the `%INCLUDE` statement. However, you should also take into consideration the programming effort needed to establish and maintain the necessary *filerefs* (as opposed to the use of an autocall library). From a practical point of view, I have not noticed a big difference between the two in terms of speed of execution.

If the code that is to be retrieved includes macro level programming statements and particularly macro definitions, the use of macro libraries offer some distinct advantages. Remember that macros retrieved through `%INCLUDE` will be compiled each time the `%INCLUDE` is executed, while using a macro library, such as an autocall library, allows the macro to be compiled once and then the compiled version can be reused.

### **Nested macro definitions**

A *nested macro definition* (see Section 5.1.3) is one that is defined inside of another macro. This programming *technique* is not as uncommon as it should be—it is rarely necessary, and it is almost always inefficient. Defining macros this way has two efficiency problems:

- The macro definition might be difficult to find for maintenance.
- The nested macro will be recompiled **every** time the outer macro is called.

### **Statement- and command-style macros**

Named-style macros always begin with a percent sign (%). This makes it easy for the macro processor (and the programmer) to find the names of the macros to be executed. Statement- and command-style macros (see Section 3.5) do not use the percent sign, and therefore additional resources are required to locate the macro calls. When they debug programs, few programmers will be looking for statement-style macros, and this can also lead to programmer confusion.

### **Multiple ampersands**

Programmers who use the macro language quickly become accustomed to seeing and using macro variable references of the form `&VAR`. Programmers who write dynamic code-building macros will become familiar (although often begrudgingly) with `&&VAR&I` macro variables.

When the code calls for three or more ampersands together, you might want to try to rethink your code, as there may be a better or easier way to handle the situation.

Macro variables of the form `&&&VAR` can, however, provide some efficiency savings in certain circumstances. This is because `&&&VAR` is an indirect macro variable reference. In the example below the macro `%NAMEONLY` is used to parse the data set name from a two-level name (see Question 7.5):

```
%macro nameonly(dsn);
 %scan(&dsn,2,.)
%mend nameonly;

%let dataset=sasclass.clinics;
%put name is %nameonly(&dataset);
```

During the execution of `%NAMEONLY`, the value `SASCLASS.CLINICS` is stored in two different macro variables. This means that the information will take up twice the memory than is necessary. In this case this is not a big deal, but if the value was long or if a large number of variables were duplicated, we could do a bit better in terms of efficiency. In the following version we pass only the name of the macro variable, **not the value**, into `%NAMEONLY`.

```
%macro nameonly(dsn);
 %scan(&&&dsn,2,.)
%mend nameonly;

%let dataset=sasclass.clinics;
%put name is %nameonly(dataset);
```

Notice the use of the triple ampersands. `&&&DSN` still resolves to the full name, however the intermediate macro variable, `&DSN`, never stores the full value, only a macro variable name.

## Macro quoting

The entire concept of macro quoting (see Section 7.1) is difficult for many macro programmers (I hesitate to say **most**, but I believe that **most** might be more accurate). Fortunately, macro quoting is not needed in most programming situations. When it is needed, ask yourself if you can code the task in a different way. If quoting is required, try to use one of the basic quoting functions such as `%BQUOTE` or `%NRSTR`.

## Macro system options

Sections 3.3 and 3.5.1 discuss several system options that are associated with the macro facility. As a general rule, when system options are turned on additional system resources are required. Some of these options are necessary; however, many options are situational. A summary of some of these system options follows:

### MACRO

If you are not using any portion of the macro facility, it can be turned off. As a general rule, you should leave it on because some statements that are not thought of as part of the macro language actually need it to be available.

### MLOGIC, MPRINT, and SYMBOLGEN

These options are used to display macro text. When a program is placed into production, there might no longer be any need to have these turned on. Because they generate text in the LOG, programs run more efficiently with these options turned off.

Often the MPRINT and NOMPRINT options are paired with the SOURCE and NOSOURCE options. For example, programs that have SOURCE turned off would probably also have MPRINT turned off.

### **CMDMAC and IMPLMAC**

Used to turn on command- and statement-style macros, these should be off unless you really, really must use one of these styles of macros. These options can cause a large efficiency loss.

### **MAUTOSOURCE, MRECALL, and SASAUTOS**

These options are used when macros are stored in AUTOCALL libraries. These libraries can be very convenient for the programmer, but some overhead is required to search the libraries. Turn off MAUTOSOURCE only if you do not want to use the AUTOCALL facility.

**CAVEAT:** Turning off the AUTOCALL facility is rarely a good idea. When turned off, all of the AUTOCALL macros supplied by SAS, such as %LEFT, %TRIM, and %VERIFY, become unavailable as well.

### **MSTORED and SASMSTORE**

When macros become permanent and are placed into production, a compiled version can be stored. Because these macros are only compiled once, you can realize a savings through the use of these libraries. If you are not using stored compiled macros, turn off MSTORED.

## **Session compiled and compiled stored macros**

Production macros can be compiled and stored in a library as a compiled stored macro. Once stored, these macros will not need to be compiled again. During a session, SAS will automatically store the compiled version (session compiled) of any called macro that is not already compiled. If you have macros in production that are not changing, they can be compiled and stored in a library. This is even more efficient than storing them uncompiled in an autocall library. See Section 3.3.4 for more on compiled stored macros.

## **Use of the autocall facility**

Macros that are used by a number of programs but are in a state of flux (or have not yet been certified for production) should be stored in an autocall library. This prevents the generation of multiple versions of the macro, and it removes the macro definition from the calling program itself. Macros defined in a calling program must be recompiled every time that program is executed, even if it has already been executed during that session. By moving the macro to an autocall library, it will only have to be compiled once in a session.

---

## **13.3.2 Programming with style**

There are a few things that you can do to make your programming life easier. The value of many of the comments in this section will be personal. Evaluate each against your programming needs, the way that you program, and your programming goals.

Try to build a personal pattern of coding. Determine which combinations of the following ideas work best for you, and apply them when developing your own style:

|                    |                                                                                                                                                                                   |
|--------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| indentation        | Use indentation or tabs to indicate changes in types of statements. For example, indent the statements inside of a %DO block.                                                     |
| capitalization     | Develop a capitalization scheme. Consider capitalizing all macro statements and using lower case for Base SAS language statements (Fehd, 2001).                                   |
| naming conventions | Think about and use naming conventions that match the flow of the program. Reusing patterns of names might make some macro programming tasks easier (Carpenter and Smith, 2001b). |
| consistency        | Try to be consistent in naming conventions used both within and across macros. Consistency can help you remember what a macro does.                                               |
| documentation      | Use comments liberally. Note each program modification in a comment header section such that the date on the modification note matches the date of the file. (See Section 5.4.1)  |
|                    | Create a standard text header section in each program that at least notes the program name, author, purpose, inputs, and outputs (Carpenter, 2003).                               |

Organize your program and your programming task by taking into consideration your programming needs and objectives.

- Plan out what you are going to do in your program (before you start coding).
- Develop a database and directory structure consistent with your programming goals (Carpenter and Smith, 2000c and 2001b).
- When writing macros look for and take advantage of patterns in your code.

As you continue to write macros, start building a macro utility tool chest. Most programmers create programs that perform similar tasks to other programs.

- Turn standard programs into macro tools so that you can avoid redundancy (See Sections 7.6 and 7.7).
- Try to use a macro library to store your utility macros (See Sections 3.3.3 and 3.3.4).

Always be on the lookout for new ways to push the boundaries of your SAS knowledge. Very often we get set in our ways. Once we learn how to do a task, we stop asking ourselves if it is the best or even the only solution. Consider the following techniques:

|                   |                                                                                                                                              |
|-------------------|----------------------------------------------------------------------------------------------------------------------------------------------|
| semicolon control | Use %DO and %END to control semicolons (See Section 5.2.2).                                                                                  |
| named parameters  | Named or keyword macro parameters provide additional documentation for your macros and often make the macro easier to use (See Section 4.3). |

|                   |                                                                                                                                                                                                                                                        |
|-------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| program structure | Minimize the number of program steps through the use of dynamic programming techniques (see Section 5.2.2 and Chapter 9, “Writing Dynamic Code”).<br>Use structured programming techniques. Avoid the %GOTO and %label statements (see Section 5.4.3). |
| %LOCAL            | Use %LOCAL to avoid macro variable collisions. Keep macro variables out of the global symbol table except when necessary (Section 5.4.2)                                                                                                               |
| functions         | Learn the Base SAS language DATA step functions. Although not part of the macro language, through the use of %SYSFUNC many of these functions can be used very effectively with the macro language (Section 7.4.2).                                    |
| PROC SQL          | Use PROC SQL to build and maintain macro variables and macro variable lists. Often a single SQL step can accomplish the same thing as a number of other Base SAS language steps (see Sections 2.7 and 6.5).                                            |

**SEE ALSO**

Levine (1989) discusses issues related to the efficient construction of macro-based systems in terms of standards and maintainability.

Fehd (2000 and 2001) has some great tips on writing good code and things to look for when writing macros. Cheng (1999) includes a number of tips on various aspects of SAS coding, including some tips on macro techniques.

A summary of things to think about before starting to write your macro can be found in Tangedal (2001).

Pochon and Burger (1998) discuss techniques that can be used to validate SAS macros.

Whitlock (2000a) discusses macro design considerations for a word-wrapping macro. Troxell (2001) discusses the use of complex macros to write code that is easier to validate. Methods of writing macros that are more resistant to user errors are presented by Troxell (2002).

Top-down programming techniques as they apply to macro programming are discussed by Heaton (2001).

Although primarily discussing non-macro issues, Levin (2001) makes a number of very good recommendations regarding SAS programming style.

---

### 13.3.3 Debugging your macros

As much as I hate to say it, your primary tool for debugging macros will be your own experience. In other words, when you need the most help you will have the least experience. Later, when you are more experienced, debugging will be easier—of course by then you will be writing more complicated macros with more complicated errors. In the meantime consider some of the following thoughts on debugging your macros.

SAS OnlineDoc has two extensive sections on debugging and troubleshooting macros. Look under the Macro Facility Error Messages and Debugging section in the SAS Macro Language Reference for a number of debugging strategies.

## Using system options

The system options MPRINT, SYMBOLGEN, and MLOGIC (see Section 3.3.2) are especially useful during the debugging process. While these options will not make debugging easy, they can assist by providing valuable insights into the code that is being generated by the macros.

If it is not clear what the resultant code that has been generated by your macro looks like, then consider using the MFILE option to write the generated code to a file.

## Using %PUT

The %PUT statement can be very useful to dump all or part of the various symbol tables to the LOG. This statement supports a number of options (see Section 2.4) that make it easy for you to display macro variables, their values, and the symbol tables in which they reside.

## Unresolved macro calls

The WARNING “Apparent invocation of macro...” indicates that SAS has been unable to find the definition for the macro that has been called. An unresolved macro call results when this definition is not available. Sections 3.3.3 and 3.3.4 discuss how SAS searches for macro definitions and some of the particular issues to consider when you are using macro libraries.

Failure to find macros supplied by SAS such as %LEFT and %TRIM can often be traced to either a failure to include SASAUTOS in the autocall path or to having turned off the autocall facility altogether.

The misspelling of macro names can also contribute to this type of error, so be careful with naming conventions.

## SEE ALSO

A number of SUGI papers address various aspects of the debugging process. Although some are based on Version 5 options, these papers can provide good insight: Frankel and Kochanski (1991); Gilmore and Helwig (1990); O'Connor (1991); and Phillips, Walgamotte, and Drummond (1993).

Chapter 10, “Macro Facility Error Messages and Debugging,” in *SAS Macro Language: Reference, First Edition* (pp. 111–130) and *SAS Macro Language: Reference, Version 8* (pp. 107–128) both discuss a variety of topics that relate to the debugging and troubleshooting of macros.

O'Connor (1991), one of the primary developers of the macro language at SAS, provides some guidelines for tracking errors and for debugging your macro code.

