**Chapter 1 Summary**

**Text Summary**

**Components of SAS Programs**

SAS programs consist of two types of steps: DATA steps and PROC (procedure) steps. These two steps, alone or combined, form most SAS programs. A SAS program can consist of a DATA step, a PROC step, or any combination of DATA and PROC steps. DATA steps typically create or modify SAS data sets, but they can also be used to produce custom-designed reports. PROC steps are pre-written routines that enable you to analyze and process the data in a SAS data set and to present the data in the form of a report. They sometimes create new SAS data sets that typically contain the results of the procedure.

**Characteristics of SAS Programs**

SAS programs consist of SAS statements. A SAS statement usually begins with a SAS keyword and always ends with a semicolon. A DATA step begins with the keyword DATA. A PROC step begins with the keyword PROC. SAS statements are free-format, so they can begin and end anywhere on a line. One statement can continue over several lines, and several statements can be on a line. Blanks or special characters separate “words” in a SAS statement.

**Processing SAS Programs**

When you submit a SAS program, SAS reads SAS statements and checks them for errors. When it encounters a subsequent DATA, PROC, or RUN statement, SAS executes the previous step in the program.

Each time a step is executed, SAS generates a log of the processing activities and the results of the processing. The SAS log collects messages about the processing of SAS programs and about any errors that occur.

The results of processing can vary. Some SAS programs open an interactive window or invoke procedures that create output in the form of a report. Other SAS programs perform tasks such as sorting and managing data, which have no visible results other than messages in the log.

**SAS Libraries**

Every SAS file is stored in a SAS library, which is a collection of SAS files such as SAS data sets and catalogs. In the Windows and UNIX environments, a SAS library is typically a group of SAS files in the same folder or directory.

Depending on the libref you use, you can store SAS files in a temporary SAS library or in permanent SAS libraries.

* Temporary SAS files that are created during the session are held in a special work space that is assigned the default libref Work. If you don't specify a libref when you create a file (or if you specify Work), the file is stored in the temporary SAS library. When you end the session, the temporary library is deleted.
* To store files permanently in a SAS library, you assign it a libref other than the default Work. For example, by assigning the libref sasuser to a SAS library, you specify that files within the library are to be stored until you delete them.

**Referencing SAS Files**

To reference a SAS file, you use a two-level name, *libref.filename*. In the two-level name, *libref* is the name for the SAS library that contains the file, and *filename* is the name of the file itself. A period separates the libref and filename.

To reference temporary SAS files, you specify the default libref Work, a period, and the filename. Alternatively, you can simply use a one-level name (the filename only) to reference a file in a temporary SAS library. Referencing a SAS file in any library *except Work*indicates that the SAS file is stored permanently.

SAS data set names can be 1 to 32 characters long, must begin with a letter (A-Z, either uppercase or lowercase) or an underscore (\_), and can continue with any combination of numerals, letters, or underscores.

**SAS Data Sets**

For many of the data processing tasks that you perform with SAS, you access data in the form of a SAS data set and use SAS programs to analyze, manage, or present the data. Conceptually, a SAS data set is a file that consists of two parts: a descriptor portion and a data portion. Some SAS data sets also contain one or more indexes, which enable SAS to locate records in the data set more efficiently.

The descriptor portion of a SAS data set contains property information about the data set.

The data portion of a SAS data set is a collection of data values that are arranged in a rectangular table. Observations in the data set correspond to rows or data lines. Variables in the data set correspond to columns. If a data value is unknown for a particular observation, a missing value is recorded in the SAS data set.

**Variable Attributes**

In addition to general information about the data set, the descriptor portion contains property information for each variable in the data set. The property information includes the variable's name, type, and length. A variable's type determines how missing values for a variable are displayed by SAS. For character variables, a *blank* represents a missing value. For numeric variables, a *period* represents a missing value. You can also specify format, informat, and label properties for variables.

**Using the Main SAS Windows**

You use the following windows to explore and manage your files, to enter and submit SAS programs, to view messages, and to view and manage your output.

***Windows and How They Are Used***

|  |  |
| --- | --- |
| **Use this window ...** | **To ...** |
| Explorer | view your SAS files  create new libraries and SAS files  perform most file management tasks such as moving, copying, and deleting files  create shortcuts to files that were not created with SAS |
| Enhanced Editor (code editing window) | enter, edit, and submit SAS program  **Note:** The Enhanced Editor window is available only in the Windows operating environment. |
| Program Editor (code editing window) | enter, edit, and submit SAS programs |
| Log | view messages about your SAS session and about any SAS programs that you submit |
| Output | browse output from SAS programs |
| Results | navigate and manage output from SAS programs  view, save, and print individual items of output |

**Points to Remember**

* Before referencing SAS files, you must assign a name (libref, or library reference) to the library in which the files are stored (or specify that SAS is to assign the name automatically).
* You can store SAS files either temporarily or permanently.
* Variable names follow the same rules as SAS data set names. However, your site may choose to restrict variable names to those valid in Version 6 SAS, to uppercase variable names automatically, or to remove all restrictions on variable names.

**Chapter 2 Summary**

**Text Summary**

**Referencing SAS Files in SAS Libraries**

To reference a SAS file, you assign a libref (library reference) to the SAS library in which the file is stored. Then you use the libref as the first part of the two-level name (*libref.filename*) for the file. To assign a libref, you can submit a LIBNAME statement. You can store the LIBNAME statement with any SAS program to assign the libref automatically when you submit the program. The LIBNAME statement assigns the libref for the current SAS session only. You must assign a libref each time you begin a SAS session in order to access SAS files that are stored in a permanent SAS library other than clinic. (Work is the default libref for a temporary SAS library.)

You can also use the LIBNAME statement to reference data in files that were created with other software products, such as database management systems. SAS can write to or read from the files by using the appropriate engine for that file type. For some file types, you need to tell SAS which engine to use. For others, SAS automatically chooses the appropriate engine.

**Viewing Librefs**

The librefs that are in effect for your SAS session are listed under Libraries in the Explorer window. To view details about a library, double-click **Libraries** (or select **Libraries**→**Open** from the pop-up menu). Then select **View**→**Details**. The library's name, engine, host pathname, and date are listed under Active Libraries.

**Viewing the Contents of a Library**

To view the contents of a library, double-click the library name in the Explorer window (or select the library name and then select **Open** from the pop-up menu). Files contained in the library are listed under Contents.

**Viewing the Contents of a File**

If you are working in a windowing environment, you can display the contents of a file by double-clicking the filename (or selecting the filename and then selecting **Open** from the pop-up menu) under Contents in the Explorer window. If you select a SAS data set, its contents are displayed in the VIEWTABLE window.

If you are working in the z/OS operating environment, you can type **?** in the selection field next to a filename in the Explorer window to display a pop-up menu with a list of options for working with that file.

**Listing the Contents of a Library**

To list the contents of a library, use the CONTENTS procedure. Append a period and the \_ALL\_ option to the libref to get a listing of all files in the library. Add the NODS option to suppress detailed information about the files. As an alternative to PROC CONTENTS, you can use PROC DATASETS.

**Specifying Result Formats**

In desktop operating environments, you can choose to create your SAS procedure output as an HTML document, a listing (traditional SAS output), or both. You choose the results format(s) that you prefer in the Preferences window. Your preferences are saved until you modify them, and they apply to all output that is created during your SAS session. To open this window, select**Tools**→**Options**→**Preferences**. Then click the **Results** tab. Choose **Create listing**, **Create HTML**, or both.

If you choose **Create HTML**, then each HTML file is displayed in the browser that you specify (in the Windows operating environment, the internal browser is the Results Viewer window). HTML files are stored in the location that you specify and are by default incrementally named sashtml.htm, sashtml1.htm, sashtml2.htm, and so on throughout your SAS session. To specify where HTML files are stored, type a path in the **Folder** box (or click **Browse** to locate a pathname). If you prefer to store your HTML files temporarily and to delete them at the end of your SAS session, click **Use WORK** folder instead of specifying a folder. To specify the presentation style for HTML output, you can select an item in the **Style** box.

**Setting System Options**

For your LISTING output, you can also control the appearance of your output by setting system options such as line size, page size, the display of page numbers, and the display of the date and time. (These options do not affect the appearance of HTML output.)

All SAS system options have default settings that are used unless you specify otherwise. For example, page numbers are automatically displayed (unless your site modifies this default). To modify system options, you submit an OPTIONS statement. You can place an OPTIONS statement anywhere in a SAS program to change the current settings. Because the OPTIONS statement is global, the settings remain in effect until you modify them or until you end your SAS session.

If you use two-digit year values in your SAS data lines, you must be aware of the YEARCUTOFF= option to ensure that you are properly interpreting two-digit years in your SAS program. This option specifies which 100-year span is used to interpret two-digit year values.

To specify the observations to process from SAS data sets, you can use the FIRSTOBS= and OBS= options.

You can also use the SAS System Options window to set system options.

**Additional Features**

You can set a number of additional SAS system options that are commonly used.

**Syntax**

**LIBNAME** *libref 'SAS-data-library'***;**

**LIBNAME** *libref engine'SAS-data-library'***;**

**OPTIONS** *options***;**

**PROC CONTENTS DATA=** *libref.***\_ALL\_ NODS;**

**PROC DATASETS;**

**CONTENTS DATA=** *libref.***\_ALL\_ NODS;**

QUIT;

**Points to Remember**

* LIBNAME and OPTIONS statements remain in effect for the current SAS session only.
* When you work with date values, check the default value of the YEARCUTOFF= system option and change it if necessary.

**Chapter 3 Summary**

**Text Summary**

**Opening a Stored SAS Program**

A SAS program that is stored in an external file can be opened in the code editing window using

* file shortcuts
* My Favorite Folders
* the Open window
* the INCLUDE command.

**Editing SAS Programs**

SAS programs consist of SAS statements. Although you can write SAS statements in almost any format, a consistent layout enhances readability and enables you to understand the program's purpose.

In the Windows operating environment, the Enhanced Editor enables you to enter and view text and

* select one or more lines of text
* use color-coding to identify code elements
* automatically indent the next line when you press the Enter key
* collapse and expand sections of SAS procedures, DATA steps, and macros
* bookmark lines of code for easy access to different sections of your program.

Using the Editor window, you can also find and replace text, use abbreviations, open multiple views of a file, and set Enhanced Editor options.

The Program Editor window enables you to edit your programs just as you would with a word processing program. You can also use text editor commands and block text editor commands to edit SAS programs. Activating line numbers can make it easier for you to edit your program regardless of your operating environment.

Remember that SAS statements disappear from the Program Editor window when they are submitted. However, you can recall a program to the Program Editor window.

To save your SAS program to an external file, first activate the Program Editor window and select **File**→**Save As**. Then specify the name of your program in the Save As window.

**Clearing SAS Programming Windows**

Text and output accumulate in the Editor, Program Editor, Log, and Output windows throughout your SAS session. You can clear a window by selecting **Edit**→**Clear All**.

**Interpreting Error Messages**

When a SAS program that contains errors is submitted, error messages appear in the Logwindow. SAS can detect three types of errors: syntax, execution-time, and data. This chapter focuses on identifying and resolving common syntax errors.

**Correcting Errors**

To modify programs that contain errors, you can correct the errors in the code editing window. In the Program Editor window, you need to recall the submitted statements before editing them.

Before resubmitting a revised program, it is a good idea to clear the messages from the Logwindow so that you don't confuse old messages with the new. You can delete any error-free steps from a revised program before resubmitting it.

**Resolving Common Problems**

You might need to resolve several types of common problems: missing RUN statements, missing semicolons, unbalanced quotation marks, and invalid options.

