# Contents

Syntax Conventions for the SAS Language .................................................. v
What’s New in SAS 9.4 DATA Step Statements ........................................... xi

**Chapter 1 / About SAS DATA Step Statements** ........................................... 1
- Definition of Statements ................................................................. 1
- DATA Step Statements ................................................................. 1

**Chapter 2 / Dictionary of SAS DATA Step Statements** ............................... 5
- SAS Statements Documented in Other SAS Publications ......................... 7
- DATA Step Statements by Category .................................................. 8
- Dictionary ..................................................................................... 15

**Chapter 3 / Dictionary of SAS Statement Environment Variables** ............... 337
- Dictionary ..................................................................................... 337
Syntax Conventions for the SAS Language

Overview of Syntax Conventions for the SAS Language

SAS uses standard conventions in the documentation of syntax for SAS language elements. These conventions enable you to easily identify the components of SAS syntax. The conventions can be divided into these parts:

- syntax components
- style conventions
- special characters
- references to SAS libraries and external files

Syntax Components

The components of the syntax for most language elements include a keyword and arguments. For some language elements, only a keyword is necessary. For other language elements, the keyword is followed by an equal sign (=). The syntax for arguments has multiple forms in order to demonstrate the syntax of multiple arguments, with and without punctuation.

**keyword**

specifies the name of the SAS language element that you use when you write your program. Keyword is a literal that is usually the first word in the syntax. In a CALL routine, the first two words are keywords.

In these examples of SAS syntax, the keywords are bold:

- **CHAR** *(string, position)*
- **CALL RANBIN** *(seed, n, p, x)*;
- **ALTER** *(alter-password)*
- **BEST w.**
- **REMOVE <data-set-name>**

In this example, the first two words of the CALL routine are the keywords:
CALL RANBIN(seed, n, p, x)

The syntax of some SAS statements consists of a single keyword without arguments:

DO;
... SAS code ...
END;

Some system options require that one of two keyword values be specified:

DUPLEX | NODUPLEX

Some procedure statements have multiple keywords throughout the statement syntax:

CREATE <UNIQUE> INDEX index-name ON table-name (column-1 <, column-2, ...>)

argument
specifies a numeric or character constant, variable, or expression. Arguments follow the keyword or an equal sign after the keyword. The arguments are used by SAS to process the language element. Arguments can be required or optional. In the syntax, optional arguments are enclosed in angle brackets (< >).

In this example, string and position follow the keyword CHAR. These arguments are required arguments for the CHAR function:

CHAR (string, position)

Each argument has a value. In this example of SAS code, the argument string has a value of 'summer', and the argument position has a value of 4:

x=char('summer', 4);

In this example, string and substring are required arguments, whereas modifiers and startpos are optional.

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

argument(s)
specifies that one argument is required and that multiple arguments are allowed. Separate arguments with a space. Punctuation, such as a comma (,) is not required between arguments.

The MISSING statement is an example of this form of multiple arguments:

MISSING character(s);

<LITERAL_ARGUMENT> argument-1 <<LITERAL_ARGUMENT> argument-2 ... >
specifies that one argument is required and that a literal argument can be associated with the argument. You can specify multiple literals and argument pairs. No punctuation is required between the literal and argument pairs. The ellipsis (...) indicates that additional literals and arguments are allowed.

The BY statement is an example of this argument:

BY <DESCENDING> variable-1 <<DESCENDING> variable-2 ...>;

argument-1 <options> <argument-2 <options> ...>
specifies that one argument is required and that one or more options can be associated with the argument. You can specify multiple arguments and associated options. No punctuation is required between the argument and the
option. The ellipsis (...) indicates that additional arguments with an associated option are allowed.

The FORMAT procedure PICTURE statement is an example of this form of multiple arguments:

```
PATTERN name <(format-options)>
<value-range-set-1 <(picture-1-options)>
<value-range-set-2 <(picture-2-options)> …>;
```

`argument-1=value-1 <argument-2=value-2 ...>`

specifies that the argument must be assigned a value and that you can specify multiple arguments. The ellipsis (...) indicates that additional arguments are allowed. No punctuation is required between arguments.

The LABEL statement is an example of this form of multiple arguments:

```
LABEL variable-1=label-1 <variable-2=label-2 ...>;
```

`argument-1 <, argument-2, ...>`

specifies that one argument is required and that you can specify multiple arguments that are separated by a comma or other punctuation. The ellipsis (...) indicates a continuation of the arguments, separated by a comma. Both forms are used in the SAS documentation.

Here are examples of this form of multiple arguments:

```
AUTHPROVIDERDOMAIN (provider-1:domain-1 <, provider-2:domain-2, …>
INTO :macro-variable-specification-1 <, :macro-variable-specification-2, …>
```

Note: In most cases, example code in SAS documentation is written in lowercase with a monospace font. You can use uppercase, lowercase, or mixed case in the code that you write.

---

**Style Conventions**

The style conventions that are used in documenting SAS syntax include uppercase bold, uppercase, and italic:

**UPPERCASE BOLD**

identifies SAS keywords such as the names of functions or statements. In this example, the keyword ERROR is written in uppercase bold:

```
ERROR <message>;
```

**UPPERCASE**

identifies arguments that are literals.

In this example of the CMPMODEL= system option, the literals include BOTH, CATALOG, and XML:

```
CMPMODEL=BOTH | CATALOG | XML |
```

**italic**

identifies arguments or values that you supply. Items in italic represent user-supplied values that are either one of the following:
nonliteral arguments. In this example of the LINK statement, the argument *label* is a user-supplied value and therefore appears in italic:

```
LINK label;
```

nonliteral values that are assigned to an argument.

In this example of the FORMAT statement, the argument DEFAULT is assigned the variable *default-format*:

```
FORMAT variable(s) <format> <DEFAULT = default-format>;
```

---

### Special Characters

The syntax of SAS language elements can contain the following special characters:

- `=`
  - an equal sign identifies a value for a literal in some language elements such as system options.
  - In this example of the MAPS system option, the equal sign sets the value of MAPS:

    ```
    MAPS=location-of-maps
    ```

- `< >`
  - angle brackets identify optional arguments. A required argument is not enclosed in angle brackets.
  - In this example of the CAT function, at least one item is required:

    ```
    CAT (item-1 <, item-2, …>)
    ```

- `|`
  - a vertical bar indicates that you can choose one value from a group of values. Values that are separated by the vertical bar are mutually exclusive.
    - In this example of the CMPMODEL= system option, you can choose only one of the arguments:

      ```
      CMPMODEL=BOTH | CATALOG | XML
      ```

- `...`
  - an ellipsis indicates that the argument can be repeated. If an argument and the ellipsis are enclosed in angle brackets, then the argument is optional. The repeated argument must contain punctuation if it appears before or after the argument.
  - In this example of the CAT function, multiple `item` arguments are allowed, and they must be separated by a comma:

    ```
    CAT (item-1 <, item-2, …>)
    ```

- `'value'` or `"value"`
  - indicates that an argument that is enclosed in single or double quotation marks must have a value that is also enclosed in single or double quotation marks.
  - In this example of the FOOTNOTE statement, the argument `text` is enclosed in quotation marks:

    ```
    FOOTNOTE <n> <ods-format-options 'text' | "text">;
    ```
a semicolon indicates the end of a statement or CALL routine.
In this example, each statement ends with a semicolon:

data namegame;
    length color name $8;
    color = 'black';
    name = 'jack';
    game = trim(color) || name;
run;

References to SAS Libraries and External Files

Many SAS statements and other language elements refer to SAS libraries and external files. You can choose whether to make the reference through a logical name (a libref or fileref) or use the physical filename enclosed in quotation marks.

If you use a logical name, you typically have a choice of using a SAS statement (LIBNAME or FILENAME) or the operating environment's control language to make the reference. Several methods of referring to SAS libraries and external files are available, and some of these methods depend on your operating environment.

In the examples that use external files, SAS documentation uses the italicized phrase file-specification. In the examples that use SAS libraries, SAS documentation uses the italicized phrase SAS-library enclosed in quotation marks:

infile file-specification obs = 100;
libname libref 'SAS-library';
What’s New in SAS 9.4 DATA Step Statements

Overview

This document supports DATA step statements for SAS 9.4 and SAS Viya.

A highlighted, abbreviated notation of the SAS version and maintenance release specifies when a feature was added to SAS. For example, SAS 9.4M6 indicates that a feature was added during the sixth maintenance release of SAS 9.4.

Beginning in SAS Viya 3.5, the DESCENDING option is supported in a DATA step that runs in CAS. When running in CAS, the DESCENDING option cannot be used on the first variable in the BY statement.

In the May 2019 release of SAS 9.4M6 and SAS Viya 3.4, the STATUS option is now documented for the INFILE statement.

These are the new and enhanced features for SAS 9.4M6:

- The WHERE statement can be specified in a DATA step that is running in CAS.
- You can enable the LIST statement to write log data in hexadecimal format using the HEXLISTALL argument in the DATA statement.

In SAS Viya 3.4, these statement options are now documented:

- The MEMVAR option is documented for the INFILE and FILE statements.
- The RESET option is documented for the INFILE statement.

Beginning in SAS 9.4M5, global statements are available in the new SAS Global Statements: Reference.

New and enhanced features enable you to perform these tasks:

- create and name a variable that contains the observation number that was just read from the data set
- control whether a KEY= search should begin at the top of the index for the data set that is being read
Enhanced SAS DATA Step Statements

In **SAS Viya 3.5**, this SAS DATA step statement is enhanced:

**BY**
- The DESCENDING option in the BY statement is now supported for a DATA step that is running in CAS. The option is supported only for the second and subsequent variables in a CAS DATA step and not for the first variable in a BY statement. For more information, see “BY-Group Processing in CAS” in *SAS Cloud Analytic Services: DATA Step Programming*.

In **SAS Viya 3.4**, these SAS DATA step statements are enhanced:

**FILE**
- The MEMVAR option specifies a file to open. When the value of the MEMVAR= variable changes, the current file is closed and the new file is opened.

**INFILE**
- The MEMVAR option enables you to open the individual files contained in a directory or contained in a directory-based file such as a ZIP file.
- The RESET option specifies whether to close the current input file and open a new file, or close and reopen the current input file.

In **SAS 9.4**, these SAS DATA step statements have been enhanced:

**INFILE**
- SAS automatically normalizes imported copies of files that are created in Windows that are being read by SAS when you are running your session in UNIX. This enables easier sharing of files between the environments.

**MODIFY**
- A new option, CUROBS, creates and names a variable that contains the observation number that was just read from the data set.
- A new option, KEYRESET, controls whether a KEY= search should begin at the top of the index for the data set that is being read. When the value of the KEYRESET variable is 1, the index lookup begins at the top of the index. When the value of the KEYRESET variable is 0, the index lookup is not reset and the lookup continues where the prior lookup ended.

**SET**
- A new option, CUROBS, creates and names a variable that contains the observation number that was just read from the data set.
- A new option, KEYRESET, controls whether a KEY= search should begin at the top of the index for the data set that is being read. When the value of the KEYRESET variable is 1, the index lookup begins at the top of the index. When the value of the KEYRESET variable is 0, the index lookup is not reset and the lookup continues where the prior lookup ended.
Definition of Statements

A SAS statement is a string of SAS keywords, SAS names, special characters, and operators that instructs SAS to perform an operation or that gives information to SAS. Each SAS statement ends with a semicolon.

This documentation covers statements that are used in DATA step programming.

DATA Step Statements

Executable and Declarative Statements

DATA step statements are executable or declarative statements that can appear in the DATA step and run in SAS. Executable statements result in some action during individual iterations of the DATA step. Declarative statements supply information to SAS and take effect when the system compiles program statements.

The following tables show the executable and declarative statements that you can use in the SAS DATA step.

Table 1.1  Executable Statements in the DATA Step

<table>
<thead>
<tr>
<th>Executable Statements</th>
<th>IF-THEN/ELSE</th>
<th>PUT, Formatted</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABORT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Array Reference</td>
<td>INFILE</td>
<td>PUT, List</td>
</tr>
</tbody>
</table>
### Executable Statements

<table>
<thead>
<tr>
<th>Statement</th>
<th>Exceptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assignment</td>
<td>INPUT</td>
</tr>
<tr>
<td>CALL</td>
<td>GO TO</td>
</tr>
<tr>
<td>CONTINUE</td>
<td>INPUT, Column</td>
</tr>
<tr>
<td>DELETE</td>
<td>INPUT, Formatted</td>
</tr>
<tr>
<td>DESCRIBE</td>
<td>INPUT, List</td>
</tr>
<tr>
<td>DISPLAY</td>
<td>INPUT, Named</td>
</tr>
<tr>
<td>DO</td>
<td>LEAVE</td>
</tr>
<tr>
<td>DO, Iterative</td>
<td>LINK</td>
</tr>
<tr>
<td>DO UNTIL</td>
<td>LIST</td>
</tr>
<tr>
<td>DO WHILE</td>
<td>LOSTCARD</td>
</tr>
<tr>
<td>ERROR</td>
<td>MERGE</td>
</tr>
<tr>
<td>EXECUTE</td>
<td>MODIFY</td>
</tr>
<tr>
<td>FILE</td>
<td>Null</td>
</tr>
<tr>
<td>FILE, ODS</td>
<td>OUTPUT</td>
</tr>
<tr>
<td>IF, Subsetting</td>
<td>PUT, Column</td>
</tr>
<tr>
<td>Table 1.2 Declarative Statements in the DATA Step</td>
<td></td>
</tr>
</tbody>
</table>

### Declarative Statements

<table>
<thead>
<tr>
<th>Statement</th>
<th>Exceptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARRAY</td>
<td>DATALINES4</td>
</tr>
<tr>
<td>ATTRIB</td>
<td>DROP</td>
</tr>
<tr>
<td>BY</td>
<td>END</td>
</tr>
<tr>
<td>CARDS</td>
<td>FORMAT</td>
</tr>
<tr>
<td>CARDS4</td>
<td>INFORMAT</td>
</tr>
<tr>
<td>DATA</td>
<td>KEEP</td>
</tr>
<tr>
<td>DATALINES</td>
<td>LABEL</td>
</tr>
</tbody>
</table>
DATA step statements can be grouped into functional categories. For a list of DATA step statements by category, see "DATA Step Statements by Category" on page 8.
Dictionary of SAS DATA Step Statements

SAS Statements Documented in Other SAS Publications .................................... 7
DATA Step Statements by Category ..................................................................... 8