Although written for Base SAS, Staum (2002) discusses a number of techniques and ideas that can be useful to the macro programmer as well.

Heaton (2001) discusses techniques that can assist with the debugging of macros.

### 13.3.4 The DATA step versus the macro language

While there are a number of similarities between the Base SAS language, especially in the DATA step and the macro language, there are enough differences that the unwary programmer can be either confused by subtle differences or caught unawares by surprising mistakes. A few of the things that have caused folks consternation are included here. Many of these points of confusion occur when the programmer has not fully assimilated the information contained in the figure in Section 1.3. A great deal of our understanding of the processes involved with the macro language is tied to the timing of these phases of SAS execution.

#### Condition macro statement execution

In the DATA step we cannot conditionally execute macros or macro statements (see Sections 6.1 and 10.2.1). The following IF statement attempts to assign a value to &GROUP based on the DATA set variable AGE.

```
if age > 55 then %let group=senior;
else %let group = ;
```

Not only will this code result in a syntax error, but you must always remember that the macro portions of these statements will be executed long before the IF statement. In this case, although the DATA step will fail, &GROUP will always have a null value. After the macro statements have executed the IF statements become

```
if age > 55 then
else
```

Of course macro variables can be conditionally specified when DATA step, rather than macro language, tools are used. The following will work correctly:

```
if age > 55 then call symput('group',senior);
else call symput('group','');
```

#### IF versus %IF in the DATA step

It is not uncommon for programmers to attempt to interchange the use of the IF and the %IF statements. When in doubt as to which you should use, remember that the %IF can **never** evaluate the value of a DATA variable. Never is a bit strong here, but you would have to go to great lengths to have the %IF evaluate the value of a DATA variable.

Macro statements are used to build code, and the %IF is used to evaluate the values of macro variables. The DATA step IF might use macro variables to build the names of variables or constants that are to be evaluated during DATA step execution, but it is primarily used to evaluate DATA step variables. In the following rather silly DATA step, AGE is a constant and is never > 55; however, every observation from OLD is written to NEW:

```
data new;
set old;
age = 10;
%if age > 55 %then %do; output new; %end;
run;
```



In the %IF statement the comparison is not between the variable AGE and 55, but rather between the letters *a-g-e* and 55. Remember when the %IF is evaluated, **no** data has been read and the variable AGE does not yet exist. Since lowercase letters (a-g-e) sort after numbers (in the ASCII collating sequence), the comparison is true. The DATA step code becomes

```
data new;
set old;
age = 10;
output new;
run;
```

The %IF is used incorrectly in an attempt to compare DATA step variables. This type of comparison should of course be made using the IF statement.

The following code (from Section 5.4.3) works correctly. However, since &DSN is a constant, the IF will be executed for every observation in the incoming data set regardless of the value of &DSN.

```
%macro smart(dsn);
data wt;
set &dsn;
if "&dsn"='FEMALE' then wt = wt*2.2;
run;
%mend smart;
```

By using an %IF statement instead of an IF (no comparison is being made on a value in the data set), the assignment statement is only added to the DATA step when &DSN is FEMALE. This is much more efficient.

```
%macro smart(dsn);
data wt;
set &dsn;
%if &dsn=FEMALE %then wt = wt*2.2;;
run;
%mend smart;
```

As in the above two versions of %SMART, you can make the correct choice between %IF and IF by looking at what is being compared. In this case it is a macro variable—not a data set value—therefore a %IF is used.

## Mixing functions

Some programmers have a tendency to mix DATA step and macro functions when in the DATA step. There are times when this is necessary; however, because the macro references are resolved first, the necessity is rare.

The following IF statement contains a number of embedded function calls. The author used macro functions (%TRIM and %LEFT) whenever the argument was a macro variable. In fact **this** was not necessary because the macro variable will be resolved to text prior to the application of the macro functions.

```
if upcase(reverse(substr(left(reverse(filename))
,1,length(%trim(%left("&TYP"))))))=%trim(%left("&TYP"));
```

Also, it is possible that this usage will yield an incorrect result. Since the macro functions are outside of the quotes in the above example, they effectively do nothing. This is because the text "&typ" is **always** left-justified when applying the %LEFT function regardless of the value of &TYPE. Remember that the quotation mark on the left is part of the text and it is already the leftmost character. The following could give a different result:

```
"%trim(%left(&TYP))"
```

There are times, however, when the macro function can be put to good use within non-macro code. In the following PROC PRINT (Stroupe, 2003) the DATA step UPCASE function could not be used.

```
proc print data=&dsn;
 title "Lowest Priced Hotels in the %upcase(&dsn) Data Set";
run;
```

## Comparisons with missing or null values

Unlike the DATA step, the macro language does not support the concept of a missing value (see Sections 5.2.1 and 7.1.2). This means that there is no automatic placeholder when making comparisons. In the DATA step we might check to see if the variable AGE is missing with the following:

```
if age = . then do;
```

Of course if we try something similar in a macro %IF we will be less successful. In this case we want to see if the macro variable &AGE has a value:

```
%if &age = . %then %do;
```

The comparison will be true only when &AGE holds the value of a period (.), and this is not the same as a null value. Since, unlike DATA step variables, macro variables can take on a null value, the %IF comparison might look like the following:

```
%if &age = %then %do;
```

Many programmers familiar with comparisons in the DATA step are uncomfortable without a value on the right side of the comparison operator. A common and fairly intuitive solution, but one which often seems to indicate a lack of understanding of how the macro language is working, is to use quotation marks to "mark" the right side of the comparison operator (see Sections 5.2.1 and Q5.9).

```
%if "&age" = "" %then %do;
```

Since in the macro language the quotation marks actually become part of what is compared, the first character on both sides of the equal sign is ". A solution that stays within the macro language, but still satisfies the desire to place something on the right side of the comparison operator, is to use a quoting function with no spaces between the parentheses (Section 7.1.2).

```
%if &age = %bquote() %then %do;
```

## Using quotes in the macro language

The third argument to the %SCAN function is used as the optional word delimiter. When used, it contains one or more characters that will be used to separate one word from the next. In the DATA step this argument is often specified as a quoted string.

```
wrd2 = scan(string,2,' ');
```

In the macro language quotation marks are, of course, used quite differently; however, sometimes DATA step programming habits **accidentally** find their way into our macros. This has happened in the following %LET statement:

```
%let wrd2 = %qscan(&string,2,' ');
```

The user expects the word delimiter to be a blank space. In fact three delimiters have been specified (two of which, the quotes, are the same). Using these delimiters, the following value of &STRING

```
AA B'C D'C
```

will have five words, not three:

```
AA B C D C
```

This could, of course, have unintended consequences. The appropriate code would use a quoting function to preserve the space:

```
%let wrd2 = %qscan(&string,2,%str());
```

## Using implied arithmetic operations

In the DATA step, variables are known to be numeric or character and all numeric operations are applied to numeric variables and constants. In the macro language the distinction between numeric and character is not as clear cut. Section 7.3.1 showed that the arithmetic in the following %LET statement will not be performed:

```
%let x = &y + 1;
```

However, this arithmetic operation is automatically done in the DATA step. The code fragment shown below contains this arithmetic operation (addition) and it would work as anticipated in a DATA step.

```
i=0;
DO WHILE((scan(xcode, i+1.0))>' ');
 i = i+1;
```

The second argument in the SCAN function adds 1 to the current value of i. In the macro language this might be rewritten as

```
%let i=0;
%DO %WHILE((%scan(&xcode, &i+1))>%bquote());
 %let i = %eval(i+1);
```

It turns out that this %SCAN will work correctly because there is an implied %EVAL (Section 7.3.2) on the second argument (%SCAN knows that the second argument has got to be treated as a number, and because it detects an arithmetic operator, the plus sign, a %EVAL is used to perform the addition). It is as if the %DO %WHILE had been coded as

```
%let i=0;
%DO %WHILE((%scan(&xcode, %eval(&i+1)))>%bquote());
 %let i = %eval(i+1);
```

Because of the implied %EVAL, which only works with integers, you would be **unable** to code the second argument as

```
%let i=0;
%DO %WHILE((%scan(&xcode, &i+1.0))>%bquote());
 %let i = %eval(i+1);
```

Remember the macro language would interpret 1.0 and even 1. as noninteger values. For noninteger arithmetic you will need to use %SYSEVALF explicitly:

```
%let i=0;
%DO %WHILE((%scan(&xcode, %sysevalf(&i+1.0)))>%bquote());
 %let i = %eval(i+1);
```

## Creating and using a macro variable

Generally, when SAS programmers create and use a macro variable in the same DATA step, they are not taking advantage of the power of the DATA step. If you think about it, we are asking to take a DATA step value, write it to the symbol table, and then later retrieve it from the symbol table. Why not use the Program Data Vector, which can do this exact same thing using the RETAIN statement without stepping out of the DATA step? Section 6.4.3 shows a fairly practical example of using the macro variable in the same step that creates it.

That said, let's look at the problem that folks often have when trying to do this sort of thing. In the following DATA step, the first observation's value of the variable AGE is written to the symbol table using the SYMPUT routine:

```
data new;
 set old;
 by age;
 if _n_=1 then call symput('firstage',put(age,3.));
run;
```

The youngest age for this patient will be stored in &FIRSTAGE. If all we wanted to do is save youngest or first age for use later within the DATA step, we could have written the step as

```
data new;
 set old;
 by age;
 retain firstage .;
 if _n_=1 then firstage = age;
run;
```

For the sake of discussion, let's assume that the youngest age is to be used to subset the data. The DATA-step-only solution might be

```
data ageerrors;
 set old;
 by age;
 retain firstage .;
 if _n_=1 then firstage = age;
 if age-firstage>10 then output;
run;
```

A common error, when using macro variables to do something similar, will be encountered in the following step:

```
data new;
 set old;
 by age;
 if _n_=1 then call symput('firstage',put(age,3.));
 if age-&firstage>10 then output;
run;
```

The macro facility will attempt to resolve &FIRSTAGE before a value has been assigned. Remember that this resolution will occur before **any** data has been read. This will, of course, cause syntax errors. Instead, you could use the SYMGET function to retrieve the macro variable during the execution of the DATA step.

```
data new;
 set old;
 by age;
 if _n_=1 then call symput('firstage',put(age,3.));
 if age-symget('firstage')>10 then output;
run;
```

By using the SYMGET, this DATA step could be modified so that it could be used with more than one patient, without creating a macro variable for each patient.

```
data new;
 set old;
 by name age;
 if first.name then call symput('firstage',put(age,3.));
 if age-symget('firstage')>10 then output;
run;
```

Again, for this example the DATA-step-only solution is more practical, but it is important to understand the timing of the process none the less.

### 13.3.5 Little things with a big bite

There are some problems that you could encounter in your macro programming career that you would not wish on your worst enemy well, maybe on your worst enemy, but no one else. These are the things that generate many hours if not days of frustration. They might not seem like much, but when they crop up, bad things can happen.

## Macro variables are in the wrong symbol table

There are a number of ways of creating macro variables, and how and when they are created will determine whether the macro variable will be written to a LOCAL or GLOBAL symbol table. The topic is complex enough to be discussed in more detail in Section 13.6.

An indication of a typical problem that occurs when variables go astray is the “undefined macro variable reference” error message in the LOG. When you get this message and you **know** that the variable should be defined, the variable might have been placed in the wrong symbol table.

Assuming that the macro variable really does exist, you might want to look for it in a symbol table that was not available when the message was generated. When you are sure that the variable has indeed been defined, consider using a %PUT statement to identify the macro variables in the various symbol tables. The following code creates

- ❶ a global macro variable (&A)
- ❷ a macro variable in the symbol table for %TRYIT (&AT)
- ❸ a macro variable in the symbol table for %INSIDE (&B).

```
%let a = global_var; ❶

%macro tryit;
 %let at = var_local_to_tryit; ❷
 %inside
%mend tryit;

%macro inside;
 %let b=var_on_the_inside_table; ❸
 %put **All Current Vars;
 %put _user_; ❹
%mend inside;

%tryit
%put **** Open code Macro vars;
%put _user_; ❹
```

The \_user\_ option on the %PUT statement ❹ writes the following to the LOG

- name of the symbol table
- name of the macro variable
- current value of the macro variable.