**Points to Remember**

* It's a good idea to begin DATA steps, PROC steps, and RUN statements on the left and to indent statements within a step.
* End each step with a RUN statement.
* Review the messages in the Log window each time you submit a SAS program.
* You can delete any error-free steps from a revised program before resubmitting it, or you can submit only the revised steps in a program.

**Chapter 4 Summary**

**Text Summary**

**Creating a Basic Report**

To list the information in a SAS data set, you can use PROC PRINT. You use the PROC PRINT statement to invoke the PRINT procedure and to specify the data set that you are listing. Include the DATA= option to specify the data set that you are using. By default, PROC PRINT displays all observations and variables in the data set, includes a column for observation numbers on the far left, and displays observations and variables in the order in which they occur in the data set. If you use a LABEL statement with PROC PRINT, you must specify the LABEL option in the PROC PRINT statement.

To refine a basic report, you can

* select which variables and observations are processed
* sort the data
* generate column totals for numeric variables.

**Selecting Variables**

You can select variables and control the order in which they appear by using a VAR statement in your PROC PRINT step. To remove the Obs column, you can specify the NOOBS option in the PROC PRINT statement. As an alternative, you can replace the Obs column with one or more variables by using the ID statement.

**Selecting Observations**

The WHERE statement enables you to select observations that meet a particular condition in the SAS data set. You use comparison operators to express a condition in the WHERE statement. You can also use the CONTAINS operator to express a condition in the WHERE statement. To specify a condition based on the value of a character variable, you must enclose the value in quotation marks, and you must write the value with lower and uppercase letters exactly as it appears in the data set. You can also use the WHERE statement to select a subset of observations based on multiple conditions. To link a sequence of expressions into compound expressions, you use logical operators. When you test for multiple values of the same variable, you specify the variable name in each expression. You can use the IN operator as a convenient alternative. To control how compound expressions are evaluated, you can use parentheses.

**Sorting Data**

To display your data in sorted order, you use PROC SORT to sort your data before using PROC PRINT to create reports. By default, PROC SORT sorts the data set specified in the DATA= option permanently. If you do not want your data to be sorted permanently, you must create an output data set that contains the data in sorted order. The OUT= option in the PROC SORT statement specifies an output data set. If you need sorted data to produce output for only one SAS session, you should specify a temporary SAS data set as the output data set. The BY statement, which is required with PROC SORT, specifies the variable(s) whose values are used to sort the data.

**Generating Column Totals**

To total the values of numeric variables, use the SUM statement in the PROC PRINT step. You do not need to specify the variables in a VAR statement if you specify them in the SUM statement. Column totals appear at the end of the report in the same format as the values of the variables. To produce subtotals, add both the SUM statement and the BY statement to your PROC PRINT step. To show BY variable headings only once, use an ID and BY statement together with the SUM statement. As another enhancement to your report, you can request that each BY group be printed on a separate page by using the PAGEBY statement.

**Double Spacing Output**

To double space your SAS LISTING output, you can specify the DOUBLE option in the PROC PRINT statement.

**Specifying Titles**

To make your report more meaningful and self-explanatory, you can associate up to 10 titles with procedure output by using TITLE statements anywhere within or preceding the PROC step. After you define a title, it remains in effect until you modify it, cancel it, or end your SAS session. Redefining a title line cancels any higher-numbered title lines. To cancel all previous titles, specify a null TITLE statement (a TITLE statement with no number or text).

**Specifying Footnotes**

To add footnotes to your output, you can use the FOOTNOTE statement. Like TITLE statements, FOOTNOTE statements are global. Footnotes appear at the bottom of each page of procedure output, and footnote lines are *pushed up* from the bottom. The FOOTNOTE statement that has the largest number appears on the bottom line. After you define a footnote, it remains in effect until you modify it, cancel it, or end your SAS session. Redefining a footnote line cancels any higher-numbered footnote lines. To cancel all previous footnotes, specify a null FOOTNOTE statement (a FOOTNOTE statement with no number or text).

**Assigning Descriptive Labels**

To label the columns in your report with more descriptive text, you use the LABEL statement, which assigns a descriptive label to a variable. To display the labels that were assigned in a LABEL statement, you must specify the LABEL option in the PROC PRINT statement.

**Formatting Data Values**

To make data values more understandable when they are displayed in your procedure output, you can use the FORMAT statement, which associates formats with variables. The FORMAT statement remains in effect only for the PROC step in which it appears. Formats affect only how the data values appear in output, not the actual data values as they are stored in the SAS data set. All SAS formats can specify the total field width (w) that is used for displaying the values in the output. For numeric variables you can also specify the number of decimal places (d), if any, to be displayed in the output.

**Using Permanently Assigned Labels and Formats**

You can take advantage of permanently assigned labels or formats without adding LABEL or FORMAT statements to your PROC step.

**Syntax**

**LIBNAME** *libref 'SAS-data-library'***;**

**OPTIONS** *options***;**

**PROC SORT DATA=***SAS-data-set* **OUT=***SAS-data-set***;**

**BY** *variable(s)***;**

**RUN;**

**TITLE**<*n* > *'text'***;**

**FOOTNOTE**<*n*>*'text'***;**

**PROC PRINT DATA=***SAS-data-set*

**BY**<DESCENDING>*BY-variable-1*<...<DESCENDING><*BY-variable-n*>>  
<NOTSORTED>;

**PAGEBY***BY-variable*;

**NOOBS LABEL DOUBLE;**

**ID** *variable(s)***;**

**VAR** *variable(s)***;**

**WHERE** *where-expression***;**

**SUM** *variable(s)***;**

**LABEL** *variable1***=***'label1'* *variable2***=***'label2'* ...**;**

**FORMAT** *variable(s) format-name;*

**RUN;**

**Sample Program**

libname clinic 'c:\stress\labdata';

options nodate number pageno=15;

proc sort data=clinic.stress out=work.maxrates;

by maxhr;

where tolerance='I' and resthr>60;

run;

title 'August Admission Fees';

footnote 'For High Activity Patients';

proc print data=work.maxrates label double noobs;

id name;

var resthr maxhr rechr;

label rechr='Recovery HR';

run;

proc print data=clinic.admit label;

var actlevel fee;

where actlevel='HIGH';

label fee='Admission Fee';

sum fee;

format fee dollar4.;

run;

**Points to Remember**

* VAR, WHERE, SUM, FORMAT and LABEL statements remain in effect only for the PROC step in which they appear.
* If you don't use the OUT= option, PROC SORT permanently sorts the data set specified in the DATA= option.
* TITLE and FOOTNOTE statements remain in effect until you modify them, cancel them, or end your SAS session.
* Be sure to match the quotation marks that enclose the text in TITLE, FOOTNOTE, and LABEL statements.
* To display labels in PRINT procedure output, remember to add the LABEL option to the PROC PRINT statement.
* To permanently assign labels or formats to data set variables, place the LABEL or FORMAT statement inside the DATA step.

**Chapter 5 Summary**

**Text Summary**

**Raw Data Files**

A raw data file is an external file whose records contain data values that are organized in fields. The raw data files in this chapter contain fixed fields.

**Steps to Create a SAS Data Set**

You need to follow several steps to create a SAS data set using raw data. You need to

* reference the raw data file to be read
* name the SAS data set
* identify the location of the raw data
* describe the data values to be read.

**Referencing a SAS Library**

To begin your program, you might need to use a LIBNAME statement to reference the SAS library in which your data set will be stored.

**Writing a DATA Step Program**

The DATA statement indicates the beginning of the DATA step and names the SAS data set(s) to be created.

Next, you specify the raw data file by using the INFILE statement. The OBS= option in the INFILE statement enables you to process a specified number of observations.

This chapter teaches column input, the most simple input style. Column input specifies actual column locations for data values. The INPUT statement describes the raw data to be read and placed into the SAS data set.

**Submitting the Program**

When you submit the program, you can use the OBS= option with the INFILE statement to verify that the correct data is being read before reading the entire data file.

After you submit the program, view the log to check the DATA step processing. You can then print the data set by using the PROC PRINT procedure.

Once you've checked the log and verified your data, you can modify the DATA step to read the entire raw data file by removing the OBS= option from the INFILE statement.

If you are working with a raw data file that contains invalid data, the DATA step continues to execute. Unlike syntax errors, invalid data errors do not cause SAS to stop processing a program. If you have a way to edit the invalid data, it's best to correct the problem and rerun the DATA step.

**Creating and Modifying Variables**

To modify existing values or to create new variables, you can use an assignment statement in any DATA step. Within assignment statements, you can specify any SAS expression.

You can use date constants to assign dates in assignment statements. You can also use SAS time constants and SAS datetime constants in assignment statements.

**Subsetting Data**

To process only observations that meet a specified condition, use a subsetting IF statement in the DATA step.

**Reading Instream Data**

To read instream data lines instead of an external file, use a DATALINES statement, a CARDS statement, or a LINES statement and enter data directly in your SAS program. Omit the RUN at the end of the DATA step.

**Creating a Raw Data File**

When the goal of your SAS program is to create a raw data file and not a SAS data set, it is inefficient to list a data set name in the DATA statement. Instead use the keyword \_NULL\_, which allows the power of the DATA step without actually creating a SAS data set. A SET statement specifies the SAS data set that you want to read from.

You can use the FILE and PUT statements to write out the observations from a SAS data set to a raw data file just as you used the INFILE and INPUT statements to create a SAS data set. These two sets of statements work almost identically.

**Microsoft Excel Files**

You can read Excel worksheets by using the SAS/ACCESS LIBNAME statement.

**Steps to Create a SAS Data Set from Excel Data**

You need to follow several steps to create a SAS data set using Excel. You need to

* provide a name for the new SAS data set
* provide the location or name of the libref and Excel worksheet

**Referencing an Excel Workbook**

To begin your program, you need to use a LIBNAME statement to reference the Excel workbook.

**Writing a DATA Step Program**

The DATA statement indicates the beginning of the DATA step and names the SAS data set(s) to be created.

Next, you specify the Excel worksheet to be read by using the SET statement. You must use a SAS name literal since SAS uses the special character ($) to name Excel worksheets.

**Submitting the Program**

When you submit the program, you can use the CONTENTS procedure to explore the new library and contents.

After you submit the program, view the log to check the DATA step processing. You can then print the data sets created from the Excel worksheets by using the PROC PRINT procedure.

Once you've checked the log and verified your data, you can modify the DATA step along with the WHERE statement to subset parts of the data as needed.

