SAS Base Programming Certification Study Guide

Written By: Paul J. Dinsmore

Date: Fall, 2007 ©

Table of Contents

Preface

The SAS code and output for all examples in this study guide can also be found on the associated electronic media.

Most of the examples follow a single theme of maintaining and calculating student grades. This was done intentionally so that the focus of the reader is on the syntax and structure of the SAS code and the available coding options, rather than deciphering the details of the examples. However, in some cases, the student grades example is not appropriate. In these cases, a more appropriate example is used. Hopefully, the details of these alternate examples are commonly understood and are not confusing in any way.

Although the same theme of grades is used throughout the text, most grade examples contain a unique name. This is for identification purposes particularly when using the SAS code on the associated electronic media. The general naming convention for the individual chapter examples is DATA = GRADES_Cn_m, where "n" represents the chapter number and "m" represents the example number within that chapter.
There also exists one generic grades dataset which is a SAS dataset called GRADES. This dataset is used across multiple chapters, contains 40 observations, and is referenced simply as GRADES. Several other example datasets exist but they are uniquely named and easily identified.

All SAS coding examples are shown in the default color scheme used by the SAS editor. The color of some individual lines may be changed to **bolded red** for emphasis. Any standalone examples of SAS statement lines are shown **bolded in dark blue**. Any SAS output resulting from the execution of the SAS coding examples is shown **bolded in dark red**.

Introduction

This study guide is intended for anyone interested in taking the SAS Base Programming Certification Test for SAS Version 9. It is written with the assumption that the reader has either taken a SAS programming course or is currently enrolled in one. It is not intended to be a course textbook.

Detailed information on the certification test as well as other SAS certification tests can be found at the SAS certification website, www.sas.com/certification (http://support.sas.com/certify/). At this website you will find specific information concerning all certification tests, locations for taking these tests, and links to schedule and register for any SAS certification test.

For the base certification test, you will need to know how to do the following:

- Import and export raw data files
- Manipulate and transform data
- Combine SAS data sets
- Create basic detail and summary reports using SAS procedures
- Identify and correct data, syntax and programming logic errors.
- Identify enhancements and new functionality in version 9.

The following chapters will focus on the details within the above topics that are required for passing the test. These detailed topics include the following:

Accessing Data

- Use FORMATTED, LIST and COLUMN input to read raw data files
- Use INFILE statement options to control processing when reading raw data files
- Use various components of an INPUT statement to process raw data files including column and line pointer controls, and trailing @ controls
- Combine SAS data sets using the DATA step

Creating Data Structures

- Create temporary and permanent SAS data sets
- Create and manipulate SAS date values
- Use DATA Step statements to export data to standard and comma delimited raw data files
- Control which observations and variables in a SAS data set are processed and output

Managing Data

- Investigate SAS data libraries using base SAS utility procedures
- Sort observations in a SAS data set

- Conditionally execute SAS statements
- Use assignment statements in the DATA step
- Modify variable attributes using options and statements in the DATA step
- Accumulate sub-totals and totals using DATA step statements
- Use SAS functions to manipulate character data, numeric data, and SAS date values
- Use SAS functions to convert character data to numeric and vice versa
- Process data using DO LOOPS
- Process data using SAS arrays

Generating Reports

- Generate list reports using the PRINT and REPORT procedures
- Generate summary reports and frequency tables using base SAS procedures
- Enhance reports through the use of labels, SAS formats, user-defined formats, titles, footnotes and SAS System reporting options
- Generate HTML reports using ODS statements

Handling Errors

- Identify and resolve programming logic errors
- Recognize and correct syntax errors
- Examine and resolve data errors

In order to pass the test, the test-taker must successfully answer at least 46 of the 70 questions within the 2 hour 15 minute time limit. This is a closed book test. No books, notes, or other reference materials will be permitted inside the test site.

There is a cost for taking the test. At the time of this writing, the cost is $180.00. Please refer to the certification site for details as to the actual cost, payment methods, and other required information.

Two forms of identification with your signature are required upon arrival at the test site.
Be aware that many people no longer sign their credit cards but write “CHECK ID” on the back of the credit card. If you do this, you will need a second form of signature identification, in addition to your driver’s license.

Access to the SAS software will assist greatly in preparing for the test. If you currently do not have access, the SAS learning edition can be purchased through the certification website. This is a full version but restricts the number of observations in a dataset. The good news is that the learning edition has a significantly reduced price.

Chapter 1: Basic Rules

One of the items for which many of us forget the detailed rules is variable names. This is particularly true for someone who writes in multiple programming languages. For SAS, the variable name rules are as follows:

Variable Names

- 1 – 32 characters - letters, numbers, underscores
- Not case-sensitive – Student, student, and STUDENT are identical
- Must start with a letter or an underscore
- No special characters except the underscore

Examples

Invalid	*Valid*	*Invalid Reason*
Sales_$	Sales_Dollars or SalesAmt	Dollar sign is an invalid special character
1Student	Student1	Cannot start with a number
2ND_Grade	Second_Grade	Cannot start with a number
Second-Grade	Second_Grade	Hyphen is an invalid special character

Note: While variable names are not case sensitive, the first instance of the variable name is saved as written and will be used as the default in any output headings.

Programming Statements

- **Must end in a semi-colon**
- Not case sensitive – can be in upper case, lower case, or a combination
- Can start in any column
- Can continue to the next line as long as you don't split a word
- Multiple statements can be on the same line

Comments

There are basically two types of comments

- Line/Statement comments
- Block comments

Line comments begin with an asterisk (*) and end with a semi-colon

For example,

```
* This is a programming comment line;

* This is also a programming comment line.  Notice
the semi-colon at the end of the line in both cases;
```

Block comments start with a slash and asterisk and end with an asterisk and slash. They also can be used to annotate a particular programming line.

```
/* This is a comment block */


/* This is also a comment block
but notice that we do not need a semi-colon*/



/*
Below is a SAS calculation statement along
with a comment block to annotate that statement.
*/

T_CRIT = TINV(PROB,DF);    /* Find the critical t distribution value */


/*
The following is also a valid form of the
above but as a line/statement comment.
Don't forget multiple statements can be on the same line.
 */

t_crit = tinv(prob,df);    * Find the critical t distribution value;
```

Options

There are numerous options that can be set inside the code available to the programmer. This is done by the "OPTIONS" statement. This statement usually appears at the beginning of the program, although that it is not a restriction. Once the options are set, those options are in place until a subsequent options statement changes them. Many of the options control the output format. Some examples are:

- DATE | NODATE – indicates whether a date appears at the top of the output page
- LINESIZE=nn – maximum length of output line is nn
- ORIENTATION=PORTRAIT | LANDSCAPE – output page orientation
- PAGESIZE = n – maximum number of lines per output page is n

For example, to set the maximum length of the output line to 80 and to have no date appear at the top of the page, use the following statement at the top of your program or somewhere before the output is produced.

```
OPTIONS  LINESIZE=80   NODATE;
```

In order to see all of the options, use the OPTIONS procedure. ***The results of this procedure appear in the LOG window.***

```
proc options;
run;
quit;
```

The following shows a sample of the output in the LOG window.

```
NOCAPS             Do not translate source input to uppercase
NOCARDIMAGE        Do not process SAS source and data lines as 80-byte records
CATCACHE=0         Number of SAS catalogs to keep in cache memory
CBUFNO=0           Number of buffers to use for each SAS catalog
CENTER             Center SAS procedure output
NOCHARCODE         Do not use character combinations as substitute for special characters not on
                   the keyboard
CLEANUP            Attempt recovery from out-of-resources condition
NOCMDMAC           Do not support command-style macros
CMPLIB=            Identify previously compiled libraries of CMP subroutines to use when linking
CMPOPT=(NOEXTRAMATH NOMISSCHECK NOPRECISE NOGUARDCHECK)
                   Enable SAS compiler performance optimizations
NOCOLLATE          Do not collate multiple copies of printed output
COLORPRINTING      Print in color if printer supports color
COMAMID=TCP        Specifies the communication access method to be used for SAS distributed
                   products
COMPRESS=NO        Specifies whether to compress observations in output SAS data sets
```

```
CONNECTPERSIST    Persist connection to remote session
CONNECTREMOTE=    Remote session ID used by SAS/CONNECT software
CONNECTSTATUS     Show the current status of a SAS/CONNECT upload or download transfer
CONNECTWAIT       Wait for a SAS/CONNECT rsubmit to finish before allowing further processing to
                  Occur
COPIES=1          Number of copies to make when printing
       :
       :
       :
ORIENTATION=PORTRAIT
                  Orientation to use when printing
NOOVP             Do not allow output lines to be overprinted
NOPAGEBREAKINITIAL
                  Do not begin SAS log and listing files on a new page.
PAGENO=1          Beginning page number for the next page of output produced by the SAS System
PAGESIZE=55       Number of lines printed per page of output
PAPERDEST=        Bin to receive printed output
PAPERSIZE=LETTER  Size of paper to print on
PAPERSOURCE=      Bin to pull paper from when printing
PAPERTYPE=PLAIN   Type of paper printer is using
```

The DATA Block

The DATA block is generally used to load data into a SAS dataset. In the simple case, an input buffer is created from some input option, either by using an external file, a SAS dataset, or directly by using the DATELINES statement. A position is allocated in a data vector for each data element from the input data as well as each additional data element assigned within the DATA block itself.

In the following example, the data vector will have positions for 5 data elements, 4 from the input buffer and 1 for the average.

```
data grades;

      input name $17. grade1 3. grade2 3. grade3 3.;

      average = mean(grade1,grade2,grade3);

datalines;
George Washington 80 90 95
Thomas Jefferson  92 86 92
James Madison     88 78 93
;
run; quit;
```

Obs	name	grade1	grade2	grade3	average
1	George Washington	80	90	95	88.3333
2	Thomas Jefferson	92	86	92	90.0000
3	James Madison	88	78	93	86.3333

The DATA block will execute for each input record available. At the beginning of each DATA block loop, the data vector will set all new data values to missing and then load each value either from the input buffer or from the assignment statement. If for some reason, one of the new data values is not addressed at all, the value will remain set to missing. At the end of the DATA block loop, the data vector will create an SAS observation associated with the new dataset. This process will continue until the end of input data is reached.

While the detailed portion of this process can be altered in several ways that will be discussed later in this document, the fundamental process always remains intact.

Chapter 2: Inputting Data – Handling Data Elements

There is a multitude of ways to get data into SAS. This chapter will first focus on the following:

- List input
- Column input
- Formatted input

The specific SAS statement required is the INPUT statement. In this statement you must identify the variable names and the type of variable. The default type is numeric. If the variable is a character type, put a dollar sign after the variable name.

List Input

Data can come in many forms. One of the simplest but, in many ways, most restrictive is the list input. This is when data is separated only by spaces. Each data element on each observation can start in a different column. For example, suppose we need to input the following data where each observation consists of a student name and his/her 3 grades for the course. Assuming that the data is not in a file yet, the data can be input using the INPUT statement and the DATALINES statement. ***Note the semi-colon at the end of the DATALINES statement and at the end of the data itself.***

```
data grades_c2_1;

      input name $ grade1 grade2 grade3;

datalines;
Washington 80 90 95
Jefferson 92   86  77
Madison 88 78  98
;

run; quit;
```

The student's name is a character variable type, while the grades are numeric variable types. Consequently a dollar sign ($) appears after the "name" variable to indicate a character type. This will create 3 observations for our data file "grades", with one observation for each student. Note also that each corresponding grade starts in a different column for each student and that there are multiple spaces between the grades for observation 2. ***This is not a problem in the list option, since a space/blank is the delimiter and the SAS column pointer will skip any number of spaces/blanks until it hits the next non-blank character.***

There are several problems with this input method.

- Character variables cannot be more than 8 characters, the default for both character and numeric variables. Characters 9 and above would be truncated.
- There can be no imbedded spaces within the name column above.
- If one of the grades were missing, it would need to be denoted by a period, which is the indicator for a missing value.

The following examples demonstrate these problems.

```
data grades_c2_2;

      input name $ grade1 grade2 grade3;

datalines;
Washington 80 90 95
Jefferson 92 77
Madison 88 78  98
;
run;
quit;
```

The following shows the resulting dataset.

Obs	name	grade1	grade2	grade3
1	Washingt	80	90	95
2	Jefferso	92	77	.

Washington and Jefferson are truncated to Washingt and Jefferso, the first 8 characters.

Jefferson has a missing grade, *grade2*, which causes a failure for observations 2 and 3. When Jefferson's observation is processed, SAS will use the value 77 as *grade2* and then go the next line to look for Jefferson's third grade, *grade3*. It tries to put Madison's name in as the third grade which causes an error because it is a character value going into numeric variable type. Now, when trying to process observation 3, SAS will put Madison's grade1 into the name field, since grade1 is the next available data element. However, there is nothing for *grade3* and this will cause a complete error for Madison's observation. The following shows the output log resulting from the execution.

```
NOTE: Invalid data for grade3 in line 23 1-7.
RULE:      ----+----1----+----2----+----3----+----4----+----5----+----6----+----7----+----8----+--
23         Madison 88 78  98
name=Jefferso grade1=92 grade2=77 grade3=. _ERROR_=1 _N_=2
NOTE: SAS went to a new line when INPUT statement reached past the end of a line.
NOTE: The data set WORK.GRADES has 2 observations and 4 variables.
```

Putting a period in for the missing grade solves this problem completely.

```
data grades_c2_3;

      input name $ grade1 grade2 grade3;

datalines;
Washington 80 90 95
Jefferson 92 . 77
Madison 88 78  98
;
run;
```

Obs	name	grade2	grade1	grade3
1	Washingt	80	90	95
2	Jefferso	92	.	77
3	Madison	88	78	98

What is also not possible with this method is to have embedded spaces within the data for a single variable. The following example would create a rather bad dataset due to the embedded space in the name values.

```
data grades_c2_4;

      input name $ grade1 grade2 grade3;

datalines;
George Washington 80 90 95
Thomas Jefferson 92 . 77
James Madison 88 78  98
;

run;
quit;
```

The DATALINES statement shows 2 character variables, first name and last name, and 3 numeric type variables, the 3 grades. However, there is only one character variable listed on the INPUT statement, *name*. Consequently, SAS will put the first name value into the *name* variable and then try to put the last name value into the *grade1* variable which will cause an error due to the mismatched types. The following shows the resulting dataset.

Obs	name	grade1	grade2	grade3
1	George	.	80	90
2	Thomas	.	92	.
3	James	.	88	78

An even bigger problem is that the routine will actually process but the output is obviously all wrong. Consequently, it is important to always check the LOG window to see if any errors or warnings have occurred and then to take the appropriate action. The following is the log information for the above execution.

```
NOTE: Invalid data for grade1 in line 6 8-17.
RULE:      ----+----1----+----2----+----3----+----4----+----5----+----6----+----7----+----8----+--
6          George Washington 80 90 95
name=George grade1=. grade2=80 grade3=90 _ERROR_=1 _N_=1
NOTE: Invalid data for grade1 in line 7 8-16.
7          Thomas Jefferson 92 . 77
name=Thomas grade1=. grade2=92 grade3=. _ERROR_=1 _N_=2
NOTE: Invalid data for grade1 in line 8 7-13.
8          James Madison 88 78  98
name=James grade1=. grade2=88 grade3=78 _ERROR_=1 _N_=3
NOTE: The data set WORK.GRADES has 3 observations and 4 variables.
```

Column Input

The column input will solve many of the previous problems. After each variable name state the start column and the end column separated by a dash. The dollar sign needs to precede the start and end column on the character type variables.

```
data grades_c2_5;

      input name $ 1-17 grade1 18-20 grade2 21-23 grade3 24-26;

datalines;
George Washington100 90 95
Thomas Jefferson  92 86 77
James Madison     88 78 98
;

run;
```

Obs	name	grade1	grade2	grade3
1	George Washington	80	90	95
2	Thomas Jefferson	92	86	77
3	James Madison	88	78	98

You must be certain that the data values are in the correct columns. However, the advantages to this format are:

- Can handle embedded spaces in a character data value
- Not limited to 8 characters in the data value
- Do not need a space between data values.
- Can move backwards and forwards

Formatted Input

Sometimes data comes in with a variety of structures and types. If this is the case, then a formatted INPUT statement might be the best method. With a formatted input statement, you use INFORMATS. This allows the programmer to state the variable type and the size all at once. ***However, the programmer must understand and control the column positioning.***

An INFORMAT, other than numeric, identifies the data type, the size and is followed by a period. The numeric data type is an exception. Numeric is the default data type and the period may not be the last character of the INFORMAT. Some examples follow.

NAME $17. - character data type called NAME of size 17.

START_DATE MMDDYY10. – date data type called START_DATE of size 10.
Format will be MM/DD/CCYY.

WEIGHT 5.1 - numeric data type called WEIGHT of 5 total places 3 whole numbers a decimal point and 1 decimal place.

GRADE 3. - numeric data type called GRADE with 3 integer positions

```
data grades_c2_6;

   input name $17. grade1 3. grade2 3. grade3 3. +3 last_test_date MMDDYY10.;

datalines;
George Washington 80 90 95   09/02/2007
Thomas Jefferson  92 86 77   09/03/2007
James Madison     88 78 98   09/04/2007
;
run;

proc print data=grades;
      format last_test_date mmddyy10.;
run; quit;
```

```
                                                    last_test_
```

Obs	name	grade1	grade2	grade3	date
1	George Washington	80	90	95	09/02/2007
2	Thomas Jefferson	92	86	77	09/03/2007
3	James Madison	88	78	98	09/04/2007

Note: There are a multitude of date formats but the numeric part of the informat controls the size of the date value. Please see the chapter 7 for a discussion of date formats.

Note: A numeric value that contains commas and/or a dollar sign must use a format statement, such as comma9 or dollar11.2. Other than the numbers 0 through 9, a period is the only character that can be used with a numeric entry without a format statement. If commas or a dollar sign are used without a format statement, the numeric variable will contain a missing value.