After executing %TRYIT the LOG shows the following:

```

33 %tryit
**All Current Vars
INSIDE B var_on_the_inside_table
TRYIT AT var_local_to_tryit
GLOBAL A global_var
34 %put **** Open code Macro vars;
**** Open code Macro vars
35 %put _user_;
GLOBAL A global_var

```

Macro variables can be written to the wrong table when

- the programmer does not understand the rules that are used to place macro variables into symbol tables (see Section 13.6)
- the programmer does not notice a %GLOBAL or %LOCAL statement that names the offending macro variable
- there is a macro variable collision (see next topic).

## Macro variable collisions

A *macro variable collision* occurs when a value of one macro variable interferes with (replaces or changes) the value of a macro variable with the same name that should be in a separate symbol table.

A simple example can be used to illustrate the problem. In the following code the programmer creates a global macro variable (&DSN) to hold the name of the data set of interest. After his program was working his boss asked him to add a call to a macro (%DATSERNUM). Now whenever he uses the macro %DATSERNUM, the DATA step that creates NEW terminates with syntax errors. Although he does not yet know it, he has experienced a macro variable collision. Here is the portion of his program that has the syntax error:

```

%* Define the data set of interest;
%let dsn = clinics;

%* Determine the Data Serial Number;
%DatSerNum(adjust=5)
* Create the new data;
data new;
set &dsn;
...code not shown...
run;

```

It turns out that the problem really resides in the macro %DATSERNUM, a portion of which is shown here:

```

%macro datsernum(adjust=0);
 %if &adjust= %then %let adjust=0;
 %let dsn = %eval(5 + &adjust); ❶
 %if &dsn > 5 %then %SerNumRpt(&dsn);
%mend datsernum;

```

When the variable `&DSN` is defined in the macro `%DATSERNUM` ❶ its value would normally be placed in the local symbol table for `%DATSERNUM`; however, since `&DSN` already exists in a higher table, the value in that table will be replaced. In this case the name of the data set (CLINICS) is replaced by a number. It is this number that causes the syntax error when it is used as a data set name in the DATA step that creates NEW.

A more subtle example, and one that can cause horrid errors while leaving the programmer blissfully unaware, is contained in the program fragments below. In this case a secondary macro is called from within a `%DO` loop.

```
%macro primary;
 ...code not shown...
 %do i = 1 %to &dsncnt;
 %chksurvey(&&dsn&i)
 %end;
 ...code not shown...
%mend primary;
```

The `%DO` seems to work without error. But a closer inspection of the inner macro `%CHKSURVEY` reveals a hidden problem:

```
%macro chksurvey(dset);
 %do i = 1 %to 5;
 ...code not shown...
 %mend chksurvey;
```

The `%DO` loop in `%CHKSURVEY` also uses 'i' as the index variable. Again, this variable would normally be local to `%CHKSURVEY`; however, since `&I` already exists in the higher table of the calling macro, `%CHKSURVEY` will modify the value of `&I` in that table. In the case shown above, `&I` will have a value of 6 after `%CHKSURVEY` has been executed. When `&DSNCNT` is less than 7, the `%DO` loop in `%PRIMARY` will terminate **after having executed only once!** If `&DSNCNT` is greater than 6 an infinite loop will have been established, since the `&I` in `%PRIMARY` will be reset to 6 over and over again. At least the programmer is likely to discover the latter problem.

The above macro variable collisions can be completely eliminated simply by using the `%LOCAL` statement to identify all macro variables that you intend to be local to the inner macro. `%CHKSURVEY` then becomes

```
%macro chksurvey(dset);
 %local i;
 %do i = 1 %to 5;
 ...code not shown...
 %mend chksurvey;
```

## Asterisk-style comments

There are three styles of comment statements that are available to SAS. Although all three can be used either in open code or within macros, they do not behave the same nor are they necessarily interchangeable. They behave differently in part because of when and how they are removed from the code (see Section 5.4.1 for more on the macro comment).

It is not unusual to write code that uses the asterisk-style comment to comment out macro statements. In the following open code statements the `%LET` statement associated with the data type of interest is left uncommented.



```

*%let dattype = ae;
%let dattype = demog;
*%let dattype = meds;
*****;

%put data type is &dattype;

```

In this case all three styles of comments will work equally well, and as we would expect, the %PUT writes to the LOG that the “data type is demog”.

When this same code is placed inside of a macro, however, the result is **not** the same. The LOG shows the outcome:

```

55 %macro tryit;
56 *%let dattype = ae;
57 %let dattype = demog;
58 *%let dattype = meds;
59 *****;
60
61 %put data type is &dattype;
62 %mend tryit;
63 %tryit
data type is meds

```

To understand the problem, we need to keep in mind how the code is being parsed. The parser breaks down the code into elemental units known as *tokens*. SAS statements start with an identifying set of characters known as a *keyword*. This keyword lets SAS know how to interpret the remaining characters through the end of the statement (semicolon).

In open code the asterisk is seen as keyword and all characters between the asterisk and the semicolon are commented out. This is even true for the %LET within the comment. Thus, in open code it is possible to comment out macro statements and macro calls using asterisk-style comments.

Inside a macro things are different. Once the %MACRO statement is encountered all the code between the %MACRO and %MEND is passed to the macro facility. Here, when the code is parsed, SAS looks for macro statements and generally ignores most of the things that otherwise have meaning in the Base SAS language (for example, asterisks are not seen as tokens and are therefore not interpreted as the keyword that starts a comment). Looking more closely at %TRYIT below, the macro statements that were executed are bolded:

```

%macro tryit;
 *%let dattype = ae;
 %let dattype = demog;
 *%let dattype = meds;
 *****;

 %put data type is &dattype;
%mend tryit;

```

All three %LET statements are executed, the last of which assigns the value of MEDS to &DATTYPE. Remember that, although macro statements such as %LET cannot be commented with an asterisk, a macro call is not seen as a macro statement and can still be commented in this manner (see example below). After the macro statements have been executed, the asterisks remain. Essentially, the code that is passed out of %TRYIT becomes

```
%macro tryit;
 *

 *
 *****;

%mend tryit;
```

As a cautionary comment on what just happened, notice that the string of asterisks end up forming a comment that starts with the very first asterisk. If this string of asterisks had not been in the macro, the first two asterisks would have been left behind without a semicolon. This could easily cause a problem by inadvertently creating a comment where it was not wanted. This is also a clue to answering the questions posed in the following example.

**QUIZ:** In order to determine your level of understanding of the use of comments within a macro, consider the macro %ABC which contains a single %PUT statement that the programmer has attempted to comment. This macro is called twice in the macro %DOIT. Both times an attempt has been made to comment out the macro call and the %LET statement that contains the second macro call.

- Will the %PUT ❶ be executed if %ABC is called?
- When %DOIT is executed how many times will %ABC ❷ be called?
- During the execution of %DOIT what value will be assigned to &X? ❸
- What, if anything, is the resolved (non-macro) code returned by %DOIT? ❹

```
%macro abc;
 *%put in abc; ❶
%mend abc;

%macro doit;
 *%abc ❷
 %put here;
 *%let x = %abc; ❷ ❸
 %put value of x is &x;
%mend doit;
%doit * run doit; ❹
```

The LOG shows the following:

```
35 %doit * run doit;
here
in abc ❷
value of x is * ❸
```

- ❶ The asterisk will not form a comment and the %PUT will be executed. This leaves the \* behind as the resolved text from the macro %ABC.
- ❷ The first call to %ABC is commented; however, neither the %PUT nor the %LET is commented. Before a value can be assigned to &X, macro %ABC is executed, and this in turn executes the %PUT that is in %ABC.
- ❸ The only non-macro text in %ABC (the asterisk) is passed back as the value for &X.
- ❹ The non-macro text in %DOIT will be \*%abc\*.

EXTRA CREDIT: What would the LOG show if you called the macro %DOIT with the following %PUT statement? (See Appendix 1 for the answer.)

```
%put %doit;
```

## Macro syntax

Debugging macros can sometimes be problematic. The error messages, even when using MPRINT, MLOGIC, and SYMBOLGEN, can occasionally be a bit cryptic. The problem can be exacerbated when your code is missing pieces of the macro language.

A simple but non-trivial example occurs when the programmer forgets to use the %MEND statement to close the macro. When programming interactively, this is an especially irritating problem as the system seems to freeze (it is actually waiting patiently for the %MEND). Resubmitting the program only makes things worse as SAS is then waiting for two %MEND statements.

You can clear these pending macro definitions by submitting a series of %MEND; statements. You might need to submit several. You will know that you need no additional %MEND statements when the following error message is displayed in the LOG:

```
ERROR: No matching %MACRO statement for this %MEND statement.
```

Related problems occur when the % sign is left off of the %MEND or when the name of the macro does not match the name on the %MEND statement.

The error in the following example also wipes out the %MEND statement, but by a very different mechanism—a missing parenthesis. Parentheses of course come in pairs and when macro functions, especially embedded macro functions do not have the correct pairs of parentheses the error messages point in every direction except to the parentheses. The following macro function counts the number of words in a list using the %SCAN and %STR functions.

```
%macro wrdcnt(string);
%let cnt=0;
%do %while(%scan(&string,&cnt+1,%str() ne %str()));
 %let cnt = %eval(&cnt+1);
%end;
&cnt
%mend wrdcnt;
```

If we “accidentally” leave out one of the closing parentheses after the first use of %STR, the code becomes

```
%macro wrdcnt(string);
%let cnt=0;
%do %while(%scan(&string,&cnt+1,%str() ne %str()));
 %let cnt = %eval(&cnt+1);
%end;
&cnt
%mend wrdcnt;
```

When this version of %WRDCNT is executed the LOG shows the following:

```
124 %macro wrdcnt(string);
125 %let cnt=0;
126 %do %while(%scan(&string,&cnt+1,%str() ne %str()));
127 %let cnt = %eval(&cnt+1);
ERROR: Macro keyword LET appears as text. A semicolon
or other delimiter may be missing.
128 %end;
ERROR: Macro keyword END appears as text. A semicolon
or other delimiter may be missing.
129 &cnt
130 %mend wrdcnt;
ERROR: Macro keyword MEND appears as text. A semicolon
or other delimiter may be missing.
131
132 %put the count is %wrdcnt(a b c d);
ERROR: Macro keyword PUT appears as text. A semicolon
or other delimiter may be missing.
ERROR: Expected semicolon not found after WHILE clause.
A dummy macro will be compiled.
WARNING: Missing %MEND statement.
```

As you can see, just about every statement is flagged except the one with the missing parentheses. Especially note that since the %MEND was also missing, an interactive session will be waiting for a %MEND statement. When you are using nested functions and you get odd messages, in addition to missing semicolons, also look for mismatched parentheses.

## Noninteger comparisons

When making numeric comparisons one of the first things to remember is that by default the macro language will only perform a numeric comparison for integer values (for example,  $9 < 10$ ). The presence of decimal values or even of the decimal point will invoke the use of a character or alphabetical comparison (for example,  $10. < 9.$ ). More details on these comparisons, the defaults, and how to override them are discussed in Section 7.3.

It of course remains your responsibility to ensure that the type of comparison is what you expect it to be. If you anticipate that the values are to be integers, you can enable the use of the implicit %EVAL. But then a comparison of noninteger values could then result in incorrect conclusions some of the time.