**Syntax**

**Reading Data from a Raw File or Reading Instream Data**

**LIBNAME** *libref 'SAS-data-library'***;**

**FILENAME** *fileref 'filename';*

**DATA** *SAS-data-set;*

**INFILE** *file-specification*<OBS=*n*>**;**

**INPUT** *variable* <$> *startcol-endcol...***;**

**IF** *expression***;**

*variable=expression;*

**DATALINES;**

*instream data goes here if used*

**;**

**RUN;** */\* not used with the DATALINES statement \*/*

**PROC PRINT DATA=** *SAS-data set***;**

**RUN;**

**Creating a Raw Data File**

**LIBNAME** *libref 'SAS-data-library'***;**

**DATA \_NULL\_;**

**SET** *SAS-data-set;*

**FILE** *file-specification;*

**PUT** *variable startcol-endcol...***;**

**RUN;**

**Reading Data from an Excel Workbook**

**LIBNAME** *libref '*<*location-of-Excel-workbook*>'**;**

**PROC CONTENTS DATA=** *libref.\_ALL\_***;**

**DATA** *SAS-data-set;*

**SET** *libref.'worksheet\_name$'n;*

**WHERE** *where-expression;*

**RUN;**

**PROC PRINT DATA=** *SAS-data set***;**

**RUN;**

**Sample Programs**

**Reading Data from an External File**

libname clinic 'c:\bethesda\patients\admit';

filename admit 'c:\clinic\patients\admit.dat';

data clinic.admittan;

infile admit obs=5;

input ID $ 1-4 Name $ 6-25 RestHR 27-29 MaxHR 31-33

RecHR 35-37 TimeMin 39-40 TimeSec 42-43

Tolerance $ 45;

if tolerance='D';

TotalTime=(timemin\*60)+timesec;

run;

proc print data=clinic.admittan;

run;

**Reading Instream Data**

libname clinic 'c:\bethesda\patients\admit';

data clinic.group1;

input ID $ 1-4 Name $ 6-25 RestHR 27-29 MaxHR 31-33

RecHR 35-37 TimeMin 39-40 TimeSec 42-43

Tolerance $ 45;

if tolerance='D';

TotalTime=(timemin\*60)+timesec;

datalines;

2458 Murray, W 72 185 128 12 38 D

2462 Almers, C 68 171 133 10 5 I

2501 Bonaventure, T 78 177 139 11 13 I

2523 Johnson, R 69 162 114 9 42 S

2539 LaMance, K 75 168 141 11 46 D

2544 Jones, M 79 187 136 12 26 N

2595 Warren, C 77 170 136 12 10 S

;

proc print data=clinic.group1;

run;

**Reading Excel Data**

libname sasuser 'c:\users\admit.xlsx' mixed=yes;

proc contents data=sasuser.\_all\_;

run;

proc print data=sasuser.'worksheet1$'n;

run;

**Creating an Excel Worksheet**

libname clinic 'c:\Users\mylaptop\admitxl.xlsx' mixed=yes;

data clinic.admit;

set work.admit;

run;

**Points to Remember**

* LIBNAME and FILENAME statements are global. Librefs and filerefs remain in effect until you change them, cancel them, or end your SAS session.
* For each field of raw data that you read into your SAS data set, you *must* specify the following in the INPUT statement: a valid SAS variable name, a type (character or numeric), a starting column, and if necessary, an ending column.
* When you use column input, you can read any or all fields from the raw data file, read the fields in any order, and specify only the starting column for variables whose values occupy only one column.
* Column input is appropriate only in some situations. When you use column input, your data *must* be standard character and numeric values, and these values *must* be in fixed fields. That is, values for a particular variable must be in the same location in all records.

**Chapter 6 Summary**

**Text Summary**

**How SAS Processes Programs**

A SAS DATA step is processed in two distinct phases. During the compilation phase, each statement is scanned for syntax errors. During the execution phase, the DATA step writes observations to the new data set.

**Compilation Phase**

At the beginning of the compilation phase, the input buffer and the program data vector are created. The program data vector is the area of memory where one observation at a time is processed. Two automatic variables are also created: \_N\_ counts the number of DATA step executions, and \_ERROR\_ signals the occurrence of a data error. DATA step statements are checked for syntax errors, such as invalid options or misspellings.

**Execution Phase**

During the execution phase, one record at a time from the input file is read into the input buffer, copied to the program data vector, and then written to the new data set as an observation. The DATA step executes once for each record in the input file, unless otherwise directed.

**Diagnosing Errors in the Compilation Phase**

Missing semicolons, misspelled keywords, and invalid options will cause syntax errors in the compilation phase. Detected errors are underlined and are identified with a number and message in the log. If SAS can interpret a syntax error, the DATA step compiles and executes; if SAS cannot interpret the error, the DATA step compiles but doesn't execute.

**Diagnosing Errors in the Execution Phase**

Illegal mathematical operations or processing a character variable as numeric will cause data errors in the execution phase. Depending on the type of error, the log might show a warning and might include invalid data from the input buffer or the program data vector, and the DATA step either stops or continues.

**Testing Your Programs**

To detect common errors and save development time, use the OBS= option in the INFILE statement to limit the number of observations that are read or created during the DATA step. You can also use the PUT statement to examine variable values and to generate your own message in the log. For information about Informats, see [Using Informats](http://proquestcombo.safaribooksonline.com.prox.lib.ncsu.edu/9781607649243/p13pda2mou79nzn183kz96dws5ea.htm).

**Syntax**

**DATA** <\_NULL\_|*SAS-data-set*>**;**

**INFILE** *file-specification* **OBS=***n***;**

**INPUT** *variable-1 informat-1*

<*. . .variable-n informat-n*;>

**PUT** *specification(s)***;**

**RUN;**

**Sample Programs**

data perm.update;

infile invent;

input Item $ 1-13 IDnum $ 15-19

InStock 21-22 BackOrd 24-25;

Total=instock+backord;

run;

data work.test;

infile loan;

input Code $ 1 Amount 3-10;

if code='1' then type='variable';

else if code='2' then type='fixed';

else put 'MY NOTE: invalid value: '

code=;

run;

**Points to Remember**

* Making, diagnosing, and resolving errors is part of the process of writing programs. However, checking for common errors will save you time and trouble. Make sure that
  + each SAS statement ends with a semicolon
  + filenames are spelled correctly
  + keywords are spelled correctly.
* In SAS output, missing numeric values are represented by periods, and missing character values are left blank.
* The order in which variables are defined in the DATA step determines the order in which the variables are stored in the data set.
* Standard character values can include numbers, but numeric values cannot include characters.

**Chapter 7 Summary**

**Text Summary**

**Invoking PROC FORMAT**

The FORMAT procedure enables you to substitute descriptive text for the values of variables. The LIBRARY= option stores the new formats in a specified format catalog; otherwise, they are stored in a default catalog named Work.Formats. The keyword FMTLIB displays the formats and values that are currently stored in the catalog. The VALUE statement defines a new format for the values of a variable.

**Defining a Unique Format**

Formats can be specified for a single value, a range of values, or a list of unique values. Unique values must be separated by commas. When character values are specified, the values must be enclosed in quotation marks, and the format name must begin with a dollar sign ($). You can specify non-inclusive numeric ranges by using the less than sign (<). The keywords HIGH, LOW, and OTHER can be used to label values that are not specifically addressed in a range.

**Associating User-Defined Formats with Variables**

To access the permanent, user-defined formats in a format catalog, you'll need to use a LIBNAME statement to reference the catalog library. To associate user-defined formats with variables in the FORMAT statement, use the same format names in both the FORMAT and VALUE statements, but place a period at the end of the format name when it is used in the FORMAT statement.

**Syntax**

**LIBNAME** *libref* *'SAS-data-library'***;**

**PROC FORMAT LIBRARY=***libref* **FMTLIB;**

**VALUE** *format-name*

*range1='label1'*

*range2 'label2'*

...**;**

**RUN;**

**DATA** *SAS-data-set;*

**INFILE** *data-file;*

**INPUT** *pointer variable-name informat.;*

**FORMAT** *variable(s) format-name.;*

**RUN;**

**Sample Programs**

libname library 'c:\sas\formats\lib';

proc format library=library fmtlib;

value jobfmt

103='manager'

105='text processor';

run;

data perm.empinfo;

infile empdata;

input @9 FirstName $5. @1 LastName $7.

+7 JobTitle 3. @19 Salary comma9.;

format salary comma9.2 jobtitle jobfmt.;

run;

**Points to Remember**

* Formats—even permanently associated ones—do not affect the values of variables in a SAS data set. Only the appearance of the values is altered.
* A user-defined format name must begin with a dollar sign ($) when it is assigned to character variables. A format name cannot end with a number.

Use two single quotation marks when you want an apostrophe to appear in a label.

* Place a period at the end of the format name when you use the format name in the FORMAT statement.

**Chapter 8 Summary**

**Text Summary**

**Purpose of PROC MEANS**

The MEANS procedure provides an easy way to compute descriptive statistics. Descriptive statistics such as the mean, minimum, and maximum provide useful information about numeric data.

**Specifying Statistics**

By default, PROC MEANS computes the *n*-count (the number of non-missing values), the mean, the standard deviation, and the minimum and maximum values for variables. To specify statistics, list their keywords in the PROC MEANS statement.

**Limiting Decimal Places**

Because PROC MEANS uses the BEST. format by default, procedure output can contain unnecessary decimal places. To limit decimal places, use the MAXDEC= option and set it equal to the length that you prefer.

**Specifying Variables in PROC MEANS**

By default, PROC MEANS computes statistics for all numeric variables. To specify the variables to include in PROC MEANS output, list them in a VAR statement.

**Group Processing Using the CLASS Statement**

Include a CLASS statement, specifying variable names, to group PROC MEANS output by variable values of classes.

**Group Processing Using the BY Statement**

Include a BY statement, specifying variable names, to group PROC MEANS output by variable values. Your data must be sorted according to those variables. Statistics are not computed for the BY variables.

**Creating a Summarized Data Set Using PROC MEANS**

You can create an output data set that contains summarized variables by using the OUTPUT statement in PROC MEANS. When you use the OUTPUT statement without specifying the statistic-keyword= option, the summary statistics N, MEAN, STD, MIN, and MAX are produced for all of the numeric variables or for all of the variables that are listed in a VAR statement.

**Creating a Summarized Data Set Using PROC SUMMARY**

You can also create a summarized output data set by using PROC SUMMARY. The PROC SUMMARY code for producing an output data set is exactly the same as the code for producing an output data set with PROC MEANS. The difference between the two procedures is that PROC MEANS produces a report by default, whereas PROC SUMMARY produces an output data set by default.

**The FREQ Procedure**

The FREQ Procedure is a descriptive procedure as well as a statistical procedure that produces one-way and n-way frequency tables. It describes your data by reporting the distribution of variable values.

**Specifying Variables**

By default, the FREQ procedure creates frequency tables for every variable in your data set. To specify the variables to analyze, include them in a TABLES statement.

**Creating Two-Way Tables**

When a TABLES statement contains two variables joined by an asterisk (\*), PROC FREQ produces crosstabulations. The resulting table displays values for

* cell frequency
* cell percentage of total frequency
* cell percentage of row frequency
* cell percentage of column frequency.

**Creating N-Way Tables**

Crosstabulations can include more than two variables. When three or more variables are joined in a TABLES statement, the result is a series of two-way tables that are grouped by the values of the first variables listed.

**Creating Tables in List Format**

To reduce the bulk of n-way table output, add a slash (/) and the LIST option to the end of the TABLES statement. PROC FREQ then prints compact, multi-column lists instead of a series of tables. Beginning in SAS®9, you can use the CROSSLIST option to format your tables in ODS column format.

**Suppressing Table Information**

You can suppress the display of specific statistics by adding one or more options to the TABLES statement:

* NOFREQ suppresses cell frequencies
* NOPERCENT suppresses cell percentages
* NOROW suppresses row percentages
* NOCOL suppresses column percentages.

**Syntax**

**PROC MEANS** <DATA=*SAS-data-set*>

<*statistic-keyword(s)*> <*option(s)*>**;**

<VAR*variable(s)*>**;**

<CLASS*variable(s)*>**;**

<BY*variable(s)*>**;**

<OUTPUT*out=SAS-data-set statistic=variable(s)*>**;**

**RUN;**

**PROC SUMMARY** <DATA=*SAS-data-set*>

<*statistic-keyword(s)*> <*option(s)*>**;**

<VAR*variable(s)*>**;**

<CLASS*variable(s)*>**;**

<BY*variable(s)*>**;**

<OUTPUT*out=SAS-data-set*>**;**

**RUN;**

**PROC FREQ** <DATA=*SAS-data-set*>**;**

**TABLES** *variable-1 \*variable-2* <*\* ... variable-n*>

**/** <NOFREQ|NOPERCENT|NOROW|NOCOL|CROSSLIST>**;**

**RUN;**

**Sample Programs**

proc means data=clinic.heart min max maxdec=1;

var arterial heart cardiac urinary;

class survive sex;

run;

proc summary data=clinic.diabetes;

var age height weight;

class sex;

output out=work.sum\_gender

mean=AvgAge AvgHeight AvgWeight;

run;

proc freq data=clinic.heart;

tables sex\*survive\*shock / nopercent list;

run;

**Points to Remember**

* In PROC MEANS, use a VAR statement to limit output to relevant variables. Exclude statistics for variables such as dates.
* By default, PROC MEANS prints the full width of each numeric variable. Use the MAXDEC= option to limit decimal places and improve legibility.
* Data must be sorted for BY group processing. You might need to run PROC SORT before using PROC MEANS with a BY statement.
* PROC MEANS and PROC SUMMARY produce the same results; however, the default output is different. PROC MEANS produces a report, whereas PROC SUMMARY produces an output data set.
* If you do not include a TABLES statement, PROC FREQ produces statistics for every variable in the data set. Variables that have continuous numeric values can create a large amount of output. Use a TABLES statement to exclude such variables, or group their values by applying a FORMAT statement.