Dictionary ........................................................................................................ 15
ABORT Statement .............................................................................................. 15
ARRAY Statement ............................................................................................... 18
Array Reference Statement .................................................................................. 24
Assignment Statement ......................................................................................... 28
ATTRIB Statement ............................................................................................... 30
BY Statement ..................................................................................................... 34
CALL Statement .................................................................................................. 41
CARDS Statement ............................................................................................... 41
CARDS4 Statement .............................................................................................. 42
CATNAME Statement ........................................................................................... 42
CHECKPOINT EXECUTE_ALWAYS Statement ....................................................... 42
Comment Statement ............................................................................................ 42
CONTINUE Statement .......................................................................................... 42
DATA Statement .................................................................................................. 44
DATALINES Statement ........................................................................................ 55
DATALINES4 Statement ....................................................................................... 57
DELETE Statement ............................................................................................... 58
DESCRIBE Statement ......................................................................................... 59
DISPLAY Statement ........................................................................................... 60
DM Statement ..................................................................................................... 62
DO Statement ...................................................................................................... 62
DO Statement: Iterative ....................................................................................... 64
DO UNTIL Statement ........................................................................................... 68
DO WHILE Statement .......................................................................................... 69
DROP Statement .................................................................................................. 71
END Statement .................................................................................................... 72
ENDSAS Statement ............................................................................................. 73
ERROR Statement ............................................................................................... 73
EXECUTE Statement ............................................................................................ 75
FILE Statement .................................................................................................... 75
FILENAME Statement ........................................................................................... 97
FILENAME Statement: Azure Access Method ..................................................... 98
FILENAME Statement: CATALOG Access Method .............................................. 98
FILENAME Statement: CLIPBOARD Access Method ........................................ 98
FILENAME Statement: DATAURL Access Method ............................................ 98
FILENAME Statement: EMAIL (SMTP) Access Method ................................... 98
<table>
<thead>
<tr>
<th>Statement</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>FILENAME Statement: FILESRCV Access Method</td>
<td>98</td>
</tr>
<tr>
<td>FILENAME Statement: FTP Access Method</td>
<td>99</td>
</tr>
<tr>
<td>FILENAME Statement: Hadoop Access Method</td>
<td>99</td>
</tr>
<tr>
<td>FILENAME Statement: S3 Access Method</td>
<td>99</td>
</tr>
<tr>
<td>FILENAME Statement: SFTP Access Method</td>
<td>99</td>
</tr>
<tr>
<td>FILENAME Statement: SOCKET Access Method</td>
<td>99</td>
</tr>
<tr>
<td>FILENAME Statement: URL Access Method</td>
<td>99</td>
</tr>
<tr>
<td>FILENAME Statement: WebDAV Access Method</td>
<td>100</td>
</tr>
<tr>
<td>FILENAME Statement: ZIP Access Method</td>
<td>100</td>
</tr>
<tr>
<td>FOOTNOTE Statement</td>
<td>100</td>
</tr>
<tr>
<td>FORMAT Statement</td>
<td>104</td>
</tr>
<tr>
<td>GO TO Statement</td>
<td>105</td>
</tr>
<tr>
<td>IF Statement: Subsetting</td>
<td>108</td>
</tr>
<tr>
<td>IF-THEN/ELSE Statement</td>
<td>110</td>
</tr>
<tr>
<td>%INCLUDE Statement</td>
<td>110</td>
</tr>
<tr>
<td>INFILE Statement</td>
<td>142</td>
</tr>
<tr>
<td>INFORMAT Statement</td>
<td>146</td>
</tr>
<tr>
<td>INPUT Statement</td>
<td>163</td>
</tr>
<tr>
<td>INPUT Statement: Column</td>
<td>167</td>
</tr>
<tr>
<td>INPUT Statement: Formatted</td>
<td>171</td>
</tr>
<tr>
<td>INPUT Statement: List</td>
<td>178</td>
</tr>
<tr>
<td>KEEP Statement</td>
<td>182</td>
</tr>
<tr>
<td>LABEL Statement</td>
<td>184</td>
</tr>
<tr>
<td>Label: Statement</td>
<td>187</td>
</tr>
<tr>
<td>LEAVE Statement</td>
<td>190</td>
</tr>
<tr>
<td>LENGTH Statement</td>
<td>191</td>
</tr>
<tr>
<td>LIBNAME Statement</td>
<td>195</td>
</tr>
<tr>
<td>LIBNAME Statement: CVP Engine</td>
<td>195</td>
</tr>
<tr>
<td>LIBNAME Statement: JMP Engine</td>
<td>195</td>
</tr>
<tr>
<td>LIBNAME Statement: JSON Engine</td>
<td>195</td>
</tr>
<tr>
<td>LIBNAME Statement: WebDAV Server Access</td>
<td>196</td>
</tr>
<tr>
<td>LINK Statement</td>
<td>196</td>
</tr>
<tr>
<td>LIST Statement</td>
<td>198</td>
</tr>
<tr>
<td>%LIST Statement</td>
<td>202</td>
</tr>
<tr>
<td>LOCK Statement</td>
<td>202</td>
</tr>
<tr>
<td>LOCKDOWN Statement</td>
<td>202</td>
</tr>
<tr>
<td>LOSTCARD Statement</td>
<td>202</td>
</tr>
<tr>
<td>MERGE Statement</td>
<td>204</td>
</tr>
<tr>
<td>MISSING Statement</td>
<td>212</td>
</tr>
<tr>
<td>MODIFY Statement</td>
<td>212</td>
</tr>
<tr>
<td>Null Statement</td>
<td>232</td>
</tr>
<tr>
<td>OPTIONS Statement</td>
<td>232</td>
</tr>
<tr>
<td>OUTPUT Statement</td>
<td>233</td>
</tr>
<tr>
<td>PAGE Statement</td>
<td>236</td>
</tr>
<tr>
<td>PUT Statement</td>
<td>236</td>
</tr>
<tr>
<td>PUT Statement: Column</td>
<td>256</td>
</tr>
<tr>
<td>PUT Statement: Formatted</td>
<td>258</td>
</tr>
<tr>
<td>PUT Statement: List</td>
<td>262</td>
</tr>
<tr>
<td>PUT Statement: Named</td>
<td>267</td>
</tr>
<tr>
<td>PUTLOG Statement</td>
<td>269</td>
</tr>
<tr>
<td>REDIRECT Statement</td>
<td>271</td>
</tr>
<tr>
<td>REMOVE Statement</td>
<td>273</td>
</tr>
<tr>
<td>RENAME Statement</td>
<td>275</td>
</tr>
</tbody>
</table>
SAS Statements Documented in Other SAS Publications

Some statements are documented with related subject matter in other SAS publications.

- SAS Global Statements: Reference
- Base SAS Procedures Guide
- SAS Companion for Windows
- SAS Companion for UNIX Environments
- SAS Companion for z/OS
- SAS DS2 Language Reference
- SAS FedSQL Language Reference
- Application Messaging with SAS
- SAS Language Interfaces to Metadata
- SAS Macro Language: Reference
- SAS Scalable Performance Data Engine: Reference
- SAS/ACCESS for Relational Databases: Reference
- SAS/CONNECT User’s Guide
- SAS/SHARE User’s Guide
**DATA Step Statements by Category**

In addition to being either executable or declarative, SAS DATA step statements can be grouped into these functional categories:

*Table 2.1 Categories of DATA Step Statements*

<table>
<thead>
<tr>
<th>Statements Category</th>
<th>Functionality</th>
</tr>
</thead>
</table>
| Action              | - Creates and modifies variables.  
                  | - Selects only certain observations to process in the DATA step.  
                  | - Looks for errors in the input data.  
                  | - Works with observations as they are being created. |
| CAS                 | - Specifies statements that run on the CAS server. |
| Control             | - Skips statements for certain observations.  
                  | - Changes the order that statements are executed.  
                  | - Transfers control from one part of a program to another. |
| File-Handling       | - Works with files used as input to the data set.  
                  | - Works with files to be written by the DATA step. |
| Information         | - Gives SAS additional information about the program data vector.  
                  | - Gives SAS additional information about the data set or data sets that are being created. |
| Window Display      | - Displays and customizes windows. |

Some statements run in SAS only, and some statements run in SAS and on the CAS server. If CAS is specified for the statement category, then the statement runs in SAS and on the CAS server. If CAS is not specified for the statement category, then the statement runs in SAS only. For example, the DO statement runs in SAS and on the CAS server, so CAS is specified as a category. The ABORT statement runs in SAS only, so CAS is not specified as a category.

This table lists and briefly describes the DATA step statements by category.
<table>
<thead>
<tr>
<th>Category</th>
<th>Language Elements</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Action</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ABORT Statement (p. 15)</td>
<td>Stops executing the current DATA step, SAS job, or SAS session.</td>
</tr>
<tr>
<td></td>
<td>Assignment Statement (p. 28)</td>
<td>Evaluates an expression and stores the result in a variable.</td>
</tr>
<tr>
<td></td>
<td>CALL Statement (p. 41)</td>
<td>Invokes a SAS CALL routine.</td>
</tr>
<tr>
<td></td>
<td>DELETE Statement (p. 58)</td>
<td>Stops processing the current observation.</td>
</tr>
<tr>
<td></td>
<td>DESCRIBE Statement (p. 59)</td>
<td>Retrieves source code from a stored compiled DATA step program or a DATA step view.</td>
</tr>
<tr>
<td></td>
<td>ERROR Statement (p. 73)</td>
<td>Sets <em>ERROR</em> to 1. A message written to the SAS log is optional.</td>
</tr>
<tr>
<td></td>
<td>EXECUTE Statement (p. 75)</td>
<td>Executes a stored compiled DATA step program.</td>
</tr>
<tr>
<td></td>
<td>IF Statement: Subsetting (p. 105)</td>
<td>Continues processing only those observations that meet the condition of the specified expression.</td>
</tr>
<tr>
<td></td>
<td>LIST Statement (p. 198)</td>
<td>Writes to the SAS log the input data record for the observation that is being processed.</td>
</tr>
<tr>
<td></td>
<td>LOSTCARD Statement (p. 202)</td>
<td>Resynchronizes the input data when SAS encounters a missing or invalid record in data that has multiple records per observation.</td>
</tr>
<tr>
<td></td>
<td>OUTPUT Statement (p. 233)</td>
<td>Writes the current observation to a SAS data set.</td>
</tr>
<tr>
<td></td>
<td>PUTLOG Statement (p. 269)</td>
<td>Writes a message to the SAS log.</td>
</tr>
<tr>
<td></td>
<td>REDIRECT Statement (p. 271)</td>
<td>Points to different input or output SAS data sets when you execute a stored program.</td>
</tr>
<tr>
<td></td>
<td>REMOVE Statement (p. 273)</td>
<td>Deletes an observation from a SAS data set.</td>
</tr>
<tr>
<td></td>
<td>REPLACE Statement (p. 277)</td>
<td>Replaces an observation in the same location.</td>
</tr>
<tr>
<td></td>
<td>STOP Statement (p. 305)</td>
<td>Stops execution of the current DATA step.</td>
</tr>
<tr>
<td></td>
<td>Sum Statement (p. 307)</td>
<td>Adds the result of an expression to an accumulator variable.</td>
</tr>
<tr>
<td></td>
<td>WHERE Statement (p. 315)</td>
<td>Selects observations from SAS data sets that meet a particular condition.</td>
</tr>
<tr>
<td><strong>CAS</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ARRAY Statement (p. 18)</td>
<td>Defines the elements of an array.</td>
</tr>
<tr>
<td></td>
<td>Array Reference Statement (p. 24)</td>
<td>Describes the elements in an array to be processed.</td>
</tr>
<tr>
<td>Category</td>
<td>Language Elements</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>------------------</td>
<td>-------------</td>
</tr>
<tr>
<td></td>
<td>Assignment Statement (p. 28)</td>
<td>Evaluates an expression and stores the result in a variable.</td>
</tr>
<tr>
<td></td>
<td>ATTRIB Statement (p. 30)</td>
<td>Associates a format, informat, label, and length with one or more variables.</td>
</tr>
<tr>
<td></td>
<td>BY Statement (p. 34)</td>
<td>Controls the operation of a SET, MERGE, MODIFY, or UPDATE statement in the DATA step and sets up special grouping variables.</td>
</tr>
<tr>
<td></td>
<td>CALL Statement (p. 41)</td>
<td>Invokes a SAS CALL routine.</td>
</tr>
<tr>
<td></td>
<td>CONTINUE Statement (p. 42)</td>
<td>Stops processing the current DO-loop iteration and resumes processing the next iteration.</td>
</tr>
<tr>
<td></td>
<td>DATA Statement (p. 44)</td>
<td>Begins a DATA step and provides names for any output such as SAS data sets, views, or programs.</td>
</tr>
<tr>
<td></td>
<td>DELETE Statement (p. 58)</td>
<td>Stops processing the current observation.</td>
</tr>
<tr>
<td></td>
<td>DO Statement (p. 62)</td>
<td>Specifies a group of statements to be executed as a unit.</td>
</tr>
<tr>
<td></td>
<td>DO Statement: Iterative (p. 64)</td>
<td>Executes statements between the DO and END statements repetitively, based on the value of an index variable.</td>
</tr>
<tr>
<td></td>
<td>DO UNTIL Statement (p. 68)</td>
<td>Executes statements in a DO loop repetitively until a condition is true.</td>
</tr>
<tr>
<td></td>
<td>DO WHILE Statement (p. 69)</td>
<td>Executes statements in a DO loop repetitively while a condition is true.</td>
</tr>
<tr>
<td></td>
<td>DROP Statement (p. 71)</td>
<td>Excludes variables from output SAS data sets.</td>
</tr>
<tr>
<td></td>
<td>END Statement (p. 72)</td>
<td>Ends DO group or SELECT group processing.</td>
</tr>
<tr>
<td></td>
<td>ERROR Statement (p. 73)</td>
<td>Sets <em>ERROR</em> to 1. A message written to the SAS log is optional.</td>
</tr>
<tr>
<td></td>
<td>FILE Statement (p. 75)</td>
<td>Specifies the current output file for PUT statements.</td>
</tr>
<tr>
<td></td>
<td>FORMAT Statement (p. 100)</td>
<td>Associates formats with variables.</td>
</tr>
<tr>
<td></td>
<td>GO TO Statement (p. 104)</td>
<td>Directs program execution immediately to the statement label that is specified and, if followed by a RETURN statement, returns execution to the beginning of the DATA step.</td>
</tr>
<tr>
<td></td>
<td>IF Statement: Subsetting (p. 105)</td>
<td>Continues processing only those observations that meet the condition of the specified expression.</td>
</tr>
<tr>
<td></td>
<td>IF-THEN/ELSE Statement (p. 108)</td>
<td>Executes a SAS statement for observations that meet specific conditions.</td>
</tr>
<tr>
<td>Category</td>
<td>Language Elements</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------</td>
<td>---------------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>INFORMAT Statement</td>
<td>Associates informats with variables.</td>
<td></td>
</tr>
<tr>
<td>KEEP Statement</td>
<td>Specifies the variables to include in output SAS data sets.</td>
<td></td>
</tr>
<tr>
<td>LABEL Statement</td>
<td>Assigns descriptive labels to variables.</td>
<td></td>
</tr>
<tr>
<td>Label: Statement</td>
<td>Identifies a statement that is referred to by another statement.</td>
<td></td>
</tr>
<tr>
<td>LEAVE Statement</td>
<td>Stops processing the current loop and resumes with the next statement in the sequence.</td>
<td></td>
</tr>
<tr>
<td>LENGTH Statement</td>
<td>Specifies the number of bytes for storing character and numeric variables, or the number of characters for storing VARCHAR variables.</td>
<td></td>
</tr>
<tr>
<td>LINK Statement</td>
<td>Directs program execution immediately to the statement label that is specified and, if followed by a RETURN statement, returns execution to the statement that follows the LINK statement.</td>
<td></td>
</tr>
<tr>
<td>LIST Statement</td>
<td>Writes to the SAS log the input data record for the observation that is being processed.</td>
<td></td>
</tr>
<tr>
<td>LOSTCARD Statement</td>
<td>Resynchronizes the input data when SAS encounters a missing or invalid record in data that has multiple records per observation.</td>
<td></td>
</tr>
<tr>
<td>MERGE Statement</td>
<td>Joins observations from two or more SAS data sets into a single observation.</td>
<td></td>
</tr>
<tr>
<td>OUTPUT Statement</td>
<td>Writes the current observation to a SAS data set.</td>
<td></td>
</tr>
<tr>
<td>PUT Statement</td>
<td>Writes lines to the SAS log, to the SAS output window, or to an external location that is specified in the most recent FILE statement.</td>
<td></td>
</tr>
<tr>
<td>PUT Statement: Column</td>
<td>Writes variable values in the specified columns in the output line.</td>
<td></td>
</tr>
<tr>
<td>PUT Statement: Formatted</td>
<td>Writes variable values with the specified format in the output line.</td>
<td></td>
</tr>
<tr>
<td>PUT Statement: List</td>
<td>Writes variable values and the specified character strings in the output line.</td>
<td></td>
</tr>
<tr>
<td>PUT Statement: Named</td>
<td>Writes variable values after the variable name and an equal sign.</td>
<td></td>
</tr>
<tr>
<td>PUTLOG Statement</td>
<td>Writes a message to the SAS log.</td>
<td></td>
</tr>
<tr>
<td>RENAME Statement</td>
<td>Specifies new names for variables in output SAS data sets.</td>
<td></td>
</tr>
<tr>
<td>Category</td>
<td>Language Elements</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-------------------</td>
<td>-------------</td>
</tr>
<tr>
<td></td>
<td>RETAIN Statement (p. 279)</td>
<td>Causes a variable that is created by an INPUT or assignment statement to retain its value from one iteration of the DATA step to the next.</td>
</tr>
<tr>
<td></td>
<td>RETURN Statement (p. 284)</td>
<td>Stops executing statements at the current point in the DATA step and returns to a predetermined point in the step.</td>
</tr>
<tr>
<td></td>
<td>SELECT Statement (p. 286)</td>
<td>Executes one of several statements or groups of statements.</td>
</tr>
<tr>
<td></td>
<td>SET Statement (p. 290)</td>
<td>Reads an observation from one or more SAS data sets.</td>
</tr>
<tr>
<td></td>
<td>STOP Statement (p. 305)</td>
<td>Stops execution of the current DATA step.</td>
</tr>
<tr>
<td></td>
<td>Sum Statement (p. 307)</td>
<td>Adds the result of an expression to an accumulator variable.</td>
</tr>
<tr>
<td></td>
<td>WHERE Statement (p. 315)</td>
<td>Selects observations from SAS data sets that meet a particular condition.</td>
</tr>
<tr>
<td>Control</td>
<td>CONTINUE Statement (p. 42)</td>
<td>Stops processing the current DO-loop iteration and resumes processing the next iteration.</td>
</tr>
<tr>
<td></td>
<td>DO Statement (p. 62)</td>
<td>Specifies a group of statements to be executed as a unit.</td>
</tr>
<tr>
<td></td>
<td>DO Statement: Iterative (p. 64)</td>
<td>Executes statements between the DO and END statements repetitively, based on the value of an index variable.</td>
</tr>
<tr>
<td></td>
<td>DO UNTIL Statement (p. 68)</td>
<td>Executes statements in a DO loop repetitively until a condition is true.</td>
</tr>
<tr>
<td></td>
<td>DO WHILE Statement (p. 69)</td>
<td>Executes statements in a DO loop repetitively while a condition is true.</td>
</tr>
<tr>
<td></td>
<td>END Statement (p. 72)</td>
<td>Ends DO group or SELECT group processing.</td>
</tr>
<tr>
<td></td>
<td>GO TO Statement (p. 104)</td>
<td>Directs program execution immediately to the statement label that is specified and, if followed by a RETURN statement, returns execution to the beginning of the DATA step.</td>
</tr>
<tr>
<td></td>
<td>IF-THEN/ELSE Statement (p. 108)</td>
<td>Executes a SAS statement for observations that meet specific conditions.</td>
</tr>
<tr>
<td></td>
<td>Label: Statement (p. 187)</td>
<td>Identifies a statement that is referred to by another statement.</td>
</tr>
<tr>
<td></td>
<td>LEAVE Statement (p. 190)</td>
<td>Stops processing the current loop and resumes with the next statement in the sequence.</td>
</tr>
<tr>
<td></td>
<td>LINK Statement (p. 196)</td>
<td>Directs program execution immediately to the statement label that is specified and, if followed by a RETURN statement, returns execution to the statement that follows the LINK statement.</td>
</tr>
<tr>
<td>Category</td>
<td>Language Elements</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------</td>
<td>--------------------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>RETURN Statement (p. 284)</strong></td>
<td><strong>Description</strong></td>
<td>Stops executing statements at the current point in the DATA step and returns to a predetermined point in the step.</td>
</tr>
<tr>
<td><strong>SELECT Statement (p. 286)</strong></td>
<td><strong>Description</strong></td>
<td>Executes one of several statements or groups of statements.</td>
</tr>
<tr>
<td><strong>FILE-Handling</strong></td>
<td><strong>BY Statement (p. 34)</strong></td>
<td><strong>Description</strong></td>
</tr>
<tr>
<td></td>
<td><strong>SELECT Statement (p. 286)</strong></td>
<td>Executes one of several statements or groups of statements.</td>
</tr>
<tr>
<td><strong>Card-Handling</strong></td>
<td><strong>BY Statement (p. 34)</strong></td>
<td><strong>Description</strong></td>
</tr>
<tr>
<td></td>
<td><strong>CARDS Statement (p. 41)</strong></td>
<td>Specifies that lines of data follow the statement.</td>
</tr>
<tr>
<td></td>
<td><strong>CARDS4 Statement (p. 42)</strong></td>
<td>Specifies that the lines of data that follow the statement contain internal semicolons.</td>
</tr>
<tr>
<td></td>
<td><strong>DATA Statement (p. 44)</strong></td>
<td>Begins a DATA step and provides names for any output such as SAS data sets, views, or programs.</td>
</tr>
<tr>
<td></td>
<td><strong>DATALINES Statement (p. 55)</strong></td>
<td>Specifies that lines of data follow the statement.</td>
</tr>
<tr>
<td></td>
<td><strong>DATALINES4 Statement (p. 57)</strong></td>
<td>Specifies that the lines of data that follow the statement contain internal semicolons.</td>
</tr>
<tr>
<td></td>
<td><strong>FILE Statement (p. 75)</strong></td>
<td>Specifies the current output file for PUT statements.</td>
</tr>
<tr>
<td></td>
<td><strong>INFILE Statement (p. 110)</strong></td>
<td>Specifies an external file to read with an INPUT statement.</td>
</tr>
<tr>
<td></td>
<td><strong>INPUT Statement (p. 146)</strong></td>
<td>Describes the arrangement of values in the input data record and assigns input values to the corresponding SAS variables.</td>
</tr>
<tr>
<td></td>
<td><strong>INPUT Statement: Column (p. 163)</strong></td>
<td>Reads input values from specified columns and assigns the values to the corresponding SAS variables.</td>
</tr>
<tr>
<td></td>
<td><strong>INPUT Statement: Formatted (p. 167)</strong></td>
<td>Reads input values with specified informats and assigns them to the corresponding SAS variables.</td>
</tr>
<tr>
<td></td>
<td><strong>INPUT Statement: List (p. 171)</strong></td>
<td>Scans the input data record for input values and assigns them to the corresponding SAS variables.</td>
</tr>
<tr>
<td></td>
<td><strong>INPUT Statement: Named (p. 178)</strong></td>
<td>Reads data values that appear after a variable name that is followed by an equal sign and assigns the values to corresponding SAS variables.</td>
</tr>
<tr>
<td></td>
<td><strong>MERGE Statement (p. 204)</strong></td>
<td>Joins observations from two or more SAS data sets into a single observation.</td>
</tr>
<tr>
<td></td>
<td><strong>MODIFY Statement (p. 212)</strong></td>
<td>Replaces, deletes, and appends observations in an existing SAS data set in place but does not create an additional copy.</td>
</tr>
<tr>
<td>Category</td>
<td>Language Elements</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>---------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>PUT Statement</td>
<td>(p. 236)</td>
<td>Writes lines to the SAS log, to the SAS output window, or to an external location that is specified in the most recent FILE statement.</td>
</tr>
<tr>
<td>PUT Statement: Column</td>
<td>(p. 256)</td>
<td>Writes variable values in the specified columns in the output line.</td>
</tr>
<tr>
<td>PUT Statement: Formatted</td>
<td>(p. 258)</td>
<td>Writes variable values with the specified format in the output line.</td>
</tr>
<tr>
<td>PUT Statement: List</td>
<td>(p. 262)</td>
<td>Writes variable values and the specified character strings in the output line.</td>
</tr>
<tr>
<td>PUT Statement: Named</td>
<td>(p. 267)</td>
<td>Writes variable values after the variable name and an equal sign.</td>
</tr>
<tr>
<td>SET Statement</td>
<td>(p. 290)</td>
<td>Reads an observation from one or more SAS data sets.</td>
</tr>
<tr>
<td>UPDATE Statement</td>
<td>(p. 309)</td>
<td>Updates a master file by applying transactions.</td>
</tr>
<tr>
<td>Information</td>
<td>ARRAY Statement (p. 18)</td>
<td>Defines the elements of an array.</td>
</tr>
<tr>
<td></td>
<td>Array Reference Statement (p. 24)</td>
<td>Describes the elements in an array to be processed.</td>
</tr>
<tr>
<td></td>
<td>ATTRIB Statement (p. 30)</td>
<td>Associates a format, informat, label, and length with one or more variables.</td>
</tr>
<tr>
<td></td>
<td>DROP Statement (p. 71)</td>
<td>Excludes variables from output SAS data sets.</td>
</tr>
<tr>
<td></td>
<td>FORMAT Statement (p. 100)</td>
<td>Associates formats with variables.</td>
</tr>
<tr>
<td></td>
<td>INFORMAT Statement (p. 142)</td>
<td>Associates informats with variables.</td>
</tr>
<tr>
<td></td>
<td>KEEP Statement (p. 182)</td>
<td>Specifies the variables to include in output SAS data sets.</td>
</tr>
<tr>
<td></td>
<td>LABEL Statement (p. 184)</td>
<td>Assigns descriptive labels to variables.</td>
</tr>
<tr>
<td></td>
<td>LENGTH Statement (p. 191)</td>
<td>Specifies the number of bytes for storing character and numeric variables, or the number of characters for storing VARCHAR variables.</td>
</tr>
<tr>
<td></td>
<td>RENAME Statement (p. 275)</td>
<td>Specifies new names for variables in output SAS data sets.</td>
</tr>
<tr>
<td></td>
<td>RETAIN Statement (p. 279)</td>
<td>Causes a variable that is created by an INPUT or assignment statement to retain its value from one iteration of the DATA step to the next.</td>
</tr>
<tr>
<td>Window Display</td>
<td>DISPLAY Statement (p. 60)</td>
<td>Displays a window that is created with the WINDOW statement.</td>
</tr>
<tr>
<td></td>
<td>WINDOW Statement (p. 323)</td>
<td>Creates customized windows for your applications.</td>
</tr>
</tbody>
</table>
ABORT Statement