The following is an example ***without*** format statements

```
data test;
      input name $ price weight;
datalines;
John $23.10   4,000
Paul 56.10    5,000
George 1,234.10 4000
Ringo 1234.10  3000
;
run;
quit;

proc print data=test;
run;
quit;
```

Obs	name	price	weight
1	John	.	.
2	Paul	56.1	.
3	George	.	4000
4	Ringo	1234.1	3000

The following is the same example ***with*** format statements

```
data test;
      input name $ price dollar9.2 weight comma5.;
datalines;
John $23.10   4,000
Paul 56.10    5,000
George 1,234.10 4000
Ringo 1234.10  3000
;
run;
quit;

proc print data=test;
run;
quit;
```

Obs	name	price	weight
1	John	23.1	4000
2	Paul	56.1	5000
3	George	1234.1	4000
4	Ringo	1234.1	3000

Column Controls

In the above example, take note of the +3 before "last_test_date". The rule is that the column pointer will go to the next column after each data value is read. Consequently, after "grade3", in columns 24-26, the column pointer will be at column 27. However, the "last_test_date" starts in column 30. So, the programmer needs to move the column pointer to column 30 before the "last_test_date" variable. This is done either by using the +3 as shown or by using @30. The following will give the identical results.

```
input name $17. grade1 3. grade2 3. grade3 3. @30 last_test_date MMDDYY10.;
```

+n - moves the pointer forward n columns

@n – moves the pointer to column n – SAS will go backward or forward to column n.

While the "n" represents a specific value, a specific value is not required. The "n" can be replaced by a numeric variable or an arithmetic expression. Notice the use of "lineno" and the expressions containing "lineno" in the following example.

lineno = 1;

```
input @lineno name $17. @(lineno+17) grade1 3. @(lineno+20)grade2 3.
@(lineno+23)grade3 3. @(lineno+29)last_test_date MMDDYY10.;
```

If you cannot be sure in what column a data value will start but you know that it starts after some specific word or character, use the @ sign with the word or character in single quotes as follows:

INPUT @'City:' C_NAME $20.

The above will read in the next 20 characters after 'City:' (no quotes in the data).

If you do not know the length of the city name and it is not column driven, then add a colon to the INFORMAT. This will read up to 20 characters but will stop if it hits a space before the 20th character.

INPUT @'City:' C_NAME :$20.

At times it may be necessary to mix and match input types. This is not a problem for SAS but the programmer must know how the column pointer is functioning.

LENGTH statement

In addition to the INFORMAT, the storage length of the input variable can be assigned by using the LENGTH statement. By default all variables are stored using 8 bytes. This may not be enough for characters data and it may be too much for numeric data. In order to control the actual storage length, use the LENGTH statement. All variables prior to the length value will have that length value assigned. In the following example the variables *first_name* and *last_name* will be stored with 15 bytes instead of the default of 8. The dollar sign ($) is required for character type data. Since the value of a student's grade cannot exceed 100, the numeric variables of grade1, grade2, grade3 will be stored only using 4 bytes instead of 8 which will save some memory space.

LENGTH first_name last_name $15 grade1 grade2 grade3 4;

Note that this is not a format statement. Consequently no period is required or used.

```
data grades_c2_7;

   length first_name last_name $15 grade1 grade2 grade3 4;
   informat grade1 grade2 grade3 3.;
   input first_name last_name grade1 grade2 grade3;

datalines;
George          Washington       80 90 95
Thomas          Jefferson        92 86 77
James           Madison          88 78 98
;
run;

proc print data=grades_c2_7;
run;
quit;
```

Obs	first_ name	last_name	grade1	grade2	grade3
1	George	Washington	80	90	95
2	Thomas	Jefferson	92	86	77
3	James	Madison	88	78	98

Note that the length value as well as the INFORMAT will be stored with the information associated with the dataset and can be viewed through PROC CONTENTS.

```
proc contents data=grades_c2_7;
run;
quit;
```

Alphabetic List of Variables and Attributes

#	Variable	Type	Len	Informat
1	first_name	Char	15	
3	grade1	Num	4	3.
4	grade2	Num	4	3.
5	grade3	Num	4	3.
2	last_name	Char	15	

The default value of 8 bytes can be changed with the LENGTH statement. Use the option DEFAULT=n to change the default storage bytes. The following example will have the same result as before. However, any new variable whose length is not directly stated, will default to 4 bytes instead of 8.

LENGTH first_name last_name $15 DEFAULT=4;

LABEL statement

Just like the LENGTH statement, the LABEL statement can associate a label to a variable to be used as the column heading when printing the variable. Otherwise, the variable name itself is used. The general format is:

LABEL variable_name_1 = 'Label 1' variable_name_2 = 'Label 2' …

For example,

```
LABEL grade1 = 'Grade 1'   grade2= 'Grade 2' grade3 = 'Grade 3'
        average="Student's Average";
```

Note: Use double quotes if a single quote is part of the label'

To remove the label just set the label value to blank.

LABEL grade1 = ' ' grade2= ' ' grade3 = ' ' average=" ";

ATTRIB statement

The ATTRIB statement sets attributes, such as LABEL, FORMAT, LENGTH, etc. for one or more variables. This is the same as setting the attributes using the individual statements. The difference is that multiple attributes for a particular variable or a list of variables can be set all at once.

The general format is:

ATTRIB variable_list LABEL='label 1' LENGTH=length_specifier
FORMAT=format_specifier …

For example,

ATTRIB grade1-grade3 LABEL='Grade' LENGTH=4 FORMAT=3.;

INFILE Statement

To this point, the data has been input to the data statement using the DATALINES statement. However, most of the time the data will come from an existing file or database of some sort.

Assuming that the data is in a standard text file, the programmer can use the INFILE statement. The basic structure is as follows:

INFILE *filename;*

The filename can be a specific filename enclosed in quotes or it can be a variable name whose data value contains the path and filename.

INFILE 'c:\mysasdata\grades.txt'; /* Be sure to use the full pathway – see below */

This will replace the DATALINES block used in the above examples. However the INFILE must be **BEFORE** the INPUT statement. The following will give the same output as the first "grades" example above.

```
data grades_c2_8;

      infile 'c:\mysasdata\grades.txt';

      input name $ 1-17 grade1 18-20 grade2 21-23 grade3 24-26;

run;
```

It is common to have data in text files delimited by commas, tabs, or some other character other than a space. When using LIST INPUT, since blank/space is the default delimiter, add the keyword phrase DLM=',' to the end of the INFILE statement to notify SAS to use a comma as the delimiter. Other delimiter values can be used as well. For tabs or other control characters use the corresponding hex value, such as '09'X for the tab character. Here are some examples.

INFILE 'c:\mysasdata\Grades.txt' DLM= ',' ; /* comma delimited */

INFILE 'c:\mysasdata\Grades.txt' DLM= '09'X ; /* tab delimited */

The one problem with the above format is that consecutive commas will be treated as one comma. Usually consecutive commas indicate a missing data value. In order to have SAS interpret the consecutive commas as a missing value, use the keyword DSD. This tells SAS to treat consecutive commas as missing data. However, it does more. The DSD keyword will:

- Treat consecutive delimiters as having a missing value
- Tell SAS to ignore delimiters in data values enclosed in quotation marks.
- Not read quotation marks as part of the data.

INFILE 'c:\mysasdata\Grades.txt' DLM= ',' DSD; /* comma delimited */

IMPORT Statement

A more powerful method to bring data into SAS is the PROC IMPORT procedure. This procedure will handle a variety of known file types as well as a variety of options. Here is an example of the general format with some of the available options:

```
PROC IMPORT  DATAFILE(TABLE) ='input file name'  OUT=dataset name;
        DBMS= database identifier;  /* for example ACCESS */
        GETNAMES=YES ; /* use first line headings as variable names */
        DELIMITER=',';
```

Chapter 3: Inputting Data – Handling Observations

While most languages expect data observations (records) to be on one line with the next observation on the next line, and so forth, SAS allows both multiple observations on a single line and a single observation on multiple lines. However, this creates some havoc for the programmer and definitely requires a whole set of rules. The following describes the rules.

Multiple Lines of Data Per Observation

By default SAS will continue to the next line of data, if necessary, when trying to process an INPUT statement. So, it is not required to have all data on a single line. A single observation can span as many lines as desired. Once the last data variable is found, SAS, again by default, will start a new line when processing the next INPUT statement for the next observation. Using the previous "grades" example, the following data, even though none of the data for the 3 observations are lined up, works just as well as the previous good examples. We still get 3 observations and each observation contains 3 correct grades.

```
data grades_c3_1;

      input name $ grade1 grade2 grade3;

datalines;
Washington 80
90
95
Jefferson 92 86
77
Madison
88
78
98
;

run; quit;
```

Obs	name	grade1	grade2	grade3
1	Washingt	80	90	95
2	Jefferso	92	86	77
3	Madison	88	78	98

MISSOVER and TRUNCOVER keywords

While continuing to the next line of information looking for data is a nice feature, it does create some problems when dealing with input files and the INFILE statement. Suppose that a text file containing the data has the data for a single observation on one line but not all of the data exists for each observation. For LIST INPUT, we would not want SAS to continue to the next line looking for the data that is missing. In order to solve this problem, use the MISSOVER keyword on the INFILE statement. This will tell SAS to not go to the next line if there is not enough data when it gets to the end of the line.
For example, suppose several students were absent for the last test. The text file containing the grades would not contain the last grade. By using the MISSOVER keyword, the last grade will automatically be identified as missing by SAS. For example, the following data is in a text file and contains each student's name and grades. Notice in the output how grade 3 is declared missing for the last 2 students and SAS did not go to the next data line looking for the last grade. Without the MISSOVER keyword, errors would have occurred.

```
George Washington 80 90 95
Thomas Jefferson  92 86
James Madison     88 78
```

```
data grades_c3_2;

    infile 'c:\mysasdata\grades_m.txt' MISSOVER;
    input name $ 1-17 grade1 grade2 grade3;

run;

proc print data=grades_c3_2;
run;
quit;
```

Obs	name	grade1	grade2	grade3
1	George Washington	80	90	95
2	Thomas Jefferson	92	86	.
3	James Madison	88	78	.

Without the MISSOVER keyword, the following output and log entries would occur.

Obs	name	grade1	grade2	grade3
1	George Washington	80	90	95
2	Thomas Jefferson	92	86	.

```
NOTE: The infile 'C:\mysasdata\grades1.txt' is:
      File Name=C:\mysasdata\grades1.txt,
      RECFM=V,LRECL=256

NOTE: Invalid data for grade3 in line 3 1-5.
RULE:     ----+----1----+----2----+----3----+----4----+----5----+----6----+----7----+----8----+---
3         James Madison     88 78 23
name=Thomas Jefferson grade1=92 grade2=86 grade3=. _ERROR_=1 _N_=2
NOTE: 3 records were read from the infile 'C:\mysasdata\grades1.txt'.
      The minimum record length was 23.
      The maximum record length was 26.
NOTE: SAS went to a new line when INPUT statement reached past the end of a line.
NOTE: The data set WORK.GRADES_C3_2 has 2 observations and 4 variables.
```

The MISSOVER works well for LIST INPUT but the TRUNCOVER keyword works for COLUMN and FORMATTED input. If some data is missing or incomplete, SAS will not continue to look beyond the end of the line, if the TRUNCOVER keyword is there. For example, if the student's name and address are being read in but the address does not take the full identified columns or formatted size, then SAS will stop and not proceed to the next line for the next observation. If there are additional data variables that do not exist on that observation, those data variables will be set as missing, in the same manner as using the MISSOVER keyword on list input.
For example, Madison's address does not take up the entire positions 18 to 32 and the data line ends. No error occurs, for either the address or the state variables, since the TRUNCOVER keyword is in place.

```
George Washington Mt. Vernon, VA
Thomas Jefferson  Montpelier, VA
James Madison     Unknown

data address;
    infile 'c:\mysasdata\address.txt' TRUNCOVER;
    input name $ 1-17 address $ 18-32;

run;

proc print data=address;
run;
quit;
```

Obs	name	address
1	George Washington	Mt. Vernon, VA
2	Thomas Jefferson	Montpelier, VA
3	James Madison	Unknown

FIRSTOBS and OBS keywords

Other keywords that allow control on the INFILE statement are FIRSTOBS and OBS.

FIRSTOBS tells SAS which data line to use to start reading the data. A good example for this use is when there are column headings that need to be skipped. Another example would be to test a program using the first few observations, instead of the whole file.

OBS tells SAS the data line at which to stop reading. **Seems like it should be LASTOBS but it is NOT.**

Both of these can be used on a same INFILE statement.

For example,

```
infile 'c:\mysasdata\grades.txt'  FIRSTOBS=3 OBS=10;
```

SAS will start reading at data line 3 and end after data line 10.

NOTE: These keywords use the starting and ending LINE number. This is NOT necessarily the starting and ending OBSERVATION. If a single observation spans multiple lines, each line counts. So, if each observation contains 2 lines of data and you want to start at the 6th observation, you need to use FIRSTOBS=11.

Line Controls for Observations

When single observations do cross multiple lines, the slash, /, and the #n control moving between lines. The slash instructs SAS to go to the next line of data. The #n tells SAS to go to line "n" of the current observation. The #n allows forward and backward control within an observation.

It is also possible to read part of the observation before reading the entire observation. This can be done with 2 INPUT statements. The first INPUT statement must end with a single "at" sign, @. This tells SAS to stay with the current observation until another I/O statement. A subsequent INPUT statement would then be able to read the rest of the current observation. Using the following text data, suppose that we are only interested in the Virginia observations. By using the "at" sign on the first INPUT statement, the program can test the state and if it matches Virginia, then it can use another INPUT statement to read the rest of the observation.

```
VA George Washington 80 90 95
VA Thomas Jefferson  92 86 98
VA James Madison     88 78 92
AR Wiliam Clinton    95 96 98
VA Dolly Madison     94 95 98
MA Paul Revere       95 96 98
```

```
data grades_c3_3;
      infile 'c:\mysasdata\grades_c3_3.txt';

      input State $2. +1 @;

      if State = 'VA'
      then
            input name $17. grade1 grade2 grade3;

      else
            delete;

run;
proc print data=grades_c3_3;
run;
quit;
```

Notice that only the Virginia records appear in the output.

Obs	State	name	grade1	grade2	grade3
1	VA	George Washington	80	90	95
2	VA	Thomas Jefferson	92	86	98
3	VA	James Madison	88	78	92
4	VA	Dolly Madison	94	95	98

Multiple Observations on a Single Line

It is possible to put multiple observations on the same line. However, you must tell SAS that this is the case. Otherwise, observations will be skipped. To do this use the double "at" sign, @@, at the end of the INPUT statement. This tells SAS to keep reading that line for observations until it runs out of data. When the line runs out of data, SAS proceeds to the next line for more data. The following illustrates the use of the double at sign. The following 2 lines of data come from a text file called *grade_c3_4.txt*. Notice that there are multiple observations on each line and that the third observation's data wraps to the next line. This is not a problem with the double "at" sign on the INPUT statement.

GRADES_C3_4 text file

```
George Washington 80 90 95 Thomas Jefferson  92 86 98 James Madison     88 78
92 William Clinton   95 96 98 Dolly Madison     94 95 98
```

```
data grades_c3_4;

      infile 'c:\mysasdata\grades_c3_4.txt';
      input name $17. grade1 grade2 grade3 @@;

run;

proc print data=grades_c3_4;
run;
quit;
```

Obs	name	grade1	grade2	grade3
1	George Washington	80	90	95
2	Thomas Jefferson	92	86	98
3	James Madison	88	78	92
4	William Clinton	95	96	98
5	Dolly Madison	94	95	98

Chapter 4: Controlling the Data

While inputting the data, there are many different options that can be taken to control the data. Here are some examples.

It may be necessary to add additional variables, possibly by calculating some value, including the run date, etc. In the grades example above, computing the average of the three grades and storing that value with the observation is a good example of computing an additional value.

If a dataset with many variables is being loaded, we may not need to keep all of the data variables. For example, a student's record may contain names, addresses, birth date, school history, etc. Yet, only the name and birth date may be of interest.

Certain observations may not be needed at all. For example, we may only want observations related to a particular state, even though the dataset contains observations for many states; or if all of the student information is in one dataset, we may only need student information for a particular class section.

SAS provides methods to handle all of these situations.

Assignment Statement

Suppose it is necessary to compute a new value, such as the average grade for each student. This can be done by doing the arithmetic calculation as part of the DATA block and assigning the resultant value to a new variable name. The assignment statement uses the equal sign, "=". The new variable will be stored as part of the dataset with the other variables.

```
data grades_c4_1;

      input name $17. grade1 3. grade2 3. grade3 3.;
      average = (grade1 + grade2 + grade3)/3.0;

datalines;
George Washington 80 90 95
Thomas Jefferson  92 86 77
James Madison     88 78 98
;
run;

proc print data=grades_c4_1;
run;
quit;
```

Obs	name	grade1	grade2	grade3	average
1	George Washington	80	90	95	88.3333
2	Thomas Jefferson	92	86	77	85.0000
3	James Madison	88	78	98	88.0000

Even though only the 3 grades were part of the input data, the new dataset "grades" also contains the average of the 3 grades. This is due to the calculation line.

Arithmetic operations do have a hierarchy of execution as in algebra. The "order of operations" is the same as in algebra:

- parenthesis
- exponentiation
- multiplication and division in order from left to right
- addition and subtraction in order from left to right

The following table shows the symbol associated with each mathematical operation.