**Chapter 9 Summary**

**Text Summary**

**The Output Delivery System**

The Output Delivery System (ODS) makes new report formatting options available in SAS. ODS separates your output into component parts so that the output can be sent to any ODS destination that you specify.

**Opening and Closing ODS Destinations**

Each ODS destination creates a different type of formatted output. By default, the HTML destination is open and SAS creates simple HTML output. Because an open destination uses system resources, it is a good idea to close the HTML destination if you don't need to create HTML output. Using ODS statements, you can create multiple output objects at the same time. When you have several ODS destinations open, you can close them all using the ODS \_ALL\_ CLOSE statement.

**Creating HTML Output with Option**

The HTML destination is open by default. However, you can use the ODS HTML statement to specify options. Use the BODY= or FILE= specification to create a custom named HTML body file containing procedure results. You can also use the ODS HTML statement to direct the HTML output from multiple procedures to the same HTML file.

**Creating HTML Output with a Table of Contents**

In order to manage multiple pieces of procedure output, you can use the CONTENTS= and FRAME = options with the ODS HTML statement to create a table of contents that links to your HTML output. The table of contents contains a heading for each procedure that creates output.

**Specifying a Path for Output**

You can also use the PATH= option to specify the directory where you want to store your HTML output. When you use the PATH= option, you don't need to specify the complete pathname for the body, contents, or frame files. By specifying the URL= suboption in the file specification, you can provide a URL that ODS uses in all the links that it creates to the file.

**Changing the Appearance of HTML Output**

You can change the appearance of your output using the STYLE= option in the ODS HTML statement. Several predefined styles are shipped with SAS.

**Additional Features**

ODS provides ways for you to customize HTML output using style definitions. Definitions are created using PROC TEMPLATE and describe how to render the HTML output or part of the HTML output.

**Syntax**

**LIBNAME** *libref 'SAS-data-library'***;**

**ODS LISTING CLOSE;**

**ODS HTML PATH=***file-specification*

<(URL=*'Uniform-Resource-Locator'* | NONE)>

**BODY=***file-specification*

**CONTENTS=**<*file-specification*>

**FRAME=**<*file-specification*>

**STYLE=**<*style-name*>;

**PROC PRINT DATA=***SAS-data-set*;

**RUN**;

**ODS HTML CLOSE**;

**ODS LISTING**;

**Sample Program**

libname clinic 'c:\data98\patients';

ods html path='c:\records'(url=none)

body='data.html'

contents='toc.html'

frame='frame.html'

style=brick;

proc print data=clinic.admit label;

var id sex age height weight actlevel;

label actlevel='Activity Level';

run;

proc print data=clinic.stress2;

var id resthr maxhr rechr;

run;

ods html close;

ods html;

**Points to Remember**

* An open destination uses system resources. Therefore, it is a good idea to close the HTML destination before you create other types of output and reopen the HTML destination after you close all destinations destination.
* You do not need the ODS HTML CLOSE statement to create simple HTML output. However, if you want to create HTML with options, then use the ODS HTML CLOSE statement to close the HTML destination. The ODS HTML CLOSE statement is added *after* the RUN statement for the procedure.
* If you use the CONTENTS= and FRAME= options, open the frame file from within your Web browser to view the procedure output *and* the table of contents.

**Chapter 10 Summary**

**Text Summary**

**Accumulating Totals**

Use a sum statement to add the result of an expression to an accumulator variable.

**Initializing and Retaining Variables**

You can use the RETAIN statement to assign an initial value to a variable whose value is assigned by a sum statement.

**Assigning Values Conditionally**

To perform an action conditionally, use an IF-THEN statement. The IF-THEN statement executes a SAS statement when the condition in the IF expression is true. You can include comparison and logical operators; logical comparisons that are enclosed in parentheses are evaluated as true or false before other expressions are evaluated. Use the ELSE statement to specify an alternative action when the condition in an IF-THEN statement is false.

**Specifying Lengths for Variables**

When creating a new character variable, SAS allocates as many bytes of storage space as there are characters in the first value that it encounters for that variable at compile time. This can result in truncated values. You can use the LENGTH statement to specify a length before the variable's first value is referenced in the DATA step.

**Subsetting Data**

To omit observations as you read data, include the DELETE statement in an IF-THEN statement. If you need to read and process variables that you don't want to keep in the data set, use the DROP= and KEEP= data set options or the DROP and KEEP statements.

**Assigning Permanent Labels and Formats**

You can use LABEL and FORMAT statements in DATA steps to permanently assign labels and formats. These do not affect how data is stored in the data set, only how it appears in output.

**Assigning Values Conditionally Using SELECT Groups**

As an alternative to IF-THEN/ELSE statements, you can use SELECT groups in DATA steps to perform conditional processing.

**Grouping Statements Using DO Groups**

You can execute a group of statements as a unit with DO groups in DATA steps. You can use DO groups in IF-THEN/ELSE statements and SELECT groups to perform many statements as part of the conditional action.

**Syntax**

**LIBNAME** *libref 'SAS-data-library'***;**

**FILENAME** *fileref 'filename'***;**

**DATA** *SAS-data-set*(**DROP=***variable(s)*|**KEEP=***variable(s)*)**;**

**INFILE** *file-specification* <OBS=*n*>**;**

**INPUT** *variable* <$> *startcol-endcol...***;**

**DROP** *variable(s)***;**

**KEEP** *variable(s)***;**

**RETAIN** *variable initial-value***;**

*variable+expression***;**

**LENGTH** *variable(s)* <$> *length***;**

**IF** *expression***THEN** *statement*;

**ELSE** *statement***;**

**IF** *expression***THEN DELETE;**

**LABEL** *variable1***=***'label1' variable2***=***'label2' ...***;**

**FORMAT** *variable(s) format-name*;

**SELECT** <*(select-expression)*>;

**WHEN**-1 (*when-expression-1* <*..., when-expression-n*>) *statement***;**

**WHEN**-n (*when-expression-1* <*..., when-expression-n*>) *statement***;**

<OTHERWISE*statement*;>

**END;**

**RUN;**

**PROC PRINT DATA=***SAS-data set* **LABEL;**

**RUN;**

**Sample Program**

data clinic.stress;

infile tests;

input ID $ 1-4 Name $ 6-25 RestHR 27-29 MaxHR 31-33

RecHR 35-37 TimeMin 39-40 TimeSec 42-43

Tolerance $ 45;

if tolerance='D'and resthr ge 70 then delete;

drop timemin timesec;

TotalTime=(timemin\*60)+timesec;

retain SumSec 5400;

sumsec+totaltime;

length TestLength $ 6;

if totaltime>800 then testlength='Long';

else if 750<=totaltime<=800 then testlength='Normal';

else if totaltime<750 then TestLength='Short';

label sumsec='Cumulative Total Seconds (+5,400)';

format sumsec comma6.;

run;

**Points to Remember**

* Like the assignment statement, the sum statement does not contain a keyword.
* If the expression in a sum statement produces a missing value, the sum statement ignores it. (Remember, however, that assignment statements assign a missing value if the expression produces a missing value.)
* Using ELSE statements with IF-THEN statements can save resources. For greater efficiency, construct your IF-THEN/ELSE statements with conditions of decreasing probability.
* Make sure the LENGTH statement appears before any other reference to the variable in the DATA step. If the variable has been created by another statement, a later use of the LENGTH statement will not change its length.
* Labels and formats do not affect how data is stored in the data set, only how it appears in output. You assign labels and formats temporarily in PROC steps and permanently in DATA steps.

**Chapter 11 Summary**

**Text Summary**

**Setting Up**

Before you can create a new data set, you must assign a libref to the SAS library that will store the data set.

**Reading a Single Data Set**

After you have referenced the library in which your data set is stored, you can write a DATA step to name the SAS data set to be created. You then specify the data set to be read in the SET statement.

**Selecting Variables**

You can select the variables that you want to drop or keep by using the DROP= or KEEP= data set options in parentheses after a SAS data set name. For convenience, use DROP= if more variables are kept than dropped.

**BY-Group Processing**

Use the BY statement in the DATA step to group observations for processing. When you use the BY statement with the SET statement, the DATA step automatically creates two temporary variables, FIRST. and LAST. When you specify multiple BY variables, a change in the value of a primary BY variable forces LAST.variable to equal **1** for the secondary BY variables.

**Reading Observations Using Direct Access**

In addition to reading input data sequentially, you can access observations directly by using the POINT= option to go directly to a specified observation. There is no end-of-file condition when you use direct access, so include an explicit OUTPUT statement and then the STOP statement to prevent continuous looping.

**Detecting the End of a Data Set**

To determine when the last observation in an input data set has been read, use the END= option in the SET statement. The specified variable is initialized to 0, then set to 1 when the SET statement reads the last observation of the data set.

**Syntax**

**LIBNAME** *libref 'SAS-data-library'***;**

**DATA** *SAS-data-set***;**

**SET** *SAS-data-set* (**KEEP=** *variable-1* <*...variable-n*>) | (**DROP=** *variable-1*  
<...variable-n>)

**POINT=***variable* | **END=***variable***;**

**OUTPUT** <*SAS-data-set*>**;**

**STOP;**

**RUN;**

**Sample Program**

proc sort data=company.usa out=work.temp2;

by manager jobtype;

run;

data company.budget2(keep=manager jobtype payroll);

set work.temp2;

by manager jobtype;

if wagecat='S' then Yearly=wagerate\*12;

else if wagecat='H' then Yearly=wagerate\*2000;

if first.jobtype then Payroll=0;

payroll+yearly;

if last.jobtype;

run;

data work.getobs5;

obsnum=5;

set company.usa(keep=manager payroll) point=obsnum;

output;

stop;

run;

data work.addtoend(drop=timemin timesec);

set sasuser.stress2(keep=timemin timesec) end=last;

TotalMin+timemin;

TotalSec+timesec;

TotalTime=totalmin\*60+totalsec;

if last;

run;

**Points to Remember**

* When you perform BY-group processing, the data sets listed in the SET statement must either be sorted by the values of the BY variable(s), or they must have an appropriate index.
* When using direct access to read data, you *must* prevent continuous looping. Add a STOP statement to the DATA step.
* Do not specify the END= option with the POINT= option in a SET statement.

**Chapter 12 Summary**

**Text Summary**

**One-to-One Merging**

You can combine data sets with one-to-one merging (combining) by including multiple SET statements in a DATA step. When you perform one-to-one merging, the new data set contains all the variables from all the input data sets. If the data sets contain same-named variables, the values that are read in from the last data set replace those that were read in from earlier ones. The number of observations in the new data set is the number of observations in the smallest original data set.

data one2one;

set a;

set b;

run;

**Concatenating**

To append the observations from one data set to another data set, you concatenate them by specifying the data set names in the SET statement. When SAS concatenates, data sets in the SET statement are read sequentially, in the order in which they are listed. The new data set contains all the variables and the total number of observations from all input data sets.

data concat;

set a b;

run;

**Appending**

Another way to combine SAS data sets is to append one data set to another using the APPEND procedure. Although appending and concatenating are similar, there are some important differences between the two methods. The DATA step creates a new data set when concatenating. PROC APPEND adds the observations of one data set to the end of a “master” (or BASE) data set. SAS does not create a new data set nor does it read the base data set when executing the APPEND procedure.