## Numeric range comparisons

The macro %CHECKIT performs a simple range check on the value passed into the macro. The intent is to identify all values that are either in the range of -5 to 0 or in the range of 1 to 5.

```
%macro checkit(val);
 %if -5 le &val le 0 %then %put &val is in neg range (-5 to 0);
 %if 1 le &val le 5 %then %put &val is in pos range (1 to 5);
%mend checkit;
```

The LOG shows the following:

```
150 %checkit(-10)
-10 is in neg range (-5 to 0)
-10 is in pos range (1 to 5)
151 %checkit(-2)
-2 is in pos range (1 to 5)
152 %checkit(2)
2 is in pos range (1 to 5)
153 %checkit(10)
10 is in pos range (1 to 5)
```

Virtually every determination that this macro makes is incorrect! What has happened?

In the DATA step this type of range check is very common and the following IF will work correctly:

```
if -5 le val le 0 then do;
```

The DATA step interprets the range comparison as if it had been coded as

```
if -5 le val and val le 0 then do;
```

The macro language, however, does not interpret the expression in the same way. The first %IF in %CHECKIT:

```
%if -5 le &val le 0 %then ...
```

is interpreted as

```
%if (-5 le &val) le 0 %then ...
```

when &VAL is -10 the %IF becomes

```
%if (-5 le -10) le 0 %then ...
```

since the comparison that checks to see if -5 was less than -10 is false, a zero is returned and the expression becomes

```
%if 0 le 0 %then ...
```

and this is of course true. This means that when you want to do a range check in the macro language you **must** use a compound expression. In order for it to be correctly evaluated, the previous %IF becomes

```
%if -5 le &val and &val le 0 %then ...
```

## Building composite macro calls

Sometimes when writing dynamic applications the name of the macro to be called might not be known when the program is written. In the following example a series of macros have been written to read the data into various data sets. The macro names include %READ\_AE, %READ\_CM, and %READ\_PT. The name of the data set is the last two letters of the macro name and these are stored in &DSN. To construct the call for the CM data set the following code might be used:

```
%let dsn = cm;
...code not shown...
%read_&dsn
```

Of course in order for the macro call to work correctly, the &DSN must be resolved **before** the macro facility attempts to execute the macro. Otherwise SAS will attempt to execute %READ\_ first, which will almost certainly cause an error.

The problem is that not all versions of SAS handle the %READ\_&DSN in the same way. Versions 5.18, and 8.0 will attempt to execute %READ\_ before resolving &DSN. Versions 6 and 7, Release 8.2, and SAS 9.0 will resolve &DSN first. There has been some discussion as to which is the “correct” way, but knowing what will happen is more important. If you are using Release 5.18 or Version 8 and need to force the resolution of &DSN before the macro is called, you could call the macro using something like the following:

```
%unquote(%nrstr(%read_)&dsn)
```

---

## 13.4 Working with Macro Parameter Buffers

A buffer is a temporary storage location that can be used to store or pass information. Macro parameter buffers enable you to create macros with a variable number of parameters. The PARMBUFF (or PBUFF) switch is used to turn on the parameter buffer and the automatic macro variable &SYSPBUFF is used to hold the buffer’s value.

**SEE ALSO**

Mace (1999) briefly discusses the automatic macro variable &SYSPBUFF.

**13.4.1 Calling a macro using /PARMBUFF**

The /PARMBUFF switch is used on the %MACRO statement to turn on the ability to load the macro variable &SYSPBUFF when the macro is called. The following simplistic macro demonstrates the process that you will use:

```
%macro demo(a=1, b=2)/parmbuff; ❶
%put buffer holds &syspbuff; ❸
%put var A is &a; ❹
%put var B is &b; ❺
%mend demo;

%demo(a=aa) ❷
```

The LOG shows the following:

```
7
8 %demo(a=aa) ❷
buffer holds (a=aa) ❸
var A is aa ❹
var B is 2 ❺
```

- ❶ The macro statement shows two named parameters and the /PARMBUFF switch.
- ❷ The macro is called with a single parameter.
- ❸ The first %PUT writes the contents of &SYSPBUFF to the LOG. Notice that the value of &SYSPBUFF contains the parentheses as well.
- ❹ The value of &A has been passed into the macro and is set to aa.
- ❺ Since &B is not included in the macro call, its value is not included in &SYSPBUFF, and the value of &B remains at its default value.

Since the parameter values being passed to the macro are coming through the buffer, you could call the macro using parameters that do not exist. The call %demo(a=aa, d=ddd) runs without error and produces the following LOG:

```
15
16 %demo(a=aa, d=ddd)
buffer holds (a=aa, d=ddd)
var A is aa
var B is 2
```

Of course in the previous example the macro variable &D is undefined. This means that you can pass values and then parse them yourself. This enables you to build a macro with a variable number of parameters, and this is done in the macro %IN in the following section.

## 13.4.2 Using the PARMBUFF macro switch

%IN

Pete Lund

Looking Glass Analytics

This macro function can be used to build a highly flexible IN operator for the DATA step IF statement. It enables you to check a character variable against a list of values of varying lengths. Additionally, you can optionally match only the first few characters of the string. It does this by building an IF expression of the following form:

```
if (code eq 'a1' or code eq 'a2' or code eq 'a3') then....
```

The variable that is to be checked (code) is the first parameter in the macro call and the remaining parameters form the values (a1, a2, and a3). The macro is

```
%macro in() / parmbuff; ❶
 %let parms = %qsubstr(&syspbuff,2,%length(&syspbuff)-2); ❷

 %let var = %scan(&parms,1,%str(,)); ❸

 %let numparms = ❹ %eval(%length(&parms) -
 %length(%sysfunc(compress(&parms,%str(,))))) ;

 %let infunc = &var eq %scan(&parms,2,%str(,)); ❺

 %do i = 3 %to (&numparms + 1); ❻
 %let thisparm = %scan(&parms,&i,%str(,)); ❼
 %let infunc = &infunc or &var eq &thisparm; ❽
 %end;

 (&infunc) ❾
%mend;
```

- ❶ The macro is specified without any parameters. The information needed by the macro will come in through &SYSPBUFF.
- ❷ The parentheses that are automatically included with &SYSPBUFF are stripped off.
- ❸ The first parameter is the variable name that will be checked.
- ❹ Count the number of parameters by counting the number of commas. In this statement the commas are counted by comparing the length of &PARM with its length after the commas have been compressed out. The number of parameters, &NUMPARMS, is one too small as there is one more parameter than there are commas and the first parameter is actually the variable name.
- ❺ The macro variable &INFUNC will be used to hold the expression that is being built. This statement creates the first expression by equating the name of the variable, &VAR, with the first parameter value (second word in the list).



- ⑥ Loop through the remaining parameters (this macro expects at least two comparison parameters).
- ⑦ Select the next value from the parameter list.
- ⑧ Add this comparison onto the growing list in &INFUNC.
- ⑨ Use &INFUNC to pass the list of comparisons back to the IF statement.

In the call to the %IN macro below, an IF statement will be built that will check a variable (CPT) against the following values 4300, 4301, 44xx, 451x. Here *x* represents a wild-card value, which is established by placing the colon operator before those values that include partial strings.

```
If %in(cpt, '4300', '4301', ':'44', ':'451') then...
```

The previous macro call would generate the following code:

```
if (cpt eq '4300' or cpt eq '4301' or cpt eq ':'44'
 or cpt eq ':'451') then ...
```

Since &SYSPBUFF is being used to pass the parameters, the macro call can contain any number of comma-separated values.

### %ORLIST

This macro is similar to %IN as it also uses the PARMBUFF switch; however, it parses &SYSPBUFF differently. The %DO %WHILE loop is used to pass through the list of parameter values and one by one the variable/value pairs are added to &ORLIST.

```
%macro ORlist() / pbuff;
 %local datvar i parm orlist;
 %let datvar = %qscan(&syspbuff,1,%str(,)); ①
 %let i = 1;
 %do %while(%qscan(&syspbuff,&i+1,%str(,%())) ne %str());
 %let parm = %qscan(&syspbuff,&i+1,%str(,%())); ②
 %if &i=1 %then %let orlist = &datvar=&parm; ③
 %else %let orlist = &orlist or &datvar=&parm;
 %let i = %eval(&i + 1);
 %end;
 &orlist ④
%mend orlist;
```

- ① The variable name is retrieved as the first word. The %( is used to mark the open parentheses as a word delimiter.
- ② %QSCAN is used to separate the values. Notice the use of %( and %) to designate the open and close parentheses as word delimiters (this prevents them from becoming a part of the first and last words selected by %QSCAN. (See Section 7.1.9 for a discussion of the marking of special characters.)
- ③ The list of comparisons is temporarily stored in &ORLIST.
- ④ The resulting list of comparisons is passed back.

**SEE ALSO**

A more detailed discussion of %IN and other user-written macro functions can be found in Lund (1998, 2000a, and 2000b). The /PARMBUFF switch and &SYSPBUFF macro variable are used by Lund (2000a) to build a formatted comment for the LOG.

---

## 13.5 Understanding Recursion in the Macro Language

*Recursion* occurs when an element of the macro language calls or references itself. Depending on the circumstances, this could be useful or it could cause errors.

Usually you will encounter recursion in the form of an error—you didn't do it on purpose. Often this is the result of a syntax error and the most likely culprit will be a missing semicolon. In the following statements, the user has attempted to use the %LET to assign a value to &DSN. This value is then to be displayed using a %PUT.

```
%let dsn = clinics
%put data set is &dsn;
```

Since the semicolon is missing from the %LET statement, the macro processor will see the %PUT as part of the value to be assigned to &DSN, but the %PUT must be a keyword, not part of a statement, and the LOG will show:

|                                                                                                                 |
|-----------------------------------------------------------------------------------------------------------------|
| <pre>211 %let dsn = clinics 212 %put data set is &amp;dsn; ERROR: Open code statement recursion detected.</pre> |
|-----------------------------------------------------------------------------------------------------------------|

This error is NOT being caused because the &DSN appears on both sides of the equal sign, but because the %PUT is out of place. There is nothing wrong with statements like

```
%let dsn = &dsn clinics;
```

In fact this is a simple example of an acceptable recursion. Other examples include macros that call themselves. The following macro, %FACT, calculates a factorial by calling itself recursively.

**%FACT**

The macro %FACT will calculate and return the factorial of the argument. The factorial for 4 is defined as  $4 \times 3 \times 2 \times 1$  which is equal to 24. The statement

```
put "The factorial of 4 is %fact(4)";
```

becomes

```
put "The factorial of 4 is 24";
```

Although the macro itself is fairly simple, there are other (better?) ways to calculate a factorial (see Section 7.6.1). This macro can, however, be used to help explain what happens when a macro calls itself.

```

%macro fact(n);❶
 %if &n > 1 %then %eval(&n * %fact(%eval(&n-1)));❷
 %else 1;❸
%mend fact;

```

❶ The parameter (&N) is the number whose factorial is to be calculated.

❷ The %EVAL function is used to multiply the current value to the factorial of one less than the current number. This macro takes advantage of the fact that  $4! = 4 \times 3!$ , which in turn is  $4! = 4 \times 3! = 4 \times 3 \times 2!$ , and so on. The “and so on” is the recursion.

❸ Always be very careful when a macro calls itself. Unless you have an escape set up, you can easily generate an infinite loop.

For %FACT(4) the %EVAL ❷ becomes:

```
%eval(4 * %fact(3))
```

which becomes

```
%eval(4 * %eval(3 * %fact(2)))
```

which becomes

```
%eval(4 * %eval(3 * %eval(2 * %fact(1))))
```

which becomes

```
%eval(4 * %eval(3 * %eval(2 * 1)))
```

which becomes

```
%eval(4 * %eval(3 * 2))
```

which becomes

```
%eval(4 * 6)
```

which becomes

```
24
```

Now that we have established that a macro **can** call itself, we should also ask **should** a macro call itself? The nesting of macros can cause problems as the number of nested calls becomes large. “Large” of course varies by a number of factors including the operating system and the version of SAS. Just be careful.