Stops executing the current DATA step, SAS job, or SAS session.

Valid in: DATA step
Category: Action
Type: Executable
Restriction: This statement is not supported in a DATA step that runs in CAS.
See: ABORT Statement in SAS Companion for Windows, SAS Companion for z/OS, and SAS Companion for UNIX Environments

Syntax

`ABORT <ABEND | CANCEL <FILE> | RETURN > <n> <NOLIST>;`

Without Arguments

If you specify no argument, the ABORT statement triggers the following actions under these methods of operation:

batch mode and noninteractive mode
- stops processing the current DATA step and writes an error message to the SAS log. Data sets can contain an incomplete number of observations or no observations, depending on when SAS encountered the ABORT statement.
- sets the OBS= system option to 0.
- continues limited processing of the remainder of the SAS job, including executing macro statements, executing system options statements, and checking the syntax of program statements.
- creates output data sets for subsequent DATA and PROC steps with no observations.

windowing environment
- stops processing the current DATA step.
- creates a data set that contains the observations that are processed before the ABORT statement is encountered.
- prints a message to the SAS log that an ABORT statement terminated the DATA step.
- continues processing any DATA and PROC steps that follow the ABORT statement.

interactive line mode
- stops processing the current DATA step. Any further DATA steps or procedures execute normally.
Arguments

**ABEND**
causes abnormal termination of the current SAS job or session. Actions depend on the method of operation.
- batch mode and noninteractive mode
  - stops processing immediately.
  - sends an error message to the SAS log that states that execution was terminated by the ABEND option of the ABORT statement.
  - does not execute any subsequent statements or check syntax.
  - returns control to the operating environment; further action is based on how your operating environment and your site treat jobs that end abnormally.
- windowing environment and interactive line mode
  - stops processing immediately and returns you to your operating environment.

**CANCEL <FILE>**
causes the execution of the submitted statements to be canceled. Actions depend on the method of operation.
- batch mode and noninteractive mode
  - terminates the entire SAS program and SAS system.
  - writes an error message to the SAS log.
- windowing environment and interactive line mode
  - clears only the current submitted program.
  - does not affect other subsequent submitted programs.
  - writes an error to the SAS log.
- workspace server and stored process server
  - clears only the currently submitted program.
  - does not affect other subsequent submit calls.
  - writes an error message to the SAS log.
- SAS IntrNet application server
  - creates a separate execution for each request and submits the request code. A CANCEL argument in the request code clears the current submitted code but does not terminate the execution or the SAS session.

**FILE**
when coded as an option to the CANCEL argument in an autoexec file or in a %INCLUDE file, causes only the contents of the autoexec file or %INCLUDE file to be cleared by the ABORT statement. Other submitted source statements are executed after the autoexec file or %INCLUDE file.

**Restriction**
The CANCEL argument cannot be submitted using SAS/SHARE, SAS/CONNECT, or SAS/AF.

**Note**
When the ABORT CANCEL FILE option is executed within a %INCLUDE file, all open macros are closed and execution resumes at the next source line of code.
RETURN
causes the immediate normal termination of the current SAS job or session. Results depend on the method of operation.
- batch mode and noninteractive mode
- stops processing immediately.
- sends an error message to the SAS log that states that execution was terminated by the RETURN option in the ABORT statement.
- does not execute any subsequent statements or check syntax.
- returns control to your operating environment with a condition code that indicates an error.
- windowing environment and interactive line mode
- stops processing immediately and returns you to your operating environment.

\( n \)
is an integer value that enables you to specify a condition code.
- when used with the CANCEL argument, the value is placed in the SYSINFO automatic macro variable.
- when not used with the CANCEL argument, the error code that is returned by SAS is ERROR. The value of ERROR depends on the operating system. The condition code \( n \) is returned to the operating system as the final SAS System exit code.

NOLIST
suppresses the output of all variables to the SAS log.

Requirement
NOLIST must be the last option in the ABORT statement.

Details
General Information
The ABORT statement causes SAS to stop processing the current DATA step. What happens next depends on the following information:
- the method that you use to submit your SAS statements
- the arguments that you use with ABORT
- your operating environment

The ABORT statement usually appears in a clause of an IF-THEN statement or in a SELECT statement that is designed to stop processing when an error occurs.

When you execute an ABORT statement in a DATA step, SAS does not use data sets that were created in the step to replace existing data sets with the same name.

Note: The return code that is generated by the ABORT statement is ignored by SAS if the system option ERRORABEND is in effect.

Operating Environment Information: The only difference between the ABEND and RETURN options is that with ABEND further action is based on how your operating environment and site treat jobs that end abnormally. RETURN simply returns a condition code that indicates an error.
Comparisons

- In batch or noninteractive mode, the ABORT and STOP statements have different effects. The ABORT statement stops processing and sets the value of the automatic variable _ERROR_ to 1. The STOP statement stops processing the current DATA step and continues processing with the next step.

- When you use the SAS windowing environment or interactive line mode, the ABORT and STOP statements stop processing. The ABORT statement sets the value of the automatic variable _ERROR_ to 1. The STOP statement does not set a value.

Example: Stopping Execution of SAS

This example uses the ABORT statement as part of an IF-THEN statement to stop processing when SAS encounters a data value that would otherwise cause a division-by-zero condition.

```sas
data test;
  mass=100;
  volume=0;
  if volume=0 then abort 255;
  density=mass/volume;
run;
```

When the ABORT statement executes, SAS returns the condition code 255 to the operating environment.

See Also

**Statements:**

- “STOP Statement” on page 305

---

**ARRAY Statement**

Defines the elements of an array.

**Valid in:** DATA step

**Categories:** CAS Information

**Type:** Declarative

**Restriction:** Variables with a VARCHAR data type are supported when an array is initialized.

**Requirement:** The CAS engine is required if you want to preserve a variable as a VARCHAR data type when reading the variable in or writing it out using the DATA step.

**Syntax**

```
ARRAY array-name { subscript }
<$ length | length | VARCHAR(length) | VARCHAR(*)>
```
Arguments

array-name

specifies the name of the array.

Restriction

Array-name must be a SAS name that is not the name of a SAS variable in the same DATA step.

CAUTION

Using the name of a SAS function as an array name can cause unpredictable results. If you inadvertently use a function name as the name of the array, SAS treats parenthetical references that involve the name as array references, not function references, for the duration of the DATA step. A warning message is written to the SAS log.

{subscript}

describes the number and arrangement of elements in the array by using an asterisk, a number, or a range of numbers. Subscript has one of these forms:

{dimension-size(s)}

specifies the number of elements in each dimension of the array. Dimension-size is a numeric representation of either the number of elements in a one-dimensional array or the number of elements in each dimension of a multidimensional array.

Tip

You can enclose the subscript in braces ({}), brackets ([ ] ) or parentheses (( )).

Examples

This ARRAY statement defines a one-dimensional array that is named SIMPLE. The SIMPLE array groups together three variables that are named RED, GREEN, and YELLOW:

array simple{3} red green yellow;

An array with more than one dimension is known as a multidimensional array. You can have any number of dimensions in a multidimensional array. For example, a two-dimensional array provides row and column arrangement of array elements. SAS places variables into a two-dimensional array by filling all rows in order, beginning at the upper left corner of the array (known as row-major order). This statement defines a two-dimensional array with five rows and three columns:

array x{5,3} score1-score15;

{<lower >:upper, ...<lower >: upper>}

are the bounds of each dimension of an array, where lower is the lower bound of that dimension and upper is the upper bound.

Range

In most explicit arrays, the subscript in each dimension of the array ranges from 1 to n, where n is the number of elements in that dimension.

Tips

For most arrays, 1 is a convenient lower bound. Therefore, you do not need to specify the lower and upper bounds. However, specifying both bounds is useful when the array dimensions have a convenient beginning point other than 1.
To reduce the computational time that is needed for subscript evaluation, specify a lower bound of 0.