**Interleaving**

If you use a BY statement when you concatenate data sets, the result is interleaving. Interleaving intersperses observations from two or more data sets, based on one or more common variables. Each input data set must be sorted or indexed based on the BY variable(s). Observations in each BY group in each data set in the SET statement are read sequentially, in the order in which the data sets and BY variables are listed, until all observations have been processed. The new data set contains all the variables and the total number of observations from all input data sets.

data interlv;

set a b;

by num;

run;

**Match-Merging**

Sometimes you need to combine observations from two or more data sets into a single observation in a new data set according to the values of a BY variable. This is match-merging, which uses a MERGE statement rather than a SET statement to combine data sets. Each input data set must be sorted or indexed in ascending order based on the BY variable(s). During match-merging, SAS sequentially checks each observation of each data set to see whether the BY values match, and then writes the combined observation to the new data set.

data merged;

merge a b;

by num;

run;

**Match-Merge Processing**

To predict the results of match-merging correctly, you need to understand how the DATA step processes data in match-merges.

**Compiling**

To prepare to merge data sets, SAS

1. reads the descriptor portions of the data sets that are listed in the MERGE statement
2. reads and compiles the rest of the DATA step program
3. creates the program data vector (PDV), an area of memory where SAS holds one observation at a time
4. assigns a tracking pointer to each data set that is listed in the MERGE statement.

If variables with the same name appear in more than one data set, the variable from the first data set that contains the variable (in the order listed in the MERGE statement) determines the length of the variable.

**Executing**

After compiling the DATA step, SAS sequentially match-merges observations by moving the pointers down each observation of each data set and checking to see whether the BY values match.

* If *Yes*, the observations are read into the PDV in the order in which the data sets appear in the MERGE statement. Values of any same-named variable are overwritten by values of the same-named variable in subsequent data sets. SAS writes the combined observation to the new data set and retains the values in the PDV until the BY value changes in all the data sets.
* If *No*, SAS determines which BY value comes first and reads the observation that contains this value into the PDV. Then the observation is written to the new data set.

When the BY value changes in all the input data sets, the PDV is initialized to missing. The DATA step merge continues to process every observation in each data set until it has processed all observations in all data sets.

**Handling Unmatched Observations and Missing Values**

All observations that are written to the PDV, including observations that have missing data and no matching BY values, are written to the output data set.

* If an observation contains missing values for a variable, then the observation in the output data set contains the missing values as well. Observations that have missing values for the BY variable appear at the top of the output data set.
* If an input data set doesn't have a matching BY value, then the observation in the output data set contains missing values for the variables that are unique to that input data set.

**Renaming Variables**

Sometimes you might have same-named variables in more than one input data set. In this case, match-merging overwrites values of the same-named variable in the first data set with values of the same-named variable in subsequent data sets. To prevent overwriting, use the RENAME= data set option in the MERGE statement to rename variables.

**Excluding Unmatched Observations**

By default, match-merging combines all observations in all input data sets. However, you might want to select only observations that match for two or more input data sets. To exclude unmatched observations, use the IN= data set option and the subsetting IF statement in your DATA step. The IN= data set option creates a variable to indicate whether the data set contributed data to the current observation. The subsetting IF statement then checks the IN= values and writes to the merged data set only matching observations.

**Selecting Variables**

You can specify the variables you want to drop or keep by using the DROP= and KEEP= data set options. When match-merging, you can specify these options in either the DATA statement or the MERGE statement, depending on whether or not you want to reference values of the variables in that DATA step. When used in the DATA statement, the DROP= option simply drops the variables from the new data set. However, they are still read from the original data set and are available within the DATA step.

**Syntax**

**One-to-One Merging**

**LIBNAME** *libref 'SAS-data-library'***;**

**DATA** *output-SAS-data-set***;**

**SET** *SAS-data-set-1***;**

**SET** *SAS-data-set-2***;**

**RUN;**

**Concatenating**

**DATA** *output-SAS-data-set***;**

**SET** *SAS-data-set-1 SAS-data-set-2***;**

**RUN;**

**Interleaving**

**PROC SORT DATA=***SAS-data-set* **OUT=***SAS-data-set***;**

**BY** *variable(s)***;**

**RUN;**

**DATA** *output-SAS-data-set***;**

**SET** *SAS-data-set-1 SAS-data-set-2***;**

**BY** *variable(s)***;**

**RUN;**

**Match-Merging**

**PROC SORT DATA=***SAS-data-set* **OUT=***SAS-data-set***;**

**BY** *variable(s)***;**

**RUN;**

**DATA** *output-SAS-data-set* **(DROP=***variable(s)* | **KEEP=***variable(s)***);**

**MERGE** *SAS-data-set-1 SAS-data-set-2*

**(RENAME=(***old-variable-name=new-variable-name***)**

**IN=** *variable* **DROP=***variable(s)* | **KEEP=***variable(s)***);**

**BY** *variable(s)***;**

**IF** *expression***;**

**RUN;**

**Sample Programs**

**One-to-One Reading**

data clinic.one2one;

set clinic.patients;

if age<60;

set clinic.measure;

run;

**Concatenating**

data clinic.concat;

set clinic.therapy1999 clinic.therapy2000;

run;

**Interleaving**

data clinic.intrleav;

set clinic.therapy1999 clinic.therapy2000;

by month;

run;

**Match-Merging**

data clinic.merged(drop=id);

merge clinic.demog(in=indemog

rename=(date=BirthDate))

clinic.visit(drop=weight in=invisit

rename=(date=VisitDate));

by id;

if indemog and invisit;

run;

**Points to Remember**

* You can rename any number of variables in each occurrence of the RENAME= option.
* In match-merging, the IN= data set option can apply to any data set in the MERGE statement. The RENAME=, DROP=, and KEEP= options can apply to any data set in the DATA or MERGE statements.
* Use the KEEP= option instead of the DROP= option if more variables are dropped than kept.
* When you specify multiple data set options for a particular data set, enclose them in a single set of parentheses.

**Chapter 13 Summary**

**Text Summary**

**Using SAS Functions**

SAS functions can be used to convert data and to manipulate the values of character variables. Functions are written by specifying the function name, then its arguments in parentheses. Arguments can include variables, constants, expressions, or other functions. Although arguments are typically separated by commas, they can also be specified as variable lists or arrays.

**Automatic Character-to-Numeric Conversion**

When character variables are used in a numeric context, SAS tries to convert the character values to numeric values. Numeric context includes arithmetic operations, comparisons with numeric values, and assignment to previously defined numeric variables. The original character values are not changed. The conversion creates temporary numeric values and places a note in the SAS log.

**Explicit Character-to-Numeric Conversion**

The INPUT function provides direct, controlled conversion of character values to numeric values. When a character variable is read with a numeric informat, the INPUT function generates numeric values without placing a note in the SAS log.

**Automatic Numeric-to-Character Conversion**

When numeric variables are used in a character context, SAS tries to convert the numeric values to character values. Character context includes concatenation operations, use in functions that require character arguments, and assignment to previously defined character variables. The original numeric values are not changed; the conversion creates temporary character values and places a note in the SAS log.

**Explicit Numeric-to-Character Conversion**

The PUT function provides direct, controlled conversion of numeric values to character values. The format specified in a PUT function must match the source, so use an appropriate numeric format to create the new character values. No note will appear in the SAS log.

**SAS Date and Time Values**

SAS date values are stored as the number of days from January 1, 1960; time values are stored as the number of seconds since midnight. These values can be displayed in a variety of forms by associating them with SAS formats.

**YEAR, QTR, MONTH, and DAY Functions**

To extract the year, quarter, month, or day value from a SAS date value, specify the YEAR, QTR, MONTH, or DAY function followed by the SAS date value in parentheses. The YEAR function returns a four-digit number; QTR returns a value of **1**, **2**, **3**, or **4**; MONTH returns a number from**1** to **12**; and DAY returns **1** to **31**.

**WEEKDAY Function**

To extract the day of the week from a SAS date value, specify the function WEEKDAY followed by the SAS date value in parentheses. The function returns a numeric value from **1** to **7**, representing the day of the week.

**MDY Function**

To create a SAS date value from a month, day, and year, specify the MDY function followed by the component values. The result can be displayed in several ways by applying a SAS date format. SAS interprets two-digit values according to the 100-year span that is set by the YEARCUTOFF= system option. The default value of YEARCUTOFF= is **1920**.

**DATE and TODAY Functions**

To convert the current date to a SAS date value, specify the DATE or TODAY function without arguments. The DATE and TODAY functions can be used interchangeably.

**INTCK Function**

To count the number of time intervals that occur in a time span, use the INTCK function and specify the interval constant or variable, the beginning date value, and the ending date value. The INTCK function counts intervals from fixed interval beginnings, not in multiples of an interval unit. Partial intervals are not counted.

**INTNX Function**

To apply multiples of an interval to a date value, use the INTNX function and specify the interval constant or variable, the start-from date value, and the increment. Include the alignment option to specify whether the date returned should be at the beginning, middle, or end of the interval, or on the same day.

**DATDIF and YRDIF Functions**

To calculate the difference between dates as a number of days or as a number of years, use the DATDIF or YRDIF function. These functions accept SAS date values and return a difference between the date values calculated according to the basis that you specify in the function.

**SCAN Function**

The SCAN function enables you to separate a character value into words and to return a word based on its position. It defines words according to delimiters, which are characters that are used as word separators. The name of the function is followed, in parentheses, by the name of the character variable, the desired word number, and the specified delimiters enclosed in quotation marks.

**SUBSTR Function**

The SUBSTR function can be used to extract or replace any portion of a character string. To extract values, place the function on the right side of an assignment statement and specify, in parentheses, the name of the character variable, the starting character position, and the number of characters to extract. To replace values, place the function on the left side of an assignment statement and specify, in parentheses, the name of the variable being modified, the starting character position, and the number of characters to replace.

**SCAN versus SUBSTR**

Both the SCAN and SUBSTR functions can extract a substring from a character value. SCAN relies on delimiters, whereas SUBSTR reads values from specified locations. Use SCAN when you know the delimiter and the order of words. Use SUBSTR when the positions of the characters don't vary.

**TRIM Function**

Because SAS pads character values with trailing blanks, unwanted spaces can sometimes appear after strings are concatenated. To remove trailing blanks from character values, specify the TRIM function with the name of a character variable. Remember that trimmed values will be padded with blanks again if they are shorter than the length of the new variable.

**CATX Function**

Beginning in SAS®9, you can concatenate character strings, remove leading and trailing blanks, and insert separators in one statement by using the CATX function. The results of the CATX function are usually equivalent to those that are produced by a combination of the concatenation operator and the TRIM and LEFT functions.

**INDEX Function**

To test character values for the presence of a string, use the INDEX function and specify, in parentheses, the name of the variable and the string enclosed in quotation marks. The INDEX function can be used with a subsetting IF statement when you create a data set. Only those observations in which the function finds the string and returns a value greater than 0 are written to the new data set.

**FIND Function**

Beginning in SAS®9, you can also use the FIND function to search for a specific substring of characters within a character string that you specify. The FIND function is similar to the INDEX function, but the FIND function enables you to ignore character case in your search and to trim trailing blanks. The FIND function can also begin the search at any position that you specify in the string.