## SEE ALSO

Ward (2001) discusses the use of recursion within the macro language as well as within SQL. A type of recursion is demonstrated by Rhoades (2001) when he re-enters and re-reads data multiple times. Adams (2003) discusses and uses recursion in a macro that builds a list of files including those in subdirectories. Benjamin (1999) demonstrates a macro that simulates recursion using the use of arrays.

## 13.6 Determining Macro Variable Scopes

The primary factor in determining the scope of a macro variable is whether it is being defined in open code or from within a macro. Secondly, the decision tree varies according to the method by which the variable is created.

Macro variables created in open code, regardless of the method of creation will be stored in the global symbol table. This should be fairly obvious since in open code there are no local symbol tables, or scopes, defined. When macro variables are defined from within a macro they will generally be placed on the symbol table local to that macro. There are exceptions, however, and these exceptions can be somewhat arcane. The decision trees discussed in the sections below assume that the macro variable is being defined within a macro.

The current scope of all macro variables can be determined by using the `%PUT _USER_;` statement and it is also stored in the view `SASHELP.VMACRO`.

### MORE INFORMATION

Section 1.4 briefly introduces the concepts of referencing environments, and Section 5.4.2 shows, through the use of the `%LOCAL` and `%GLOBAL` statements, how to override and control the default behaviors discussed in the sections below.

### SEE ALSO

The *SAS Macro Language Reference Manual Version 8* (pp. 42–43) and the SAS OnlineDoc both do a good job of diagramming the decision trees discussed in these sections. These decision trees can also be found in Burlew (1998, Chapter 5).

### Nested or layered symbol tables

It is not at all unusual to have macros that call macros, and since there is likely to be a symbol table associated with each of these macros, the symbol tables are essentially nested or layered. The symbol table associated with the calling macro is said to be the “next higher” symbol table or scope.

When a macro variable is created it must be assigned to a symbol table, and when nested symbol tables exist the process of deciding which symbol table to use often includes a check of the next higher symbol table. SAS automatically keeps track of the number of levels and of the macro variables associated with each symbol table. You can determine which variables are in which symbol tables by browsing `SASHELP.VMACRO`.

### Macro parameters

Macro variables (positional or named) created through the use of macro statement parameters will have a scope that is local to that macro. This will be true even if the macro variable already exists in a higher symbol table including the global symbol table.

## %LET and %DO

Macro variables created with both the %LET and %DO statements follow the same rules. These rules are applied in the following order:

1. For each assignment of a macro variable value, the local table is checked first to see if the variable already exists on the local table. If it does exist the assignment is made.
2. If the variable is not already on the local table, each higher table is checked successively up to and including the global table. If the variable is found to exist on any of these tables, the assignment is made to the first table (the lowest) that already contains the variable.
3. If variable is not found on any higher table the assignment is made to the local table.

## The CALL SYMPUT DATA step routine

The rules for the CALL SYMPUT DATA step routine are not as easily anticipated as are those for the %LET statement described above. In part this is because the routine is **not** a macro language element. Usually SYMPUT writes the macro variable to the local symbol table (whether it already exists on the table or not) and does not check higher tables as does the %LET statement.

A special case rule comes into play if the local table is completely empty (no macro variables have been assigned to the local table). When this happens, SYMPUT writes to the first higher table that is not empty. Since this rule is a rare exception to the usual case, it is probably only fitting that it too should have an even rarer exception.

This last, rarely invoked, rule is that SYMPUT will write to the local table, even if it is empty, when either of two conditions apply:

1. The macro containing the DATA step creates &SYSPBUFF when the macro is called.
2. The macro contains a computed %GOTO statement.

In the following example the CALL SYMPUT seems to violate these rules. The DATA step is used to create a macro variable CNTMALES which contains the number of male patients.

```
%macro cntmales;
 %local cntmales; ❶
 data malesonly;
 set sasclass.clinics end=eof;
 if sex='M' then cnt+1;
 if eof then call symput('cntmales',left(put(cnt,3)));
 %mend cntmales;

 %cntmales
 title "Number of Males is &cntmales"; ❷
 proc print data=malesonly;
 run;
```

Since &CNTMALES is declared as local ❶, it should come as no surprise that the use of &CNTMALES in the TITLE statement ❷ produces an uninitialized macro variable message. However, when the TITLE statement is moved inside the PROC step ❸, the &CNTMALES is no longer uninitialized and is therefore obviously **not** local to the macro %CNTMALES.

```

&cntmales
proc print data=malesonly;
 title "Number of Males is &cntmales"; ❸
run;

```

What has happened? Moving the TITLE statement won't make a difference as to which symbol table contains &CNTMALES. In fact, the local version of &CNTMALES was never created. Since the DATA step does not have a RUN statement, it is not terminated and therefore not executed until the PROC statement is encountered. In this case the DATA step is not executed until after %CNTMALES has completed. Since the PROC statement is in open code the SYMPUT writes to the GLOBAL table (the symbol table for %CNTMALES no longer even exists).

Placing the TITLE statement before the PROC statement ❷ causes an error because as a global statement it does not trigger the execution of the DATA step. Consequently the TITLE statement is executed **before** the DATA step. Like a RUN statement, the PROC statement does signify that the DATA step has been fully specified, and the DATA step is executed when the PROC statement is encountered. With the TITLE statement inside the PROC step ❸, the macro variable &CNTMALES is now available.

### The INTO: operator

Like the CALL SYMPUT routine the INTO: operator is not a macro language element. However, its rules for the assignment of macro variable values are most similar to those of the %LET statement. In part this is due to the fact that, unlike DATA steps that contain a CALL SYMPUT, the local table is not going to be empty because the SQL step itself will automatically generate a minimum set of local macro variables (&SQLOBS, &SQLOOPS, &SQLXOBS, and &SQLRC).

---

## 13.7 Chapter Summary

The complexity and flexibility of the macro language make it difficult to categorize many of its capabilities. A number of considerations when using the macro language have less to do with the language itself and more to do with how it is used.

Specialized tasks include the use of macro variables that require three adjacent ampersands. These are used when the macro variable is needed to store the name of another macro variable.

There are a number of efficiency issues to take into consideration when working with the macro language. Several system options are available that can be used to tailor the resources that are to be used by your application and some of these options can also assist with the debugging of your macros. The techniques that you use to physically code your macro can also make it easier to debug, as well as more efficient when it executes.

Like macro parameters, macro buffers can be used to pass values into a macro. Through the use of &SYSPBUFF it is possible to create macros with a variable number of parameters.

Since macro variables are associated with specific scopes and since a given macro variable name can be used in multiple scopes, it is important to be able to determine into which scope or symbol table a macro variable will be stored. Writing a macro variable to the wrong symbol table can cause macro variable collisions, and can result in unanticipated and even incorrect results.

# Index

## A

- %ABORT statement 95
- aborting macro execution 95–96
- %ADDYEARS function (example) 190
- \_all\_ option, %PUT statement 17
- ampersand (&)
  - multiple ampersands, efficiency of 369–370
  - quoting 145–159
  - resolving && into 21
  - resolving subscripts for macro arrays 358
  - triple-ampersand variables 348
- appending macro variables to SAS code 19
- appending unknown data sets 246–251
- arithmetic evaluations
  - floating point evaluation 168–170
  - implied arithmetic operations 378
  - integer evaluations 166–168
  - totals based on macro variable lists 349–351
- ARRAY statement, macro equivalent to 112
- array subscripts, loop index as 117
- arrays, macro
  - See macro arrays
- assigning values 119–127
  - assignment statements 119
  - in SQL steps 128
  - macro variables in SCL programs 202
  - quoting 145–159
  - RESOLVE function 122–127
  - RETAIN statement 119
  - series of macro variables 129–132
  - SYMGET function 120–121
- assignment statements 119
- asterisk-style comments 383–386
  - See also comments in macro code
- attitude, characters with 145
  - See also quoting functions
- attributes, macro functions 179
- ATTRN function 320
- autocall facility 331
  - efficiency of 371
  - interactive macro development 334
  - locations for autocall libraries 41, 336, 371
  - macro search order 44
  - permanently storing macros in 344
  - predefined macros in 335–345
  - structure and strategy 333
  - system options for 41–42, 371
- AUTOEXEC initialization option 49
- AUTOEXEC.SAS program 47–49

- automatic macro variables 5, 22–27
  - accessing outside DATA step 172–178
  - generated with SQL procedure 133
  - SCL programs and 203
- \_automatic\_ option, %PUT statement 17
- automatic system initialization 47–49

## B

- batch file, for executing series of SAS programs 252–254
- binary number systems, converting 190
- blanks in character strings, removing 339, 342
- %BLDDSN macro (example) 232
- %BLDLIVE macro (example) 224
- blind quotes
  - See %BQUOTE function
- BOOLEAN0 conversion type 169
- %BQUOTE function 146–149
  - %NRBQUOTE function vs. 155
  - when quoting is executed 158
- branching program flow
  - See program flow
- bridging functions 170–178
- buffers for macro parameters 389–393
- %BUILDVARLIST macro (example) 299–303

## C

- calling macros
  - See invoking (executing) macros
- capitalization of code, style for 372
- capitalization of strings, converting 165, 195, 340–342
- catalogs, copying members of 240–241
- CATALOGS table 132
- %CATCOPY macro (example) 240–241
- CEIL conversion type 169
- character strings
  - converting case of 165, 195, 340–342
  - converting current date to 185
  - converting macro variable values to 120–121
  - determining if macro variables contain numbers 343
  - left-justifying macro values 338
  - length of, determining 161
  - placing commas between words 308–311
  - quoting functions 145–159
  - removing blanks 339, 342
  - repeating 194

- retrieving last word in list 188
- searching for characters within 160, 337
- searching for words 162–164, 188, 193–194
- text functions 160–165
- unequal length, comparing 196
- charts, controlling
  - See* Output Delivery System (ODS)
- %CHECKIT macro (example) 388
- %CHKCOPY macro (example) 245
- %CHKDSN macro (example) 231
- %CHKDUP macro (example) 222–224
- classification variables, list of 316–319
- clock values, storing in macro variables 244
- %CMBNSUBJ macro (example) 246–251
- CMD option, %MACRO statement 47
- CMDMAC system option 38, 46, 371
- %CMPRES macro 339
- code
  - See* dynamic programs
  - See* SAS programs
- coding tips 368–389
  - style 371–373
- collisions, macro variable 382–383
- %COLONCMR function (example) 196
- column indicators for macro arrays 358–361
- column-specified flat files, creating 280–284
- COLUMNS table 132
- combining titles in output 254–260
- comma-delimited files, creating 284–286
  - placing commas between words 308–311
  - separating and quoting words 311–313
- command-line macros 45
- command-style macros 46, 369
- commands, issuing to operating environment (%SYSEXEC) 26, 93
- commas, placing between words 308–311
- comments in macro code 86–88
  - asterisk-style comments 70–71, 383–386
  - compiling with macros 70–71
  - documenting macro parameters 58
  - logging with MPRINT system option 39, 86, 370, 374
- comments in SAS code, creating with macros 36–37
- comparing values
  - >= operator 84
  - floating point evaluations 168–170
  - integer evaluations 166–168
  - noninteger comparisons 388
  - range comparisons 388
  - strings of unequal length 196
- compile-time macros 205–207
- compiling macros
  - noting complications in LOG 40, 370
  - quoting when 156
  - SCL programs and 199–201
  - storing comments as part of 70–71
  - storing macros permanently in autocall library 344
- composite macro calls 389
- %COMPSTR macro 344
- conditional macro execution 375
- conditional macro statements 72–77
  - dynamic programs 213
  - %RETURN to terminate macro execution 96
- conditional SAS code
  - commenting for clarity 86–88
  - %DO statements for 79–82
  - %IF-%THEN/%ELSE statements 74
- consistency in coding 372
- constant text 34
- CONTENTS procedure
  - %CMBNSUBJ macro 247–248
  - flat files, creating 280–286
- control data sets and files 107–115, 218–220
  - assigning macro variable names 113–115
  - assigning macro variable values 107–113
  - building macro variables 220–221
  - SASHELP views as 236–237
  - sources of control information 118
- controlled repeatability in dynamic programs 217
- controlling SAS program execution
  - See* EXECUTE routine
- converting
  - case of character strings 165, 195, 340–342
  - date and time information 172
  - floating point values to integers 169
  - macro variable values to character strings 120–121
  - number systems 190
- coordinating lists of macro variables 224–227
- %COPY statement 331
- copying catalog members 240–241
- %COUNTER macro (example) 324–326
- counting
  - observations in data sets 28, 319–326
  - words in macro variables 306–308
- %CSTR macro (example) 308
- %CSTR2 macro (example) 309
- %CSTR3 macro (example) 310
- %CURRDATE function (example) 184
- custom macro functions
  - building 178–182
  - DATA step functions 190–191
  - examples of 183–188
  - logical evaluations with 191–197
- customized formats 265–268