**Examples**

By default, the value of each dimension in this example is the upper bound of that dimension.

```
array x{5,3} score1-score15;
```

As an alternative, this ARRAY statement is a longhand version of the previous example:

```
array x{1:5,1:3} score1-score15;
```

{**} specifies that SAS is to determine the subscript by counting the variables in the array. When you specify the asterisk, also include array-elements.

**Restriction**

You cannot use the asterisk with _TEMPORARY_ arrays or when you define a multidimensional array.

$ specifies that the elements in the array are character elements.

**Tip**

The dollar sign is not necessary if the elements have been previously defined as character elements.

**length**

specifies the length of elements in the array that have not been previously assigned a length. For numeric and character variables, this is the maximum number of bytes stored in the variable.

**Range**

For numeric variables, 2 to 8 bytes or 3 to 8 bytes, depending on your operating environment. For character variables, 1 to 32767 bytes under all operating environments.

**VARCHAR**

specifies that the preceding variables are character variables of type VARCHAR.

**length**

specifies the maximum number of characters stored for VARCHAR variables.

**Range**

1 to 536,870,911 characters for VARCHAR variables.

**Restriction**

When assigning a character constant to a VARCHAR variable, the character constant is limited to 32767 bytes.

**** specifies to support the maximum length allowed for VARCHAR variables: 536,870,911 characters.

**array-elements**

specifies the names of the elements that make up the array. Array-elements must be either all numeric or all character, and they can be listed in any order. The elements can be named variables or temporary data elements.

**variables**

lists variable names.

**Range**

The names must be either variables that you define in the ARRAY statement or variables that SAS creates by concatenating the array name and a number. For example,
when the subscript is a number (not the asterisk), you do not need to name each variable in the array. Instead, SAS creates variable names by concatenating the array name and the numbers 1, 2, 3, …n.

**Restriction**

If you use _ALL_, all the previously defined variables must be of the same type.

**Tips**

These SAS variable lists enable you to reference variables that have been previously defined in the same DATA step:

- `_NUMERIC_` specifies all numeric variables.
- `_CHARACTER_` specifies all character variables.
- `_ALL_` specifies all variables.

Name range lists enable you to define a range of numeric or character variables based on the order in which the variables are defined.

**See**

“SAS Variable Lists” in *SAS Language Reference: Concepts*

**Example**

“Example 1: Defining Arrays” on page 23

---

**_TEMPORARY_**

creates a list of temporary data elements.

**Range**

Temporary data elements can be numeric or character.

**Tips**

Temporary data elements behave like DATA step variables with these exceptions:

- They do not have names. Refer to temporary data elements by the array name and dimension.
- They do not appear in the output data set.
- You cannot use the special subscript asterisk (*) to refer to all the elements.
- Temporary data element values are always automatically retained, rather than being reset to missing at the beginning of the next iteration of the DATA step.
- Arrays of temporary elements are useful when the only purpose for creating an array is to perform a calculation. To preserve the result of the calculation, assign it to a variable. You can improve performance time by using temporary data elements.

**_(initial-value-list)_**

gives initial values for the corresponding elements in the array. The values for elements can be numbers or character strings. You must enclose all character strings in quotation marks. To specify one or more initial values directly, use this format:

**_(initial-value(s))_**
To specify an iteration factor and nested sublists for the initial values, use this format:

<constant-iter-value*> <(>constant value | constant-sublist<)>>

**Restriction**
If you specify both an initial-value-list and array-elements, then array-elements must be listed before initial-value-list in the ARRAY statement.

**Tips**
You can assign initial values to both variables and temporary data elements.

Elements and values are matched by position. If there are more array elements than initial values, the remaining array elements receive missing values and SAS issues a warning.

You can separate the values in the initial value list with either a comma or a blank space.

You can also use a shorthand notation for specifying a range of sequential integers. The increment is always +1.

If you have not previously specified the attributes of the array elements (such as length or type), the attributes of any initial values that you specify are automatically assigned to the corresponding array element. Initial values are retained until a new value is assigned to the array element.

When any (or all) elements are assigned initial values, all elements behave as if they were named in a RETAIN statement.

**Example**
These examples show how to use the iteration factor and nested sublists. All of these ARRAY statements contain the same initial value list.

```sas
ARRAY x{10} x1-x10 (10*5);
ARRAY x{10} x1-x10 (5*(5 5));
ARRAY x{10} x1-x10 (5 5 3*(5 5) 5 5);
ARRAY x{10} x1-x10 (2*(5 5) 5 2*(5 5));
ARRAY x{10} x1-x10 (2*(5 2*(5 5)));
```

**Examples**
“Example 2: Assigning Initial Numeric Values” on page 23

“Example 3: Defining Initial Character Values” on page 23

**Details**
The ARRAY statement defines a set of elements that you plan to process as a group. You refer to elements of the array by the array name and subscript. Because you usually want to process more than one element in an array, arrays are often referenced within DO groups.

**Comparisons**
- Arrays in the SAS language are different from arrays in many other languages. A SAS array is simply a convenient way of temporarily identifying a group of variables. It is not a data structure, and array-name is not a variable.
An ARRAY statement defines an array. An array reference uses an array element in a program statement.

Examples:

Example 1: Defining Arrays
These examples show how to use the ARRAY statement to define an array.

- array rain {5} janr febr marr aprr mayr;
- array days{7} d1-d7;
- array month{*} jan feb jul oct nov;
- array x{*} _NUMERIC_;
- array qbx{10};
- array meal{3};

Example 2: Assigning Initial Numeric Values
These examples show how to define an array and assign initial numeric values to the elements in the array.

- array test{4} t1 t2 t3 t4 (90 80 70 70);
- array test{4} t1-t4 (90 80 2*70);
- array test{4} _TEMPORARY_ (90 80 70 70);

Example 3: Defining Initial Character Values
This example shows how to define an array and assign initial character values to the elements in the array.

- array test2{*} $ a1 a2 a3 ('a','b','c');

Example 4: Defining More Advanced Arrays

- array new{2:5} green jacobs denato fetzer;
- array x{5,3} score1-score15;
- array test{3:4,3:7} test1-test10;
- array temp{0:999} _TEMPORARY_;
- array x{10} (2*1:5);

Example 5: Creating a Range of Variable Names That Have Leading Zeros
This example shows that you can create a range of variable names that have leading 0s. Each variable name has a length of three characters, and the names sort correctly (A01, A02, ... A10). Without leading 0s, the variable names would sort in this order: A1, A10, A2, ... A9.

- data test (drop=i);
array a{10} A01-A10;
do i=1 to 10;
   a{i}=i;
end;
run;
proc print noobs data=test;
run;

Output 2.1   Array Names That Have Leading Zeros

The SAS System

<table>
<thead>
<tr>
<th></th>
<th>A01</th>
<th>A02</th>
<th>A03</th>
<th>A04</th>
<th>A05</th>
<th>A06</th>
<th>A07</th>
<th>A08</th>
<th>A09</th>
<th>A10</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
</tr>
</tbody>
</table>

See Also

- "Array Processing” in SAS Language Reference: Concepts

Statements:

- “Array Reference Statement” on page 24

Array Reference Statement

Describes the elements in an array to be processed.

Valid in: DATA step
Categories: CAS Information
Type: Declarative
Restriction: Variables with a VARCHAR data type are not supported.

Syntax

array-name {subscript};

Arguments

array-name

is the name of an array that was previously defined with an ARRAY statement in the same DATA step.

{subscript}

specifies the subscript. Any of these forms can be used:
{variable-1< , …variable-n>}
specifies a variable or variable list that is usually used with DO-loop processing. For each execution of the DO loop, the current value of this variable becomes the subscript of the array element that is being processed.

Tip
You can enclose a subscript in braces ( { } ), brackets ( [ ] ), or parentheses ( ( )).

Example
“Example 1: Using Iterative DO-Loop Processing” on page 26

{ * }
forces SAS to treat the elements in the array as a variable list.

Restriction
When you define an array that contains temporary array elements, you cannot reference the array elements with an asterisk.

Tips
The asterisk can be used with the INPUT and PUT statements, and with some SAS functions.

This syntax is provided for convenience and is an exception to usual array processing.

Example
“Example 4: Using the Asterisk References as a Variable List” on page 28

expression-1< , …expression-n>
specifies a SAS expression.

Range
The expression must evaluate to a subscript value when the statement that contains the array reference executes. The expression can also be an integer with a value between the lower and upper bounds of the array, inclusive.

Example
“Example 3: Specifying the Subscript” on page 27

Details
- To refer to an array in a program statement, use an array reference. The ARRAY statement that defines the array must appear in the DATA step before any references to that array. An array definition is in effect only for the duration of the DATA step. If you want to use the same array in several DATA steps, redefine the array in each step.

CAUTION
Using the name of a SAS function as an array name can cause unpredictable results. If you inadvertently use a function name as the name of the array, SAS treats parenthetical references that involve the name as array references, not function references, for the duration of the DATA step. A warning message is written to the SAS log.

- You can use an array reference anywhere that you can write a SAS expression, including SAS functions and these SAS statements:
  - assignment statement
  - sum statement
The DIM function is often used with the iterative DO statement to return the number of elements in a dimension of an array, when the lower bound of the dimension is 1. If you use DIM, you can change the number of array elements without changing the upper bound of the DO statement. For example, because DIM(NEW) returns a value of 4, these statements process all the elements in the array:

```
array new{*} score1-score4;
  do i=1 to dim(new);
    new{i}=new{i}+10;
  end;
```

You can use the IN operator to determine whether a value exists in an array. For example:

```
data test;
  array v[3] {1 2 3};
  if 2 in v then flag = 5;
run;
```

Comparisons
An ARRAY statement defines an array, whereas an array reference defines the members of the array to process.

Examples:

Example 1: Using Iterative DO-Loop Processing
In this example, the statements process each element of the array, using the value of variable I as the subscript on the array references for each iteration of the DO loop. If an array element has a value of 99, the IF-THEN statement changes that value to 100.

```
data test;
  input d1 d2 d3 d4 d5 d6 d7;
  datalines;
  1 2 99 3 99 0 99
  ;
  run;
```

```
data test;
  set test;
  array days{7} d1-d7;
  do i=1 to 7;
```
Example 2: Referencing Many Arrays in One Statement

You can refer to more than one array in a single SAS statement. In this example, you create two arrays, DAYS and HOURS. The statements inside the DO loop substitute the current value of variable I to reference each array element in both arrays.

```sas
data test;
  input d1 d2 d3 d4 d5 d6 d7;
datalines;
  1 2 99 3 99 0 99
;run;

data test;
  set test;
  array days{7} d1-d7;
  array hours{7} h1-h7;
  do i=1 to 7;
    if days{i}=99 then days{i}=100;
    hours{i}=days{i}*24;
  end;
  put d1= d2= d3= d4= d5= d6= d7=;
  put h1= h2= h3= h4= h5= h6= h7=;
run;
```

SAS writes the following output to the log:

```
d1=1 d2=2 d3=100 d4=3 d5=100 d6=0 d7=100
```

Example 3: Specifying the Subscript

In this example, the INPUT statement reads in variables A1, A2, and the third element (A3) of the array named ARR1.

```sas
data test;
  array arr1{*} a1-a3;
x=1;
  input a1 a2 arr1{x+2};
datalines;
  1 2 3
;run;
```
Example 4: Using the Asterisk References as a Variable List

```sas
data test;
  input cost1 cost2 cost3 cost4 cost5 cost6 cost7 cost8 cost9 cost10;
  datalines;
  10 20 30 40 50 60 70 80 90 100
; run;

data test;
  set test;
  array cost{10} cost1-cost10;
  totcost=sum(of cost {*});
  put totcost=;
run;

data test;
  array days{7} d1-d7;
  input days {*};
  datalines;
  1 2 3 4 5 6 7
; run;

data test;
  array hours{7} h1-h7;
  input h1 h2 h3 h4 h5 h6 h7;
  datalines;
  1 2 3 4 5 6 7
; run;

data test;
  set test;
  array hours{7} h1-h7;
  put hours {*};
run;
```

See Also

- “Array Processing” in SAS Language Reference: Concepts
- “DIM Function” in SAS Functions and CALL Routines: Reference

Functions:

- “DIM Function” in SAS Functions and CALL Routines: Reference

Statements:

- “ARRAY Statement” on page 18
- “DO Statement: Iterative” on page 64

Assignment Statement

Evaluates an expression and stores the result in a variable.
Assignment Statement

Valid in: DATA step
Categories: Action
          CAS
Type: Executable

Syntax

\[ \text{variable}=\text{expression}; \]

Arguments

\textbf{variable}

names a new or existing variable.

Range \textit{variable} can be a variable name, array reference, or SUBSTR function.

Tip Variables that are created by the Assignment statement are not automatically retained.

\textbf{expression}

is any SAS expression.

Tip \textit{expression} can contain the variable that is used on the left side of the equal sign. When a variable appears on both sides of a statement, the original value on the right side is used to evaluate the expression. The result is stored in the variable on the left side of the equal sign. For more information, see “Expressions” in \textit{SAS Language Reference: Concepts}.

Details

Assignment statements evaluate the expression on the right side of the equal sign and store the result in the variable that is specified on the left side of the equal sign.

Example: Various Expressions in Assignment Statements

These assignment statements use different types of expressions:

- \texttt{name='Amanda Jones';}
- \texttt{WholeName='Ms. '||name;}
- \texttt{a=a+b;}

See Also

\textbf{Statements}:

- “Sum Statement” on page 307
ATTRIB Statement

Associates a format, informat, label, and length with one or more variables.

Valid in: DATA step or PROC step
Categories: CAS
Information
Type: Declarative
Restriction: You cannot declare a VARCHAR variable with the ATTRIB statement.
See: ATTRIB Statement under Windows, UNIX, and z/OS

Syntax

**ATTRIB variable-list(s) attribute-list(s);**

Arguments

*variable-list(s)*

names the variables that you want to associate with the attributes.

**TIP** List the variables in any form that SAS allows.

*attribute-list(s)*

specifies one or more attributes to assign to *variable-list*. Specify one or more of these attributes in the ATTRIB statement:

**FORMAT=**<sup>*</sup>*format*

associates a format with variables in *variable-list*.

**Tip** The format can be either a standard SAS format or a format that is defined with the FORMAT procedure.

**INFORMAT=**<sup>*</sup>*informat*

associates an informat with variables in *variable-list*.

**Tip** The informat can be either a standard SAS informat or an informat that is defined with the FORMAT procedure.

**LABEL=**<sup>*</sup>*label*

associates a label with variables in *variable-list*.

**LENGTH=**<sup>*</sup>*length*

specifies the length of variables in *variable-list*.

**Range** For character variables, the range is from 1 to 32767 bytes for all operating environments.

**Restrictions** You cannot change the length of a variable using LENGTH= from PROC DATASETS.
You cannot declare a VARCHAR variable by using the ATTRIB statement.

**Requirement**
Insert a dollar sign ($) in front of the length of character variables.

**Operating environment**
For numeric variables, the minimum length that you can specify with the LENGTH= specification is 2 bytes in some operating environments and 3 bytes in other operating environments.

**Tip**
Use the ATTRIB statement to specify the length of variables in your output data set before you use a SET statement to bring the variables in from an existing data set.

**TRANSCODE=**

### YES | NO
specifies whether character variables can be transcoded. Use TRANSCODE=NO to suppress transcoding.

**Default**
YES

**Restriction**
The TRANSCODE=NO attribute is not supported by some SAS Workspace Server clients. Variables with TRANSCODE=NO are not returned in SAS 9.4. Prior to SAS 9.4, variables with TRANSCODE=NO are transcoded. Earlier releases of SAS cannot access a SAS 9.4 data set that contains a variable with a TRANSCODE=NO attribute.

**Interactions**
You can use the "VTRANSCODE Function" and the "VTRANSCODEX Function" in SAS National Language Support (NLS): Reference Guide to return a value that indicates whether transcoding is turned on or off for a character variable.

If the TRANSCODE= attribute is set to NO for any character variable in a data set, PROC CONTENTS prints a transcoded column that contains the TRANSCODE= value for each variable in the data set. If all character variables in the data set are set to the default TRANSCODE= value (YES), no transcoded column is printed.

**See**

---

**Details**

**The Basics**

Using the ATTRIB statement in the DATA step permanently associates attributes with variables by changing the descriptor information of the SAS data set that contains the variables.