**UPCASE Function**

Lowercase letters in character values can be converted to uppercase by using the UPCASE function. Include the function in an assignment statement, and specify the variable name in parentheses.

**LOWCASE Function**

Uppercase letters in character values can be converted to lowercase by using the LOWCASE function. Include the function in an assignment statement, and specify the variable name in parentheses.

**PROPCASE Function**

Beginning in SAS®9, character values can be converted to proper case by using the PROPCASE function. Include the function in an assignment statement, and specify the variable name in parentheses. Remember that you can specify delimiters or use the default delimiters.

**TRANWRD Function**

You can replace or remove patterns of characters in the values of character variables by using the TRANWRD function. Use the function in an assignment statement, and specify the source, target, and replacement strings or variables in parentheses.

**INT Function**

To return the integer portion of a numeric value, use the INT function. Any decimal portion of the INT function argument is discarded.

**ROUND Function**

To round values to the nearest specified unit, use the ROUND function. If a round-off unit is not provided, the argument is rounded to the nearest integer.

**Nesting SAS Functions**

To write more efficient programs, you can nest functions as appropriate. You can nest any functions as long as the function that is used as the argument meets the requirements for the argument.

**Syntax**

**INPUT(***source,informat***)**

**PUT(***source,format***)**

**YEAR(***date***)**

**QTR(***date***)**

**MONTH(***date***)**

**DAY(***date***)**

**WEEKDAY(***date***)**

**MDY(***month,day,year***)**

**DATE()**

**TODAY()**

**INTCK(***'interval',from,to***)**

**INTNX(***'interval',start-from,increment*<*,'alignment'*>**)**

**DATDIF(***start\_date,end\_date,basis***)**

**YRDIF(***start\_date,end\_date,basis***)**

**SCAN(***argument,n,*<*delimiters*>)

**SUBSTR(***argument,position,*<*n*>**)**

**TRIM(***argument***)**

**CATX(***separator,string-1*<*,...string-n*>**)**

**INDEX(***source,excerpt***)**

**FIND(***string,substring*<*,modifiers*><*,startpos*>**)**

**UPCASE(***argument***)**

**LOWCASE(***argument***)**

**PROPCASE(***(argument*<*,delimiter(s)*>**)**

**TRANWRD(***source,target,replacement***)**

**INT(***argument***)**

**ROUND(***argument,round-off-unit***)**

**Sample Programs**

data hrd.newtemp;

set hrd.temp;

Salary=input(payrate,2.)\*hours;

run;

data hrd.newtemp;

set hrd.temp;

Assignment=put(site,2.)||'/'||dept;

run;

data hrd.tempnov;

set hrd.temp;

if month(startdate)=11;

run;

data hrd.temp2010;

set hrd.temp;

if year(startdate)=2010;

run;

data radio.schwkend;

set radio.sch;

if weekday(airdate)in(1,7);

run;

data hrd.newtemp(drop=month day year);

set hrd.temp;

Date=mdy(month,day,year);

run;

data hrd.newtemp;

set hrd.temp;

EditDate=today();

run;

data work.anniv20;

set flights.mechanics(keep=id lastname firstname hired);

Years=intck('year',hired,today());

if years=20 and month(hired)=month(today());

proc print data=work.anniv20;

title '20-Year Anniversaries this Month';

run;

data work.after;

set work.before;

TargetYear=intnx('year','05feb94'd,3);

run;

data hrd.newtemp(drop=name);

set hrd.newtemp;

length LastName FirstName MiddleName $ 10;

lastname=scan(name,1,' ,');

firstname=scan(name,2,' ,');

middlename=scan(name,3,' ,');

run;

data hrd.temp2(drop=exchange);

set hrd.temp;

exchange=substr(phone,1,3);

if exchange='622' then substr(phone,1,3)='433';

run;

data hrd.newtemp(drop=address city state zip);

set hrd.temp;

NewAddress=trim(address)||', '||trim(city)||', '||zip;

run;

data hrd.datapool;

set hrd.temp;

if index(job,'word processing') > 0;

run;

data hrd.newtemp;

set hrd.temp;

Job=upcase(job);

run;

data hrd.newtemp;

set hrd.temp;

Contact=lowcase(contact);

run;

data hrd.newtemp;

set hrd.temp;

location=propcase(city);

run;

data work.after;

set work.before;

name=tranwrd(name,'Miss','Ms.');

name=tranwrd(name,'Mrs.','Ms.');

run;

data hrd.newtemp(drop=address city state zip);

set hrd.temp;

NewAddress=catx(', ',address,city,zip);

run;

data work.after;

set work.before;

Examples=int(examples);

run;

data work.after;

set work.before;

Examples=round(examples,.2);

run;

data hrd.datapool;

set hrd.temp;

if find(job,'word processing','t')>0;

run;

**Points to Remember**

* Even if a function doesn't require arguments, the function name must still be followed by parentheses.
* When specifying a variable list or an array as a function argument, be sure to precede the list or the array with the word OF.
* To remember which function requires a format versus an informat, note that the INPUT function requires the informat.
* If you specify an invalid date in the MDY function, a missing value is assigned to the target variable.
* The SCAN function treats contiguous delimiters as one delimiter; leading delimiters have no effect.
* When using the SCAN function, you can save storage space by adding a LENGTH statement to your DATA step to set an appropriate length for your new variable(s). Place the LENGTH statement before the assignment statements that contain the SCAN function.
* When the SUBSTR function is on the left side of an assignment statement, it replaces variable values. When SUBSTR is on the right side of an assignment statement, it extracts variable values. The syntax of the function is the same; only the placement of the function changes.
* The INDEX function is case sensitive. To ensure that all forms of a character string are found, use the UPCASE or LOWCASE function with the INDEX function. You can also use the FIND function with the i modifier.

**Chapter 14 Summary**

**Text Summary**

**Purpose of DO Loops**

DO loops process groups of SAS statements repeatedly, reducing the number of statements required in repetitive calculations.

**Syntax of Iterative DO Loops**

To construct an iterative DO loop, specify an index variable and the conditions that will execute the loop. These conditions include a start value for the index variable, a stop value, and an increment value. Start, stop, and increment values can be any number, numeric variable, or SAS expression that results in a number.

**DO Loop Execution**

During each iteration of a DO loop, statements within the loop execute. Once the loop's index value exceeds the stop value, the DO loop stops, and processing continues with the following DATA step statement.

**Generating Observations with DO Loops**

Include an OUTPUT statement within the DO loop to write an observation for each iteration. This overrides the automatic generation of output at the end of the DATA step.

**Decrementing DO Loops**

You can decrement a DO loop by specifying a negative value for the BY clause. The start value must be greater than the stop value.

**Specifying a Series of Items**

You can specify how many times a DO loop executes by listing items in a series; the DO loop will execute once for each item, with the index variable equal to the value of each item. A series can consist of all numeric values, all character values (enclosed in quotation marks), or all variable names (without quotation marks).

**Nesting DO Loops**

DO loops can run within DO loops, as long as you assign a unique index variable to each loop and terminate each DO loop with its own END statement.

**Iteratively Processing Observations from a Data Set**

You can use a DO loop to read a data set and compute the value of a new variable. DO loop start and stop values, for example, can be read from a data set.

**Conditionally Executing DO Loops**

The DO UNTIL statement executes a DO loop until a condition becomes true. Because the expression is not evaluated until the bottom of the loop, a DO UNTIL loop will execute at least once. The DO WHILE statement is used to execute a DO loop while a condition is true. Because the DO WHILE statement is evaluated at the top of the DO loop, if the expression is initially false, the DO loop never executes.

**Using Conditional Clauses with the Iterative DO Statement**

DO WHILE and DO UNTIL statements can be used within iterative DO loops to combine conditional and unconditional execution.

**Creating Samples**

DO loops provide an efficient way to create samples from data sets. Enclose the SET statement in a DO loop, using the start, stop, and increment values to select the observations. Add the POINT= option to the SET statement, setting it equal to the index variable of the DO loop. Then add a STOP statement to prevent DATA step looping, and add an OUTPUT statement to write observations.

**Syntax**

**DO** *index-variable=start* **TO** *stop* **BY** *increment***;**

*SAS statements*

**END;**

**DO UNTIL(***expression***);**

**DO WHILE(***expression***);**

**DO** *index-variable=start* **TO** *stop* **BY** *increment* **UNTIL(***expression***);**

**DO** *index-variable=start* **TO** *stop* **BY** *increment* **WHILE(***expression)***;**

**Sample Programs**

**Simple DO Loop**

data work.earn;

Value=2000;

do year=1 to 20;

Interest=value\*.075;

value+interest;

end;

run;

**Nested DO Loops**

data work.earn;

do year=1 to 20;

Capital+2000;

do month=1 to 12;

Interest=capital\*(.075/12);

capital+interest;

end;

end;

run;

**Conditional Clause**

data work.invest;

do year=1 to 10 until(Capital>=50000);

capital+2000;

capital+capital\*.10;

end;

if year=11 then year=10;

run;

**Creating Samples**

data work.subset;

do sample=10 to 5000 by 10;

set factory.widgets point=sample;

output;

end;

stop;

run;

**Points to Remember**

* If you do not specify a BY clause, the increment value for DO loops is **1**.
* In most cases, the index variable is needed only for processing the DO loop and can be dropped from the data set.
* The index variable is always incremented by one value beyond the stop value unless you terminate the DO loop in some other manner.
* It's easier to manage nested DO loops if you indent the statements in each loop.

**Chapter 15 Summary**

**Text Summary**

**Purpose of SAS Arrays**

An array is a temporary grouping of variables under a single name. This can reduce the number of statements needed to process variables and can simplify the maintenance of DATA step programs.

**Defining an Array**

To group variables into an array, use an ARRAY statement that specifies the array's name; its dimension enclosed in braces, brackets, or parentheses; and the elements to include. For example: array sales{4} qtr1 qtr2 qtr3 qtr4;

**Variable Lists as Array Elements**

You can use a variable list to specify array elements. Depending on the form of the variable list, it can specify all numeric or all character variables, or a numbered range of variables.

**Referencing Elements of an Array**

When you define an array in a DATA step, a subscript is assigned to each element. During execution, you can use an array reference to perform actions on specific array elements. When used in a DO loop, for example, the index variable of the iterative DO statement can reference each element of the array.

**The DIM Function**

When using DO loops to process arrays, you can also use the DIM function to specify the TO clause of the iterative DO statement. When you use the DIM function, you do not have to re-specify the stop value of a DO statement if you change the dimension of the array.

**Creating Variables with the ARRAY Statement**

If you don't specify array elements in an ARRAY statement, SAS automatically creates variable names for you by concatenating the array name and the numbers 1, 2, 3 ... up to the array dimension. To create an array of character variables, add a dollar sign ($) after the array dimension. By default, all character variables that are created with an ARRAY statement are assigned a length of 8; however, you can specify a different length after the dollar sign.

**Assigning Initial Values to Arrays**

To assign initial values in an ARRAY statement, place the values in parentheses after the array elements, specifying one initial value for each array element and separating each value with a comma or blank. To assign initial values to character variables, enclose each value in quotation marks.

**Creating Temporary Array Elements**

You can create temporary array elements for DATA step processing without creating additional variables. Just specify \_TEMPORARY\_ after the array name and dimension. This is useful when the array is needed only to perform a calculation.

**Multidimensional Arrays**

To define a multidimensional array, specify the number of elements in each dimension, separated by a comma. For example, **array new{3,4} x1-x12;** defines a two-dimensional array, with the first dimension specifying the number of rows (3) and the second dimension specifying the number of columns (4).

**Referencing Elements of a Two-Dimensional Array**

Multidimensional arrays are typically used with nested DO loops. If a DO loop processes a two-dimensional array, you can reference any element within the array by specifying the two dimensions.