*	Multiplication
/	Division
+	Addition
-	Subtraction
**	Exponentiation

The value assigned to new character variable types must be enclosed in single quotes. In the following example "Letter_Grade" will be added to the database as a character type variable.

```
data grades_c4_2;

      input name $17. grade1 3. grade2 3. grade3 3.;
      average = (grade1 + grade2 + grade3)/3.0;

      if average > 90 then Letter_Grade = 'A';
      else if average > 80 then Letter_Grade = 'B';
      else if average > 70 then Letter_Grade = 'C';
      else if average > 60 then Letter_Grade = 'D';
      else Letter_Grade = 'F';

datalines;
George Washington 80 90 95
Thomas Jefferson  92 93 97
James Madison     88 78 98
;
run;
quit;
```

```
proc print data=grades_c4_2;
run;
quit;
```

Obs	name	grade1	grade2	grade3	average	Letter_ Grade
1	George Washington	80	90	95	88.3333	B
2	Thomas Jefferson	92	93	97	94.0000	A
3	James Madison	88	78	98	88.0000	B

Note that the size of a character variable created by an assignment statement becomes the size needed for the first assignment or 8, whichever is greater. Subsequent assignments will NOT expand the size of the variable beyond that initial value.

Date variable types can also be added. The following two statements are examples of assigning values and creating new variables that are date type variables.

```
d7 = '01sep20'd;
d8 = date();
```

Please see chapter 7 for more information on dates.

Calculation Functions

Several other useful options are available when controlling the data. An earlier example in this chapter showed how a new average variable could be added when processing the data by mathematically calculating the three grades.

This method is fine if there are only a few grades or if no grades are missing. If some of the grades are missing, then the average will also be missing. If some grades are missing but the programmer still wants the average of the non-missing grades, the MEAN function is better. The MEAN function will include only the non-missing grades. Other functions, such as SUM, also handle the missing grades. For example, the SUM function will give the total for the non-missing values and will ignore the missing values.

Take the grades example from earlier in this chapter, where we just added the 3 grades together. If grade 3 is missing because the student is absent, the following would be the result.

Obs	name	grade1	grade2	grade3	average
1	George Washington	80	90	95	88.3333
2	Thomas Jefferson	92	86	.	.
3	James Madison	88	78	.	.

Notice the missing "grade3" for 2 of the students and the associated missing average.
Now if the code were changed to use the MEAN function as follows, the output would be very different.

```
data grades_c4_3;

      input name $17. grade1 3. grade2 3. grade3 3.;

      average = mean(grade1,grade2,grade3);

datalines;
George Washington 80 90 95
Thomas Jefferson  92 86 .
James Madison     88 78 .
;
run;
```

Obs	name	grade1	grade2	grade3	average
1	George Washington	80	90	95	88.3333
2	Thomas Jefferson	92	86	.	89.0000
3	James Madison	88	78	.	83.0000

Notice that the correct average (mean) was calculated even for the students with missing grades.

Accumulators

Suppose that a running total is desired while inputting the data. This can be done by using a statement that does the accumulation but does NOT use an equal sign. The structure of the statement tells SAS that your intent is a running total. Suppose in the previous example you wanted a running total for grade 1. This can easily be accomplished by creating an accumulator variable such as the one in the red line below, "sum_g1".

```
data grades_c4_4;
      input name $17. grade1 3. grade2 3. grade3 3.;
      average = mean(grade1,grade2,grade3);

      sum_g1+grade1;

datalines;
George Washington 80 90 95
Thomas Jefferson  92 86 .
James Madison     88 78 .
;
run; quit;
```

Obs	name	grade1	grade2	grade3	average	sum_g1
1	George Washington	80	90	95	88.3333	80
2	Thomas Jefferson	92	86	.	89.0000	172
3	James Madison	88	78	.	83.0000	260

RETAIN statement

While a running total is a common occurrence, the underlying method implies that the value of the variable, sum_g1 in the previous example is not cleared for each new observation. That is, the sum_g1 value is not reset to missing for a new observation as is normally the case for all variables associated with observations. The value of the variable is retained.
The value of any variable can be retained by using the RETAIN statement.

The following code will produce the same results as above. However, it uses the RETAIN statement. The zero after the "sum_g1" variable, in the RETAIN statement, tells SAS to initialize the variable to zero.

```
data grades_c4_5;

      input name $17. grade1 3. grade2 3. grade3 3.;

      average = mean(grade1,grade2,grade3);


      sum_g1 = sum_g1+grade1;

      retain sum_g1 0;


datalines;
George Washington 80 90 95
Thomas Jefferson  92 86 .
James Madison     88 78 .
;
run;
quit;

proc print data=grades_c4_5;
run;
quit;
```

Obs	name	grade1	grade2	grade3	average	sum_g1
1	George Washington	80	90	95	88.3333	80
2	Thomas Jefferson	92	86	.	89.0000	172
3	James Madison	88	78	.	83.0000	260

IF-THEN-ELSE statement

In several of the program examples already shown, an IF-THEN statement was used. This statement allows a single line of code to be executed based on some logical condition. The general format is

IF *condition* THEN *action*;

For example,

IF average >= 90 THEN letter_grade = 'A';

The conditional portion of the statement must evaluate to TRUE or FALSE. If TRUE, then the THEN part of the statement is executed. If FALSE, the THEN part of the statement is NOT executed.

Symbols or mnemonics can be used for the conditional portion. The following is a table of symbols and corresponding mnemonics, either of which is recognized by SAS.

Symbol	Mnemonic	Meaning
=	EQ	Equals
^= ~=	NE	Not equal
>	GT	Greater than
<	LT	Less than
>=	GE	Greater than or equal to
<=	LE	Less than or equal to

Conditional expressions can be built in a way similar to mathematical expressions. Just as there are mathematical operators, there are logical operators. There are three primary logical operators AND, OR, and NOT. Just as with the mathematical operators, there is a symbol associated with each operation. Either the symbol or the mnemonic can be used. Note there are multiple symbols for OR and NOT. Only use one or the other.

&	AND
! |	OR
^ ~	NOT

When two conditions are used and operated on by an AND operator, both conditions must evaluate to TRUE in order for the THEN statement to execute. If the OR is used, only one condition must evaluate to TRUE in order for the THEN statement to execute. The NOT operator will give the opposite value of the conditional result.

The following table shows possible condition combinations and results of the operations on those conditions.

Condition A	Condition B	Operator	Result
TRUE	TRUE	AND	TRUE
TRUE	FALSE	AND	FALSE
FALSE	TRUE	AND	FALSE
FALSE	FALSE	AND	FALSE
TRUE	TRUE	OR	TRUE
TRUE	FALSE	OR	TRUE
FALSE	TRUE	OR	TRUE
FALSE	FALSE	OR	FALSE
TRUE		NOT	FALSE
FALSE		NOT	TRUE

Using the grades example again, suppose that an "A" was only to be assigned if the average was 90 or above AND all grades were recorded. Assuming that we had a variable called "num_grades" whose value contained the number of non-missing grades, then the following would be the appropriate IF statement.

```
IF    average >= 90 AND num_grades = 3    THEN    letter_grade = 'A';
```

There is also an **implied AND**. If there are two comparisons with a common variable, the AND is not required. The following conditional statements are equivalent.

```
IF    average >= 80 AND average <=89    THEN    letter_grade = 'B';

IF    80 <=  average  <=89    THEN    letter_grade = 'B';
```

Note: Please see the WHERE statement in Chapter 5 for other conditional operators.

Suppose that we want one statement to be assigned if the conditional expression is true and a different statement to be assigned if the conditional statement is false. In this case we would use an IF-THEN-ELSE statement.

```
 IF    average >= 90 AND num_grades = 3    THEN    letter_grade = ‘A’;
ELSE    letter_grade = ‘I’;              /* I = Incomplete */
```

Returning to the A,B,C,D,F letter grade assignments, we can assign different letter grades by checking the grade for multiple conditions and using the ELSE statement with the IF-THEN.

The general structure is

IF condition THEN action;
ELSE IF condition THEN action;
ELSE IF condition THEN action;
:
:
:
ELSE action;

```
IF     average >=90     THEN      letter_grade = 'A';
ELSE  IF    average > 80    THEN    letter_grade = 'B';
ELSE  IF    average > 70    THEN    letter_grade = 'C';
ELSE  IF    average > 60     THEN    letter_grade = 'D';
ELSE      letter_grade = 'F';
```

Note that the capitals letters for IF, THEN, and ELSE are only for clarity here and are NOT required in SAS.

There is an order of operations for logical operators just as there is an order of operations for mathematical operators. The order of precedence is:

- ***Inside parenthesis***
- ***NOT***
- ***AND***
- ***OR***

Despite the defined precedence, good programming practices state to use parenthesis to establish clarity as it can be get confusing quickly with the ANDs and ORs.

DO-END statement

The above IF-THEN-ELSE statements only show one action statement being performed. However, multiple action statements can be performed. State the multiple action statements with a DO statement and end the multiple actions with an END statement. The general format is as follows:

```
IF   condition   THEN

        DO;
                action statement 1;
                action statement 2;
                :
                :
                action statement n;

        END;

ELSE

        DO;

                action statement 1;
                action statement 2;
                :
                :
                action statement n;

        END;
```

Iterative DO loops

A specific version of the DO-END statement is the "Iterative DO" statement. This statement allows a block of code to be processed multiple times before proceeding past the END statement. The general format is as follows:

```
DO index_variable = start TO stop BY increment

                action statement 1;
                action statement 2;
                :
                :
                action statement n;
END
```

The *index_variable* is the control variable for the count. The *start* and *stop* contain the starting numeric value for the *index_variable* and the ending numeric value for the *index_variable*. The increment contains the amount to increment the *index_variable* for each new iteration of the loop. This is optional. If not stated the default value is one. Once the *index_variable* goes beyond the *stop* value, the loop stops processing and the next line of code after the END statement processes.

In the following example, 10 grades will be read in for each student and assigned to an array called grade. Please see chapter 7 for a description on the use of arrays.

Using the iterative DO statement, these 10 grades will be added to obtain a total value of the grades for use in computing the average. The DO will cycle 10 times adding the grade corresponding to the value of "i" for that particular iteration.

```
data grades_c4_7;
      input name $17. grade1 - grade10;

      ARRAY grade(10) grade1-grade10;

      total = 0;

      do i = 1 to 10;
            total = grade(i) + total;
      end;

      average = total/10.0;

datalines;
George Washington 80 90 95 95 84 86 77 92 86 93
Thomas Jefferson  92 86 90 95 84 86 77 92 88 94
James Madison     88 78 98 95 84 86 77 92 89 94
Martha Washington 85 94 93 95 84 86 87 95 88 98
John Adams        94 88 91 95 84 86 87 96 89 99
Dolly Madison     92 80 99 95 84 86 87 92 88 93
;
run;
quit;
```

Note that the start, stop, and increment values can be in ascending or descending order and that these values can either be constants or numeric variables.

The following example will output to the log file the even numbers in descending order from 20 to 2.

```
data _NULL_;

      do i = 20 to 1 BY -2;

            put i;

      end;

run;
quit;
```

SAS Generated Variables

At times SAS generates variables. The following are two very common ones. However, there are others that will be discussed in specific sections of this guide. Many SAS generated variables are recognizable by the prefix and suffix underscores.

_N_

The _N_ variable's value contains the number of times SAS has looped through the entire DATA block. This value will not necessarily be equal to the observation number for the output data. If the path through the DATA block is prematurely terminated, then the _N_ will be incremented but an observation may not be written. An example is when a "Subsetted IF" occurs. Please see the next chapter for a discussion of the "Subsetted IF."

The following is an example. In the original "grades" routine, there are 40 total observations, 10 for 4 different sections. The following routine is only selecting 2 of the sections. In the subsequent dataset, "grades_c4_8", note the value of the observation number in the first column and the value of the corresponding N in the last column. Even though the program did not write the first 20 observations from the "grades" dataset to the new dataset, the _N_ was still incremented.

```
data grades_c4_8;
      set mysas.grades;
      N = _N_;
      if section >= 30;
run; quit;
```

```
proc print data=grades_c4_8;
run;
quit;
```

Obs	name	section	grade1	grade2	grade3	average	last_name	N
1	Chester Arthur	30	85	98	85	89.33	Arthur	21
2	Grover Cleveland1	30	93	93	81	89.00	Cleveland1	22
3	Benjamin Harrison	30	87	90	88	88.33	Harrison	23
4	Grover Cleveland2	30	96	93	81	90.00	Cleveland2	24
5	William McKinley	30	97	99	81	92.33	McKinley	25
6	Theodore Roosevelt	30	98	81	88	89.00	Roosevelt	26
7	William Howard	30	90	82	94	88.67	Howard	27
8	Woodraw Wilson	30	98	89	84	90.33	Wilson	28
9	Warren Harding	30	92	87	81	86.67	Harding	29
10	Calvin Coolidge	30	97	97	97	97.00	Coolidge	30
11	Herbert Hoover	40	99	96	92	95.67	Hoover	31
12	Franklin Roosevelt	40	98	82	86	88.67	Roosevelt	32
13	Harry Truman	40	88	94	93	91.67	Truman	33
14	Dwight Eisenhower	40	88	90	83	87.00	Eisenhower	34
15	John Kennedy	40	80	83	84	82.33	Kennedy	35
16	Lyndon Johnson	40	85	95	98	92.67	Johnson	36
17	Richard Nixon	40	85	91	98	91.33	Nixon	37
18	Gerald Ford	40	81	88	84	84.33	Ford	38
19	Jimmy Carter	40	94	96	84	91.33	Carter	39
20	Ronald Reagan	40	88	81	98	89.00	Reagan	40

ERROR

The _ERROR_ variable's value is a 1 if there is an error for the current observation and a 0 if there is no error.

FIRST.variable_name
LAST.variable_name

When a particular file is sorted by one or more grouping variables, it may be necessary to know when the first observation for that group is being processed and/or when the last observation for that group is being process. SAS automatically creates two variables for the group. When the first observation for a group is being processed, the *FIRST.variable_name* variable will be set to 1. Otherwise it will be set to 0. The *LAST.variable_name* variable works the same way when the last observation for a group is being processed.

Converting Variables

At times it may be necessary to convert character variables to numeric variables and numeric variables to character variables. This can be done with the INPUT function and the PUT function respectively.

Note that this is NOT the same as the INPUT and PUT statements.

The general format is:

new_num_variable = INPUT (old_char_variable, numeric format specifier);

new_char_variable = PUT (old_num_variable, numeric format specifier);

Note that the format specifier in both cases is a numeric format specifier.

The following example shows both conversion options with the results of PROC CONTENTS confirming the variable types and sizes.

```
data test;
input charv $4. +1 numv;

c2n=input(charv,4.); /* convert a character variable to a numeric variable */
n2c=put(numv,6.2);   /* convert a numeric variable to a character variable */

datalines;
1234 134.01
2222 143.02
4444 153.03
;
run;
quit;

proc print data=test;
run;
quit;

proc contents data=test;
run;
quit;
```

Obs	charv	numv	c2n	n2c
1	1234	134.01	1234	134.01
2	2222	143.02	2222	143.02
3	4444	153.03	4444	153.03

```
Alphabetic List of Variables and Attributes

#    Variable    Type    Len

3    c2n         Num       8
1    charv       Char      4
4    n2c         Char      6
2    numv        Num       8
```

Within specific functions, such as the SUBSTR function, SAS will automatically convert the variable, if the receiving variable has not been previously defined.

Chapter 5: Datasets

Most of the previous discussions concerning data have dealt with inputting various types of data files into the SAS system. However, once a data file is incorporated into the SAS system it becomes a SAS dataset. When the dataset is saved, unless otherwise specified, it is saved as a SAS file type with the extension, sas7bdat.

Once this happens it is no longer necessary to INPUT or IMPORT a file to bring in data. When the data file on the disk is a SAS dataset simply use the SET command with the dataset name. All necessary information concerning the data file, variables, and other information is stored within that file. For example, once our "grades" file was saved as a SAS dataset, simply use the following set command to load the data into the work area, "grades".

```
data grades_c5_1;
      set grades;
run;
quit;
```

There will now be two datasets. The data on the file grades.sas7bdat will be loaded into the work area with a dataset name of grades. If the actual grades dataset was to be modified directly, then use direct referencing or use the library option described after the example.

```
data 'c:\mysasdata\grades.sasb7dat';  /* an example of direct referencing */
      set 'c:\mysasdata\grades.sasb7dat';
run;
quit;
```

Libraries

All datasets belong to a library. The naming convention for a dataset inside of SAS is the library name and the dataset name separated by a period.

```
library_name.dataset_name

For example,

mysas.grades
```

In the above "grades" examples, the dataset associated with the "data" statement, is only referenced by a dataset name with no library name. When this happens, SAS uses the default library name. ***The default library name is WORK. This is a temporary library and the dataset is considered a temporary dataset. When SAS is closed, this dataset will be deleted.***

In order to make a dataset permanent, either move the temporary dataset to a permanent library or use a permanent library to start. From a PC point of view, a permanent library is just a directory on the PC. Assign a library name to the desired directory using the LIBNAME statement. Then use that library name as part of the full dataset name in the DATA statement. Any modifications will be saved in that dataset on the PC directory.

In the following example, the SAS database called grades is modified using the DATA statement and the SET statement. The existing data file does not have the average value included. The average value will be added to the original grades data file on the disk drive by the following statements.

```
libname mysas 'C:\mysasdata\';

data mysas.grades;

      set mysas.grades;

      average = mean(grade1, grade2, grade3);

run;
quit;

proc print data=mysas.grades;
run;
quit;
```

If you do NOT want the file directly updated, then do NOT use the direct referencing or the permanent library name in the DATA statement.

Additional Notes:

The LIBNAME statement has a multitude of options available for use. Many are associated with linking to database management systems.

FILENAME statement

The FILENAME statement is used to assign a short reference name to a physical file name. The general format is:

FILENAME ***reference_name 'physical_file_name';***

For example, when using the INFILE statement previously, a direct reference to the physical file name was used.

```
data grades_c2_8;

      infile 'c:\mysasdata\grades_c2_8.txt';

      input name $ 1-17 grade1 18-20 grade2 21-23 grade3 24-26;
run;
quit;
```

Using a FILENAME statement, the example changes to:

```
filename gc28 'c:\mysasdata\grades_c2_8.txt';

data grades_c2_8;

      infile gc28;

      input name $ 1-17 grade1 18-20 grade2 21-23 grade3 24-26;
run;
quit;
```

For one instance of the filename, this is not a savings. However, if there are multiple references to the filename, the FILENAME statement can save a considerable amount of time.