**D**

- data dependencies, hard-coded 216–221
- data dictionaries
  - See* control data sets and files
- DATA\_NULL\_step 228–230, 261
  - counting observations in data sets 323
- data set variables 296–319
  - creating lists of names, from PDV 296–303
  - creating lists of names, with SQL procedure 304
  - creating macro variables from 304–306
  - existence, checking for 313
  - removing repeated variables 314–316
- data sets 279–326
  - appending unknown data sets 246–251
  - as control files 107–115, 218–221, 236–237
  - building formats from 267–268
  - classification variables, list of 316–319
  - commas between words, adding 308–311
  - counting observations in 28, 319–326
  - counting words in macro variables 306–308
  - creating lists of variable names 296–304
  - creating macro variables from variable names 304–306
  - existence of, checking 178–179, 183, 295
  - existence of data set variables, checking 313
  - flat files as, creating 280–286
  - hard-coded data dependencies, avoiding 216–221
  - macro variables for most recent data set 23
  - outliers, finding 294–295
  - quoting words in lists 311–313
  - repeated words, removing from list 314–316
  - stepping through list of 222
  - subsetting 286–295
  - write access, checking for 245
- DATA step
  - assigning variable values to macro variables 102–106, 138, 396
  - assignment statements 119
  - building dynamically 215, 228–230
  - calling functions outside DATA step 172–178
  - calling routines outside, with %SYSCALL 171
  - difference from macro language 375–380
  - order of execution 5–6
  - return code for 24
- DATA step functions and routines
  - custom (examples) 190–191
  - mixing with macro functions 376
  - outside DATA step 170–178
- DATA step variables
  - assigning values to macro variables 102–106, 138, 396
  - assignment statements 119
  - evaluating with %IF 375–376
  - moving macro variable values to 119–127
- %DATATYP macro 343
- date and time information
  - automatic macro variables for 22–23
  - converting 172
  - converting to string for SQL pass-through 185
  - including in titles 172, 184
  - incrementing week-ending date 191
  - incrementing year values 190
  - storing system clock values 244
  - suspending SAS programs 186–187
  - timing issues with CALL EXECUTE 138–141
- day, macro variable for 22
- %DB2DATE function (example) 185
- DBDIR data set (example) 219
- debugging macros 373–374
  - See also* errors and troubleshooting
  - MERROR and SERROR system options 39
  - SAS system options for 39–41, 374
- decimal systems, converting 190
- defining macro parameters
  - keyword (named) parameters 57
  - positional parameters 54–55
- defining macro variables 13
- defining macros 34–50
  - nesting macro definitions 71, 369
  - without % token (statement- and command-style) 47
- %DELIM macro (example) 284–286
- DESCRIBE statement, SQL procedure 133
- DICTIONARY tables, SQL procedure 132
- DICTIONARY.TITLES table 261
- directories, creating 251
- disabling macro facility 39
- display manager, command-line macros for 45
- %DISPLAY statement 94–95
- displaying macro variables 16–19
- %DISTINCTLIST macro (example) 314–316
- %DO statements 77–85
  - coordinating lists of macro variables 224
  - debugging with MLOGIC and MLOGICNEST 39–40, 370, 374
  - %DO blocks 78–80
  - %DO loops, iterative 80–83
  - %DO %UNTIL loops 83–84
  - %DO %WHILE loops 84
  - dynamic programs 213–214
  - loop index as array subscript 117
  - macro variable scope 396
- documentation of program code 372
- documenting macros 58

- dot (.)
  - appending to macro variables 20
  - terminating macro variable names 14, 20
- doubly subscripted macro arrays 357–368
- drill-down graphs and charts 275–276
- %DUMPIT macro (example) 242–243
- duplicate KEY values, checking for 222–224
- dynamic programs 116–119, 211–237
  - building SAS statements 214–215, 227–234
  - commenting for clarity 86–88
  - composite macro calls 389
  - %DO statements for 79–82
  - elements of 212–215
  - hard-coded data dependencies, avoiding 216–221
  - %IF-%THEN/%ELSE statement for 74
  - macro arrays (&&VAR&I reference) 118, 204, 221–227, 233–234, 351–352, 357–368
  - SASHELP views 234–237
  - sources of control information 118

## E

- efficient programming tips 368–371
- %ELSE statements
  - See* %IF-%THEN/%ELSE statements
- enabling macro facility 39
- %END statement 77
- environment control 239–277
  - adapting SAS environment 262–269
  - operating system operations 240–255
  - output control 255–262
  - Output Delivery System (ODS) 269–276
- environment variables
  - accessing 255
  - maintaining 262–265
- %EOW function (example) 191
- errors and troubleshooting
  - See also* debugging macros
  - See also* syntax errors
  - hanging semicolon problem 75–76
  - macro variable collisions 382–383
  - macro variables in wrong symbol table 381
  - mismatched symbols 159
  - return codes 24
  - syntax errors 36, 386–387
  - undefined macro parameters 55, 57
- %EVAL function 166–168
- evaluation functions 166–170
- EXECUTE routine 134–141
  - building SAS statements dynamically 231–233
  - timing issues 138–141

- executing
  - operating system commands (%SYSEXEC) 26, 93
  - routines outside DATA step 172–178
  - SAS programs, controlling with macro calls 70–71
  - series of SAS programs 252–254
- executing macros
  - See* invoking (executing) macros
- execution phases (macros) 5–7
  - calling macros from within SCL programs 205–207
  - when quoting is executed 158
- %EXIST macro (example) 178–179, 183, 295
- existence, checking for
  - data set variables 313
  - data sets 178–179, 183, 295
  - directories 251
  - macro variables 193–194, 352–354
- explicit use of %EVAL function 166
- expressions, macro
  - defined 5
  - including in macro definitions 35
- EXTFILES table 132

## F

- %FACT macro (example) 183–184, 393–394
- factorials, computing 183–184, 393–394
  - calculating permutations with 192
- file location, ODS 270–271
- FILEEXIST function 270
- FILENAME statement 250
- finding strings
  - See* searching character strings
- %FINDOUTLIERS macro (example) 294–295
- fixed columns, creating flat files with 280–284
- flat files
  - See also* data sets
  - creating 280–286
  - writing first lines of 242–243
- FLDDIR data set (example) 219
- floating point evaluation 168–170
- FLOOR conversion type 169
- FMTSEARCH system option 265–266
- %FMSTRCH macro (example) 265–267
- footnotes, coordinating 261
- FORMAT procedure 265–268
- formats, customized 265–268
- functions
  - See* macro functions
- %FUZZRNGE function (example) 186

**G**

generalized repeatability in dynamic programs 217  
 %GETKEYS macro (example) 351–352  
 %GETVARS macro (example) 298  
 global macro variables 8, 395–397  
   creating 88–91  
   macro variables in wrong symbol table 381  
   removing 30, 355–357  
 \_global\_ option, %PUT statement 17  
 %GLOBAL statement 88–91  
 global statements, order of execution 5–6  
 %GOTO statement 91–93  
 graphics devices 271–272  
 graphs, controlling  
   *See* Output Delivery System (ODS)  
 GRSEG catalog entries, WEBFRAME device with  
   272–274

**H**

halting SAS programs 186–187  
 hanging semicolon problem 75–76  
 hard-coded data dependencies, avoiding 216–221  
 hard-coded programs 212  
 hexadecimal number systems, converting 190  
 %HOLDOPT macro (example) 264–265  
 host operating environment  
   issuing commands to (%SYSEXEC) 26, 93  
   obtaining name for 27  
 host system variables  
   accessing 255  
   maintaining 262–265  
 HTML files 271

**I**

%IF-%THEN/%ELSE statements 72–77  
   debugging with MLOGIC and MLOGICNEST  
     39–40, 370, 374  
   dynamic programs 213  
   evaluating DATA step variables 375–376  
   IF statement vs. 375  
   IN (#) operator 76–77, 391–392  
   %RETURN to terminate macro execution 96  
   semicolon management 74–76, 372  
 implicit use of %EVAL function 167  
 implied arithmetic operations 378  
 IMPLMAC system option 39, 46, 371  
 %IN macro (example) 391–392  
 IN operator 76–77, 391–392  
 %INCLUDE function 228–230  
   as macro library 329, 333  
   interactive macro development 334  
   %MACRO statement vs. 369

indentation style 372  
 %INDEX function 160  
 indexes, building with PRINT procedure 274  
 INDEXES table 132  
 %INDEXW function (example) 188  
 indicator variables, assigning to classifications  
   316–319  
 indicators for macro arrays  
   naming 358–361  
   resolving 358  
 indirect macro variables references 118  
 %INDVAR macro (example) 316–319  
 information sources for dynamic programs  
   216–217  
 initializing system 47–49  
 inserting commas between words 308–311  
 INTEGER conversion type 169  
 integer evaluations 166–168  
   computing factorials 183–184, 393–394  
   noninteger comparisons (accidental) 388  
 interactive macro development 334  
 INTO: operator 397  
 INTO operator, SELECT statement 127  
 invisible quote marks, viewing 153–154  
 invoking (executing) macros 37–38  
   composite macro calls 389  
   conditionally, in DATA step 375  
   controlling SAS program execution 70  
   debugging unresolved macro calls 374  
   EXECUTE routine 134–141  
   execution phases 5–7, 158, 205–207  
   from other macros (nested) 40, 59, 66–72  
   from within SCL programs 199–201, 205–207  
   order of execution 5–6  
   quoting when 156  
   recursively 393–395  
   terminating macro execution 95–96  
   with % (named-style) 37–38, 46, 369  
   without % (statement- and command-style)  
     46, 369  
 invoking routines outside DATA step 172–178  
 issuing commands to operating environment  
   (%SYSEXEC) 26, 93  
 iterative execution (loops) 77–85  
   commenting for clarity 86–88  
   dynamic programs 213–214

**K**

KEY values, checking for duplicate 222–224  
 key variables, selecting from macro arrays  
   351–352  
 %KEYLIST macro (example) 305  
 keyword (named) parameters 56–59, 372  
   positional parameters and 59–61

**L**

- `%label` statement 91–93
- labels for macros 91–93
- layered symbols tables 395
- leading blanks in character strings, removing 339
- left-justifying macro values 338
- `%LEFT` macro 338
- `%LENGTH` function 161
- length of character strings, determining 161
- `%LET` statement 13
  - macro parameters instead of 54
  - macro variable scope 396
- `LIB=` option, `%COPY` statement 331
- libraries 268–269
  - See also* macro libraries
- `LINESIZE` option 256
- listing values within macro variables 28–29
- listings, controlling observations in 321–323
  - See also* output control
- `%LISTLAST` function (example) 188
- local macro variables 8, 395–397
- `_local_` option, `%PUT` statement 17
- local scope
  - creating macro variables 88–91
  - custom macro functions 180
  - macro variables in wrong symbol table 381
- `%LOCAL` statement 88–91, 373
- `LOG` file, debugging 39, 370
- logical evaluations
  - example macro functions 191–197
  - floating point evaluation 168–170
  - `%FUZZRNGE` function (example) 186
  - integer evaluations 167–168
- logical program flow
  - See* program flow
- `%LOOK` macro (example) 321–323
- loops 77–85
  - commenting for clarity 86–88
  - dynamic programs 213–214
- `%LOWCASE` macro 340–342
- lowercase characters, converting to 340–342