You can use ATTRIB in a PROC step, but the rules are different.
How SAS Treats Variables When You Assign Informats with the INFORMAT= Option in the ATTRIB Statement

Informats that are associated with variables that use the INFORMAT= option in the ATTRIB statement behave like informats that are used with modified list input.

**Note:** Modified list input uses format modifiers to help read in data. For example, use the : modifier with an informat to read character values that are longer than 8 bytes or that vary in length. For numeric values, you can use the : modifier to read nonstandard values.

Here is how SAS treats variables in modified list input.

- uses the value of $w$ in an informat to specify the length of previously undefined character variables
- does not use the value of $w$ in an informat to specify input field widths or column positions in an external file
- ignores the value of $w$ in numeric informats
- uses the value of $d$ in an informat in the same way SAS usually does for numeric informats
- treats blanks that are embedded as input data as delimiters unless you change their status with the DLM= or DLMSTR= option specification in an INFILE statement

If you have coded the INPUT statement to use another style of input such as formatted input or column input, that style of input is not used when you use the INFORMAT= option in the ATTRIB statement.

How SAS Treats Transcoded Variables When You Use the SET Statement or MERGE Statement

When you use the SET statement or MERGE statement to create a data set from several data sets, SAS makes the TRANSCODE= attribute of the variable in the output data set equal to the TRANSCODE= value of the variable in the first data set. See “Example 2: Using the SET Statement with Transcoded Variables” on page 33 and “Example 3: Using the MERGE Statement with Transcoded Variables” on page 33.

**Note:** The TRANSCODE= attribute is set when the variable is first seen in an input data set or in an ATTRIB TRANSCODE= statement. If a SET statement or MERGE statement comes before an ATTRIB TRANSCODE= statement and the TRANSCODE= attribute contradicts the SET statement or MERGE statement, a warning occurs.

Comparisons

You can use either an ATTRIB statement or an individual attribute statement such as FORMAT, INFORMAT, LABEL, or LENGTH to change an attribute that is associated with a variable.
Examples:

Example 1: Examples of ATTRIB Statements with Varying Numbers of Variables and Attributes

Here are examples of ATTRIB statements that contain varying numbers of variables and attributes.

- **single variable and single attribute:**
  ```plaintext
  attrib cost length=4;
  ```

- **single variable with multiple attributes:**
  ```plaintext
  attrib saleday informat=mmddyy.
  format=worddate.;
  ```

- **multiple variables with the same multiple attributes:**
  ```plaintext
  attrib x y length=$4 label='TEST VARIABLE';
  ```

- **multiple variables with different multiple attributes:**
  ```plaintext
  attrib x length=$4 label='TEST VARIABLE'
  y length=$2 label='RESPONSE';
  ```

- **variable list with single attribute:**
  ```plaintext
  attrib month1-month12
  label='MONTHLY SALES';
  ```

Example 2: Using the SET Statement with Transcoded Variables

This example uses the SET statement. The variable Z's TRANSCODE= attribute in data set A is NO because B is the first input data set and Z's TRANSCODE= attribute in data set B is NO.

```plaintext
data b;
  length z $4;
  z = 'ice';
  attrib z transcode = no;
data c;
  length z $4;
  z = 'snow';
  attrib z transcode = yes;
data a;
  set b;
  set c;
  /* Check transcode setting for variable Z */
  rcl = vtranscode(z);
  put rcl=;
run;
```

Example 3: Using the MERGE Statement with Transcoded Variables

This example uses the MERGE statement. The variable Z's TRANSCODE= attribute in data set A is YES because C is the first data set in the MERGE statement and Z's TRANSCODE= attribute in data set C is YES.

```plaintext
data b;
  length z $4;
```
z = 'ice';
attrib z transcode = no;
data c;
  length z $4;
z = 'snow';
attrib z transcode = yes;
data a;
  merge c b;
  /* Check transcode setting for variable Z */
  rcl = vtranscode(z);
  put rcl=;
run;

See Also

- “How Many Characters Can I Use When I Measure SAS Name Lengths in Bytes?” in *SAS Language Reference: Concepts*

Functions:


Statements:

- “FORMAT Statement” on page 100
- “INFORMAT Statement” on page 142
- “LABEL Statement” on page 184
- “LENGTH Statement” on page 191

---

**BY Statement**

Controls the operation of a SET, MERGE, MODIFY, or UPDATE statement in the DATA step and sets up special grouping variables.

<table>
<thead>
<tr>
<th>Valid in:</th>
<th>DATA step or PROC step</th>
</tr>
</thead>
<tbody>
<tr>
<td>Categories:</td>
<td>CAS</td>
</tr>
<tr>
<td></td>
<td>File-Handling</td>
</tr>
<tr>
<td>Type:</td>
<td>Declarative</td>
</tr>
<tr>
<td>See:</td>
<td>“BY-Group Processing in CAS” in <em>SAS Cloud Analytic Services: DATA Step Programming</em></td>
</tr>
</tbody>
</table>

Example:

data ShoesByRegion;
set sashelp.shoes;
  by descending Region
    descending Subsidiary;
run;
Syntax

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

Required Argument

**variable**

names each variable by which the data set is sorted or indexed. These variables are referred to as BY variables for the current DATA or PROC step.

**Requirement**
If you designate a name literal as the BY variable in BY-group processing and you want to refer to the corresponding FIRST. or LAST. temporary variable, you must include the FIRST. or LAST. portion of the two-level variable name within single quotation marks. For example:

```sas
data sedanTypes;
  set cars;
  by 'Sedan Types'n;  
  if 'first.Sedan Types'n then type=1;
run;
```

**Tip**
The data set can be sorted or indexed by more than one variable.

**Examples**

- “Example 1: Specifying One or More BY Variables” on page 38
- “Example 2: Specifying Sort Order” on page 38
- “Example 3: BY-Group Processing with Nonsorted Data” on page 39
- “Example 4: Grouping Observations by Using Formatted Values” on page 39

Optional Arguments

**DESCENDING**
specifies that the data sets are sorted in descending order by the variable that is specified. DESCENDING means largest to smallest numerically, or reverse alphabetical for character variables.

**Restrictions**
Beginning in SAS Viya 3.5, the DESCENDING option is supported in a DATA step that runs in CAS. When running in CAS, the DESCENDING option cannot be used on the first variable in the BY statement.

You cannot use the DESCENDING option with data sets that are indexed because indexes are always stored in ascending order.

**Example**

“Group and Sort Variables in Descending Order” in SAS Cloud Analytic Services: DATA Step Programming
**GROUPFORMAT**

uses the formatted values, instead of the internal values, of the BY variables to determine where BY groups begin and end. GROUPFORMAT also determines how FIRST.<variable> and LAST.<variable> are assigned. Although the GROUPFORMAT option can appear anywhere in the BY statement, the option applies to all variables in the BY statement.

<table>
<thead>
<tr>
<th>Restrictions</th>
<th>You must sort the observations in a data set based on the value of the BY variables before using the GROUPFORMAT option in the BY statement.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>You can use the GROUPFORMAT option in a BY statement only in a DATA step.</td>
</tr>
<tr>
<td>Interaction</td>
<td>If you also use the NOTSORTED option, you can group the observations in a data set by the formatted value of the BY variables without requiring that the data set be sorted or indexed.</td>
</tr>
<tr>
<td>Note</td>
<td>BY-group processing in the DATA step using the GROUPFORMAT option is the same as BY-group processing with formatted values in SAS procedures.</td>
</tr>
<tr>
<td>Tips</td>
<td>Using the GROUPFORMAT option is useful when you define your own formats to display data that is grouped.</td>
</tr>
<tr>
<td></td>
<td>Using the GROUPFORMAT option in the DATA step ensures that BY groups that you use to create a data set match the BY groups in PROC steps that report grouped, formatted data.</td>
</tr>
<tr>
<td>See</td>
<td></td>
</tr>
<tr>
<td>Example</td>
<td>“Example 4: Grouping Observations by Using Formatted Values” on page 39</td>
</tr>
</tbody>
</table>

**NOTSORTED**

specifies that observations with the same BY value are grouped together but are not necessarily sorted in alphabetical or numeric order.

<table>
<thead>
<tr>
<th>Restriction</th>
<th>You cannot use the NOTSORTED option with the MERGE and UPDATE statements.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tips</td>
<td>The NOTSORTED option can appear anywhere in the BY statement.</td>
</tr>
<tr>
<td></td>
<td>Using the NOTSORTED option is useful if you have data that falls into other logical groupings such as chronological order or categories.</td>
</tr>
<tr>
<td>Example</td>
<td>“Example 3: BY-Group Processing with Nonsorted Data” on page 39</td>
</tr>
</tbody>
</table>

**Details**

How SAS Identifies the Beginning and End of a BY Group

SAS identifies the beginning and end of a BY group by creating two temporary variables for each BY variable: FIRST.<variable> and LAST.<variable>. The value of these variables is either 0 or 1. SAS sets the value of FIRST.<variable> to 1 when it
reads the first observation in a BY group, and sets the value of LAST.\texttt{variable} to 1 when it reads the last observation in a BY group. These temporary variables are available for DATA step programming but are not added to the output data set.

For a complete explanation of how SAS processes grouped data and how to prepare your data, see “BY-Group Processing in the DATA Step” in \textit{SAS Language Reference: Concepts}.

Using a BY Statement in a DATA Step

The BY statement applies only to the SET, MERGE, MODIFY, or UPDATE statement that precedes it in the DATA step. Only one BY statement can accompany each of these statements in a DATA step.

The data sets that are listed in the SET, MERGE, or UPDATE statement must be sorted by the values of the variables that are listed in the BY statement or that have an appropriate index. As a default, SAS expects the data sets to be arranged in ascending numeric order or in alphabetical order. The observations can be arranged by using one of these methods:

- sort the data set
- create an index for the variables
- read in the observations in order

\textbf{Note:} MODIFY does not require sorted data, but sorting can improve performance.

\textbf{Note:} The BY statement honors the linguistic collation of data that is sorted by using the SORT procedure with the SORTSEQ=LINGUISTIC option.

For more information, see “How to Prepare Your Data Sets” in \textit{SAS Language Reference: Concepts}.

Using a BY Statement in a PROC Step

You can specify the BY statement with some SAS procedures to modify their action. For more information, see the individual procedures in \textit{Base SAS Procedures Guide} for a discussion of how the BY statement affects processing for SAS procedures.

Using a BY Statement with SAS Views

If you create a DATA step view by reading from a DBMS and the SET, MERGE, UPDATE, or MODIFY statement is followed by a BY statement, the BY statement might cause the DBMS to sort the data in order to return the data in sorted order. Sorting the data could increase execution time.

How SAS Processes BY Groups

SAS assigns these values to FIRST.\texttt{variable} and LAST.\texttt{variable}:

- \texttt{FIRST.\texttt{variable}} has a value of 1 under these conditions:
  - when the current observation is the first observation that is read from the data set.
when you do not use the GROUPFORMAT option and the internal value of
the variable in the current observation differs from the internal value in the
previous observation.

If you use the GROUPFORMAT option, FIRST.\textit{variable} has a value of 1 when
the formatted value of the variable in the current observation differs from the
formatted value in the previous observation.

\textbf{FIRST.\textit{variable} has a value of 1 for any preceding variable in the BY
statement.}

In all other cases, FIRST.\textit{variable} has a value of 0.

\textbf{LAST.\textit{variable} has a value of 1 under these conditions:}

\begin{itemize}
\item when the current observation is the last observation that is read from the data
set.
\item when you use the GROUPFORMAT option and the internal value of the
variable in the current observation differs from the internal value in the next
observation.
\end{itemize}

If you use the GROUPFORMAT option, LAST.\textit{variable} has a value of 1 when
the formatted value of the variable in the current observation differs from the
formatted value in the next observation.

\textbf{LAST.\textit{variable} has a value of 1 for any preceding variable in the BY
statement.}

In all other cases, LAST.\textit{variable} has a value of 0.

\section*{Examples:}

\subsection*{Example 1: Specifying One or More BY Variables}

\begin{itemize}
\item Observations are in ascending order of the variable \textit{DEPT}.
\begin{verbatim}
by dept;
\end{verbatim}
\item Observations are in alphabetical (ascending) order by \textit{CITY} and, within each
value of \textit{CITY}, in ascending order by \textit{ZIPCODE}.
\begin{verbatim}
by city zipcode;
\end{verbatim}
\end{itemize}

\subsection*{Example 2: Specifying Sort Order}

\textbf{Note: DESCENDING is not supported in a DATA step that runs in CAS.}

\begin{itemize}
\item Observations are in ascending order of \textit{SALESREP} and, within each
\textit{SALESREP} value, in descending order of the values of \textit{JANSALES}.
\begin{verbatim}
by salesrep descending jansales;
\end{verbatim}
\item Observations are in descending order of \textit{BEDROOMS}, and, within each value of
\textit{BEDROOMS}, in descending order of \textit{PRICE}.
\begin{verbatim}
by descending bedrooms descending price;
\end{verbatim}
\end{itemize}
Example 3: BY-Group Processing with Nonsorted Data

Observations are ordered by the name of the month in which the expenses were accrued.

by month notsorted;

Example 4: Grouping Observations by Using Formatted Values

This example illustrates the use of the GROUPFORMAT option.

```
proc format;
  value range
    low -55 = 'Under 55'
    55-60  = '55 to 60'
    60-65  = '60 to 65'
    65-70  = '65 to 70'
    other  = 'Over 70';
run;
proc sort data=sashelp.class out=sorted_class;
  by height;
run;
data _null_;
  format height range.;
  set sorted_class;
  by height groupformat;
  if first.height then
      put 'Shortest in ' height 'measures ' height:best12.;
run;
```

SAS writes the following output to the SAS log:

```
Shortest in Under 55 measures 51.3
Shortest in 55 to 60 measures 56.3
Shortest in 60 to 65 measures 62.5
Shortest in 65 to 70 measures 65.3
Shortest in Over 70 measures 72
```

Example 5: Combining Multiple Observations and Grouping Them Based on One BY Value

This example shows how to use FIRST."variable" and LAST."variable" with BY-group processing.

```
data Inventory;
  length RecordID 8 Invoice $ 30 ItemLine $ 50;
  infile datalines;
  input RecordID Invoice ItemLine &;
  drop RecordID;
  datalines;
A74  A5296  Highlighters
A75  A5296  Lot # 7603
A76  A5296  Yellow Blue Green
A77  A5296  24 per box
A78  A5297  Paper Clips
A79  A5297  Lot # 7423
```
data combined;
  array Line{4} $ 60 ;
  retain Line1-Line4;
  keep Invoice Line1-Line4;
  set Inventory;
  by Invoice;
  if first.Invoice then do;
    call missing(of Line1-Line4);
    records = 0;
  end;
  records + 1;
  Line[records]=ItemLine;
  if last.Invoice then output;
run;
proc print data=combined;
  title 'Office Supply Inventory';
run;

Output 2.2  Output from Combining Multiple Observations

<table>
<thead>
<tr>
<th>Obs</th>
<th>Line1</th>
<th>Line2</th>
<th>Line3</th>
<th>Line4</th>
<th>Invoice</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Highlighters</td>
<td>Lot # 7603</td>
<td>Yellow Blue Green</td>
<td>24 per box</td>
<td>A5295</td>
</tr>
<tr>
<td>2</td>
<td>Paper Clips</td>
<td>Lot # 7423</td>
<td>Small Medium Large</td>
<td></td>
<td>A5297</td>
</tr>
<tr>
<td>3</td>
<td>Gluestick</td>
<td>Lot # 4422</td>
<td>New item</td>
<td></td>
<td>A5298</td>
</tr>
<tr>
<td>4</td>
<td>Rubber bands</td>
<td>Lot # 7892</td>
<td>Wide width, Narrow width</td>
<td>1000 per box</td>
<td>A5299</td>
</tr>
</tbody>
</table>

See Also

Statements:
- “MERGE Statement” on page 204
- “MODIFY Statement” on page 212
- “SET Statement” on page 290
- “UPDATE Statement” on page 309
CALL Statement

Invokes a SAS CALL routine.

Valid in: DATA step

Categories: Action
CAS

Type: Executable

Syntax

CALL routine(parameter-1 <, …parameter-n>);

Arguments

routine
specifies the name of the SAS CALL routine that you want to invoke.

See For information about available routines, see SAS Functions and CALL Routines: Reference

(parameter)
is a piece of information to be passed to or returned from the routine.

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Enclose this information, which depends on the specific routine, in parentheses.</th>
</tr>
</thead>
</table>

Tip You can specify additional parameters, separated by commas.