Sorting Datasets

In many cases a dataset must be sorted first before being used by certain procedures. Just use the SORT procedure to sort the data. The general form is PROC SORT with the BY keyword to define the variables to use for sorting and the sorting order. These variables together are called the sort key.

PROC SORT DATA= in_dataset_name OUT=sorted_dataset_name;

BY var1 var2 …;

The input dataset observations will be sorted to the output dataset name in sequential order by the variables listed in the BY statement. The default order is low to high. If the desired order for a particular variable is high to low, use the DESCENDING keyword ***before*** that particular variable name.

The following example will sort students in descending order (high to low) based on the average grade stored in the variable AVG. If two or more students have the same average, the observations will be in ascending order by name within that particular average score.

PROC SORT DATA=mysas.grades OUT=mysas.avg_grades;

BY descending avg name;

If there is no output name the input and output datasets will be the same.

If there is the possibility of duplicate records for a sort key and the duplicate records are not desired in the output, use the NODUPKEY keyword option on the PROC SORT statement.

Subsetting IF statement

While retrieving data from the outside dataset into the dataset defined in the DATA statement, it is possible to be selective about which observations are brought into the new dataset, thus creating a subset of the original dataset. The first method is to use the "subsetted IF". The structure is:

IF *condition*;

By default the DATA step will process and load each individual observation from the input dataset. However, by using an IF with a condition statement, SAS interprets the statement as criteria for loading that observation or not loading that observation. If the conditional part of the IF evaluates to TRUE, the observation is loaded. If it evaluates to FALSE, the observation is not loaded and processing immediately jumps to the next observation. This also helps with performance. Place the IF as high in the DATA block as possible so that needless statements are not executed.

In general, by subsetting the data there will most likely be a performance increase when using the subsetted dataset in other procedures due to the reduced number of observations.

WHERE statement

The WHERE statement works in the same manner as the Subsetted IF but has a broader range of uses. The structure is:

WHERE *condition*;

If the condition evaluates to TRUE, the observation is kept, otherwise it is not kept.

The WHERE statement lends itself well to conditional operators other than AND, OR, and NOT. The following are some of the other conditional operators.

IN – An expression with the IN operator evaluates to TRUE if the value of the variable matches one of the values within the list of values in parentheses. For example,

where state IN ('NC','TX') ;

If the value of the variable "*state*" is NC or TX the condition evaluates to TRUE.

BETWEEN – AND – An expression with the BETWEEN- AND operator evaluates to TRUE if the value of the variable lies between the two values, inclusively. For example,

where age BETWEEN 13 AND 64; /* 13 and 64 are included */

CONTAINS (?) – An expression with the CONTAINS operator evaluates to TRUE if the any part of the value of the variable matches the characters after the CONTAINS operator. The question mark can be used in place of the word "contains". For example,

where name CONTAINS 'bay';
where name ? 'bay';

LIKE with a % (percent sign) – An expression with the LIKE operator and a % evaluates to TRUE if the beginning value of the variable matches the characters after the "LIKE" operator and before the % sign. For example, the following is TRUE for all *lastnames* beginning with the letter "R".

where lastname like 'R%';

LIKE with an _ (underscore) – An expression with the LIKE operator and an underscore evaluates to TRUE if the value of the variable matches the corresponding positional characters after the "LIKE" operator, with any other characters matching the positions containing the underscore(s). For example, the following will take all first names that begin with a "D" and have an "a" in the third position and an "n" in the fourth position. The % indicates that the remaining values do not matter.

where firstname like 'D_an%';

A **prefix equal** (an equal sign followed by a colon) is also for partial character comparisons. The prefix equal will only compare only the beginning characters of the variable. The following example will evaluate all *lastname* values that begin with the letter "S" as TRUE.

where lastname =: 'S' ;

MIN – The MIN operator returns the lower value.

where z = (x MIN y);

MAX – The MAX operator returns the higher value.

where z = (x MAX y);

|| – The concatenation operator concatenates two strings.

where fullname = 'James ' || 'Brown';

IS MISSING– A condition with the IS MISSING operator evaluates to TRUE if the associated variable contains a missing value.

where grade3 IS MISSING;

Note that these conditional operators are not limited to the WHERE statement.

The WHERE statement can be used in any of the following SAS procedures, to selectively process data observations.

CALENDAR	RANK
CHART	REPORT
COMPARE	SORT
CORR	SQL
DATASETS (APPEND statement)	STANDARD
FREQ	TABULATE
MEANS/SUMMARY	TIMEPLOT
PLOT	TRANSPOSE
PRINT	UNIVARIATE

OUTPUT Statement

When a DATA block processes a given set of statements and comes to the end of the block, it automatically outputs an observation to the output dataset(s) defined on the DATA statement line. There is no need to actually put an OUTPUT statement in the code.

However, there are times when selective output is desired. If the selection is simple, the "subsetted IF" can be used as previously described. However, it is also possible to specifically direct the output using the OUTPUT statement. If an OUTPUT statement appears anywhere within the DATA block, there will NOT be an automatic OUTPUT at the end of the DATA block.

In addition, the user can direct output to a particular dataset based on certain conditions. This can be done by naming the dataset within the OUTPUT statement. The following example shows two new datasets, SECT80 and SECT90, created from an existing dataset called "grades_c5_2." If value of the section variable is 80, the observation is sent to dataset SECT80. If the value of section variable is 90, the observation is sent to dataset SECT90.

The "grades_c5_2" dataset

Obs	name	studid	section	grade1	grade2	grade3	average
1	George Washington	101	80	80	90	95	88.3333
2	Thomas Jefferson	220	80	92	86	77	85.0000
3	James Madison	854	80	88	78	98	88.0000
4	Martha Washington	420	90	84	92	98	91.3333
5	John Adams	119	90	95	89	84	89.3333
6	Dolly Madison	750	90	92	95	96	94.3333

```
data sect80 sect90;
      set mysas.grades_c5_2;

      if section = 80 then output sect80;
      else output sect90;

run;
quit;
```

```
proc print data=sect80;
run;
quit;
```

Obs	name	studid	section	grade1	grade2	grade3	average
1	George Washington	101	80	80	90	95	88.3333
2	Thomas Jefferson	220	80	92	86	77	85.0000
3	James Madison	854	80	88	78	98	88.0000

```
proc print data=sect90;
run;
quit;
```

Obs	name	studid	section	grade1	grade2	grade3	average
1	Martha Washington	420	90	84	92	98	91.3333
2	John Adams	119	90	95	89	84	89.3333
3	Dolly Madison	750	90	92	95	96	94.3333

DROP, KEEP, RENAME

DROP, KEEP, and RENAME can be either options or statements. The basic formats are:

Statement Format

DROP variable_list;
KEEP variable_list;
RENAME from_name= to_name;

Option Format

PROC PRINT DATA=dataset (DROP=variable_list);
PROC PRINT DATA=dataset (KEEP=variable_list);

DATA new_dataset (RENAME=(from_name=to_name));

As a statement DROP and/or KEEP can only be added as part of a DATA statement block. The DROP statement will cause the new dataset(s) being written to not include the variables listed in the variable list and include all other variables from the input file(s). The KEEP statement will cause the new dataset being written to only include the variables listed in the variable list and not include all other variables from the input file(s).

The RENAME statement simply changes the variable name from the input dataset to the new name on the output dataset.

Note that the DROP/KEEP/RENAME statement applies to ALL datasets in the DATA statement.

As an option on a given statement the DROP or KEEP applies to the dataset associated with the statement to which it is an option. For example, if the DROP/KEEP is part of the PROC PRINT statement, then the DROP/KEEP will apply to the dataset associated with the PROC PRINT.

The DROP or KEEP as an option can be part of the following statements, DATA, SET, MERGE, UPDATE, and various PROC statements, such PRINT, MEAN, etc.

The following code shows an example without the DROP option and then an example with the DROP option.

Without the DROP option – all variables are shown

```
proc print data=grades_c5_2;
run;
quit;
```

Obs	name	studid	section	grade1	grade2	grade3	average
1	George Washington	101	80	80	90	95	88.3333
2	Thomas Jefferson	220	80	92	86	77	85.0000
3	James Madison	854	80	88	78	98	88.0000
4	Martha Washington	420	90	84	92	98	91.3333
5	John Adams	119	90	95	89	84	89.3333
6	Dolly Madison	750	90	92	95	96	94.3333

With the DROP option – only name, studid, section, and average, the remaining variables, are shown

```
proc print data=grades_c5_2 (drop= grade1 grade2 grade3);
run;
quit;
```

Obs	name	studid	section	average
1	George Washington	101	80	88.3333
2	Thomas Jefferson	220	80	85.0000
3	James Madison	854	80	88.0000
4	Martha Washington	420	90	91.3333
5	John Adams	119	90	89.3333
6	Dolly Madison	750	90	94.3333

With the KEEP option – the same output as above can be produced

```
proc print data=grades_c5_2 (keep=name studid section average);
run;
quit;
```

Obs	name	studid	section	average
1	George Washington	101	80	88.3333
2	Thomas Jefferson	220	80	85.0000
3	James Madison	854	80	88.0000
4	Martha Washington	420	90	91.3333
5	John Adams	119	90	89.3333
6	Dolly Madison	750	90	94.3333

Additional Notes:

- ***The DROP and KEEP are also performance factors. A significant process time improvement can occur by eliminating unnecessary variables from a large dataset. This can occur either through the DROP or KEEP, whichever makes sense in context.***

- ***The input data files remain unchanged in all cases of the DROP/KEEP/RENAME usage.***

Chapter 6: Handling Multiple Datasets

With the SET command and with some other commands, the programmer now has the ability to combine, merge, and update datasets. The following discusses these concepts.

Stacking Datasets

It is possible to combine two (or more) datasets into one. The first option is to "stack" the datasets. This means putting the second dataset's observations right after the first dataset's observations to create one new large dataset. Suppose that a teacher has 2 different sections of the same course and wants to analyze both sections together. By simply putting the datasets for both sections in the SET command, SECT80 and SECT90 separated by a blank, the dataset named in the data statement, BOTHSECT, will contain the information for both sections.

SECT80 dataset

Obs	name	studid	section	grade1	grade2	grade3	average
1	George Washington	101	80	80	90	95	88.3333
2	Thomas Jefferson	220	80	92	86	77	85.0000
3	James Madison	854	80	88	78	98	88.0000

SECT90 dataset

Obs	name	studid	section	grade1	grade2	grade3	average
1	Martha Washington	420	90	84	92	98	91.3333
2	John Adams	119	90	95	89	84	89.3333
3	Dolly Madison	750	90	92	95	96	94.3333

```
data bothsect;
      set sect80 sect90; /* stack the two datasets */
run;
quit;


proc print data=bothsect;
run;
quit;
```

BOTHSECT dataset

Obs	name	studid	section	grade1	grade2	grade3	average
1	George Washington	101	80	80	90	95	88.3333
2	Thomas Jefferson	220	80	92	86	77	85.0000
3	James Madison	854	80	88	78	98	88.0000
4	Martha Washington	420	90	84	92	98	91.3333
5	John Adams	119	90	95	89	84	89.3333
6	Dolly Madison	750	90	92	95	96	94.3333

Note that there are a total of 6 observations, 3 from SECT80 and 3 from SECT90. Also note that all students from section 90 follow all of the students from section 80.

Interleaving Datasets

By stacking the datasets the second section is behind the first section. However, suppose that the instructor needs the students' information in student id order/sequence. This would mean having all students in one dataset but this time, the two sections would be mixed together in student id sequence. To do this simply use the BY statement with the sequencing variable(s). However, to accomplish this, the two input datasets must already be sorted in that sequence. If one or both datasets are not sorted, the data must be sorted first. Use the PROC SORT procedure to sort the data. Then use the DATA statement with the BY statement to interleave the two datasets. SECT80 and SECT90 datasets will be used again. The SAS program will do the following.

SECT80 dataset is already sorted by "studid", student id. SECT90 is not. Consequently, SECT90 needs to be sorted but there is no need to sort SECT80.

The following SAS program sorts each the SECT90 dataset by student id and then interleaves the two datasets, SECT80 and SECT90, into one dataset called BOTHSECT.

```
proc sort data=sect90;
      by studid;
run;
quit;

proc print data=sect90;
run;
quit;
```

SECT90 after the sort

Obs	name	studid	section	grade1	grade2	grade3	average
1	John Adams	119	90	95	89	84	89.3333
2	Martha Washington	420	90	84	92	98	91.3333
3	Dolly Madison	750	90	92	95	96	94.3333

```
data bothsect;
      set sect80 sect90;
      by studid;  /* interleave the two datasets by student id */
run;
quit;

proc print data=bothsect;
run;
quit;
```

BOTHSECT after BY statement

Obs	name	studid	section	grade1	grade2	grade3	average
1	George Washington	101	80	80	90	95	88.3333
2	John Adams	119	90	95	89	84	89.3333
3	Thomas Jefferson	220	80	92	86	77	85.0000
4	Martha Washington	420	90	84	92	98	91.3333
5	Dolly Madison	750	90	92	95	96	94.3333
6	James Madison	854	80	88	78	98	88.0000

Note that there are 6 observations but they are now in sequence by "studid", student id. Also note that the two sections are intermingled.

Merging Datasets

Again, suppose that we have two separate datasets but they do not contain the same type of information. For example, suppose that one dataset contains general information about the students, such as name, city, and state, and the second dataset contains each student's grades. The instructor now wants a report showing both sets of information as a single observation for each student. To accomplish this use the MERGE statement. The syntax is almost identical to the SET statement and BY statement when interleaving datasets. However, the results are very different. Instead of putting one dataset's observations after the other one's observations, the two datasets will be joined. Here is an example.

Suppose that we have two datasets for section 80. The one dataset, STUDENT80, contains the student id and general information concerning each student. The second dataset, GRADES80,

contains only the student id and the grades for each student. In order to link these two datasets together and get the correct name and address with that student's grades we use the MERGE statement and the BY statement with the student id to give a new dataset, ALLSECT80. ALLSECT80 contains all variables from the 2 input datasets. The result is a single observation for each student. That single observation contains all of the variables from the 2 input datasets.

STUDENT80 dataset – contains name and address

Obs	studid	name	city	state
1	101	George Washington	Mt. Vernon	VA
2	220	Thomas Jefferson	Montecello	VA
3	854	James Madison	Somewhere	VA

GRADES80 dataset – contains grades

Obs	studid	section	grade1	grade2	grade3
1	101	80	80	90	95
2	220	80	92	86	77
3	854	80	88	78	98

```
data allsect80;
      merge mysas.student80 mysas.grades80;
      by studid;  /* merge the two datasets by student id */
run;
quit;

proc print data=allsect80;
run;
quit;
```

ALLSECT80 dataset – contains both datasets merged

Obs	studid	name	city	state	section	grade1	grade2	grade3
1	101	George Washington	Mt. Vernon	VA	80	80	90	95
2	220	Thomas Jefferson	Montecello	VA	80	92	86	77
3	854	James Madison	Somewhere	VA	80	88	78	98

In order for the merge to work properly:

- There must be a common variable(s) between the datasets in order to link them, "studid" in the previous example.
- The datasets being merged must be sorted by the common variable(s). If not, use the PROC SORT procedure before merging.

Other important notes:

The BY statement can contain more than one variable but these variables need to define the sorted order of the input datasets.

The MERGE statement can contain more than two datasets but they all need to be linked by the same variable(s) used in the BY statement.

If a variable, other than a linking variable, is common to both datasets, the second dataset's value for that variable will be the value in the resulting merged dataset. This is true even if the second dataset's variable has a missing value.

The number of output observations will be equal to the same number of observations as the largest number of observations of the two (or more) merged datasets. In our example, both datasets being merged had 3 observations and our resulting merged dataset had 3 observations. Had one of our datasets contained 3 observations and the other dataset only 2 observations, our merged dataset would still contain 3 observations. However, the variables from the missing observation in the 2 observation dataset would be set to missing in the newly merged dataset.
For example, suppose the grades for studid = 220, Thomas Jefferson, were missing. The output would have been:

Obs	studid	name	city	state	section	grade1	grade2	grade3
1	101	George Washington	Mt. Vernon	VA	80	80	90	95
2	220	Thomas Jefferson	Montecello	VA	.	.	.	.
3	854	James Madison	Somewhere	VA	80	88	78	98

The previous example referenced two databases that were linked one observation to one observation. **This is NOT a requirement**. There can be multiple records for the same linking variable(s) in one of the databases. This would mean a one-to-many relationship, instead of a one-to-one relationship.

This would be the case if each student's grade were on a single observation. Assuming no grades were missing, there would have been 3 observations for each the 3 students for a total of 9 grade observations. The resulting merged dataset would also contain 9 observations. Here is an example.

STUDENT80 dataset – contains the student's name and address

Obs	studid	name	city	state
1	101	George Washington	Mt. Vernon	VA
2	220	Thomas Jefferson	Montecello	VA
3	854	James Madison	Somewhere	VA

GRADES80 dataset – contains the student's grades

Obs	studid	section	test	grade
1	101	80	1	80
2	101	80	2	90
3	101	80	3	95
4	220	80	1	92
5	220	80	2	86
6	220	80	3	77
7	854	80	1	88
8	854	80	2	78
9	854	80	3	98

ALLSECT80 dataset – contains both datasets merged

Obs	studid	name	city	state	section	test	grade
1	101	George Washington	Mt. Vernon	VA	80	1	80
2	101	George Washington	Mt. Vernon	VA	80	2	90
3	101	George Washington	Mt. Vernon	VA	80	3	95
4	220	Thomas Jefferson	Montecello	VA	80	1	92
5	220	Thomas Jefferson	Montecello	VA	80	2	86
6	220	Thomas Jefferson	Montecello	VA	80	3	77
7	854	James Madison	Somewhere	VA	80	1	88
8	854	James Madison	Somewhere	VA	80	2	78
9	854	James Madison	Somewhere	VA	80	3	98

Updating Datasets

Another scenario is when there is a master dataset with a second dataset providing updated information to the master dataset. In this case the UPDATE statement will take the information from the second dataset and modify the corresponding master dataset's observation with the new

information. Just like using the MERGE, a linking variable(s) needs to exist between the two datasets.
For example, suppose that we have the student information dataset and the student's average is kept in that dataset. As grades are added to the grades file, the averages are recomputed and stored in a temporary dataset called AVERAGE80. However, we need to get the updated average from the temporary dataset onto the corresponding student information observation. The UPDATE statement with the BY statement will accomplished this.