**M**

- macro arrays 118, 221–227
  - `&&&VAR&I` variable form 361–363
  - coordinating lists of macro variables 224–227
  - doubly subscripted 357–368
  - in SCL programs 204
  - macro lists instead of 233–234
  - observations with duplicate KEY values 222–224
  - resolving 222
  - resolving subscripts for 358

- row indicators for 358–361
- selecting elements of 351–352
- macro calls, order of execution 5–6
- macro definitions
  - including macro expressions in 35
  - order of execution 5–6
- macro execution
  - See* invoking (executing) macros
- macro expressions
  - defined 5
  - including in macro definitions 35
- macro facility 4
  - enabling/disabling 39
- macro functions 5, 143–198
  - bridging functions 170–178
  - calling DATA step functions outside DATA step 172–178
  - evaluation functions 166–170
  - including in macro definitions 35
  - interesting examples of 183–197
  - mixing with DATA functions 376
  - quoting functions 145–159
  - text functions 160–165
- macro functions, custom
  - building 178–182
  - examples of 183–188
  - for DATA step 190–191
  - logical evaluations with 191–197
- macro labels 91–93
- macro language 3
  - differences from DATA step 375–380
  - programming tips 368–389
  - recursion 393–395
  - terminology 4
  - when to use 368
- macro libraries 327–345
  - DATA step functions for 268–269
  - essential information 332–335
  - strategy for 333
  - structure 333
- macro lists 233–234
- macro names 34
  - invoking macros without % token 39, 46, 369
  - obtaining 27
- macro parameters 54–61
  - buffers for 389–393
  - determining macro variable scope 395
  - documenting 58
  - keyword (named) parameters 56–61
  - passing between macros 66–69
  - passing between macros, quoting when 151–153
  - passing between SCL entries 204
  - positional parameters 54–56, 59–61

- macro program statements 4
  - conditional macro statements 72–77
  - including in macro definitions 34–35
  - iterative execution (loops) 77–85, 213–214
  - list of 85–96
  - order of execution 5–6
- macro references 4
  - compiling SCL programs and 199–201
  - indirect 118
  - order of execution 5
  - scope of (referencing environment) 8, 88–91, 395–397
  - warnings for unresolved references 39
- macro references, resolving 5
  - &&VAR&I macro reference 222
  - debugging unresolved macro calls 374
  - execution control with CALL EXECUTE 134–141
  - quoting when passing parameters 151–153
  - SCL programs and 199–208
  - warnings for unresolved references 39
- %MACRO statement 34–37
  - CMD option 47
  - %INCLUDE function vs. 369
  - nesting macro definitions 71, 369
  - /PARMUFF option 390–393
  - STMT option 47
- macro symbol tables
  - creating macro variables 88–91
  - macro variables in wrong symbol table 381
  - maximum size of 44
  - scope and nesting 8, 395
- MACRO system option 39, 370
- macro system options, list of 370–371
- macro text functions 160–165
- macro variable collisions 382–383
- macro variable references
  - See* macro references
- macro variables 4, 11–32
  - See also* automatic macro variables
  - &&& (triple-ampersand) variables 348
  - as prefixes 20
  - as suffixes 19
  - assigning values in SQL steps 127–128
  - assigning variable values to 102–106, 138, 396
  - building from control files 107–113, 220–221
  - collisions 382–383
  - converting values to character strings 120–121
  - counting words in 306–308
  - creating 88–91, 379–380
  - creating multiple variables with SELECT statements 128–129
  - data set variables and 296–319
  - DATA step functions on 172–178
  - displaying contents of 16–19
  - dynamic programs, elements of 116–119
  - executing routines outside DATA step 172–178
  - existence, checking for 193–194, 352–354
  - hard-coded data dependencies, avoiding 216–221
  - in PROC SQL step 27–29
  - in wrong symbol table 381
  - including in macro definitions 34
  - lists of, coordinating 224–227
  - lists of, totals based on 349–351
  - maximum size of 44
  - moving values to DATA step variables 119–127
  - naming and defining 12–13, 113–115, 220–221
  - nested 8
  - numeric, determining if 343
  - parameters for 54–61, 66–69, 151–153, 204, 389–393, 395
  - placing values into series of 129–132
  - preceding SAS code with 20
  - quoting values 145–159
  - removing 30, 355–357
  - reserved names 12
  - SCL programs and 199–208
  - scope of (referencing environment) 8, 88–91, 395–397
  - storing clock values 244
- macro variables, resolving 19–22
  - custom macro functions 180–182
  - debugging unresolved macro calls 374
  - noting resolutions in LOG 39, 370
  - quoting when passing parameters 151–153
  - SCL programs and 199–208
  - warnings for unresolved references 39
- macro windows, displaying 94–95
- macros
  - See also* compiling macros
  - See also* invoking (executing) macros
  - See also* stored compiled macros
  - command-line macros 45
  - debugging 39–41, 373–374
  - defined 4
  - documenting 58
  - interactive development of 334
  - named-style 37–38, 46, 369
  - nested 40, 59, 66–72, 370
  - predefined autocall macros 335–345
  - run-time 205
  - SAS system options for 37–38
  - search order 44

macros, creating/defining 34–50  
   nesting macro definitions 71, 369  
   to comment code 36–37  
   without % token (statement- and command-style) 47  
 MACROS table 132  
 %MAKECSV macro (example) 292–294  
 %MAKEDIR macro (example) 251  
 %MAKERUNBAT macro (example) 252–254  
 %MATRIXPRINT macro (example) 362  
 MAUTOSOURCE system option 41, 332, 371  
 MCOMPILENOTE system option 40  
 MEMBERS table 132  
 memory control options 44  
   *See also* performance  
 %MEND statement 34–37, 384  
   nesting macro definitions 71, 369  
 MERROR system option 39  
 message boxes, creating with %WINDOW 94–95  
 metadata (control data sets)  
   *See* control data sets and files  
 MFILE system option 40  
 mismatched symbols 159  
 missing values 14, 377  
 mixed-case strings  
   converting strings to 195  
   converting to lowercase 340–342  
 %MIXEDCASE function (example) 195  
 %MKFMT macro (example) 267–268  
 %MKLIB macro (example) 268–269  
 MLOGIC system option 39, 370, 374  
   viewing invisible quote marks 153–154  
 MLOGICNEST system option 40  
 MPRINT system option 39, 86, 370, 374  
 MPRINTNEST system option 40  
 MRECALL system option 41, 371  
 MSTORED system option 43, 371  
 MSYNTABMAX system option 44  
 multiple blanks in character strings, removing 339, 342  
 multiply subscripted macro arrays 357–368  
 MVARSIZE system option 44

## N

name of host operating environment, obtaining 27  
 named parameters  
   *See* keyword (named) parameters  
 named-style macros 37–38, 46, 369  
 names, data set variables  
   creating lists of, from PDV 296–303  
   creating lists of, with SQL procedure 304  
   creating macro variables from 304–306  
 names, macro variables 12  
   assigning with control data sets 113–115

  avoiding hard-coded 217  
   collisions between macro variables 382–383  
   DICTIONARY tables to create 132  
 names, macros 34  
   invoking macros with % token 37–38, 46  
   invoking macros without % token 39, 46, 369  
   obtaining 27  
 naming  
   conventions for 372  
   macro array indicators 358–361  
   ODS output, standardizing 270  
   recently used data sets 23  
 nested macro definitions 71, 369  
 nested macros  
   debugging 40, 370  
   invoking 66–72  
   macro parameters 59  
 nested variables 8, 395  
 no-rescan (no-resolve) functions 155  
 NOBS= option  
   counting observations in data sets 323  
   subsetting data set with 288  
 noninteger comparisons (accidental) 388  
 NR functions 155  
 %NRBQUOTE function 146–147, 155, 158  
 %NRQUOTE function 156, 158  
 %NRSTR function 147, 155, 158  
 null values 14, 377  
   undefined macro parameters 55, 57  
 numbers for output pages, renumbering 260  
 numeric operations  
   *See also* date and time information  
   converting number systems 190  
   determining if macro variables contain numbers 343  
   factorials, computing 183–184, 393–394  
   floating point evaluation 168–170  
   implied arithmetic operations 378  
   integer evaluations 166–168  
   noninteger comparisons 388  
   permutations, calculating 192  
   range comparisons 388  
   renumbering listing pages 260  
   rounding numbers 177  
 %NUMOBS macro (example) 323  
 %NW macro (example) 306

## O

%OBSCNT macro (example) 320  
 observations  
   *See also* data sets  
   counting in data sets 28, 319–326  
   finding duplicate KEY values 222–224  
   finding outliers 294–295

observations (*continued*)  
 random subsets of data sets 288–292  
 ODS  
   *See* Output Delivery System (ODS)  
 ODS files, physical location of 270–271  
 ODS HTML statement 271  
 open code, defined 5  
 operating environment  
   issuing commands to (%SYSEXEC) 26, 93  
   obtaining name for 27  
 operating system operations 240–255  
 # operator (%IF-%THEN /%ELSE statements)  
   76–77  
 operators, defined 5  
 OPTION settings, managing 262–265  
 OPTIONS statement  
   *See* system options  
 OPTIONS table 132  
 %ORLIST macro (example) 392–393  
 OUT= option, CONTENTS procedure 247–248  
 OUTFILE= option, %COPY statement 331  
 outliers, finding 294–295  
 output control 255–262  
   accessing system variables 255  
   combining titles 254–260  
   graphs, charts, reports (ODS) 269–276  
   renumbering listing pages 260  
 Output Delivery System (ODS) 269–276  
   controlling ODS locations 270–271  
   drill-down graphs and charts 275–276  
   graphics devices 271–272  
   GSREG catalog entries 272–274  
   HTML files, controlling 271  
   indexes, building with PRINT procedure 274  
   physical location of ODS files 270–271  
 outputting lines from flat files 242–243  
 overlapping variable names, avoiding 314–316

## P

page numbers, renumbering 260  
 pairs of mismatched symbols, quoting 159  
 parameters  
   *See* macro parameters  
 /PARMUFF option, %MACRO statement  
   390–393  
 PATH= option (ODS HTML statement) 271  
 PATHNAME function 174–175  
 PATTERN statements 176  
 PDV, creating list of variable names from  
   296–303  
 percent sign (%)  
   identifying macro statements 35  
   invoking macros with (named-style) 37–38,  
   46, 369

invoking macros without 39, 46, 369  
 quoting 145–159  
 percentage subset of data set, creating 287–288  
 performance  
   compiling comments with macros 70–71  
   invoking macros without % token 46, 369  
   macros for comments 37  
   memory control options 44  
   programming tips 368–371  
 period (.)  
   appending to macro variables 20  
   terminating macro variable names 14, 20  
 %PERM function (example) 192  
 permutations, calculating 192, 352–354  
 PIPE device type, in %CMBNSUBJ macro 250  
 POINT option, subsetting data set with 288  
 positional parameters 54–56  
   keyword (named) parameters and 59–61  
 predefined autocall macros 335–345  
 prefixes, macro variables as 20  
 PRINT listings, controlling observations in  
   321–323  
 PRINT procedure, building indexes with 274  
 PROC steps  
   order of execution 5–6  
   return code for 24  
 program flow 66–69  
   commenting for clarity 86–88  
   conditional macro statements 72–77  
   debugging with MLOGIC and MLOGICNEST  
   39–40, 370, 374  
   efficiency tips 368–371  
   %GOTO and %label statements 91–93  
   iterative execution (loops) 77–85, 213–214  
   logical branching 213  
   recursion 393–395  
   terminating macro execution 95–96  
 program statements, macros 4  
   conditional macro statements 72–77  
   including in macro definitions 34–35  
   iterative execution (loops) 77–85, 213–214  
   list of 85–96  
   order of execution 5–6  
 programming tips 368–389  
   style 371–373  
 programs, SAS  
   *See* SAS programs  
 %PUT statement 16–19, 374  
 PUTN function 176