**Rotating Data Sets**

You can use arrays to rotate a data set. Rotating a data set changes variables to observations or observations to variables.

**Syntax**

**ARRAY** *array-name***{***dimension***}** <*elements*>**;**

*array-name***(***subscript***)**

**DIM(***array-name***)**

**Sample Programs**

data work.report(drop=i);

set master.temps;

array wkday{7} mon tue wed thr fri sat sun;

do i=1 to 7;

wkday{i}=5\*(wkday{i}-32)/9;

end;

run;

data hrd.convert(drop=i);

set hrd.fitclass;

array wt{6} weight1-weight6;

do i=1 to dim(wt);

wt{i}=wt{i}\*2.2046;

end;

run;

data hrd.diff(drop=i);

set hrd.convert;

array wt{6} weight1-weight6;

array WgtDiff{5};

do i=1 to 5;

wgtdiff{i}=wt{i+1}-wt{i};

end;

run;

data finance.report(drop=i);

set finance.qsales;

array sale{4} sales1-sales4;

array goal{4} \_temporary\_ (9000 9300 9600 9900);

array Achieved{4};

do i=1 to 4;

achieved{i}=100\*sale{i}/goal{i};

end;

run;

data finance.quarters(drop=i j);

set finance.monthly;

array m{4,3} month1-month12;

array Qtr{4};

do i=1 to 4;

qtr{i}=0;

do j=1 to 3;

qtr{i}+m{i,j};

end;

end;

run;

**Points to Remember**

* A SAS array exists only for the duration of the DATA step.
* Do not give an array the same name as a variable in the same DATA step. Also, avoid using the name of a SAS function as an array name; the array will be correct, but you won't be able to use the function in the same DATA step, and a warning will be written to the SAS log.
* You can indicate the dimension of a one-dimensional array with an asterisk (\*) as long as you specify the elements of the array.
* When referencing array elements, be careful not to confuse variable names with the array references.

**Chapter 16 Summary**

**Text Summary**

**Review of Column Input**

When data is arranged in columns or fixed fields, you can use column input to read them. With column input, the beginning and ending column are specified for each field. Character variables are identified by a dollar ($) sign.

Column input has several features.

* Fields can be read in any order.
* It can be used to read character variables that contain embedded blanks.
* No placeholder is required for missing data. A blank field is read as missing and does not cause other fields to be read incorrectly.
* Fields or parts of fields can be re-read.
* Fields do not have to be separated by blanks or other delimiters
* It can be used to read standard character and numeric data.

**Identifying Nonstandard Numeric Data**

Standard numeric data values are values that contain only numbers, scientific notation, decimal points, and plus and minus signs. When numeric data contains characters such as commas or dollar signs, the data is considered to be nonstandard.

Nonstandard numeric data includes

* values that contain special characters, such as percent signs, dollar signs, and commas
* date and time values
* data in fraction, integer binary, real binary, and hexadecimal forms.

**Choosing an Input Style**

SAS provides two input styles for reading data in fixed fields: column input and formatted input. You can use

* column input to read standard data only
* formatted input to read both standard and nonstandard data.

**Using Formatted Input**

Formatted input uses column pointer controls to position the input pointer on a specified column. A column pointer control is optional when the first variable begins in the first column.

The @*n* is an absolute pointer control that moves the input pointer to a specific column number. You can read columns in any order with the @*n* column pointer control.

The +*n* is a relative pointer control that moves the input pointer forward to a column number that is relative to the current position. The +*n* pointer control cannot move backwards. However, you can use the notation +(-*n*) to move the pointer control backwards.

**Using Informats**

An informat tells SAS how to read raw data. There are informats for reading standard and nonstandard character and numeric values.

Informats always contain a *w* value to indicate the width of the raw data field. A period (.) ends the informat or separates the *w* value from the optional *d* value, which specifies the number of implied decimal places.

**Record Formats**

A record format specifies how records are organized in a file. The two most common are fixed-length records and variable-length records.

When you read variable-length records that contain fixed-field data into a SAS data set, there might be values that are shorter than others or that are missing. The PAD option pads each record with blanks so that all data lines have the same length.

**Syntax**

**LIBNAME** *libref 'SAS-data-library'***;**

**FILENAME** *fileref 'filename'***;**

**DATA** *SAS-data-set***;**

**INFILE** *file-specification***;**

**INPUT** <*pointer-control*> *variable informat.***;**

**RUN;**

**PROC PRINT DATA=***SAS-data-set***;**

**RUN;**

**Sample Program**

libname sasuser 'c:\data\sales';

filename vandata 'c:\records\vans.dat';

data sasuser.vansales;

infile vandata;

input +12 Quarter 1. @1 Region $9.

+6 TotalSales comma11.;

run;

proc print data=sasuser.vansales;

run;

**Points to Remember**

* When you use column input or formatted input, the input pointer stops on the column following the last column that was read.
* When you use informats, you do not need to specify a *d* value if the data values already contain decimal places.
* Column input can be used to read standard character or numeric data only.
* Formatted input can be used to read both standard and nonstandard data.
* When reading variable-length records that contain fixed-field data, you can avoid problems by using the PAD option in the INFILE statement.

**Chapter 17 Summary**

**Text Summary**

**Free-Format Data**

External files can contain raw data that is free-format. That is, the data is not arranged in fixed fields. The fields can be separated by blanks, or by some other delimiter, such as commas.

**Using List Input**

Free-format data can easily be read with list input because you do not need to specify column locations of the data. You simply list the variable names in the same order as the corresponding raw data fields. You must distinguish character variables from numeric variables by using the dollar ($) sign.

When characters other than blanks are used to separate the data values, you can specify the field delimiter by using the DLM= option in the INFILE statement.

You can also specify a range of variables in the INPUT statement when the variable values in the raw data file are sequential and are separated by blanks (or by some other delimiter). This is especially useful if your data contains similar variables, such as the answers to a questionnaire.

In its simplest form, list input places several limitations on the types of data that can be read.

**Reading Missing Values**

If your data contains missing values at the end of a record, you can use the INFILE statement with the MISSOVER option to prevent SAS from reading the next record to find the missing values.

If your data contains missing values at the beginning or in the middle of a record, you might be able to use the DSD option in the INFILE statement to correctly read the raw data. The DSD option sets the default delimiter to a comma and treats two consecutive delimiters as a missing value.

If the data uses multiple delimiters or a single delimiter other than a comma, you can use both the DSD option and the DLM= option in the INFILE statement.

The DSD option can also be used to read raw data when there is a missing value at the beginning of a record, as long as a delimiter precedes the first value in the record.

**Specifying the Length of Character Values**

You can specify the length of character variables by using the LENGTH statement. The LENGTH statement enables you to use list input to read character values that are longer than eight characters without truncating them.

Because variable attributes are defined when the variable is first encountered in the DATA step, the LENGTH statement precedes the INPUT statement and defines both the length and the type of the variable.

When you use the LENGTH statement, you do not need to specify the variable type again in the INPUT statement.

**Modifying List Input**

Modified list input can be used to read values that contain embedded blanks and nonstandard values. Modified list input uses two format modifiers:

* the ampersand (&) modifier enables you to read character values that contain single embedded blanks
* the colon (:) modifier enables you to read nonstandard data values and character values that are longer than eight characters, but which contain no embedded blanks.

Remember that informats work differently in modified list input than they do in formatted input.

**Creating Free-Format Data**

You can create a raw data file using list output. With list output, you simply list the names of the variables whose values you want to write. The PUT statement writes a value, leaves a blank, and then writes the next value.

You can use the DLM= option with a FILE statement to create a delimited raw data file. You can use the DSD option in a FILE statement to specify that data values containing commas should be enclosed in quotation marks. You can also use the DSD option to read values that contain delimiters within a quoted string.

**Mixing Input Styles**

With some file layouts, you might need to mix input styles in the same INPUT statement in order to read the data correctly.

**Syntax**

**Reading Free-Format Data**

**LIBNAME** *libref 'SAS-data-library'***;**

**FILENAME** *fileref 'filename'***;**

**DATA** *SAS-data-set*;

**INFILE** *file-specification* <DLM *'delimiter'*> <MISSOVER><DSD>**;**

**LENGTH** *variable* **$** *length***;**

**INPUT** *variable*<$> <&|:><*informat*>**;**

**RUN;**

**PROC PRINT DATA=***SAS-data-set***;**

**RUN;**

**Creating Free-Format Data**

**LIBNAME** *libref 'SAS-data-library'***;**

**DATA \_NULL\_**;

**SET** *SAS-data-set***;**

**FILE** *file-specification* <DLM *'delimiter'*> <DSD>**;**

**PUT** *variable*<*: format*>;

**RUN;**

**Sample Programs**

**Reading Free-Format Data**

libname sasuser 'c:\records\data';

filename credit 'c:\records\credit.dat';

data sasuser.carduse;

infile creditcard.dat dlm='#' missover;

length LastName $ 14;

input lastname $ Gender $ Age CardType $

Total : comma.;

run;

proc print data=sasuser.carduse;

run;

**Creating Raw Data Using List Output**

libname sasuser 'c:\records\data';

data \_null\_;

set sasuser.finance;

file 'c:\accounts\newdata.txt' dsd;

put ssn name salary : comma. date : date9.;

run;

**Points to Remember**

* When you use list input,
  + fields must be separated by at least one blank or other delimiter.
  + fields must be read in order, left to right. You cannot skip or re-read fields.
  + use a LENGTH statement to avoid truncating character values that are longer than eight characters.
* In formatted input, the informat determines both the length of character variables and the number of columns that are read. The same number of columns are read from each record.
* The informat in modified list input determines only the length of the variable value, not the number of columns that are read.

**Chapter 18 Summary**

**Text Summary**

**How SAS Stores Date and Time Values**

SAS stores dates as numeric SAS date values, which represent the number of days from January 1, 1960. SAS time values are the number of seconds since midnight.

**Reading Dates and Times with Informats**

Use SAS informats to read date and time expressions and convert them to SAS date and time values.

* MMDDYY*w*. reads dates such as 053090, 05/30/90, or 05 30 1990.
* DATE*w*. reads dates such as 30May1990, 30May90, or 30-May-1990.
* TIME*w*. reads times such as 17:00, 17:00:01.34, or 2:34.
* DATETIME*w*. reads dates and times such as 30May1990:10:03:17.2, 30May90 10:03:17.2, or 30May1990/10:03.

Two-digit year values require special consideration. When a two-digit year value is read, SAS defaults to a year within a 100-year span that is determined by the YEARCUTOFF= system option. The default value of YEARCUTOFF= is **1920**. You can check or reset the value of this option in your SAS session to use a different 100-year span for date informats.

**Using Dates and Times in Calculations**

Date and time values can be used in calculations like other numeric values. In addition to tracking time intervals, SAS date and time values can be used with SAS functions and with complex calculations.

**Using Date and Time Formats**

SAS provides many specialized date and time formats that enable you to specify how date and time values are displayed. You can use the WEEKDATE*w*. format to write date values in the form *day-of-week*, *month-name dd*, *yy* (or *yyyy*). You can use the WORDDATE*w*. format to write date values in the form *month-name dd*, *yyyy*.

**Syntax**

**OPTIONS YEARCUTOFF=***yyyy*;

**DATA** *SAS-data-set***;**

**INFILE** *file-specification***;**

**INPUT** <*pointer-control*> *variable informat.***;**

**RUN;**

**PROC PRINT DATA=***SAS-data-set***;**

**FORMAT** *variable format.***;**

**RUN;**

**Sample Program**

options yearcutoff=1920;

data perm.aprbills;

infile aprdata;

input LastName $8. @10 DateIn mmddyy8.

+1 DateOut mmddyy8. +1 RoomRate 6.