STUDENT80 dataset – contains name, address, and average grade

Obs	studid	name	city	state	average
1	101	George Washington	Mt. Vernon	VA	0
2	220	Thomas Jefferson	Montecello	VA	0
3	854	James Madison	Somewhere	VA	0

AVERAGE80 dataset – contains average grade

Obs	studid	average
1	101	88.3333
2	220	85.0000
3	854	88.0000

```
data student80;
   update student80 average80;
   by studid;/* update the student80 average with averages from average80 */
run;
quit;

proc print data=student80;
run;
quit;
```

STUDENT80 dataset – contains name, address, and average grade after UPDATE

Obs	studid	name	city	state	average
1	101	George Washington	Mt. Vernon	VA	88.3333
2	220	Thomas Jefferson	Montecello	VA	85.0000
3	854	James Madison	Somewhere	VA	88.0000

Transposing a Dataset

Suppose multiple records exist within a dataset for a single student but the user only wants one record per student. Suppose grades and the average are stored in a single numeric variable called "grade" with a type variable. The type variable is a character variable whose value identifies the meaning of the numeric grade field.

The following data is an example of a dataset called "GRADES_C6_1", which contains an observation for each individual grade and the average grade.

Obs	name	type	grade
1	Jefferson	grade1	86.0
2	Jefferson	grade2	90.0
3	Jefferson	grade3	90.0
4	Jefferson	average	88.7
5	Washington	grade1	80.0
6	Washington	grade2	90.0
7	Washington	grade3	85.0
8	Washington	average	85.0

Using the TRANSPOSE procedure with a BY statement, an ID statement, and a VAR statement, a new dataset with one observation for each student can be created. The input data must be sorted by the variable that defines the common record, the student name field in this case. The TRANSPOSE uses the individual TYPE variable to define the new variable name and then stores the value of the numeric GRADE variable into the new variable defined by the TYPE variable.

The variable identified in the BY statement defines the link for the common observations. The variable identified in the ID statement defines the TYPE variable. The variable identified in the VAR statement defines the numeric variable.

Using the above dataset, 4 observations for each student are transposed into 1 observation. The following shows the SAS code and the resultant output.

```
proc transpose data=mysas.grades_c6_1 out=grades_c6_2;
      by name;
      id type;
      var grade;
run;
quit;

proc print data=grades_c6_2;
run;
quit;
```

Obs	name	_NAME_	grade1	grade2	grade3	average
1	Jefferson	grade	86	90	90	88.7
2	Washington	grade	80	90	85	85.0

The _NAME_ variable is automatically generated by SAS and is part of the transposed dataset.

Chapter 7: Outputting the Data - Fundamentals

Before discussing the specific output methods, let's look at some overall functionality.

Dates

SAS stores all dates as a numeric value. The zero or base date is January 1, 1960. This date is stored as zero and any date forward is stored as a positive numeric value representing the number of days after that date. Any date prior to that date is stored as a negative numeric value representing the number of days before that date. The range of valid dates/years extends from 1582 A.D. to 19,900 A.D. Some examples follow:

January 2,1960 = 1
January 1, 1961 = 366 /* 1960 was a leap year/
February 1, 2007 = 17198

January 1, 1959 = -365
February 1, 1950 = -3621

There are a multitude of format statements available for any date format imaginable. The basic structure of the predefined formats is some form of month, day, and year with the number of characters to be used. Some examples follow for September 30, 2007:

MMDDYY8. formats as 09/30/07
MMDDYY10. formats as 09/30/2007
DATE7. formats as 30SEP07
DATE9. formats as 30SEP2007
WORDDATE18. formats as September 30, 2007

One major issue is the 2 digit year. When you see a two digit year, you do not really know the intended century. For example, 09/01/10 could be the ending date for a mortgage that occurs on September 1, 2010 or it could be the birth date of someone who was born on September 1, 1910. In order to solve this problem, SAS stores a year cutoff value to determine which century to use as the default. The system default cutoff is 1920. This means that for years between 0 and 19, SAS will assume century 20. For years 20 through 99, SAS will assume century 19. So, in the above example, 09/01/10 would be interpreted by SAS as September 1, 2010.
The value for the year cutoff can be changed through the YEARCUTOFF option. Use the option statement with the YEARCUTOFF keyword. For the birth date scenario above, set YEARCUTOFF=1905 in order to default to 1910 for the birth date. Of course you can always directly indicate 1910.

Time is also stored as a numeric value. This is stored as the number of seconds since midnight. Again, a variety of formats are available. The default format for TIME shows the time as HH:MM:SS.

Don't forget… all formats need the ending period.

The following are some SAS examples. Note the method and format for entering a date as a constant, the date is in single quotes in "ddmmmyy" format followed by a "d" outside the single quotes.

In the output, the first five date variables, d1 – d5, show the different stored numeric values. The date variables d6 and d7 show the different century assignments based on the YEARCUTOFF option of 1920. The variables d6, d7, and d8 also show different output formats. The t1 variable shows one of the time output formats.

```
data test;
d1 = '01jan60'd;
d2 = '01jan61'd;
d3 = '01jan59'd;
d4 = '01feb07'd;
d5 = '01feb50'd;
d6 = '01sep10'd;
d7 = '01sep20'd;
d8 = date();
t1 = time();

run;
quit;

proc print;
      format  d6 mmddyy10. d7 mmddyy10. d8 worddate18. t1 time.;
run;
quit;
```

Obs	d1	d2	d3	d4	d5	d6	d7	d8	t1
1	0	366	-365	17198	-3621	09/01/2010	09/01/1920	September 15, 2007	15:54:47

Custom Formats

While a variety of standard format options exists, it may be necessary to create custom formats. This is particularly true for formalizing the output for presentations, etc. This can be done with the FORMAT procedure. The general format for the PROC FORMAT procedure is as follows:

PROC FORMAT;

```
VALUE     format_name     range-1 = 'formatted text 1'
                          range-2 = 'formatted text 2'
                          :
                          :
                          range-n = 'formatted text n'
```

Multiple formats can be defined within a single PROC FORMAT procedure.

The following example says to substitute the text string "Male" if the value of the variable is a 1, M, or m and to substitute the text string "Female" if the value of the variable is 2, F, or f.

Numeric and character formats can NOT be mixed. Note the dollar sign in the second VALUE statement.

```
PROC FORMAT;

    VALUE gender1     1  = 'Male'
                      2  = 'Female';

    VALUE $gender2    'M', 'm' = 'Male'
                      'F', 'f' = 'Female';
```

- ***Be sure to end each value set with a semi-colon.***
- ***When the variable being translated contains character data, the format name must begin with a dollar sign, such as "$gender2" above.***
- ***Format names must be no more than 32 characters and follow the same rules as variable names except that a character type must start with a dollar sign.***
- ***When actually using the format specifier, be sure to put a period just as you would with any format specifier.***

If the variable sex contains is a numeric type, use the following for output.

```
    FORMAT sex gender1.
```

If the variable sex is a character type, use the following for output.

```
    FORMAT sex $gender2.
```

There are a variety of ways to identify the actual values. The "$gender2" uses a listing method; that is, the desired values to be formatted are separated by a comma. The following shows other methods.

If an inclusive range of values is appropriate use a hyphen.

```
VALUE  movie_discount     0  - 12 = 'Child Discount'
                         13 - 64 = 'No Discount'
                         65 - 100 = 'Senior Discount';
```

An exclusive range is also possible using the less than or greater than signs with the hyphen.

```
VALUE  movie_discount     0 -< 13 = 'Child Discount'
                         13 -< 65 = 'No Discount'
                         65 - 100 = 'Senior Discount';
```

Appropriate keywords are also available, for example LOW, HIGH, and OTHER.

```
VALUE  movie_discount    LOW -< 13 = 'Child Discount'
                         13 -< 65 = 'No Discount'
                         65 - HIGH = 'Senior Discount'
                         OTHER = 'No Discount';
```

TITLE and FOOTNOTE statements

The TITLE and FOOTNOTE statements allow the user to set up to 10 heading lines and/or 10 footnotes lines. The form is

```
TITLE1 'This is the first heading line';
TITLE2 'This is the second heading line';
:
:
TITLE10 'This is the tenth heading line';


FOOTNOTE1 'This is the first footnote line';
FOOTNOTE2 'This is the second footnote line';
:
:
FOOTNOTE10 'This is the tenth footnote line';
```

The first TITLE line can be entered as TITLE1 or just TITLE.

The TITLE and FOOTNOTE statements are global statements. It is more clear if they are defined within the particular output procedure but this is not required. Since they are global statements they are carried across procedures and remain in effect until specifically cleared or changed.

To clear a title line, enter the keyword for that title line with nothing behind it. Note however that the lines have a relationship. Clearing or even redefining a particular title line clears all lines numerically after it. For example, if you clear TITLE5 or enter a new value for TITLE5, then you have cleared TITLE lines 6 through 10. Consequently, if you want to clear all lines enter:

TITLE;

The FOOTNOTE statement works identically as the TITLE statement.

Use double quotes if a single quote is part of the TITLE or FOOTNOTE content.

TITLE1 "These are the section80's grades.";

```
proc print data=grades_c7_1;
      title 'Grade Report';
      title2 'for all Students';

      footnote '3 Grades Reported to Date';
run;
quit;
```

```
                                  Grade Report
                                for all Students

          Obs          name           grade1    grade2    grade3    average

           1     George Washington       80        90        95      88.3333
           2     Thomas Jefferson        92        86        92      90.0000
           3     James Madison           88        78        93      86.3333



                                3 Grades Reported to Date
```

PUT statement

The PUT statement is probably the easiest way to get output from SAS. By default the PUT statement writes to the log file. The PUT statement follows the same rules as the INPUT statement in regards to formatting. Please review chapter 2 if you are not familiar with those rules.

The following example shows an INPUT statement and then a corresponding PUT statement. In the PUT statement some spacing has been added along with the computed average formatted as 5.2. The results appear in the log file as the notes following the data output indicate.

```
data grades_c7_2;

      input name $17. grade1 3. grade2 3. grade3 3.;
      average = mean(grade1,grade2,grade3);

      put name $17. +2 grade1 3. +2 grade2 3. +2 grade3 3. +2 average 5.2;

datalines;
George Washington 80 90 95
Thomas Jefferson  92 86 90
James Madison     88 78 93
;
run; quit;
```

```
George Washington   80   90   95  88.33
Thomas Jefferson    92   86   90  89.33
James Madison       88   78   93  86.33

NOTE: The data set WORK.GRADES_C7_2 has 3 observations and 5 variables.
NOTE: DATA statement used (Total process time):
      real time           0.01 seconds
      cpu time            0.01 seconds
```

To output the information to a text file (non SAS dataset) on the disk, use the FILE statement. The FILE statement mirrors the INFILE statement just as the PUT statement mirrors the INPUT statement.

The file TESTFILE.TXT can be found in the associated directory with the same format as shown in the example above. The log file will NOT contain the data but will contain the log information denoting that the file was created and that the 3 records were written as indicated after the following program.

```
data grades_c7_3;

      input name $17. grade1 3. grade2 3. grade3 3.;

      average = mean(grade1,grade2,grade3);

      file 'C:\mysasdata\testfile.txt';

      put name $17. +2 grade1 3. +2 grade2 3. +2 grade3 3. +2 average 5.2;

datalines;
George Washington 80 90 95
Thomas Jefferson  92 86 90
James Madison     88 78 93
;
run;
quit;
```

```
NOTE: The file 'C:\mysasdata\testfile.txt' is:
      File Name=C:\mysasdata\testfile.txt,
      RECFM=V,LRECL=256

NOTE: 3 records were written to the file 'C:\mysasdata\testfile.txt'.
      The minimum record length was 39.
      The maximum record length was 39.
NOTE: The data set WORK.GRADES_C7_3 has 3 observations and 5 variables.
NOTE: DATA statement used (Total process time):
      real time           0.07 seconds
      cpu time            0.03 seconds
```

NULL dataset

This indicator is used when output is needed but a specific dataset is not needed. The DATA block is needed in order to produce the output values. However, these values will go to either the log file or a non-SAS data file using the PUT statement. The following example is a repeat of the earlier examples. In this example, the even numbers from 20 to 2 are output to the log file. Other examples might include finding critical t-values, or other statistical type calculations, where the results just need to go to the log file and no specific SAS dataset is necessary.

```
data _NULL_;

      do i = 20 to 1 BY -2;

            put i;

      end;

run;
quit;
```

Numbered and Name Range Lists

While all of the examples in this document have only a few variable names, the reality is that it is very possible to have a large number of variable names. However, many times there is a numerical pattern to the variable names. For example, if there were 10 grades per student, the names might follow a numeric pattern such as, grade1, grade2, …, grade10. It is not necessary to type all ten variable names. SAS allows a shortcut. The following is equivalent to typing all 10 variable names.

INPUT grade1 - grade10;

By using a single dash between the first and last variable name, SAS knows that you really want all 10 grade variable names. This is called a ***numbered range list***.

It is also possible to have a ***name range list***. While there may not be an obvious pattern, as long as you know the internal order of the variable names, using two dashes will provide a shortcut entry option. For example, if the variables, name, address, city, state, zip are in order internally, then the following, using 2 dashes, is identical to typing all 5 variables.

INPUT name - - zip;

In order to determine the internal order, use the PROC CONTENTS procedure with the POSITON option.

The above options are available anywhere a list of variables is appropriate. However, certain functions may require the "OF" keyword for proper interpretation. In the following example, the "OF" is necessary within the MEAN function. If the "OF" were not there, the interpretation would be the mean of the difference of the grade1 and grade3 variables.

```
data grades_c7_4;
      input name $17. grade1 3. grade2 3. grade3 3.;

      average = MEAN( OF grade1 - grade3);

datalines;
George Washington 80 90 95
Thomas Jefferson  92 86 90
James Madison     88 78 98
;
run; quit;
```

Arrays

Arrays in SAS do NOT follow the traditional 3GL array type. In SAS, arrays are just another naming shortcut, primarily for use with the iterative DO loop. In certain cases, there may be a need to progress through a list of variables performing the same test on all variables. In this case, it would be easier to use an iterative DO loop, instead of trying to test each variable individually. This can be done by assigning an array to a variable list using the following format.

ARRAY arr_name (n) $ variable_list;

- ***ARRAY*** – required statement keyword
- ***arr-name*** – the array name to be used.
- ***(n)*** – n is the number of elements in the array which should match the number of variables in the variable list.
- ***$*** - is only necessary if the variable list has character types which have NOT been previously declared.
- ***variable_list*** – list of variable names to be associated with the array element. These variables must be either all numeric types or all character types.

The following is an example using both the ARRAY statement and the numbered list shortcut.

```
data grades_c7_5;
      input name $17. grade1 - grade10;

      ARRAY grade(10) grade1-grade10;

      total = 0;
      do i = 1 to 10;
            total = grade(i) + total;
      end;

      average = total/10.0;

datalines;
George Washington 80 90 95 95 84 86 77 92 86 93
Thomas Jefferson  92 86 90 95 84 86 77 92 88 94
James Madison     88 78 98 95 84 86 77 92 89 94
Martha Washington 85 94 93 95 84 86 87 95 88 98
John Adams        94 88 91 95 84 86 87 96 89 99
Dolly Madison     92 80 99 95 84 86 87 92 88 93
;
run;
```

```
quit;

proc print data=grades_c7_5;
run;
quit;
```

Obs	name	grade1	grade2	grade3	grade4	grade5	grade6	grade7	grade8	grade9	grade10	total	i	average
1	George Washington	80	90	95	95	84	86	77	92	86	93	878	11	87.8
2	Thomas Jefferson	92	86	90	95	84	86	77	92	88	94	884	11	88.4
3	James Madison	88	78	98	95	84	86	77	92	89	94	881	11	88.1
4	Martha Washington	85	94	93	95	84	86	87	95	88	98	905	11	90.5
5	John Adams	94	88	91	95	84	86	87	96	89	99	909	11	90.9
6	Dolly Madison	92	80	99	95	84	86	87	92	88	93	896	11	89.6

Note the _TEMPORARY_ option creates a list of temporary data elements. Use this option when creating an array that is used to calculate some other variable within the data block. These array data elements are not included in the output dataset but their values are retained for each iteration of the data step.

EXPORT Statement

To export from SAS use the PROC EXPORT procedure. This procedure will handle a variety of known file types as well as a variety of options. Here is an example of the general format with some of the available options:

```
PROC EXPORT  DATA =SAS Dataset
        OUTFILE= 'output filename'|OUTTABLE='tablename'
        DBMS= database identifier;  /* for example ACCESS */
        REPLACE /* option to automatically replace an existing file */
        DELIMITER=','
        ;
```

Chapter 8: Outputting the Data – PROC PRINT

Probably the easiest and most powerful way to output SAS datasets is through the PRINT procedure. The following is the simplest form of the PROC PRINT.

```
PROC PRINT DATA=dataset_name;
run;
quit;
```

This routine will output all of the variables on all of the observations for the dataset named. If you do not use the DATA=dataset_name option, the last dataset used will be used for this procedure. If output formats are defined for the dataset, they will be used for data. If you want previously defined labels for the column headings add the option "LABEL" on the PROC PRINT statement.

By default, observation numbers are printed for each observation. To remove the observation numbers, add the option "NOOBS" on the PROC PRINT statement line.

There are several statements available that are commonly used. However, there are some subtle default rules, especially with formats and labels.