Details

SAS CALL routines can assign variable values and perform other system functions.

See Also

SAS Functions and CALL Routines: Reference

CARDS Statement

Specifies that lines of data follow the statement.

Valid in: DATA step

Category: File-Handling

Type: Declarative

Alias: DATALINES, LINES

Restriction: This statement is not supported in a DATA step that runs in CAS.
CARDS4 Statement

Specifies that the lines of data that follow the statement contain internal semicolons.

Valid in: DATA step
Category: File-Handling
Type: Declarative
Alias: DATALINES4, LINES4
Restriction: This statement is not supported in a DATA step that runs in CAS.
See: “DATALINES4 Statement” on page 57

CATNAME Statement

Logically combines two or more catalogs into one by associating them with a catref (a shortcut name); clears one or all catrefs; lists the concatenated catalogs in one concatenation or in all concatenations.

Note: The CATNAME Statement has moved to SAS Global Statements.

CHECKPOINT EXECUTE_ALWAYS Statement

Indicates to execute the DATA step or PROC step that immediately follows the statement without considering the checkpoint-restart data.

Note: The CHECKPOINT EXECUTE_ALWAYS Statement has moved to SAS Global Statements.

Comment Statement

Specifies the purpose of the statement or program.

Note: The Comment Statement has moved to SAS Global Statements.

CONTINUE Statement

 Stops processing the current DO-loop iteration and resumes processing the next iteration.

Valid in: DATA step
Categories: CAS
Control
Type: Executable
Restriction: Can be used only in a DO loop

Syntax

CONTINUE;

Without Arguments

The CONTINUE statement has no arguments. It stops processing statements within the current DO-loop iteration based on a condition. Processing resumes with the next iteration of the DO loop.

Comparisons

- The CONTINUE statement stops processing the current iteration of a loop and resumes with the next iteration. The LEAVE statement causes processing of the current loop to end.
- You can use the CONTINUE statement only in a DO loop. You can use the LEAVE statement in a DO loop or a SELECT group.

Example: Preventing Other Statements from Executing

This DATA step creates a report of benefits for new full-time employees. If an employee's status is PT (part-time), the CONTINUE statement prevents the second INPUT statement and the OUTPUT statement from executing.

```plaintext
data new_emp;
  drop i;
  do i=1 to 5;
    input name $ idno status $;
    /* return to top of loop */
    /* when condition is true */
    if status='PT' then continue;
    input benefits $10.;
    output;
  end;
  datalines;
  Becker 9011 PT
  Tanaka 876 PT
  Green 1002 FT
  Eye/Dental
  Kim 85111 PT
  Patel 433 FT
  HMO
;```

See Also

Statements:
- “DO Statement: Iterative” on page 64
- “LEAVE Statement” on page 190
DATA Statement

Begins a DATA step and provides names for any output such as SAS data sets, views, or programs.

Valid in: DATA step
Categories: CAS
File-Handling
Type: Declarative

Syntax

Form 1: DATA statement for creating output data sets
   DATA <data-set-name-1 <(data-set-options-1)>> >
   <…data-set-name-n <(data-set-options-n)>> >
   </<DEBUG> <NESTING> <STACK=stack-size> <HEXLISTALL>> <NOLIST>>;

Form 2: DATA statement for creating a DATA step view
   DATA view-name <data-set-name-1 <(data-set-options-1)>> >
   <…data-set-name-n <(data-set-options-n)>> >/ VIEW=view-name
   <(<password-option> <SOURCE=source-option> )>
   <NESTING> <NOLIST>>;

Form 3: DATA statement for creating a stored compiled DATA step program
   DATA data-set-name / PGM=program-name
   <(<password-option> <SOURCE=source-option> )>
   <NESTING> <NOLIST>>;

Form 4: DATA statement for describing a DATA step view
   DATA VIEW=view-name <(password-option)> <NOLIST>>;
   DESCRIBE;

Form 5: DATA statement for use with a stored compiled DATA step program
   DATA PGM=program-name <(password-option)> <NOLIST>>;
   <DESCRIBE>;
   <REDIRECT INPUT | OUTPUT old-name-1 = new-name-1<… old-name-n = new-name-n> >;
   <EXECUTE>;

Form 6: DATA statement for use on the CAS server
   DATA <data-set-name-1 <(data-set-options-1)>> >
   <…data-set-name-n <(data-set-options-n)>> >
   / <SESSREF=session-reference-name> | <SESSUUID=session-uuid > >
   <THREADS=number-of-threads> <SINGLE= NO | YES | NOINPUT>
   <NESTING> <STACK=stack-size><HEXLISTALL> <NOLIST>>;

Without Arguments

If you omit the arguments, the DATA step automatically names each successive data set that you create as DATAn, where n is the smallest integer that makes the name unique.
Arguments

**data-set-name**

names the SAS data file or DATA step view that the DATA step creates. To create a DATA step view, you must specify at least one *data-set-name* and that *data-set-name* must match *view-name*.

**Restriction**

*data-set-name* must conform to the rules for SAS names, and additional restrictions might be imposed by your operating environment.

**Tips**

Instead of using a data set name, you can specify the physical pathname to the file using syntax that your operating system understands. The pathname must be enclosed in single or double quotation marks.

If _NULL_ is specified as the data set name, SAS does not create a data set when it executes the DATA step. For an example, see “Example 5: Creating a Custom Report” on page 53.

**See**

“Names in the SAS Language” in SAS Language Reference: Concepts

(data-set-options)

specifies optional arguments that the DATA step applies when it writes observations to the output data set.

**See**

SAS Data Set Options: Reference for a definition and list of data set options.

**Example**

“Example 1: Creating Multiple Data Files and Using Data Set Options” on page 52

**NOLIST**

suppresses the output of all variables to the SAS log when the value of _ERROR_ is 1.

**Restriction**

NOLIST must be the last option in the DATA statement.

**password-option**

assigns a password to a stored compiled DATA step program or a DATA step view.

---

**Note:** To describe a password-protected DATA step program, you must specify its password. If the program has more than one password, you must specify the most restrictive password. ALTER is the most restrictive, and READ is the least restrictive. For more information, see “DESCRIBE Statement” on page 59.

---

These password options are available:

**ALTER=alter-password**

assigns an ALTER password to a SAS data file. The password enables you to protect or replace a stored compiled DATA step program or a DATA step view.

**Alias**

PROTECT=
PW=password
assigns a READ= password and an ALTER= password. Both passwords are set to the same value.

READ=read-password
assigns a READ password to a SAS data file. The password enables you to read or execute a stored compiled DATA step program or a DATA step view.

PGM=program-name
names the stored compiled program that SAS creates or executes in the DATA step. To create a stored compiled program, specify a slash (/) before the PGM= option. To execute a stored compiled program, specify the PGM= option without a slash (/).

SOURCE=source-option
specifies one of these source options:

ENCERYPT
encrypts and saves the source code that created a stored compiled DATA step program or a DATA step view.

Tip If you encrypt source code, use the ALTER password option as well. SAS issues a warning message if you do not use ALTER.

NOSAVE
does not save the source code.
**CAUTION**  
If you use the NOSAVE option for a DATA step view, the view cannot be migrated or copied from one version of SAS to another version.

**SAVE**  
saves the source code that created a stored compiled DATA step program or a DATA step view.

**Default**  
SAVE  

**Restriction**  
This argument is not supported in a DATA step that runs in CAS.

**VIEW=view-name**  
names a view that the DATA step uses to store the input DATA step view.

**Restrictions**  
This argument is not supported in a DATA step that runs in CAS.

- **view-name** must match one of the output data set names.
- SAS creates only one view in a DATA step.

**Tips**  
If you specify additional data sets in the DATA statement, SAS creates these data sets when the view is processed in a subsequent DATA or PROC step. Views have the capability of generating other data sets when the view is executed.

- SAS macro variables resolve when the view is created. Use the SYMGET function to delay macro variable resolution until the view is processed.

**Examples**  
"Example 2: Creating Input DATA Step Views" on page 52  
"Example 3: Creating a View and a Data File" on page 53

**Optional Arguments**

**DEBUG**  
enables you to debug your program interactively by helping identify logic errors, and sometimes data errors.

**Restriction**  
This argument is not supported in a DATA step that runs in CAS.

**See**  
Using the DATA Step Debugger  

**Example**  
data test / debug; x=round(height); run;

**HEXLISTALL**  
enables the LIST statement to write log data in hexadecimal format for all lines of input data. This argument is available beginning with SAS 9.4M6.

**See**  
For more information about using the HEXLISTALL option with the LIST statement, see "Example 3: Listing Hexadecimal Values for Input Data" on page 200.
NESTING
specifies that a note is printed to the SAS log for the beginning and end of each DO-END and SELECT-END nesting level. This option enables you to debug mismatched DO-END and SELECT-END statements and is particularly useful in large programs where the nesting level is not obvious.

STACK=stack-size
specifies the maximum number of nested LINK statements.

Arguments for DATA Step Running in CAS

SESSREF=session-reference-name
associates a DATA step with a CAS session. The value for the SESSREF= option represents the current active session.

Interactions
If the DSCAS system option is set to NODSCAS, either the SESSREF= or SESSUUID= option is required to run the DATA step in CAS.

If the DATA step running in CAS has no input tables, it runs in a single thread unless the SESSREF or SESSUUID option is specified.

SESSUUID=session-uuid
specifies the CAS session where the DATA step runs. The session-uuid is the universally unique identifier (UUID) that is associated with a CAS session. One way to create a session and retrieve the UUID for the session is with the CAS statement or LIBNAME statement.

Interactions
If the DSCAS system option is set to NODSCAS, either the SESSREF= or SESSUUID= option is required to run the DATA step in CAS.

If the DATA step running in CAS has no input tables, it runs in a single thread unless the SESSREF or SESSUUID option is specified.

Tip
One way to create a session and retrieve the UUID for the session is with the CAS statement or LIBNAME statement.

SINGLE=NO | NOINPUT | YES
specifies whether to run the DATA step in a single thread in CAS. By default, a DATA step running in CAS runs in a single thread when there is no CAS input table specified. The DATA step running in CAS runs in multiple threads by default when there is an input CAS table specified. For example, the following DATA step runs by default in a single thread in CAS:

libname mycas cas;
data mycas.test;
x='No input table specified';
run;

The following note is returned in the log: NOTE: The DATA step has no input data set and will run in a single thread.

This DATA step runs in multiple threads in CAS by default:

libname mycas cas;
data mycas.class;
set mycas.class;
When you run the DATA step in a single thread in CAS, processing is localized to a single machine regardless of the number of available worker nodes in your environment. If a single-threaded DATA step processes large tables, then not only does the program run slowly, the machine executing the single thread can run out of disk or memory resources. Running the DATA step in a single thread is not recommended for very large tables.

To redistribute a CAS table, you can use the Partition action in the Tables action set. For information about redistributing a table, see “Partition table” in SAS Viya: System Programming Guide.

If the DATA step fails to run in CAS because the requirements for running in the server are not met, then the DATA step runs in SAS automatically. When the DATA step runs in SAS, it always runs in a single thread. See “Running the DATA Step in CAS” in SAS Cloud Analytic Services: DATA Step Programming for a list of requirements for running the DATA step in CAS.

For more information about the DATA step and processing in CAS, see “Single-threaded Processing” in SAS Cloud Analytic Services: DATA Step Programming and “Controlling the Number of Threads” in SAS Cloud Analytic Services: DATA Step Programming.

NOINPUT runs the DATA step program in a single thread only when there are no CAS input tables. This option is useful when writing a DATA step that generates only output data to a table. NOINPUT is the default for DATA step processing in CAS when there is no input table specified.

CAUTION
When you run the DATA step in a single thread in CAS, processing is localized to a single machine regardless of the number of available worker nodes in your environment. If a single-threaded DATA step processes large tables, then not only does the program run slowly, the machine executing the single thread can run out of disk or memory resources. Running the DATA step in a single thread is not recommended for very large tables.

NO runs the DATA step program in all threads. If there are 4 workers and each worker supports 32 threads, the DATA step program runs in 128 threads. NO is the default for DATA step processing in CAS as long as there is an input table specified. Otherwise, without an input table, the DATA step in CAS runs in a single thread.

YES runs the DATA step program in a single thread. As rows from input tables are processed, a copy of every row is transferred to a single node and processed in the single thread. This action incurs a performance penalty for the data transfer, data duplication, and limited processing that is possible with a single thread. A DATA step that is running in SAS always runs in a single thread.
worker nodes in your environment. If a single-threaded DATA step processes large tables, then not only does the program run slowly, the machine executing the single thread can run out of disk or memory resources. Running the DATA step in a single thread is not recommended for very large tables.

Default NOINPUT


THREADS=number-of-threads
specifies the number of threads to use to run the DATA step in CAS.

Requirement This number must be greater than or equal to 0. A value of 0 indicates to run the DATA step program using the maximum number of threads allowed.

Details

Using the DATA Statement

The DATA step begins with the DATA statement. You use the DATA statement to create these types of output: SAS data sets, data views, and stored programs. You can specify more than one output in a DATA statement. However, only one of the outputs can be a data view. You create a view by specifying the VIEW= option on page 47 and create a stored program by specifying the PGM= option on page 46.

Using Both a READ Password and an ALTER Password

If you use both a READ password and an ALTER password in creating a stored compiled DATA step program or a DATA step view, these items apply:

- A READ or ALTER password is required to execute the stored compiled DATA step program or DATA step view.
- A READ or ALTER password is required if the stored compiled DATA step program or DATA step view contains both DESCRIBE and EXECUTE statements.
- If you use an ALTER password with the DESCRIBE and EXECUTE statements, these items apply:
  - SAS executes both the DESCRIBE and EXECUTE statements.
  - If you execute a stored compiled DATA step program or DATA step view with an invalid ALTER password:
    - The DESCRIBE statement does not execute.
    - In batch mode, the EXECUTE statement has no effect.
    - In interactive mode, SAS prompts you for a READ password. If the READ password is valid, SAS processes the EXECUTE statement. If it is invalid, SAS does not process the EXECUTE statement.
If you use a READ password with the DESCRIBE and EXECUTE statements, these items apply:

- In interactive mode, SAS prompts you for the ALTER password.
  - If you enter a valid ALTER password, SAS executes both the DESCRIBE and EXECUTE statements.
  - If you enter an invalid ALTER password, SAS processes the EXECUTE statement but not the DESCRIBE statement.
- In batch mode, SAS processes the EXECUTE statement but not the DESCRIBE statement.
- In both interactive and batch modes, if you specify an invalid READ password SAS does not process the EXECUTE statement.

- An ALTER password is required if the stored compiled DATA step program or DATA step view contains a DESCRIBE statement.
- An ALTER password is required to replace the stored compiled DATA step program or DATA step view.

Creating an Output Data Set (Form 1)

Use the DATA statement to create one or more output data sets. You can use data set options to customize the output data set.

Here is a DATA step that creates two output data sets, EXAMPLE1 and EXAMPLE2. This DATA step example uses the data set option DROP to prevent the variable IDNUMBER from being written to the EXAMPLE2 data set.

```sas
data example1 example2 (drop=IDnumber);
  set sample;
  . . .more SAS statements . .
run;
```

Creating a DATA Step View (Form 2)

You can create DATA step views and execute them at a later time. This DATA step example creates a DATA step view. It uses the SOURCE=ENCRYPT option to both save and encrypt the source code.

```sas
data phone_list / view=phone_list (source=encrypt);
  set customer_list;
  . . .more SAS statements . .
run;
```

For more information, see "DATA Step Views" in SAS Language Reference: Concepts.

Creating a Stored Compiled DATA Step Program (Form 3)

The ability to compile and store DATA step programs enables you to execute the stored programs later. Stored compiled DATA step programs can reduce processing costs by eliminating the need to compile DATA step programs repeatedly. This DATA step example compiles and stores a DATA step program. It uses the ALTER password option, which enables the user to replace an existing stored program, and to protect the stored compiled program from being replaced.

```sas
data testfile / pgm=stored.test_program (alter=sales);
  set sales_data;
```
Describing a DATA Step View (Form 4)

This example uses the DESCRIBE statement in a DATA step view to write a copy of the source code to the SAS log.

```sas
data view=inventory;
describe;
run;
```

For more information, see “DESCRIBE Statement” on page 59.

Executing a Stored Compiled DATA Step Program (Form 5)

This example executes a stored compiled DATA step program. It uses the DESCRIBE statement to write a copy of the source code to the SAS log.