@35 EquipCost 6.;

Days=dateout-datein+1;

RoomCharge=days\*roomrate;

Total=roomcharge+equipcost;

run;

proc print data=perm.aprbills;

format datein dateout worddate12.;

run;

**Points to Remember**

* SAS makes adjustments for leap years, but not for leap seconds or daylight saving time.
* The minimum acceptable field width for the TIME*w*. informat is 5. If you specify a *w* value less than **5**, you'll receive an error message in the SAS log.
* The default value of the YEARCUTOFF= option is **1920**. When you work with two-digit year data, remember to check the default value of the YEARCUTOFF= option, and change it if necessary.
* The value of the YEARCUTOFF= system option does not affect four-digit year values. Four-digit values are always read correctly.
* Be sure to specify the proper informat for reading a date value, and specify the correct field width so that the entire value is read.
* If SAS date values appear in your program output, use a date format to display them in legible form.

**Chapter 19 Summary**

**Text Summary**

**Multiple Records per Observation**

Information for one observation can be spread out over several records. You can write one INPUT statement that contains line pointer controls to specify the record(s) from which values are read.

**Reading Multiple Records Sequentially**

The forward slash (/) line pointer control is used to read multiple records sequentially. Each time a / pointer is encountered, the input pointer advances to the next line.

**Reading Multiple Records Non-Sequentially**

The #*n* line pointer control is used to read multiple records non-sequentially. The #*n* specifies the absolute number of the line to which you want to move the pointer.

**Combining Line Pointer Controls**

The / line pointer control and the #*n* line pointer control can be combined within a SAS program to read multiple records both sequentially and non-sequentially.

**Syntax**

**LIBNAME** *libref 'SAS-data-library'***;**

**FILENAME** *fileref 'filename'***;**

**DATA** *SAS-data-set***;**

**INFILE** *file-specification***;**

**INPUT** *#n variable ...*

*#n @n variable ... / variable ... / variable ...* **;**

**RUN;**

**PROC PRINT DATA=***SAS-data-set***;**

**RUN;**

**Sample Program**

libname perm 'c:\records\empdata';

filename personel 'c:\records\empdata\new.dat';

data perm.emplist3;

infile personel;

input #2 Department $ 5-16

#1 @16 ID $4. @1 Name $14. /

JobCode 3. /

Salary comma9.;

run;

proc print data=perm.emplist3;

run;

**Points to Remember**

* When a file contains multiple records per observation, depending on the program, the file might need to contain the same number of records for each observation.
* Because the / pointer control can move forward only, the pointer control is specified *after* the values in the current record are read.
* The #*n* pointer control can read records in any order and must be specified *before* the variable names are defined.
* A semicolon should be placed at the end of the *complete* INPUT statement.

**Chapter 20 Summary**

**Text Summary**

**File Formats**

One raw data record might contain data for several observations. Data might be stored in this manner in order to reduce the size of the entire file. The data can be organized into

* repeating blocks of data
* an ID field followed by the same number of repeating fields
* an ID field followed by a varying number of repeating fields.

**Reading Repeating Blocks of Data**

To create multiple observations from a record that contains repeating blocks of data, the INPUT statement needs to hold the current record until each block of data has been read and written to the data set as an observation. The DATA step should include statements that

1. read the first block of values and hold the current record with the double trailing at sign (@@) line-hold specifier
2. optionally add a FORMAT statement to display date or time values with a specified format
3. write the first block of values as an observation
4. execute the DATA step until all repeating blocks have been read.

**Reading the Same Number of Repeating Fields**

To create multiple observations from a record that contains an ID field and the same number of repeating fields, you must execute the DATA step once for each record, repetitively reading and writing values in one iteration. The DATA step should include statements that

1. read the ID field and hold the current record with the single trailing at sign (@) line-hold specifier
2. execute SAS statements using an iterative DO loop. The iterative DO loop repetitively processes statements that
   * read the next value of the repeating field and hold the record with the @ line-hold specifier
   * explicitly write an observation to the data set by using an OUTPUT statement.
3. complete the iterative DO loop with an END statement.

**Reading a Varying Number of Repeating Fields**

To create multiple observations from a record that contains an ID field and a varying number of repeating fields, you must execute the DATA step once for each record, repetitively reading and writing values in one iteration while the value of the repeating field is nonmissing. The DATA step should include statements that

1. prevent SAS from reading the next record if missing values were encountered in the current record. The MISSOVER option on the INFILE statement prevents reading the next record.
2. read the ID field and the first repeating field, and then hold the record with the single trailing at sign (@) line-hold specifier
3. optionally create a counter variable
4. execute SAS statements while a condition is true, using a DO WHILE loop. A DO WHILE loop repetitively processes statements that
   * optionally increment the value of the counter variable by using a sum statement
   * explicitly add an observation to the data set by using an OUTPUT statement
   * read the next value of the repeating field and hold the record with the single trailing at sign (@) line-hold specifier.
5. complete the DO WHILE loop with an END statement.

**Syntax**

**Repeating Blocks of Data**

**LIBNAME** *libref 'SAS-data-library'***;**

**FILENAME** *fileref 'filename'***;**

**DATA** SAS-data-set**;**

**INFILE** *file-specification***;**

**INPUT** *variables @@***;**

**FORMAT** *date/time-variable format***;**

**RUN;**

**An ID Field Followed by the Same Number of Repeating Fields**

**LIBNAME** *libref 'SAS-data-library'***;**

**FILENAME** *fileref 'filename'***;**

**DATA** *SAS-data-set***;**

**INFILE** *file-specification***;**

**INPUT** *id-variable @***;**

**DO** *index-variable specification***;**

**INPUT** *repeating-variable @***;**

**OUTPUT;**

**END;**

**RUN;**

**An ID Field Followed by a Varying Number of Repeating Fields**

**LIBNAME** *libref 'SAS-data-library'***;**

**FILENAME** *fileref 'filename'***;**

**DATA** *SAS-data-set***;**

**INFILE** *file-specification* **MISSOVER;**

**INPUT** *id-variable repeating-variable @***;**

*counter-variable***=**0;

**DO WHILE** *(expression)***;**

*counter-variable***+**1;

**OUTPUT;**

**INPUT** *repeating-variable @***;**

**END;**

**RUN;**

**Sample Programs**

**Repeating Blocks of Data**

libname perm 'c:\records\weather';

filename tempdata 'c:\records\weather\tempdata';

data perm.april90;

infile tempdata;

input Date : date. HighTemp @@;

format date date9.;

run;

**An ID Field Followed by the Same Number of Repeating Fields**

libname perm 'c:\records\sales';

filename data97 'c:\records\sales\1997.dat';

data perm.sales97;

infile data97;

input ID $ @;

do Quarter=1 to 4;

input Sales : comma. @;

output;

end;

run;

**An ID Field Followed by a Varying Number of Repeating Fields**

libname perm 'c:\records\sales';

filename data97 'c:\records\sales\1997.dat';

data perm.sales97;

infile data97 missover;

input ID $ Sales : comma. @;

Quarter=0;

do while (sales ne .);

quarter+1;

output;

input sales : comma. @;

end;

run;

**Points to Remember**

* The double trailing at sign (@@) holds a record across multiple iterations of the DATA step until the end of the record is reached.
* The single trailing at sign (@) releases a record when control returns to the top of the DATA step.
* Use an END statement to complete DO loops and DO WHILE loops.

**Chapter 21 Summary**

**Text Summary**

**Hierarchical Raw Data Files**

Raw data files can be hierarchical in structure, consisting of a header record and one or more detail records. You can build a SAS data set from a hierarchical file by creating one observation

* per detail record, storing header data with each observation
* per header record, combining the information from detail records into summary variables.

**Creating One Observation per Detail Record**

To create one observation per detail record, it is necessary to distinguish between header and detail records. Having a field that identifies the type of the record makes this task easier.

As you write the DATA step, use a RETAIN statement to retain header data in the PDV until the next header record is encountered.

Next, you need to read the field in each record that identifies the record's type. Remember to use the @ line-hold specifier to hold the current value of each record type so that the other values in the record can be read by subsequent INPUT statements.

Use an IF-THEN statement to check for the condition that the record is a header record. If the record is a header record, you need to execute an INPUT statement to read values for that record.

You can use a subsetting IF statement to check for the condition that the record is a detail record. If the record is a detail record, use another INPUT statement to read values in that record.

Use the DROP= option to exclude the variable that identifies each record's type from the output data set.

**Creating One Observation per Header Record**

Creating one observation per header record condenses a large amount of information into a concise data set. As you write the DATA step, you need to think about performing several tasks.

As with creating one observation per detail record, use a RETAIN statement to retain header data in the PDV until the next header record is encountered. Then read the field in each record that identifies the record's type. Remember to use the @ line-hold specifier to hold the current record so that the other values in the record can be read by subsequent INPUT statements.

When the record is a header record, multiple statements need to be executed. You can do this by adding a simple DO group to an IF-THEN statement. Within the DO group, you need to

1. determine whether this is the first header record in the external file by using the automatic variable \_N\_
2. use an OUTPUT statement to write each header record except for the first one to the data set
3. use an assignment statement to create a summary variable, and set its value to *0*
4. add an INPUT statement to read values in the header record
5. close the DO group with an END statement.

When the record is a detail record, you need to define an alternative set of actions. You can do this by adding an ELSE statement after the DO group. As each detail record is read, increment the value of the summary variable by using a sum statement.

After the last detail record is read, there are no more header records to cause the last observation to be written to the data set. Determine when the current record is the last record in an external file by specifying the END= option in the INFILE statement. Again, you can use the DROP= option to exclude the variable that identifies each record's type from the output data set.

**Syntax**

**Syntax to Create One Observation for Each Detail Record**

**LIBNAME** **libref** *'SAS-data-library'***;**

**FILENAME** *fileref* *'filename'***;**

**DATA** *SAS-data-set* **(DROP=***variable***);**

**INFILE** *file-specification***;**

**RETAIN** *variable***;**

**INPUT** *variable***;**

**IF** *variable***=***'condition'* **THEN** *SAS statement***;**

**IF** *variable***=***'condition'***;**

*SAS-statement***;**

**RUN;**

**Syntax to Create One Observation for Each Header Record**

**LIBNAME** *libref 'SAS-data-library'***;**

**FILENAME** fileref *'filename'***;**

**DATA** *SAS-data-set* **(DROP=** *variable***);**

**INFILE** *file-specification* **END=***variable***;**

**RETAIN** *variable***;**

**INPUT** *variable***;**

**IF** *variable***=***'condition'* **THEN DO;**

**IF** \_N\_ > 1 **THEN OUTPUT;**

*summary-variable***=**0;

**INPUT** *variable*;

**END;**

ELSE **IF** *variable***=***'condition'* **THEN**

*summary-variable+expression***;**

**IF** *variable* **THEN OUTPUT;**

**RUN;**

**Sample Programs**

**Program to Create One Observation for Each Detail Record**

libname perm 'c:\records\census2k';

filename census 'c:\records\census2k\survey.dat';

data perm.people(drop=type);

infile census;

retain Address;

input type $1. @;

if type='H' then input @3 address $15.;

if type='P';

input @3 Name $10. @13 Age 3. @16 Gender $1.;

run;

**Program to Create One Observation for Each Header Record**

libname perm 'c:\records\census2k';

filename census 'c:\records\census2k\survey.dat';

data perm.residnts (drop=type);

infile census end=last;

retain Address;

input type $1. @;

if type='H' then do;

if \_n\_ > 1 then output;

Total=0;

input address $ 3-17;

end;

else if type='P' then total+1;

if last then output;

run;

**Points to Remember**

* As with automatic variables, the END= variable is not written to the data set.
* Values are automatically retained when using a sum statement. Therefore, it may be necessary to set the value of the counter variable back to **0** when a new header is encountered.