- VAR variable_list; – only show the variables listed in the "variable_list."
- ID variable_list; - do not show the observation numbers but show the variables listed in the "variable_list" on the **LEFT SIDE** of the output.
- SUM variable_list; - prints totals for any variable in the "variable_list."
- BY variable_list; - creates a new section for every change in value of the variables in the "variable list". This is for sub-grouping. The dataset must sorted in the same order as the "variable_list" in the BY statement.

The following examples use the "grades" dataset. Upon input, more descriptive labels and a format for average were attached to the variables as part of the dataset. Below is output from the PROC CONTENTS procedure showing the defined format and descriptive labels.

Alphabetic List of Variables and Attributes

#	Variable	Type	Len	Format	Label
5	average	Num	8	5.2	Student Average
2	grade1	Num	8		Grade 1
3	grade2	Num	8		Grade 2
4	grade3	Num	8		Grade 3
1	name	Char	17		Student Name

Some examples of the PRINT procedure

```
proc print data=grades_c8_1;
run;
quit;
```

Notice the observation numbers and the defined format are used but the labels were not.

Without a previously defined format attribute, the average would have shown with 4 decimal places.

Obs	name	grade1	grade2	grade3	average
1	George Washington	80	90	95	88.33
2	Thomas Jefferson	92	86	90	89.33
3	James Madison	88	78	93	86.33

```
proc print data=grades_c8_1 noobs label;
run;
quit;
```

Notice that both the defined format and the label settings are used but there are no longer observation numbers.

Student Name	Grade 1	Grade 2	Grade 3	Student Average
George Washington	80	90	95	88.33
Thomas Jefferson	92	86	90	89.33
James Madison	88	78	93	86.33

```
proc print data=grades_c8_1 label;
      id name average;
run;
quit;
```

Notice that the ID statement removed the observation numbers and put both the name and average on the left side.

Student Name	Student Average	Grade 1	Grade 2	Grade 3
George Washington	88.33	80	90	95
Thomas Jefferson	89.33	92	86	90
James Madison	86.33	88	78	93

```
proc print data=grades_c8_1 label;
      var name average;
run;
quit;
```

Notice that only the name and average are printed due to the VAR statement.

Obs	Student Name	Student Average
1	George Washington	88.33
2	Thomas Jefferson	89.33
3	James Madison	86.33

The following example uses the BOTHSECT dataset to show the use of the BY and SUM statements.

The following information is contained in the dataset BOTHSECT.

Obs	name	section	grade1	grade2	grade3	average
1	George Washington	80	80	90	95	88.33
2	Thomas Jefferson	80	92	86	90	89.33
3	James Madison	80	88	78	93	86.33
4	Martha Washington	90	80	95	95	90.00
5	Dolly Madison	90	92	90	93	91.67
6	John Adams	90	89	84	93	88.67

```
proc print data=bothsect noobs label;
      by section;
      sum grade1 grade2 grade3;
run;
quit;
```

Notice the section breaks due to the BY statement. Also notice the summation for each grade after each section and a grand total. This is due to the SUM statement.

```
---------------------------------------- section=80 ----------------------------------------

                                Grade     Grade     Grade     Student
Student Name                      1         2         3       Average

George Washington                80        90        95        88.33
Thomas Jefferson                 92        86        90        89.33
James Madison                    88        78        93        86.33
----------------------------    -----     -----     -----
section                         260       254       278


---------------------------------------- section=90 ----------------------------------------

                                Grade     Grade     Grade     Student
Student Name                      1         2         3       Average

Martha Washington                80        95        95        90.00
Dolly Madison                    92        90        93        91.67
John Adams                       89        84        93        88.67
----------------------------    -----     -----     -----
section                         261       269       281
                                =====     =====     =====
                                521       523       559
```

Chapter 9: Outputting the Data – PROC REPORT

REPORT procedure

While the PRINT procedure gives a lot of options and flexibility, the REPORT procedure really allows a user to create professional reports with minimal effort. The basic structure is similar to the PROC PRINT procedure.

```
PROC REPORT DATA=dataset_name NOWINDOWS;
        COLUMN var-list;
run;
quit;
```

By default, you will get an interactive report window. The NOWINDOWS option prevents this.

Without the COLUMN statement, all variables in the dataset will be shown. To only show selective variables use the COLUMN statement. This is similar to the VAR statement in the PRINT procedure.

Other helpful options on the procedure line are:

- HEADLINE – inserts a line under the column headings.
- HEADSKIP – inserts a blank line under the column headings.
- COLWIDTH = n – specifies the default number of characters for computed and numeric variables.
- SPACING = n – determines the number of blank characters between each column.
- MISSING – includes observations with missing values; by default they are excluded.

```
proc report data=grades_c9_1 nowindows headline headskip;
      title 'Report with all Variables';
      format average 10.2;
run;
quit;
```

Report with all Variables

Student Name	Grade 1	Grade 2	Grade 3	Student Average
George Washington	80	90	95	88.33
Thomas Jefferson	92	86	77	85.00
James Madison	88	78	98	88.00
Martha Washington	85	92	94	90.33
John Adams	96	87	79	87.33
Dolly Madison	89	80	99	89.33

```
proc report data=grades_c9_1 nowindows headline headskip;
      title 'Report With Two Variables';
      column name average;
      format average 10.2;
run;
quit;
```

Report With Two Variables

Student Name	Student Average
George Washington	88.33
Thomas Jefferson	85.00
James Madison	88.00
Martha Washington	90.33
John Adams	87.33
Dolly Madison	89.33

If only numeric variables are being reported, only summary totals will show.

```
proc report data=grades_c9_1 nowindows headline headskip;
      title 'Report With Only Numeric Variables';
      column grade1 grade2 grade3 average;
      format average 10.2;
run;
quit;
```

Report With Only Numeric Variables

Grade 1	Grade 2	Grade 3	Student Average
530	513	542	528.33

The format statement in the above examples was added for better spacing and readability. Without the format statement, the column heading for average would look something like the following.

Student Name	Grade 1	Grade 2	Grade 3	Stude nt Avera ge
George Washington	80	90	95	88.33

```
Thomas Jefferson          92          86          77  85.00
James Madison             88          78          98  88.00
Martha Washington         85          92          94  90.33
John Adams                96          87          79  87.33
Dolly Madison             89          80          99  89.33
```

DEFINE statement

The real difference between PROC REPORT and PROC PRINT is the DEFINE statement. The DEFINE statement allows a multitude of options for a professionally formatted the report. The general form of the DEFINE statement is:

DEFINE *variable_name* / *options 'column_heading'*;

The column heading field allows a specific column heading and control over the output of that heading. By using the slash you control on which line each word or phrase of the heading appears. In the following example, notice how "Average", "For", and "Student" each appear on a separate line in the column heading.

```
proc report data=grades_c9_1 nowindows headline headskip;
      title 'Report With Two Variables';
      column name average;

      define average / 'Average/For/Student';

      format average 10.2;
run;
quit;
```

```
                                Average
                                    For
Student Name                    Student
──────────────────────────────────────

George Washington                 88.33
Thomas Jefferson                  85.00
James Madison                     88.00
Martha Washington                 90.33
John Adams                        87.33
Dolly Madison                     89.33
```

Other options for controlling the totals and output format are available through the DEFINE statement. The following list shows the primary options with examples.

- DISPLAY – creates one row for each observation. This is the default except when showing all numeric variables. Use this to see the individual data lines when showing only numerics.
- ORDER – rows are arranged/sorted according to the value of the variable.
- GROUP – creates a row for each new value of the variable.
- ANALYSIS – calculates statistics for the variable. Summation is the default.
- ACROSS – for the variable listed, this option creates a column for each unique value.

DISPLAY option

```
proc report data=grades_c9_1 nowindows headline headskip;

      title 'Report With Only Numeric Variables';
      column grade1 grade2 grade3 average;

      define average / display;

      format average 10.2;
run;
quit;
```

Without the DISPLAY option in the DEFINE statement only the summary totals would show.

Report With Only Numeric Variables

Grade 1	Grade 2	Grade 3	Student Average
530	513	542	528.33

With the DEFINE statement and the DISPLAY option, all observations are shown.

Grade 1	Grade 2	Grade 3	Student Average
80	90	95	88.33
92	86	77	85.00
88	78	98	88.00
85	92	94	90.33
96	87	79	87.33
89	80	99	89.33

The following is an input dataset "grades40" for the subsequent DEFINE examples. Notice the 4 different sections with 10 students in each section.

Obs	name	section	grade1	grade2	grade3	average	last_name
1	George Washington	10	90	91	100	93.67	Washington
2	John Adams	10	82	89	97	89.33	Adams
3	Thomas Jefferson	10	92	89	83	88.00	Jefferson
4	James Madison	10	87	83	86	85.33	Madison
5	James Monroe	10	84	81	86	83.67	Monroe
6	John Quincy Adams	10	85	91	96	90.67	Quincy Adams
7	Andrew Jackson	10	87	87	81	85.00	Jackson
8	Martin Van Buren	10	95	85	81	87.00	Van Buren
9	William Henry Harrison	10	94	83	93	90.00	Henry Harrison
10	John Tyler	10	86	94	85	88.33	Tyler
11	James Polk	20	93	95	99	95.67	Polk
12	Zachary Taylor	20	84	97	80	87.00	Taylor
13	Millard Fillmore	20	96	84	89	89.67	Fillmore
14	Franklen Pierce	20	85	82	95	87.33	Pierce
15	James Buchanan	20	82	98	95	91.67	Buchanan
16	Abraham Lincoln	20	90	98	91	93.00	Lincoln
17	Andrew Johnson	20	82	94	87	87.67	Johnson
18	Ulysses Grant	20	82	90	83	85.00	Grant
19	Rutherford Hayes	20	88	97	85	90.00	Hayes
20	James Garfield	20	90	92	96	92.67	Garfield
21	Chester Arthur	30	85	98	85	89.33	Arthur
22	Grover Cleveland1	30	93	93	81	89.00	Cleveland1
23	Benjamin Harrison	30	87	90	88	88.33	Harrison
24	Grover Cleveland2	30	96	93	81	90.00	Cleveland2
25	William McKinley	30	97	99	81	92.33	McKinley
26	Theodore Roosevelt	30	98	81	88	89.00	Roosevelt
27	William Howard	30	90	82	94	88.67	Howard
28	Woodraw Wilson	30	98	89	84	90.33	Wilson
29	Warren Harding	30	92	87	81	86.67	Harding
30	Calvin Coolidge	30	97	97	97	97.00	Coolidge
31	Herbert Hoover	40	99	96	92	95.67	Hoover
32	Franklin Roosevelt	40	98	82	86	88.67	Roosevelt
33	Harry Truman	40	88	94	93	91.67	Truman
34	Dwight Eisenhower	40	88	90	83	87.00	Eisenhower
35	John Kennedy	40	80	83	84	82.33	Kennedy
36	Lyndon Johnson	40	85	95	98	92.67	Johnson
37	Richard Nixon	40	85	91	98	91.33	Nixon
38	Gerald Ford	40	81	88	84	84.33	Ford
39	Jimmy Carter	40	94	96	84	91.33	Carter
40	Ronald Reagan	40	88	81	98	89.00	Reagan

ORDER option

The following example shows the ORDER option. The DEFINE statement with the ORDER option automatically sorts the data into that order and then reports the information. In the example, the data is sorted by the "last_name" variable.
Note also the "SPACING=1" option. This will put 1 additional space between each column.

```
proc report data=mysas.grades nowindows headskip spacing=1;
      column last_name name section grade1 grade2 grade3 average;
      define last_name /order;
      format last_name $15.;
run;
quit;
```

last_name	name	section	grade1	grade2	grade3	average
Adams	John Adams	10	82	89	97	89.33
Arthur	Chester Arthur	30	85	98	85	89.33
Buchanan	James Buchanan	20	82	98	95	91.67
Carter	Jimmy Carter	40	94	96	84	91.33
Cleveland1	Grover Cleveland1	30	93	93	81	89.00
Cleveland2	Grover Cleveland2	30	96	93	81	90.00
Coolidge	Calvin Coolidge	30	97	97	97	97.00
Eisenhower	Dwight Eisenhower	40	88	90	83	87.00
Fillmore	Millard Fillmore	20	96	84	89	89.67
Ford	Gerald Ford	40	81	88	84	84.33
Garfield	James Garfield	20	90	92	96	92.67
Grant	Ulysses Grant	20	82	90	83	85.00
Harding	Warren Harding	30	92	87	81	86.67
Harrison	Benjamin Harrison	30	87	90	88	88.33
Hayes	Rutherford Hayes	20	88	97	85	90.00
Henry Harrison	William Henry Harrison	10	94	83	93	90.00
Hoover	Herbert Hoover	40	99	96	92	95.67
Howard	William Howard	30	90	82	94	88.67
Jackson	Andrew Jackson	10	87	87	81	85.00
Jefferson	Thomas Jefferson	10	92	89	83	88.00
Johnson	Andrew Johnson	20	82	94	87	87.67
	Lyndon Johnson	40	85	95	98	92.67
Kennedy	John Kennedy	40	80	83	84	82.33
Lincoln	Abraham Lincoln	20	90	98	91	93.00
Madison	James Madison	10	87	83	86	85.33
McKinley	William McKinley	30	97	99	81	92.33
Monroe	James Monroe	10	84	81	86	83.67
Nixon	Richard Nixon	40	85	91	98	91.33
Pierce	Franklen Pierce	20	85	82	95	87.33
Polk	James Polk	20	93	95	99	95.67
Quincy Adams	John Quincy Adams	10	85	91	96	90.67
Reagan	Ronald Reagan	40	88	81	98	89.00
Roosevelt	Theodore Roosevelt	30	98	81	88	89.00
	Franklin Roosevelt	40	98	82	86	88.67
Taylor	Zachary Taylor	20	84	97	80	87.00
Truman	Harry Truman	40	88	94	93	91.67
Tyler	John Tyler	10	86	94	85	88.33
Van Buren	Martin Van Buren	10	95	85	81	87.00
Washington	George Washington	10	90	91	100	93.67
Wilson	Woodraw Wilson	30	98	89	84	90.33

GROUP option

The following example uses the GROUP option. The DEFINE statement with the GROUP option allows the user to define groups for display, analysis, total processing, etc. The following example is a simple example of grouping the data by section number. More complicated examples appear with the other options later in this chapter.

```
proc report data=mysas.grades nowindows headskip;
      column section name grade1 grade2 grade3 average;
      define section/group;
run;
quit;
```

Note that the section number is not repeated for each observation.

section	name	grade1	grade2	grade3	average
10	George Washington	90	91	100	93.67
	John Adams	82	89	97	89.33
	Thomas Jefferson	92	89	83	88.00
	James Madison	87	83	86	85.33
	James Monroe	84	81	86	83.67
	John Quincy Adams	85	91	96	90.67
	Andrew Jackson	87	87	81	85.00
	Martin Van Buren	95	85	81	87.00
	William Henry Harrison	94	83	93	90.00
	John Tyler	86	94	85	88.33
20	James Polk	93	95	99	95.67
	Zachary Taylor	84	97	80	87.00
	Millard Fillmore	96	84	89	89.67
	Franklen Pierce	85	82	95	87.33
	James Buchanan	82	98	95	91.67
	Abraham Lincoln	90	98	91	93.00
	Andrew Johnson	82	94	87	87.67
	Ulysses Grant	82	90	83	85.00
	Rutherford Hayes	88	97	85	90.00
	James Garfield	90	92	96	92.67
30	Chester Arthur	85	98	85	89.33
	Grover Cleveland1	93	93	81	89.00
	Benjamin Harrison	87	90	88	88.33
	Grover Cleveland2	96	93	81	90.00
	William McKinley	97	99	81	92.33
	Theodore Roosevelt	98	81	88	89.00
	William Howard	90	82	94	88.67
	Woodraw Wilson	98	89	84	90.33
	Warren Harding	92	87	81	86.67
	Calvin Coolidge	97	97	97	97.00
40	Herbert Hoover	99	96	92	95.67
	Franklin Roosevelt	98	82	86	88.67

Harry Truman	88	94	93	91.67
Dwight Eisenhower	88	90	83	87.00
John Kennedy	80	83	84	82.33
Lyndon Johnson	85	95	98	92.67
Richard Nixon	85	91	98	91.33
Gerald Ford	81	88	84	84.33
Jimmy Carter	94	96	84	91.33
Ronald Reagan	88	81	98	89.00

ANALYSIS option

The ANALYSIS option allows statistics for numeric variables to be calculated. The general form of the define statement for the ANALYSIS option is:

DEFINE *variable_name* / *ANALYSIS statistics*;

Available statistics for the ANALYSIS option are:

- MAX – highest value
- MIN – lowest value
- MEAN – the arithmetic mean
- MEDIAN – the median
- N – number of non-missing values
- NMISS – number of missing values
- P90 – the 90th percentile
- PCTN – the percentage of observations for that group
- PCTSUM – the percentage of a total sum represented by that group
- STD – the standard deviation
- SUM – the sum – ***this is the default value***

Suppose that for each section in our grades example, the mean for grade3 and the mean for the average are desired. The following code will produce the subsequent report.

```
proc report data=mysas.grades nowindows headskip spacing=3;
      column section grade3 average;
      define section /group;
      define grade3 /analysis mean;
      define average /analysis mean;
run;
quit;
```

section	grade3	average
10	88.8	88.10
20	90	89.97
30	86	90.07
40	90	89.40

Note that the group option is required to analyze by section.

ACROSS option

The ACROSS option gives the flexibility to produce reports in a horizontal format rather than a vertical format. Both the DEFINE statement and the COLUMN statement have specific formats. The basic format for the DEFINE statement is:

DEFINE *variable_name* / *ACROSS*;

The COLUMN statement format is:

COLUMN *grouping_variable_name*, (*numeric_variable1*, *numeric_variable2*, ...);

Using the above example, suppose the section means for grade3 and the average were to be horizontally written instead of vertically written. The following code will accomplish this task.

```
proc report data=mysas.grades nowindows headskip spacing=3;
      column section , (grade3 average);
      define section /across;
      define grade3 /analysis mean;
      define average /analysis mean;
run;
quit;
```

section							
10		20		30		40	
grade3	average	grade3	average	grade3	average	grade3	average
88.8	88.10	90	89.97	86	90.07	90	89.40

Note the section variable is the ACROSS variable and note the format of the COLUMN statement.