```sas
libname stored 'SAS library';
data pgm=stored.employee_list;
describe;
execute;
run;
```

```sas
libname stored 'SAS library';
data pgm=stored.test_program;
describe;
execute;
...more SAS statements...
run;
```

For more information, see “DESCRIBE Statement” on page 59 and “EXECUTE Statement” on page 75.

Examples:

Example 1: Creating Multiple Data Files and Using Data Set Options

This DATA statement creates more than one data set, and it changes the contents of the output data sets.

```sas
data error (keep=subject date weight)
  fitness(label='Exercise Study'
    rename=(weight=pounds));
```

The ERROR data set contains three variables. SAS assigns a label to the FITNESS data set and renames the variable weight to pounds.

Example 2: Creating Input DATA Step Views

This DATA step creates an input DATA step view instead of a SAS data file.

```sas
libname ourlib 'SAS-library';
data ourlib.test / view=ourlib.test;
  set ourlib.fittest;
```
Example 3: Creating a View and a Data File

This DATA step creates an input DATA step view named THEIRLIB.TEST and an additional temporary SAS data set named SCORETOT.

```sas
libname ourlib 'SAS-library-1';
libname theirlib 'SAS-library-2';
data theirlib.test scoretot
  / view=theirlib.test;
set ourlib.fittest;
tot=sum(of score1-score10);
run;
```

SAS does not create the data file SCORETOT until a subsequent DATA or PROC step processes the view THEIRLIB.TEST.

Example 4: Storing and Executing a Compiled Program

The first DATA step produces a stored compiled program named STORED.SALESFIG.

```sas
libname in 'SAS-library-1 ';
libname stored 'SAS-library-2 ';
data salesdata / pgm=stored.salesfig;
set in.sales;
qtr1tot=jan+feb+mar;
run;
```

SAS creates the data set SALESDATA when it executes the stored compiled program STORED.SALESFIG.

```sas
data pgm=stored.salesfig;
run;
```

Example 5: Creating a Custom Report

The second DATA step in this program produces a custom report and uses the _NULL_ keyword to execute the DATA step without creating a SAS data set.

```sas
data sales;
  input dept : $10. jan feb mar;
datalines;
shoes 4344 3555 2666
housewares 3777 4888 7999
appliances 53111 7122 41333
;
data _null_
  set sales;
  qtr1tot=jan+feb+mar;
  put 'Total Quarterly Sales: ' qtr1tot dollar12.;
run;
```
Example 6: Using a Password with a Stored Compiled DATA Step Program

The first DATA step creates a stored compiled DATA step program called STORED.ITEMS. This program includes the ALTER password, which limits access to the program.

```sas
data sample;
  input Name $ TotalItems $;
  datalines;
  Lin 328
  Susan 433
  Ken 156
  Pat 340
;
proc print data=sample;
run;
libname stored 'SAS-library';
data employees / pgm=stored.items (alter=klondike);
  set sample;
  if TotalItems > 200 then output;
run;
```

This DATA step executes the stored compiled DATA step program STORED.ITEMS. It uses the DESCRIBE statement to print the source code to the SAS log. Because the program was created with the ALTER password, you must use the password if you use the DESCRIBE statement. If you do not enter the password, SAS prompts you for it.

```sas
data pgm=stored.items (alter=klondike);
  describe;
  execute;
run;
```

Example 7: Displaying Nesting Levels

This program has two nesting levels. SAS generates four log messages, one begin and end message for each nesting level.

```sas
data _null_ /nesting;
  do i = 1 to 10;
    do j = 1 to 5;
      put i= j=;
    end;
  end;
run;
```
Example Code 2.1  Nesting Level Debug (Partial SAS Log)

```sas
6 data _null_/nesting;
7   do i = 1 to 10;
    -
719
    NOTE 719-185: *** DO begin level 1 ***.
8      do j = 1 to 5;
719
    NOTE 719-185: *** DO begin level 2 ***.
9        put i= j=;
10      end;
    ---
720
    NOTE 720-185: *** DO end level 2 ***.
11    end;
    ---
720
    NOTE 720-185: *** DO end level 1 ***.
12  run;
```

See Also

- “DATA Step Programming in Hadoop” in SAS In-Database Products: User’s Guide
- “Definition of Data Set Options” in SAS Data Set Options: Reference

Statements:

- “DESCRIBE Statement” on page 59
- “EXECUTE Statement” on page 75
- “LINK Statement” on page 196

DATALINES Statement

Specifies that lines of data follow the statement.

Valid in: DATA step
Category: File-Handling
Type: Declarative
Alias: CARDS, LINES
Restriction: This statement is not supported in a DATA step that runs in CAS.
See: Data lines cannot contain semicolons. Use the “DATALINES4 Statement” on page 57 when your data contains semicolons.
Syntax

```
DATALINES;
```

Without Arguments

Use the DATALINES statement with an INPUT statement to read data that you enter directly in the program, rather than data stored in an external file.

Details

Using the DATALINES Statement

The DATALINES statement is the last statement in the DATA step and immediately precedes the first data line. Use a null statement (a single semicolon) to indicate the end of the input data.

You can use only one DATALINES statement in a DATA step. Use separate DATA steps to enter multiple sets of data.

Reading Long Data Lines

SAS handles data line length with the `CARDIMAGE` system option. If you use `CARDIMAGE`, SAS processes data lines exactly like 80-byte punched card images padded with blanks. If you use `NOCARDIMAGE`, SAS processes data lines longer than 80 columns in their entirety.

Using Input Options with In-Stream Data

The DATALINES statement does not provide input options for reading data. However, you can access some options by using the DATALINES statement in conjunction with an INFILE statement. Specify DATALINES in the INFILE statement to indicate the source of the data, and then use the options that you need. For more information, see “Example 2: Reading In-Stream Data with Options” on page 57.

Comparisons

- Use the DATALINES statement whenever data does not contain semicolons. If your data contains semicolons, use the DATALINES4 statement.
- These SAS statements also read data or point to a location where data is stored:
  - The INFILE statement points to raw data lines stored in another file. The INPUT statement reads those data lines.
  - The %INCLUDE statement brings SAS program statements or data lines stored in SAS files or external files into the current program.
  - The SET, MERGE, MODIFY, and UPDATE statements read observations from existing SAS data sets.

Examples:

Example 1: Using the DATALINES Statement

In this example, SAS reads a data line and assigns values to two character variables, NAME and DEPT, for each observation in the DATA step.
Example 2: Reading In-Stream Data with Options

This example takes advantage of options that are available with the INFILE statement to read in-stream data lines. With the DELIMITER= option, you can use list input to read data values that are delimited by commas instead of blanks.

```sas
data person;
  infile datalines delimiter=',';
  input name $ dept $;
  datalines;
John,Sales
Mary,Acctng
;
```

See Also

**Statements:**
- “DATALINES4 Statement” on page 57
- “INFILE Statement” on page 110

**System Options:**
- “CARDIMAGE System Option” in SAS System Options: Reference

---

**DATALINES4 Statement**

Specifies that the lines of data that follow the statement contain internal semicolons.

- **Valid in:** DATA step
- **Category:** File-Handling
- **Type:** Declarative
- **Alias:** CARDS4, LINES4
- **Restriction:** This statement is not supported in a DATA step that runs in CAS.

**Syntax**

```
DATALINES4;
```

**Without Arguments**

Use the DATALINES4 statement with an INPUT statement to read data that contains semicolons that you enter directly in the program.
Details

The DATALINES4 statement is the last statement in the DATA step and immediately precedes the first data line. Follow the data lines with four consecutive semicolons that are located in columns 1 through 4.

Comparisons

Use the DATALINES4 statement when your data contains semicolons. If your data does not contain semicolons, use the DATALINES statement.

Example: Reading Data Lines That Contain Semicolons

In this example, SAS reads data lines that contain internal semicolons until it encounters a line of four semicolons. Execution continues with the rest of the program.

```
data biblio;
  input number citation $50. ;
  datalines4;
  KIRK, 1988
  LIN ET AL., 1995; BRADY, 1993
  BERG, 1990; ROA, 1994; WILLIAMS, 1992
 ;;;;
```

See Also

**Statements:**

- “DATALINES Statement” on page 55

---

**DELETE Statement**

Stops processing the current observation.

Valid in: DATA step

Categories: Action

CATEGORIES: CAS

Type: Executable

**Syntax**

```
DELETE;
```

Without Arguments

When DELETE executes, the current observation is not written to a data set, and SAS returns immediately to the beginning of the DATA step for the next iteration.
Details

The DELETE statement is often used in a THEN clause of an IF-THEN statement or as part of a conditionally executed DO group.

Comparisons

- Use the DELETE statement when it is easier to specify a condition that excludes observations from the data set or when there is no need to continue processing the DATA step statements for the current observation.
- Use the subsetting IF statement when it is easier to specify a condition for including observations.
- Do not confuse the DROP statement with the DELETE statement. The DROP statement excludes variables from an output data set; the DELETE statement excludes observations.

Examples:

Example 1: Using the DELETE Statement as Part of an IF-THEN Statement

When the value of LEAFWT is missing, the current observation is deleted.

```sql
if leafwt=. then delete;
```

Example 2: Using the DELETE Statement to Subset Raw Data

```sql
data topsales;
  infile file-specification;
  input region office product yrsales;
  if yrsales<100000 then delete;
run;
```

See Also

**Statements:**

- “DO Statement” on page 62
- “DROP Statement” on page 71
- “IF Statement: Subsetting” on page 105
- “IF-THEN/ELSE Statement” on page 108
### DESCRIBE Statement

This statement is not supported in a DATA step that runs in CAS. Use DESCRIBE only with stored compiled DATA step programs and DATA step views. You must specify the PGM= option or the VIEW= option in the DATA statement.

#### Syntax

**DESCRIBE;**

#### Without Arguments

Use the DESCRIBE statement to retrieve program source code from a stored compiled DATA step program or a DATA step view. SAS writes the source statements to the SAS log.

#### Details

Use the DESCRIBE statement without the EXECUTE statement to retrieve source code from a stored compiled DATA step program or a DATA step view. Use the DESCRIBE statement with the EXECUTE statement to retrieve source code and execute a stored compiled DATA step program. For information about how to use these statements with the DATA statement, see “DATA Statement” on page 44.

**Note:** To DESCRIBE a password-protected view or DATA step program, you must specify its password. If the view or program was created with more than one password, you must specify the most restrictive password. As with other SAS files, the ALTER password is the most restrictive, and the READ password is the least restrictive. For more information, see “Executing a Stored Compiled DATA Step Program” in SAS Language Reference: Concepts and “Using Passwords with Views” in SAS Language Reference: Concepts.

#### See Also

**Statements:**

- “DATA Statement” on page 44
- “EXECUTE Statement” on page 75

---

### DISPLAY Statement

Displays a window that is created with the WINDOW statement.

- **Valid in:** DATA step
- **Category:** Window Display
- **Type:** Executable
- **Restriction:** This statement is not supported in a DATA step that runs in CAS.
Syntax

DISPLAY window.<group> <NOINPUT> <BLANK> <BELL> <DELETE>;

Arguments

window.<group>
names the window and group of fields to be displayed. This field is preceded by a period (.).

Tip If the window has more than one group of fields, give the complete window.group specification. If a window contains a single unnamed group, use only window.

NOINPUT
specifies that you cannot enter values into fields that are displayed in the window.

Default If you omit NOINPUT, you can input values into unprotected fields that are displayed in the window.

Restriction If you use NOINPUT in all DISPLAY statements in a DATA step, you must include a STOP statement to stop processing the DATA step.

Tip The NOINPUT option is useful when you want to allow values to be entered into a window at some times but not others. For example, you can display a window once for entering values and a second time for verifying them.

BLANK
clears the window.

Tip Use the BLANK option when you want different groups of fields in a window to be displayed and you do not want text from the previous group to appear in the current display.

BELL
produces an audible alarm, beep, or bell sound when the window is displayed if your personal computer is equipped with a speaker device that provides sound.

DELETE
deletes the display of the window after processing passes from the DISPLAY statement on which the option appears.

Details

You must create a window in the same DATA step that you use to display it. When you display a window, the window remains visible until you display another window over it or until the end of the DATA step. When you display a window that contains fields where you enter values, either enter a value or press Enter at each unprotected field to cause SAS to proceed to the next display. You cannot skip any fields.

While a window is being displayed, use commands and function keys to view other windows, to change the size of the current window, and so on.

A DATA step that contains a DISPLAY statement continues execution until the last observation that is read by a SET, MERGE, UPDATE, MODIFY, or INPUT statement has been processed or until a STOP or ABORT statement is executed. You can also
issue the END command on the command line of the window to stop the execution of the DATA step.

You must create a window before you can display it. For a description of how to create windows, see "WINDOW Statement" on page 323. A window that is displayed with the DISPLAY statement does not become part of the SAS log or output file.

Example

This DATA step creates and displays a window named start. The start window fills the entire screen. Both lines of text are centered.

```sas
data _null_;
window start
   #5 @28 'WELCOME TO THE SAS SYSTEM'
   #12 @30 'PRESS ENTER TO CONTINUE';
display start;
stop;
run;
```

Although the start window in this example does not require you to enter any values, you must press Enter to cause the execution to proceed to the STOP statement. If you omit the STOP statement, the DATA step executes endlessly unless you enter END on the command line of the window.

**Note:** Because this DATA step does not read any observations, SAS cannot detect an end-of-file to cause DATA step execution to cease. If you add the NOINPUT option to the DISPLAY statement, the window is displayed quickly and is removed.

**See Also**

**Statements:**
- "WINDOW Statement" on page 323

---

**DM Statement**

Submits SAS Program Editor, Log, Procedure Output, or text editor commands as SAS statements.

**Note:** The DM Statement has moved to SAS Global Statements.

**DO Statement**

Specifies a group of statements to be executed as a unit.

**Valid in:** DATA step

**Categories:** CAS

**Type:** Executable
Syntax

```sas
DO;
...more SAS statements...
END;
```

Without Arguments

Use the DO statement for simple DO group processing.

Details

The DO statement is the simplest form of DO group processing. The statements between the DO and END statements are called a **DO group**. You can nest DO statements within DO groups.

**Note:** The memory capabilities of your system can limit the number of nested DO statements that you can use. For more information, see the SAS documentation about how many levels of nested DO statements your system's memory can support.

A simple DO statement is often used within IF-THEN/ELSE statements to designate a group of statements to be executed depending on whether the IF condition is true or false.

Comparisons

There are three other forms of the DO statement:

- The iterative DO statement executes statements between DO and END statements repetitively based on the value of an index variable. The iterative DO statement can contain a WHILE or UNTIL clause.
- The DO UNTIL statement executes statements in a DO loop repetitively until a condition is true, checking the condition after each iteration of the DO loop.
- The DO WHILE statement executes statements in a DO loop repetitively while a condition is true, checking the condition before each iteration of the DO loop.

Example: Using the DO Statement

In this simple DO group, the statements between DO and END are performed only when YEARS is greater than 5. If YEARS is less than or equal to 5, statements in the DO group do not execute, and the program continues with the assignment statement that follows the ELSE statement.