## Q

%QSCAN function 162–164  
 %QSTR macro (example) 311–313  
 %QSUBSTR function 164

%QSYSFUNC function 172–178  
 %QUOTE function 146–147, 156  
   mismatched symbols 159  
   when quoting is executed 158  
 quoting 145, 370, 378  
   avoiding need for 150  
   items that require quoting 158  
   mismatched symbols 159  
   removing or changing quoting effects 156–157  
   types of quoting functions 155  
   viewing invisible quote marks 153–154  
   when passing values 151–153  
   words, in list 311–313  
 quoting functions 145–159  
   items that require quoting 158  
   removing or changing effects of 156–157  
   types of 155  
 %QUPCASE function 165

## R

random subsetting of data sets 288–292  
 %RAND\_W macro (example) 290–292  
 %RAND\_WO macro (example) 288–290  
 recursion 393–395  
 referencing environment (scope) 8  
   creating macro variables 88–91  
   determining 395–397  
 renumbering listing pages 260  
 %REPEAT function 194  
 repeatability in dynamic programs 217  
 repeated words, removing from lists 314–316  
 repeating character strings 194  
 replacement, random selection with or without 288–292  
 %REPORT macro (example) 260  
 reports, formatting  
   *See* output control  
 reserved names for macro variables 12  
 RESOLVE function 122–127  
 RETAIN statement 119  
 return codes 24, 26  
 %RETURN statement 96  
 %REVSAN function 193–194, 366  
 %RGBHEX function (example) 190  
 ROUND function 177  
 rounding numbers 177  
 row indicators for macro arrays 358–361  
 run-time macros 205

## S

SAS Component Language (SCL) 199–208  
 SAS environment, adapting 262–269

  customized formats 265–268  
   libraries 268–269  
   maintaining system options 262–265  
 SAS programs  
   *See also* comments in macro code  
   controlling execution with macro calls 70–71  
   executing series of 252–254  
   execution control with CALL EXECUTE 134–141  
   halting 186–187  
 SAS programs, dynamic  
   *See also* macro arrays  
   building SAS statements 214–215, 227–234  
   building with %DO 79–82  
   building with %IF-%THEN 74  
   commenting process of 86–88  
   elements of 212–215  
   hard-coded data dependencies, avoiding 216–221  
   SASHELP views 234–237  
 SAS statements, building dynamically 214–215, 227–234  
   CALL EXECUTE routine 227–234  
   DATA\_NULL\_step and %INCLUDE 228–230  
 %SAS2RAW macro (example) 280–284  
 SASAUTOS environment variable, modifying 335  
 SASAUTOS= system option 41, 336, 371  
 SASHELP views 234–237  
 SASMACR library 330  
   *See also* stored compiled macros  
   submitting macros to 334  
 SASMSTORE= system option 43, 330, 371  
   structure and strategy 333  
 %SCAN function 162–164, 233–234  
   identifying macro array elements 363–368  
   quoting in 378  
 SCL (SAS Component Language) 199–208  
 scope (referencing environment) 8  
   creating macro variables 88–91  
   determining 395–397  
 search order for macro libraries 332  
 search order for macros 44  
 searching character strings  
   determining word number from list of words 193–194  
   for character substrings 160  
   for characters outside a list 337  
   for substring segments 164–165  
   for words 162–164, 188  
 SELECT statement, creating multiple macro variables 128–129  
 semicolon management 74–76, 372  
 SERROR system option 39  
 session compiled macros 43, 371



- sessions, return code for 24, 26
- SET statement
  - building dynamically (example) 215
  - subsetting data sets with POINT option 288
- SGEN system option 39
- site number, obtaining 27
- size
  - macro variables 44
  - string length, determining 161
- %SLEEP function (example) 186–187
- %SMARTPERM macro (example) 352–354
- %SMARTPERM2 macro (example) 352–354
- SOURCE option, %COPY statement 331
- spaces in character strings, removing 339, 342
- special characters, requiring quoting 158
- SQL procedure 127–134, 373
  - assigning values to individual macro variables 127–128
  - automatic macro variables 133
  - converting date for SQL pass-through 185
  - counting observations in data sets 28, 324–326
  - creating lists of variable names 304
  - creating multiple macro variables 128–129
  - DICTIONARY tables 132
  - macro variables in 27–29
  - placing values into series of macro variables 129–132
  - subsetting data set by top percentage 287
- SQLOBS macro variable 133
- SQLOOPS macro variable 133
- SQLRC macro variable 133
- SQLXMSG macro variable 133
- SQLXOBS macro variable 133
- SQLXRC macro variable 133
- SRC option, %COPY statement 331
- statement-style macros 46, 369
- statements, building 214–215, 227–234
  - See* macro program statements
- static programs 212
  - avoiding hard-coded data dependencies 216–221
- STMT option, %MACRO statement 47
- /STORE option, %MACRO statement 331
- stored compiled macros 138–141, 330–331, 371
  - deleting from SASMACR catalog 334–335
  - macro search order 44
  - SAS system facility options 42–43
  - storing permanently in autocall library 344
- %STOREOPT macro (example) 262–263
- %STR function 146–147, 149
  - mismatched symbols 159
  - %NRSTR function vs. 155
  - when quoting is executed 158

- strings
  - See* character strings
- structured programming 373
- style, programming 371–373
- SUBMIT blocks, macro variables in 202–203
- submitting macros to SASMACR library 334
- subscripts, loop index as 117
- subscripts, multiple for macro arrays 357–368
- subsetting data sets 286–295
  - randomly 288–292
  - top percentages of data sets 287–288
  - WHERE clause, building dynamically 292–295
- %SUBSTR function 164
- substrings
  - See* character strings
- suffixes, macro variables as 19
- SUMMARY procedure, creating macro variables 131
- summing macro variable values 349–351
- %SUMS macro (example) 349–351
- %SUPERQ function 146–147, 157–158
- suspending SAS programs 186–187
- symbol tables
  - creating macro variables 88–91
  - macro variables in wrong symbol table 381
  - maximum size of 44
  - scope and nesting 8, 395
- SYMBOLGEN system option 370, 374
- symbolic variables
  - See* macro variables
- %SYMBOLSYNC macro (example) 364–366
- %SYMBOLSYNC2 macro (example) 366–368
- %SYMCHECK macro 193–194, 352–354
- %SYMDEL macro statement 30, 355–357
- SYMGET function 120–121, 138
- SYMPUT routine 102–106
  - CALL EXECUTE routine with 138
  - macro variable scope 396
- SYMPUTX routine 105–106
- syntax errors 36, 386–387
  - See also* errors and troubleshooting
  - hanging semicolon problem 75–76, 372
  - undefined macro parameters 55, 57
- %SYSCALL function 171
- &SYSCC macro variable 24
- &SYSDATE macro variable 22, 172
  - storing system clock values 244
- &SYSDATE9 macro variable 22, 172
- &SYSDAY macro variable 22
- &SYSDSN macro variable 23
- &SYSERR macro variable 24
- %SYSEVALF function 168–170
- %SYSEXEC macro statement 26, 93
- %SYSFUNC function 172–178, 320

- %SYSGET function (example) 255
- &SYSLAST macro variable 23
- &SYSMACRONAME macro variable 27
- &SYSPARM macro variable 25
- SYSPARM= system option 25
- %SYSPBUFF macro variable 390–393
- &SYSRC macro variable 26, 93
- &SYSSCP macro variable 27
- &SYSSCPL macro variable 27
- &SYSSITE macro variable 27
- system clock values, storing in macro variables 244
- system initialization 47–49
- system options for macros 38–45, 370–371
  - accessing with %SYSGET 255
  - debugging macros 374
  - maintaining 262–265
- system variables 255, 262–265
- &SYSTIME macro variable 22, 244
- &SYSUSERID macro variable 27

## T

- TABLES table 132
- terminating macro execution 95–96
- %TESTPRT macro (example) 321–323
- text
  - constant text 34
  - defined 4
  - maximum storage space in macro variables 12
  - passing character strings into SAS programs 25–26
- text functions 160–165
- %TF macro (example) 254–260
- %THEN statements
  - See* %IF-%THEN/%ELSE statements
- time information
  - See* date and time information
- timing issues with CALL EXECUTE 138–141
- titles
  - combining in output 254–260
  - coordinating multiple titles 261
  - specifying dates in 172, 184
- TITLES table 132, 261
- tokens 5, 384
- top percentage subset of data set, creating 287–288
- totals based on macro variable lists 349–351
- trailing blanks in character strings, removing 342
- TRANSPOSE procedure 296
- TRIM function 105
- %TRIM macro 342
- triple-ampersand variables 348
  - efficiency savings over &VAR 370

- troubleshooting
  - See* errors and troubleshooting

## U

- undefined macro parameters 55, 57
- unknown data sets, appending 246–251
- unmatched symbols 159
- %UNQUOTE function 147, 156–157
- unresolved macro calls, debugging 374
- %UNTIL statements 83–84
- %UPCASE function 165
- %UPDATE macro (example) 244
- %UPDATE2 macro (example) 245
- uppercase characters
  - converting strings to 165
  - converting to lowercase 340–342
- URL= option (ODS HTML statement) 271
- %USELIST macro (example) 233–234
- user ID, obtaining 27
- \_user\_ option, %PUT statement 17

## V

- value comparisons
  - floating point evaluations 168–170
  - integer evaluations 166–168
- values, assigning
  - See* assigning values
- values, comparing
  - See* comparing values
- values, left-justifying 338
- values, lists of
  - determining word number from list of words 193–194
  - placing commas between words 308–311
  - placing values into series of macro variables 129–132
  - retrieving last 188
  - totals based on macro variable lists 349–351
- &&&VAR macro reference 118, 348
  - &&&VAR&I variable form 361–363
  - efficiency savings over &VAR 370
- &&VAR&I macro reference 118
  - as macro arrays 221–227
  - doubly subscripted macro arrays 357–368
  - in SCL programs 204
- %VAREXIST macro (example) 313
- variable names, lists of
  - creating from PDV 296–303
  - creating macro variables from 304–306
  - creating with SQL procedure 304
- variables
  - See also* data set variables
  - See also* DATA step variables

Variables (*continued*)

- See also* macro variables
- class variables 316–319
- system variables 255, 262–265
- VARNAME function 299
- vectors, macro equivalent to 112
- %VERIFY macro 337
- verifying existence
  - See* existence, checking for
- views, SASHELP 234–237
- VIEWS table 132
- VOPTION table (SASHELP) 235, 256, 262–265
- VRDIR data set (example) 219
- VTABLE view 235, 246–247
- VTITLE table (SASHELP) 235, 261

**W**

- %WAKEUPAT function (example) 187
- WEBFRAME device 271–274
- weekday, macro variable for 22
- WHERE clause, building dynamically 292–295
- %WHILE statements 84
- %WINDOW statement 94–95
- windows, displaying with %WINDOW 94–95
- word scanner 5
- %WORDCOUNT macro (example) 307, 310
- WORDDATE function 172
- words
  - adding commas between 308–311
  - counting in macro variables 306–308
  - determining sequence number in lists 193–194
  - quoting 311–313
  - removing repeated words from lists 314–316
  - searching for 162–164, 188
- work.sasmacr catalog 43–44
- write access, checking for 245
- writing lines from flat files 242–243

**X**

- X command, last return code after 26
- X statement
  - in %CMBNSUBJ macro 249–250
  - last return code after 26

**Symbols**

- & (ampersand)
  - See* ampersand (&)
- >= comparison operator 84
- % (percent sign)
  - See* percent sign (%)
- %\* statement 86–88
- . (period)
  - appending to macro variables 20
  - terminating macro variable names 14, 20
- ; (semicolons), managing 74–76, 372
- /\*...\*/ comments
  - See* comments in macro code