This will make each individual section value a heading with the mean for grade3 and the section average underneath that value. In the COLUMN statement, the desired dataset variables (***grade3 and average***) appear in parentheses behind the variable being grouped (***section***) with a comma in between.

BREAK and RBREAK statements

Once items are in a group, the analysis for each group and an overall analysis can easily occur. The following example shows the average for each group as well as an overall average for each of the grade variables and the individual's average grade variable. This is done using the GROUP option on the DEFINE statement, the ANALYSIS option on the DEFINE statement, the BREAK statement, and the RBREAK statement. The total lines for each section are controlled by the BREAK statement. The overall total lines are controlled by the RBREAK section. The ANALYSIS option on the DEFINE statement controls which statistic will appear in the group totals and the overall total lines controlled by the BREAK and RBREAK statements respectively. The default value is the sum. However, in the following example, the MEAN statistic is used.

```
proc report data=mysas.grades nowindows headskip spacing=3 colwidth=10;
column section name grade1 grade2 grade3 average;
define section/group;
define grade1 /analysis mean ;
define grade2 /analysis mean;
define grade3 /analysis mean;
define average /analysis mean;

break after section/ol skip summarize suppress;
rbreak after/ol skip summarize;

run;
quit;
```

section	name	grade1	grade2	grade3	average
10	George Washington	90	91	100	93.67
	John Adams	82	89	97	89.33
	Thomas Jefferson	92	89	83	88.00
	James Madison	87	83	86	85.33
	James Monroe	84	81	86	83.67
	John Quincy Adams	85	91	96	90.67
	Andrew Jackson	87	87	81	85.00
	Martin Van Buren	95	85	81	87.00
	William Henry Harrison	94	83	93	90.00
	John Tyler	86	94	85	88.33
		88.2	87.3	88.8	88.10

20	James Polk	93	95	99	95.67
	Zachary Taylor	84	97	80	87.00
	Millard Fillmore	96	84	89	89.67
	Franklen Pierce	85	82	95	87.33
	James Buchanan	82	98	95	91.67
	Abraham Lincoln	90	98	91	93.00
	Andrew Johnson	82	94	87	87.67
	Ulysses Grant	82	90	83	85.00
	Rutherford Hayes	88	97	85	90.00
	James Garfield	90	92	96	92.67
		87.2	92.7	90	89.97
30	Chester Arthur	85	98	85	89.33
	Grover Cleveland1	93	93	81	89.00
	Benjamin Harrison	87	90	88	88.33
	Grover Cleveland2	96	93	81	90.00
	William McKinley	97	99	81	92.33
	Theodore Roosevelt	98	81	88	89.00
	William Howard	90	82	94	88.67
	Woodraw Wilson	98	89	84	90.33
	Warren Harding	92	87	81	86.67
	Calvin Coolidge	97	97	97	97.00
		93.3	90.9	86	90.07
40	Herbert Hoover	99	96	92	95.67
	Franklin Roosevelt	98	82	86	88.67
	Harry Truman	88	94	93	91.67
	Dwight Eisenhower	88	90	83	87.00
	John Kennedy	80	83	84	82.33
	Lyndon Johnson	85	95	98	92.67
	Richard Nixon	85	91	98	91.33
	Gerald Ford	81	88	84	84.33
	Jimmy Carter	94	96	84	91.33
	Ronald Reagan	88	81	98	89.00
		88.6	89.6	90	89.40
		89.325	90.125	88.7	89.38

The BREAK and RBREAK statements have the following general format:

BREAK location *group_variable / options*

RBREAK location */ options*

Location is either the keyword BEFORE or AFTER. This controls whether the summary statistics for the group occur at the beginning of the group's output, BEFORE, or at the end of the group's output, AFTER.

The ***group_variable*** is one of the dataset variables that are defined as a GROUP in a DEFINE statement.

Note that there can be multiple group variables. Each group variable will require a separate BREAK statement if summary output is desired.

The following are the available **options** for the BREAK and RBREAK statements.

- OL - inserts a line of hyphens (-) above each value that appears in the summary line.
- UL - inserts a line of hyphens (-) under each value that appears in the summary line.
- SKIP - writes a blank line for the last break line.
- SUMMARIZE - writes a summary line in each group of break lines.
- SUPPRESS - suppresses the printing of the value of the break variable in the summary line, and any underlining or overlining in the break lines.

The following statements are the BREAK and RBREAK statements from the previous SAS example, showing the usage of some of the above options.

```
break after section/ol skip summarize suppress;
rbreak after/ol skip summarize;
```

Statistics with the COLUMN statement

Individual variable statistics can also be shown by associating the desired statistic(s) with the variable name in the COLUMN statement. The general formats are:

COLUMN numeric_variable_name, (list of statistics keywords);

COLUMN (list of numeric_variable_names), statistic_keyword;

For example, the following SAS code lists six different statistics by section for the variable *average*. ***Note the format of the COLUMN statement and the similarity to the COLUMN statement when using the ACROSS option.***

```
proc report data=mysas.grades nowindows headskip spacing=3 colwidth=10;
      column section average,(mean std min max median p90);
      define section /group;
run;
quit;
```

	average					
section	mean	std	min	max	median	p90
10	88.10	3.00	83.67	93.67	88.17	92.17
20	89.97	3.29	85.00	95.67	89.83	94.33
30	90.07	2.84	86.67	97.00	89.17	94.67
40	89.40	4.01	82.33	95.67	90.17	94.17

Chapter 10: Outputting the Data – ODS

Whether specifically directed or not, all SAS output goes through ODS, the Output Delivery System. ***The ODS has two basic parts, the destination and the templates***. The destination tells SAS what file type is to be created, while the templates define the rules for formatting and presenting the data. The following is a list of the major destinations available in SAS.

- LISTING – basic SAS outout – used throughout this document
- OUTPUT – SAS dataset
- HTML – Hypertext Markup Language
- RTF – Rich Text Format for MSWord
- PRINTER – high resolution printer output
- PS – PostScript
- PCL – Printer Control Language
- PDF – Portable Document Format - ADOBE
- MARKUP – XML, CSV, etc.
- DOCUMENT – please see the following description

The MARKUP and DOCUMENT destinations are new to SAS version 9.0. The DOCUMENT destination allows output to be produced in a general format when a user is not sure of the particular desired format. Once a final format is determined, the DOCUMENT can be converted to the final format without recollecting all of the data. This can save considerable processing time.

There are two types of templates, table templates and style templates. Table templates determine a structure while the style templates determine how the data will look, font, color, etc. Styles can be customized but for a list of available pre-determined styles, use the following:

```
proc template;
      list styles;
run;
quit;
```

```
Listing of: SASHELP.TMPLMST
Path Filter is: Styles
Sort by: PATH/ASCENDING

Obs    Path                          Type
-------------------------------------------
 1     Styles                        Dir
 2     Styles.Analysis               Style
 3     Styles.Astronomy              Style
 4     Styles.Banker                 Style
 5     Styles.BarrettsBlue           Style
 6     Styles.Beige                  Style
 7     Styles.Brick                  Style
 8     Styles.Brown                  Style
 9     Styles.Curve                  Style
```

```
10      Styles.D3d                Style
11      Styles.Default            Style
12      Styles.EGDefault          Style
13      Styles.Education          Style
14      Styles.Electronics        Style
15      Styles.Festival           Style
16      Styles.FestivalPrinter    Style
17      Styles.Gears              Style
18      Styles.Journal            Style
19      Styles.Magnify            Style
20      Styles.Meadow             Style
21      Styles.MeadowPrinter      Style
22      Styles.Minimal            Style
23      Styles.Money              Style
24      Styles.NoFontDefault      Style
25      Styles.Normal             Style
26      Styles.NormalPrinter      Style
27      Styles.Printer            Style
28      Styles.Rsvp               Style
29      Styles.Rtf                Style
30      Styles.Sasweb             Style
31      Styles.Sasweb2            Style
32      Styles.Science            Style
33      Styles.Seaside            Style
34      Styles.SeasidePrinter     Link
35      Styles.Sketch             Style
36      Styles.Statdoc            Style
37      Styles.Statistical        Style
38      Styles.Theme              Style
39      Styles.Torn               Style
40      Styles.Watercolor         Style
41      Styles.blockPrint         Style
42      Styles.fancyPrinter       Style
43      Styles.sansPrinter        Style
44      Styles.sasdocPrinter      Style
45      Styles.serifPrinter       Style
```

To select an overall type of output use the ODS statement. The general format is:

ODS *destination* FILE='filename.ext' *options;*

The ***destinations*** were listed above. The ***filename.ext*** is the name of the output disk file with the appropriate extension. ***Options*** will vary based on the destination. However, STYLE= is one of the available options for almost all destinations. For example,

ODS RTF FILE='grades.rtf' STYLE=Education;

ODS HTML FILE='grades.html' STYLE=Banker;

The above statements will open the file *grades* with either the extension RTF or HTML and produce the report with the stated style. When finished, issue the following statement to close the output file.

ODS *destination* CLOSE;

For example…

ODS RTF CLOSE;

ODS HTML CLOSE;

The PRINT, REPORT, and TABULATE procedures allow a STYLE option on the procedure statement line. When specifically using the STYLE option in particular procedures such as PRINT or REPORT, there are two parts, the location list and the attribute list. By location is meant the particular location on the report. The location keywords may vary from procedure to procedure. The following are the location keywords for the PRINT procedure.

DATA
HEADER
OBS
OBSHEADER
TOTAL
GRANDTOTAL

Style attributes include font, color, bold, italic, etc. The general format within the procedure line is:

STYLE(location_list) = {style_attribute= value};

The following will show output data with blue characters on a yellow background.

```
ods rtf file='c:\mysasdata\grades.rtf';

proc print data=mysas.grades style(data)={foreground=blue background=yellow};
      var name average;
run;
quit;

ods rtf close;
```

Note: The style attributes are surrounded by { } and NOT parentheses, while the location list is surrounded by parentheses.

The following table is a portion of the rtf output from the above example.

Obs	name	average
1	George Washington	93.67
2	John Adams	89.33
3	Thomas Jefferson	88.00
4	James Madison	85.33
5	James Monroe	83.67

The STYLE option is also available on a variable by variable basis. Just attach the STYLE option to the VAR or DEFINE statement for the PRINT and REPORT procedures respectively.

Different styles are also available with the TITLE and FOOTNOTE statements. However, the format is simpler. The general format is:

TITLE options 'Text String 1' options 'Text String2' … ;

In the options portion use the style attribute keywords such as COLOR=, BCOLOR=, FONT=, etc. Different options can apply to different portions of the text. For example,

TITLE COLOR=BLUE 'The quick ' COLOR=RED 'red fox' COLOR=GREEN 'jumped over the fence';

will show as:

The quick red fox jumped over the fence

Chapter 11: PROC MEANS

The PROC MEANS procedure allows a user to get a number of statistics for a given dataset with very few instructions. With just the following statements, the number, mean, standard deviation, minimum and maximum values for each numeric variable in the entire dataset will be outputted.

```
proc means data=mysas.grades;
run;
quit;
```

The MEANS Procedure

Variable	N	Mean	Std Dev	Minimum	Maximum
section	40	25.0000000	11.3227703	10.0000000	40.0000000
grade1	40	89.3250000	5.5487929	80.0000000	99.0000000
grade2	40	90.1250000	5.7342314	81.0000000	99.0000000
grade3	40	88.7000000	6.3698005	80.0000000	100.0000000
average	40	89.3833333	3.2829960	82.3333333	97.0000000

These statistics for some numeric variables may not make sense, such as "section" in the example above. Specific numeric variables can be chosen using the VAR statement. Note that the following example is the same except for the absence of the "section" statistics.

```
proc means data=mysas.grades;
      var grade1 grade2 grade3 average;
run;
quit;
```

The MEANS Procedure

Variable	N	Mean	Std Dev	Minimum	Maximum
grade1	40	89.3250000	5.5487929	80.0000000	99.0000000
grade2	40	90.1250000	5.7342314	81.0000000	99.0000000
grade3	40	88.7000000	6.3698005	80.0000000	100.0000000
average	40	89.3833333	3.2829960	82.3333333	97.0000000

At times it is desired to have the statistics for subgroups within the dataset. While the mean of the variable "section", in the above example, has no meaning, the mean of the other numeric variables within each section may have a lot of meaning. By using the BY statement or the CLASS statement, statistics for subgroups defined by the variable(s) associated with the BY or CLASS statement can be obtained. The following shows the statistics for each unique "section".

```
proc means data=mysas.grades;
      by section;
run;
quit;
```

```
---------------------------------------- section=10 ----------------------------------------

                                The MEANS Procedure

Variable     N            Mean         Std Dev         Minimum         Maximum
-------------------------------------------------------------------------------------------
grade1      10      88.2000000       4.3665394      82.0000000      95.0000000
grade2      10      87.3000000       4.2176876      81.0000000      94.0000000
grade3      10      88.8000000       7.0521864      81.0000000     100.0000000
average     10      88.1000000       2.9981476      83.6666667      93.6666667
-------------------------------------------------------------------------------------------

---------------------------------------- section=20 ----------------------------------------

Variable     N            Mean         Std Dev         Minimum         Maximum
-------------------------------------------------------------------------------------------
grade1      10      87.2000000       4.9844202      82.0000000      96.0000000
grade2      10      92.7000000       5.7551909      82.0000000      98.0000000
grade3      10      90.0000000       6.2538877      80.0000000      99.0000000
average     10      89.9666667       3.2940275      85.0000000      95.6666667
-------------------------------------------------------------------------------------------

---------------------------------------- section=30 ----------------------------------------

Variable     N            Mean         Std Dev         Minimum         Maximum
-------------------------------------------------------------------------------------------
grade1      10      93.3000000       4.7152236      85.0000000      98.0000000
grade2      10      90.9000000       6.3148854      81.0000000      99.0000000
grade3      10      86.0000000       5.7542255      81.0000000      97.0000000
average     10      90.0666667       2.8406224      86.6666667      97.0000000
-------------------------------------------------------------------------------------------

---------------------------------------- section=40 ----------------------------------------

Variable     N            Mean         Std Dev         Minimum         Maximum
-------------------------------------------------------------------------------------------
grade1      10      88.6000000       6.5353738      80.0000000      99.0000000
grade2      10      89.6000000       5.8727241      81.0000000      96.0000000
grade3      10      90.0000000       6.4807407      83.0000000      98.0000000
average     10      89.4000000       4.0086327      82.3333333      95.6666667
-------------------------------------------------------------------------------------------
```

Note that the dataset must be sorted in the order of the variable(s) used in the BY statement. If the dataset is NOT sorted, the CLASS statement MUST be used.

```
proc means data=mysas.grades;
      class section;
run;
quit;
```

The MEANS Procedure

section	N Obs	Variable	N	Mean	Std Dev	Minimum	Maximum
10	10	grade1	10	88.2000000	4.3665394	82.0000000	95.0000000
		grade2	10	87.3000000	4.2176876	81.0000000	94.0000000
		grade3	10	88.8000000	7.0521864	81.0000000	100.0000000
		average	10	88.1000000	2.9981476	83.6666667	93.6666667
20	10	grade1	10	87.2000000	4.9844202	82.0000000	96.0000000
		grade2	10	92.7000000	5.7551909	82.0000000	98.0000000
		grade3	10	90.0000000	6.2538877	80.0000000	99.0000000
		average	10	89.9666667	3.2940275	85.0000000	95.6666667
30	10	grade1	10	93.3000000	4.7152236	85.0000000	98.0000000
		grade2	10	90.9000000	6.3148854	81.0000000	99.0000000
		grade3	10	86.0000000	5.7542255	81.0000000	97.0000000
		average	10	90.0666667	2.8406224	86.6666667	97.0000000
40	10	grade1	10	88.6000000	6.5353738	80.0000000	99.0000000
		grade2	10	89.6000000	5.8727241	81.0000000	96.0000000
		grade3	10	90.0000000	6.4807407	83.0000000	98.0000000
		average	10	89.4000000	4.0086327	82.3333333	95.6666667

The default statistics are N, mean, standard deviation, minimum, and maximum. However, by placing one or more of the following statistics on the PROC MEANS statement line, only the statistics listed on the statement line will be shown.

Descriptive statistic keywords	
CLM	RANGE
CSS	SKEWNESS|SKEW
CV	STDDEV|STD
KURTOSIS|KURT	STDERR
LCLM	SUM
MAX	SUMWGT
MEAN	UCLM
MIN	USS
N	VAR
NMISS	
Quantile statistic keywords	
MEDIAN|P50	Q3|P75
P1	P90
P5	P95
P10	P99
Q1|P25	QRANGE
Hypothesis testing keywords	
PROBT	T

The following two examples show selected percentile statistics across the dataset for the variables listed.

```
proc means data=mysas.grades median p90;
      var grade1 grade2 grade3 average;
run;
quit;
```

The MEANS Procedure

Variable	Median	90th Pctl
grade1	88.0000000	97.5000000
grade2	90.5000000	97.5000000
grade3	86.5000000	98.0000000
average	89.1666667	93.3333333

```
proc means data=mysas.grades p25 p50 p75;
      var grade1 grade2 grade3 average;
run;
quit;
```

The MEANS Procedure

Variable	25th Pctl	50th Pctl	75th Pctl
grade1	85.0000000	88.0000000	94.0000000
grade2	84.5000000	90.5000000	95.0000000
grade3	83.5000000	86.5000000	95.0000000
average	87.1666667	89.1666667	91.5000000

The data shown as output from the PROC MEANS procedure can also be sent to a dataset by using the OUTPUT statement. The general form is:

OUTPUT OUT=dataset statistic(variable_list) = output_variable_name_list

The following shows an example with the data being sent to an output dataset called *percentiles.* The NOPRINT option stops the standard output from the PROC MEANS procedure.

```
proc means data=mysas.grades p25 p50 p75 NOPRINT;
      var grade1 grade2 grade3 average;
      output out=percentiles
      p25(grade1 grade2 grade3 average)=p25_g1 p25_g2 p25_g3 p25_avg
      p50(grade1 grade2 grade3 average)=p50_g1 p50_g2 p50_g3 p50_avg
      p75(grade1 grade2 grade3 average)=p75_g1 p75_g2 p75_g3 p75_avg ;

run;
quit;


proc print data=percentiles;
      format p25_g1 p25_g2 p25_g3 p25_avg 5.2;
      format p50_g1 p50_g2 p50_g3 p50_avg 5.2;
      format p75_g1 p75_g2 p75_g3 p75_avg 5.2;

run;
quit;
```

Obs	_TYPE_	_FREQ_	p25_g1	p25_g2	p25_g3	p25_avg	p50_g1	p50_g2	p50_g3	p50_avg	p75_g1	p75_g2	p75_g3	p75_avg
1	0	40	85.00	84.50	83.50	87.17	88.00	90.50	86.50	89.17	94.00	95.00	95.00	91.50

Chapter 12: PROC FREQ

The PROC FREQ procedure shows a user the counts for particular categories within the data. If the desired count is for categories within a single variable, called a one-way table, then the output will present a standard frequency and cumulative frequency table. The output will look similar to the output from PROC PRINT. If counts across multiple variables, called multi-way tables, are desired, the output format will be in table form.

```
proc freq data=mysas.grades;
run;
quit;
```

last_name	Frequency	Percent	Cumulative Frequency	Cumulative Percent
Adams	1	2.50	1	2.50
Arthur	1	2.50	2	5.00
Buchanan	1	2.50	3	7.50
Carter	1	2.50	4	10.00
Cleveland1	1	2.50	5	12.50
Cleveland2	1	2.50	6	15.00
Coolidge	1	2.50	7	17.50
Eisenhower	1	2.50	8	20.00
Fillmore	1	2.50	9	22.50
Ford	1	2.50	10	25.00
Garfield	1	2.50	11	27.50
Grant	1	2.50	12	30.00
Harding	1	2.50	13	32.50
Harrison	1	2.50	14	35.00
Hayes	1	2.50	15	37.50
Henry Harrison	1	2.50	16	40.00
Hoover	1	2.50	17	42.50
Howard	1	2.50	18	45.00
Jackson	1	2.50	19	47.50
Jefferson	1	2.50	20	50.00
Johnson	2	5.00	22	55.00
Kennedy	1	2.50	23	57.50
Lincoln	1	2.50	24	60.00
Madison	1	2.50	25	62.50
McKinley	1	2.50	26	65.00
Monroe	1	2.50	27	67.50
Nixon	1	2.50	28	70.00
Pierce	1	2.50	29	72.50
Polk	1	2.50	30	75.00
Quincy Adams	1	2.50	31	77.50
Reagan	1	2.50	32	80.00
Roosevelt	2	5.00	34	85.00
Taylor	1	2.50	35	87.50
Truman	1	2.50	36	90.00
Tyler	1	2.50	37	92.50
Van Buren	1	2.50	38	95.00
Washington	1	2.50	39	97.50
Wilson	1	2.50	40	100.00

The BY statement shows the frequency table for the variable(s) in the BY statement. However, the data must already be sorted in the order of the variable(s) within the BY statement. If the data is not in sorted order, use the TABLES statement.

The FREQ Procedure

section	Frequency	Percent	Cumulative Frequency	Cumulative Percent
10	10	25.00	10	25.00
20	10	25.00	20	50.00
30	10	25.00	30	75.00
40	10	25.00	40	100.00

The TABLES statement can handle one-way or n-way frequency tables but the data does not need to be in any sorted order. The asterisk between variables identifies the cross tabulation desired for multi-way tables. When there are more than 2 categories desired, the following table shows the rules for the proper syntax of the TABLES statement to obtain the desired outcome tables.

Table 2.8: Grouping Syntax

Request	Equivalent to
tables A*(B C);	tables A*B A*C;
tables (A B)*(C D);	tables A*C B*C A*D B*D;
tables (A B C)*D;	tables A*D B*D C*D;
tables A - - C;	tables A B C;
tables (A - - C)*D;	tables A*D B*D C*D;

As a memory technique, note that the rules above are similar to the distribution property in algebra.

Use the /MISSING option on the TABLES statement if a missing column is desired in the table. If not, a message showing the missing number will be shown at the bottom of the table, but no column will be included in the table itself.

Each observation will be considered a single count for the desired frequency combination unless the WEIGHT statement is employed. The WEIGHT statement identifies a variable whose value contains the desired count for the categorical combination on each particular observation.

The following example reflects a survey at a fitness club where males and females were asked whether they preferred a particular facility at the club, specifically the tennis courts or the swimming pool. The results were counted and stored in a data file called facilities. Using PROC FREQ, a table is displayed showing the resulting counts from the survey with row, column, and total percentages for each combination.

Note that if each individual survey was being tabulated, the WEIGHT statement would not be necessary. However, since only the summary numbers of each group are entered into the data statement, the WEIGHT statement is required within the PROC FREQ procedure.

```
data facilities;
      input facility$ gender$ count;
datalines;
tennis   male     51
pool     male     30
tennis   female   43
pool     female   48
;
run;

proc freq data=facilities order=data;
      weight count;
      tables gender*facility;
run;
```

Table of gender by facility

gender	facility		
Frequency Percent Row Pct Col Pct	tennis	pool	Total
male	51 29.65 62.96 54.26	30 17.44 37.04 38.46	81 47.09
female	43 25.00 47.25 45.74	48 27.91 52.75 61.54	91 52.91
Total	94 54.65	78 45.35	172 100.00

Note that row and column percentages can be suppressed with the NOROWand/or NOCOL options on the TABLES statement.

Now, suppose there are multiple site locations for the tennis court versus swimming pool survey. By structuring the TABLES statement as shown in the following example, the results for two sets of tables can be shown; the first is the gender distribution between locations and then the overall preference for tennis versus swimming between locations.

```
data facilities;
      input location $11. facility$ gender$ count;
datalines;
WChester    tennis   male     51
WChester    pool     male     30
WChester    tennis   female   43
WChester    pool     female   48
Downingtown tennis   male     45
Downingtown pool     male     20
Downingtown tennis   female   35
Downingtown pool     female   45

;
run;


proc freq data=facilities order=data;
      weight count;
      tables location*gender*facility;
run;
quit;
```

Table 1 of gender by facility
Controlling for location=WChester

gender	facility		
Frequency Percent Row Pct Col Pct	tennis	pool	Total
male	51 29.65 62.96 54.26	30 17.44 37.04 38.46	81 47.09
female	43 25.00 47.25 45.74	48 27.91 52.75 61.54	91 52.91
Total	94 54.65	78 45.35	172 100.00

Table 2 of gender by facility
Controlling for location=Downingtown

gender	facility		
Frequency Percent Row Pct Col Pct	tennis	pool	Total
male	45 31.03 69.23 56.25	20 13.79 30.77 30.77	65 44.83
female	35 24.14 43.75 43.75	45 31.03 56.25 69.23	80 55.17
Total	80 55.17	65 44.83	145 100.00

If a combination of the two locations is desired, the first example of PROC FREQ will produce those results.

```
proc freq data=facilities order=data;
      weight count;
      tables gender*facility;
run;
quit;
```

The FREQ Procedure

Table of gender by facility

gender	facility		
Frequency Percent Row Pct Col Pct	tennis	pool	Total
male	96 30.28 65.75 55.17	50 15.77 34.25 34.97	146 46.06
female	78 24.61 45.61 44.83	93 29.34 54.39 65.03	171 53.94
Total	174 54.89	143 45.11	317 100.00

Note that all values include both locations.

The OUTPUT option also exists for the PROC FREQ procedure. Please see PROC MEANS for a description and some examples. The options are different for the PROC FREQ procedure but the overall statement structure is the same.

Chapter 13: PROC TABULATE

The PROC TABULATE procedure is used to produce sophisticated tables. The general form of the procedure is:

PROC TABULATE;

CLASS class_variable(s);

TABLE page_dimension expression, row dimension expression, column dimension expression;

Note that the page, row, and column dimensions are separated by commas.

Just as in PROC FREQ and PROC MEANS the CLASS statement determines the categorical variables. The TABLE statement defines how to present the data with the row and column dimensions producing the standard two-dimension table which will be presented for each different category within the page dimension variable. This is similar to last example in PROC FREQ where the gender (row dimension) versus facility (column dimension) preference was shown within each location (page dimension).

There are a variety of operators available within the TABLE statement for each expression. The following is a table showing the available operators and a description of the resulting action.

Operator	**Action**
, comma	separates dimensions of the table
* asterisk	crosses elements within a dimension
blank space	concatenates elements within a dimension
= equal	overrides default cell format or assigns label to an element
( ) parentheses	groups elements and associates an operator with each concatenated element in the group
[ ] square brackets	groups the STYLE= option for crossing, and groups style attribute specifications within the STYLE= option

{ } braces	groups the STYLE= option for crossing, and groups style attribute specifications within the STYLE= option

Missing values are excluded from the table(s) by default. Use the MISSING option on the procedure statement line if the missing values need to be included.

Just as in PROC MEANS, summary statistics can be computed and shown. The numeric variable for which the statistic is to be calculated must be listed in a VAR statement. Then the desired statistic can be used within the TABLE statement.

The available statistics are:

COLPCTN	PCTSUM
COLPCTSUM	RANGE
CSS	REPPCTN
CV	REPPCTSUM
KURTOSIS | KURT	ROWPCTN
LCLM	ROWPCTSUM
MAX	SKEWNESS | SKEW
MEAN	STDDEV|STD
MIN	STDERR
N	SUM
NMISS	SUMWGT
PAGEPCTN	UCLM
PAGEPCTSUM	USS
PCTN	VAR

Quantile statistic keywords

MEDIAN|P50	Q3|P75
P1	P90
P5	P95

P10	P99
Q1|P25	QRANGE

Hypothesis testing keywords

PROBT	T

The following examples show the use of the statistics option. The ALL keyword will give an additional row for the overall values.

The first example, using the grades data, shows the mean by section and overall, for each of the 3 grades and the average. The second example shows the mean and median for grade3 and the average variables.

Note the use of the space operator between SECTION and ALL and the asterisk operator between the statistics and the grades on the TABLE statement.

```
proc tabulate data=mysas.grades;
      var grade1 grade2 grade3 average;
      class section ;
      table section all, mean*grade1 mean*grade2 mean*grade3 mean*average;
run;
quit;
```

	Mean	Mean	Mean	Mean
	grade1	grade2	grade3	average
section				
10	88.20	87.30	88.80	88.10
20	87.20	92.70	90.00	89.97
30	93.30	90.90	86.00	90.07
40	88.60	89.60	90.00	89.40
All	89.33	90.13	88.70	89.38

```
proc tabulate data=mysas.grades;
      var grade3 average;
      class section ;
      table section all, mean*grade3 median*grade3 mean*average
median*average ;
run;
quit;
```

	Mean	Median	Mean	Median
	grade3	grade3	average	average
section				
10	88.80	86.00	88.10	88.17
20	90.00	90.00	89.97	89.83
30	86.00	84.50	90.07	89.17
40	90.00	89.00	89.40	90.17
All	88.70	86.50	89.38	89.17

In many ways PROC TABULATE is similar to PROC FREQ. Here are two of the PROC FREQ examples using PROC TABULATE to produce the same table results. However, the row, column, and total percentages do not show automatically, only the tabulation values. The following is the gender* facility table for both locations. Only the column and row are used in the TABLE statement.

```
proc tabulate data=facilities order=data;
      class gender facility;
      table gender, facility;
      freq count;
run;
quit;
```

	facility	
	tennis	pool
	N	N
gender		
male	96.00	50.00
female	78.00	93.00

Note the different output format with the box around the data. Also, note that the FREQ keyword is used instead of the WEIGHT keyword to identify the count variable as the total accumulation for that grouping.

The following is an example showing the table for each of the two locations. Thus we need a page, row, and column expression for the TABLE statement.

```
proc tabulate data=facilities order=data;
      class location gender facility;
      table location, gender, facility;
      freq count;
run;
quit;
```

location WChester

	facility	
	tennis	pool
	N	N
gender		
male	51.00	30.00
female	43.00	48.00

location Downingtown

	facility	
	tennis	pool
	N	N
gender		
male	45.00	20.00
female	35.00	45.00

In the above example, the commas separate the page, row, and column class variables. It may be desired to only have a row and column table but to have two sets of class variables for each row variable. For example,

```
proc tabulate data=facilities order=data;
      class location gender facility;
      table location, gender facility;
      freq count;
run; quit;
```

	gender		facility	
	male	female	tennis	pool
	N	N	N	N
location				
WChester	81.00	91.00	94.00	78.00
Downingt	65.00	80.00	80.00	65.00

Note that the only difference in syntax is that there is no comma between gender and facility. This makes the location become the row variable and both gender and facility become column variables. There is no page variable.

Even though many of the same results can be produced by the PROC FREQ and other procedures, the PROC TABULATE contains a multitude of additional options available for formatting and presenting the tables, such as colors, a box around the data, STYLE options, etc.

Chapter 14: Preventing and Debugging Errors

While it is impossible to give a full list of rules to completely prevent errors or to solve every possible error scenario, the following ten rules will help.

1) Be sure to ***use an editor that color codes*** the different portions of the SAS code. This will immediately inform the programmer of a mistyped statement name, option name, etc.

2) Always ***check the end of each statement for the semi-colon***. A missing semi-colon is probably the most common syntactical error. Item 1 will also help with this.

3) ***Use spacing and indentations*** in order to make the code very readable for yourself and others. In addition to the readability, it will make it much easier to identify missing block ending statements and the like.

4) ***ALWAYS check the log file and each entry in the log file***. There are 3 different types of log file entries, errors, warnings, or informational. Each one must be checked. Even though SAS thinks that it is informational, it may actually be a logic error with what the programmer is trying to do.

5) ***Test each program in small sections***.

6) When testing large databases, ***use a small representative subset of the large database***. In many cases, this can be done by using the OBS and FIRSTOBS options.

7) When testing multi-level groupings, ***use a range of observations that includes breaks across the levels***, so that each level total or statistical summary value can be tested.

8) ***Use PUT statements*** to track specific values that do not appear to be correct.

9) ***Use the debugging option*** to research what is changing with each line of code.

10) ***Test, test, test…***

Chapter 15: Specific SAS Version 9 Changes

The following is a specific short list of specific new functions and new statement options now available in SAS version 9. A more extensive function list can be found in the "*What's New in the Base SAS 9.0, 9.1, 9.1.2, and 9.1.3 Language*" section of the SAS online documentation.

Functions

FIND function – looks for a given substring within a given string and returns the index position if found. Otherwise it returns a zero.

string_position = FIND (string, substring, <modifier>, <start_position>)

<modifer> can be

- 'i' for ignore case or
- 't' for trim blanks from both string and substring

<start_position> indicates where in the string to start the search with position 1 being the default value

```
data _NULL_;

      comment1 = 'This is a test';
      position = find(comment1,'th');
                  /* not found -     position = 0 */

      put position;



      comment1 = 'This is a test';
      position = find(comment1,'th','i');
                  /* ignore case -    position = 1   */

      put position;



      comment1 = 'This is a test';
      position = find(comment1,'t');
                  /* skip capital T and find lower case t - position = 11 */

      put position;



      comment1 = 'This is a test';
      position = find(comment1,'t','i');
                  /* ignore case - find any t -    position = 1 */

      put position;
```

```
run;
quit;
```

TRANWRD function – replaces a given target string within a source string with a given replacement string, returning a new resultant string.

new-string = TRNWORD (source, target, replacement)

```
data test;

      input name $11.;

      new_name = tranwrd(name,'Mr.','Dr.');

datalines;
Mr. Smith
Mr. Jones
Mr. Adams
Mr. Franks
;
run;
quit;

proc print data=test;
run;
quit;
```

Obs	name	new_name
1	Mr. Smith	Dr. Smith
2	Mr. Jones	Dr. Jones
3	Mr. Adams	Dr. Adams
4	Mr. Franks	Dr. Franks

PROPCASE function – converts all words in a string to their proper case, returning a new resultant string

new_string = PROPCASE (string, <delimiter>)

<delimiter> identifies valid special characters, such as hyphens, etc.

```
data _NULL_;

      comment1 = 'this is a test';

      new_comment = propcase(comment1);

      put new_comment;

run;
quit;
```

```
Output from the PUT statement…

This Is A Test
```

CATX function - concatenates multiple strings by trimming the blanks and inserting a separator between the strings

new_cat_string = CATX (separator, string1, string2, …, stringn)

```
data test;
      input ssprefix $5. ssmiddle $5. sssuffix $5.;

      new_ss = CATX('-',ssprefix,ssmiddle, sssuffix);
                        /* trim, add hyphen, and concatenate */
datalines;
111   22    3333
222   88    4444
333   77    5555
444   99    7777
;
run;
quit;

proc print data=test;
run;
quit;
```

Obs	ssprefix	ssmiddle	sssuffix	new_ss
1	111	22	3333	111-22-3333
2	222	88	4444	222-88-4444
3	333	77	5555	333-77-5555
4	444	99	7777	444-99-7777

WEEKDAY function – returns a numerical value for the day of the week for a given date.
1 = Sunday, 2 = Monday, …, 7 = Saturday.

num_day_of_week = WEEKDAY (date)

Options

PAGEBY – will start a new page in the PRINT procedure for each change in the BY variable.

PAGEBY variable

BLANKLINES – as an option in PRINT procedure, this option will automatically insert a blank line after every "n" lines of data.

BLANKLINES=n

CROSSLIST – as an option in the TABLES statement, this option will produce tables information across the page in report style, instead of in table format.

CROSSLIST

Index

K

L

M

N

O

P

R

S

T

U

V

W

X

Y