```sas
if years>5 then
    do;
        months=years*12;
        put years= months=;
    end;
else yrsleft=5-years;
```
DO Statement: Iterative

Executes statements between the DO and END statements repetitively, based on the value of an index variable.

Valid in: DATA step
Categories: CAS Control
Type: Executable

Syntax

```sas
DO index-variable=specification-1 <, ...specification-n>; …more SAS statements…
END;
```

Arguments

- **index-variable**
  names a variable whose value governs execution of the DO group.

  - **Tip**
    Unless you specify to drop it, the index variable is included in the data set that is being created.

  - **CAUTION**
    Changing the index variable within the iterative DO group might cause infinite looping.

- **specification**
  denotes an expression or a series of expressions in this form:
  
  ```sas
  start <TO stop> <BY increment> <WHILE(expression) | UNTIL(expression)>
  ```

  The DO group is executed first with index-variable equal to `start`. The value of `start` is evaluated before the first execution of the loop.

  - **start**
    specifies the initial value of the index variable.

    When it is used without TO stop or BY increment, the value of start can be a series of items expressed in this form:

    ```sas
    item-1 <, ...item-n>,
    ```

    The items can be either all numeric or all character constants, or they can be variables. Enclose character constants in quotation marks. The DO group is
executed once for each value in the list. If a WHILE condition is added, it applies only to the item that it immediately follows.

**Requirement**
When it is used with **TO stop** or **BY increment**, **start** must be a number or an expression that yields a number.

**Example**
“Example 1: Using Various Forms of the Iterative DO Statement” on page 66

**TO stop**
specifies the ending value of the index variable.

When both **start** and **stop** are present, execution continues (based on the value of **increment**) until the value of **index-variable** passes the value of **stop**. When only **start** and **increment** are present, execution continues (based on the value of **increment**) until a statement directs execution out of the loop, or until a WHILE or UNTIL expression that is specified in the DO statement is satisfied. If neither **stop** nor **increment** is specified, the group executes according to the value of **start**. The value of **stop** is evaluated before the first execution of the loop.

**Requirement**
**Stop** must be a number or an expression that yields a number.

**Tip**
Any changes to **stop** made within the DO group do not affect the number of iterations. To stop iteration of a loop before it finishes processing, change the value of **index-variable** so that it passes the value of **stop**, or use a LEAVE statement to go to a statement outside the loop.

**Example**
“Example 1: Using Various Forms of the Iterative DO Statement” on page 66

**BY increment**
specifies a positive or negative number (or an expression that yields a number) to control the incrementing of **index-variable**.

The value of **increment** is evaluated before the execution of the loop. Any changes to the increment that are made within the DO group do not affect the number of iterations. If no increment is specified, the index variable is increased by 1. When **increment** is positive, **start** must be the lower bound and **stop**, if present, must be the upper bound for the loop. When **increment** is negative, **start** must be the upper bound and **stop**, if present, must be the lower bound for the loop.

**Example**
“Example 1: Using Various Forms of the Iterative DO Statement” on page 66

**WHILE(expression) UNTIL(expression)**
evaluates, either before or after execution of the DO group, any SAS expression that you specify. Enclose the expression in parentheses.

A WHILE expression is evaluated before each execution of the loop so that the statements inside the group are executed repetitively while the expression is true. An UNTIL expression is evaluated after each execution of the loop so that the statements inside the group are executed repetitively until the expression is true.
A WHILE or UNTIL specification affects only the last item in the clause in which it is located.

See “DO WHILE Statement” on page 69 and “DO UNTIL Statement” on page 68

Example “Example 1: Using Various Forms of the Iterative DO Statement” on page 66

The iterative DO statement requires at least one specification argument.

The order of the optional TO and BY clauses can be reversed.

When you use more than one specification, each one is evaluated before its execution.

Comparisons

There are three other forms of the DO statement:

- The DO statement, the simplest form of DO-group processing, designates a group of statements to be executed as a unit, usually as a part of IF-THEN/ELSE statements.
- The DO UNTIL statement executes statements in a DO loop repetitively until a condition is true, checking the condition after each iteration of the DO loop.
- The DO WHILE statement executes statements in a DO loop repetitively while a condition is true, checking the condition before each iteration of the DO loop.

Examples:

Example 1: Using Various Forms of the Iterative DO Statement

- These iterative DO statements use a list of items for the value of start.
  
  do month='JAN','FEB','MAR';
  do count=2,3,5,7,11,13,17;
  do i=5;
  do i=var1, var2, var3;
  do i='01JAN2001'd,'25FEB2001'd,'18APR2001'd;

- These iterative DO statements use the start TO stop syntax.
  
  do i=1 to 10;
  do i=1 to exit;
  do i=1 to x-5;
  do i=1 to k-1, k+1 to n;
  do i=k+1 to n-1;

- These iterative DO statements use the BY increment syntax:
do i=n to 1 by -1;

do i=.1 to .9 by .1, 1 to 10 by 1,
   20 to 100 by 10;

do count=2 to 8 by 2;

These iterative DO statements use WHILE and UNTIL clauses.

do i=1 to 10 while(x<y);

do i=2 to 20 by 2 until((x/3)>y);

do i=10 to 0 by -1 while(month='JAN');

In this example, the DO loop is executed when I=1 and I=2. The WHILE
condition is evaluated when I=3, and the DO loop is executed if the WHILE
condition is true.

do i=1,2,3 while (condition);

Example 2: Using the Iterative DO Statement without Infinite Looping

In each of these examples, the DO group executes 10 times. The first example
demonstrates the preferred approach.

/* correct coding */
do i=1 to 10;
   ...more SAS statements...
end;

This example uses the TO and BY arguments.

do i=1 to n by m;
   ...more SAS statements...
   if i=10 then leave;
end;
   if i=10 then put 'EXITED LOOP';

Example 3: Stopping the Execution of the DO Loop

In this example, setting the value of the index variable to the current value of EXIT
causes the loop to terminate.

data iterate1;
   input x;
   exit=10;
   do i=1 to exit;
      y=x*normal(0);
      /* if y>25, */
      /* changing i's value */
      /* stops execution */
      if y>25 then i=exit;
      output;
   end;
   datalines;
5
000
2500
;
DO UNTIL Statement

Executes statements in a DO loop repetitively until a condition is true.

Valid in: DATA step
Categories: CAS
Control
Type: Executable

Syntax

DO UNTIL (expression);

...more SAS statements...

END;

Arguments

(expression)

is any SAS expression, enclosed in parentheses. You must specify at least one expression.

Details

The expression is evaluated at the bottom of the loop after the statements in the DO loop have been executed. If the expression is true, the DO loop does not iterate again.

Note: The DO loop always iterates at least once.

Comparisons

There are three other forms of the DO statement:
The DO statement, the simplest form of DO-group processing, designates a group of statements to be executed as a unit, usually as a part of IF-THEN/ELSE statements.

The iterative DO statement executes statements between DO and END statements repetitively based on the value of an index variable.

The DO WHILE statement executes statements in a DO loop repetitively while a condition is true, checking the condition before each iteration of the DO loop. The DO UNTIL statement evaluates the condition at the bottom of the loop; the DO WHILE statement evaluates the condition at the top of the loop.

Note: The statements in a DO UNTIL loop always execute at least one time, whereas the statements in a DO WHILE loop do not iterate even once if the condition is false.

Example: Using a DO UNTIL Statement to Repeat a Loop

These statements repeat the loop until N is greater than or equal to 5. The expression N>=5 is evaluated at the bottom of the loop. There are five iterations in all (0, 1, 2, 3, 4).

```sas
n=0;
do until(n>=5);
   put n=;
   n+1;
end;
```

See Also

Statements:

- "DO Statement" on page 62
- "DO Statement: Iterative" on page 64
- "DO WHILE Statement" on page 69

DO WHILE Statement

Executes statements in a DO loop repetitively while a condition is true.

Valid in: DATA step

Categories: CAS
            Control

Type: Executable

Syntax

```sas
DO WHILE (expression);
   ...more SAS statements...
```
END;

Arguments

(expression)

is any SAS expression, enclosed in parentheses. You must specify at least one expression.

Details

The expression is evaluated at the top of the loop before the statements in the DO loop are executed. If the expression is true, the DO loop iterates. If the expression is false the first time it is evaluated, the DO loop does not iterate even once.

Comparisons

There are three other forms of the DO statement:

- The DO statement, the simplest form of DO-group processing, designates a group of statements to be executed as a unit, usually as a part of IF-THEN/ELSE statements.
- The iterative DO statement executes statements between DO and END statements repetitively based on the value of an index variable.
- The DO UNTIL statement executes statements in a DO loop repetitively until a condition is true, checking the condition after each iteration of the DO loop. The DO WHILE statement evaluates the condition at the top of the loop; the DO UNTIL statement evaluates the condition at the bottom of the loop.

Note: If the expression is false, the statements in a DO WHILE loop do not execute. However, because the DO UNTIL expression is evaluated at the bottom of the loop, the statements in the DO UNTIL loop always execute at least once.

Example: Using a DO WHILE Statement

These statements repeat the loop while N is less than 5. The expression N<5 is evaluated at the top of the loop. There are five iterations in all (0, 1, 2, 3, 4).

```sas
n=0;
do while(n<5);
   put n=;
   n+1;
end;
```

See Also

Statements:

- "DO Statement” on page 62
- ”DO Statement: Iterative" on page 64
- “DO UNTIL Statement” on page 68
**DROP Statement**

Excludes variables from output SAS data sets.

Valid in:   DATA step  
Categories: CAS  
Information  
Type:      Declarative

**Syntax**

```
DROP variable-list;
```

**Arguments**

*variable-list*

specifies the names of the variables to omit from the output data set.

**Tip** You can list the variables in any form that SAS allows.

**Details**

The DROP statement applies to all the SAS data sets that are created within the same DATA step and can appear anywhere in the step. The variables in the DROP statement are available for processing in the DATA step. If no DROP or KEEP statement appears, all data sets that are created in the DATA step contain all variables. Do not use both DROP and KEEP statements within the same DATA step.

**Comparisons**

- The DROP statement differs from the DROP= data set option in these ways:
  - You cannot use the DROP statement in SAS procedure steps.
  - The DROP statement applies to all output data sets that are named in the DATA statement. To exclude variables from some data sets but not from others, use the DROP= data set option in the DATA statement.

- The KEEP statement is a parallel statement that specifies a list of variables to write to output data sets. Use the KEEP statement instead of the DROP statement if the number of variables to include is significantly smaller than the number to omit.

- Do not confuse the DROP statement with the DELETE statement. The DROP statement excludes variables from output data sets; the DELETE statement excludes observations.
Examples:

Example 1: Basic DROP Statement Usage
These examples show the correct syntax for listing variables with the DROP statement.

```sas
    drop time shift batchnum;
    drop grade1-grade20;
```

Example 2: Dropping Variables from the Output Data Set
In this example, the variables PURCHASE and REPAIR are used in processing but are not written to the output data set INVENTORY.

```sas
    data inventory;
    drop purchase repair;
    infile file-specification;
    input unit part purchase repair;
    totcost=sum(purchase,repair);
    run;
```

See Also

**Data Set Options:**
- "DROP= Data Set Option" in SAS Data Set Options: Reference

**Statements:**
- “DELETE Statement” on page 58
- “KEEP Statement” on page 182

---

**END Statement**

Ends DO group or SELECT group processing.

**Valid in:** DATA step

**Categories:** CAS
Control

**Type:** Declarative

**Syntax**

```sas
    END;
```

**Without Arguments**
Use the END statement to end DO group or SELECT group processing.
Details

The END statement must be the last statement in a DO group or a SELECT group.

Example: Using the END Statement

This example shows how to use the END statement to end a simple DO group.

```sas
do;
    ...more SAS statements...
end;
```

This example shows how to use the END statement to end a simple SELECT group.

```sas
select(expression);
    when(expression) SAS statement;
    otherwise SAS statement;
end;
```

See Also

**Statements:**
- “DO Statement” on page 62
- “SELECT Statement” on page 286

---

**ENDSAS Statement**

Terminates a SAS job or session after the current DATA or PROC step executes.

*Note:* The **ENDSAS Statement** has moved to **SAS Global Statements**.

---

**ERROR Statement**

Sets _ERROR_ to 1. A message written to the SAS log is optional.

**Valid in:** DATA step
**Categories:** Action
**Type:** CAS
**Executable**

**Syntax**

```sas
ERROR <message>;
```

**Without Arguments**

Using ERROR without an argument sets the automatic variable _ERROR_ to 1 and writes a blank message to the SAS log.
Arguments

**message**

writes a message to the SAS log.

Tip  *Message* can include character literals (enclosed in quotation marks), variable names, formats, and pointer controls.

Details

The ERROR statement sets the automatic variable _ERROR_ to 1. Writing a message that you specify to the SAS log is optional. When _ERROR_ = 1, SAS writes the data lines that correspond to the current observation in the SAS log.

Using ERROR is equivalent to using these statements in combination:

- an assignment statement setting _ERROR_ to 1
- a FILE LOG statement
- a PUT statement (if you specify a message)
- a PUT; statement (if you do not specify a message)
- another FILE statement resetting FILE to any previously specified setting

Example: Writing Error Messages

SAS writes the error message and the variable name and value to the SAS log for each observation that satisfies the condition in the IF-THEN statement.

- In this example, the ERROR statement automatically resets the FILE statement specification to the previously specified setting.

```sas
file file-specification;
  if type='teen' & age > 19 then
    error 'type and age don"t match ' age=;
```

- This example uses a series of statements to produce the same results.

```sas
file file-specification;
  if type='teen' & age > 19 then
    do;
      file log;
      put 'type and age don"t match ' age=;
      _error_=1;
      file file-specification;
    end;
```

See Also

**Statements:**

- “PUT Statement” on page 236
EXECUTE Statement

Executes a stored compiled DATA step program.

Valid in: DATA step
Category: Action
Type: Executable
Restrictions: This statement is not supported in a DATA step that runs in CAS.
Use EXECUTE with stored compiled DATA step programs only.
Requirement: You must specify the PGM= option in the DATA step.

Syntax

EXECUTE;

Without Arguments

The EXECUTE statement executes a stored compiled DATA step program.

Details

Use the DESCRIBE statement with the EXECUTE statement in the same DATA step to retrieve the source code and execute a stored compiled DATA step program. If you do not specify either statement, EXECUTE is assumed. The order in which you use the statements is interchangeable. The DATA step program executes when it reaches a step boundary. For information about how to use these statements with the DATA statement, see “DATA Statement” on page 44.

See Also

Statements:

- “DATA Statement” on page 44
- “DESCRIBE Statement” on page 59

FILE Statement

Specifies the current output file for PUT statements.

Valid in: DATA step
Categories: CAS
File-Handling
Type: Executable
Restriction: When SAS is in a locked-down state, the FILENAME statement is not available for files that are not in the locked-down path list. For more information, see “SAS Processing Restrictions for Servers in a Locked-Down State” in SAS Language Reference: Concepts.

See: FILE Statement under Windows, UNIX, and z/OS

Syntax

FILE file-specification <device-type> <options> <operating-environment-options>;

Arguments

file-specification

identifies an external file that the DATA step uses to write output from a PUT statement. File-specification can have these forms:

'external-file'

specifies the physical name of an external file, which is enclosed in quotation marks. The physical name is the name by which the operating environment recognizes the file.

fileref

specifies the fileref of an external file.

Range 1 to 8 bytes

Restriction This argument is not supported in a DATA step that runs in CAS.

Requirement You must have associated fileref with an external file in a FILENAME statement or function in a previous step or in an appropriate operating environment command. The only way to assign fileref at run time is to use the FILEVAR= option in the FILE statement.

Note If your fileref is associated with the EMAIL access method, additional options are available for the FILE statement. For a complete list of options, see “FILENAME Statement: EMAIL (SMTP) Access Method” on page 98.

See “FILENAME Statement” on page 97

fileref(file)

specifies a fileref that is previously assigned to an external file that is an aggregate grouping of files. Follow the fileref with the name of a file or member, which is enclosed in parentheses.

Restriction This argument is not supported in a DATA step that runs in CAS.

Requirement You must have associated fileref with an external file in a FILENAME statement or function in a previous step or in an appropriate operating environment command.

Operating environment Different operating environments call an aggregate grouping of files by different names such as a directory, a MACLIB, or
a partitioned data set. For more information, see the SAS documentation for your operating environment.

### Note
A file that is located in an aggregate storage location and has a name that is not a valid SAS name must have its name enclosed in quotation marks.

### See
“FILENAME Statement” on page 97

## LOG

is a reserved fileref that directs the output that is produced by any PUT statements to the SAS log.

At the beginning of each execution of a DATA step, the fileref that indicates where the PUT statements write is automatically set to LOG. Therefore, the first PUT statement in a DATA step always writes to the SAS log, unless the PUT statement is preceded by a FILE statement that specifies otherwise.

### Tip
Because output lines are by default written to the SAS log, use a FILE LOG statement to restore the default action or to specify additional FILE statement options.

## PRINT

is a reserved fileref that directs the output that is produced by any PUT statements to the same file as the output that is produced by SAS procedures.

### Restriction
This argument is not supported in a DATA step that runs in CAS.

### Interaction
When you write to a file, the value of the N= option must be either 1 or PAGESIZE.

### Operating environment
The carriage-control characters that are written to a file can be specific to the operating environment. For more information, see the SAS documentation for your operating environment.

### Tip
When PRINT is the fileref, SAS uses carriage-control characters and writes the output with the characteristics of a print file.

### See
A complete discussion of print files in *SAS Language Reference: Concepts*

### Tip
If the file does not exist in the directory that you specify for file-specification, SAS creates the file. If the directory that is specified in file-specification does not exist, SAS sets the SYSERR macro variable, which can be checked if the ERRORCHECK option is set to STRICT.

### device-type
specifies the type of device or the access method that is used if the fileref points to an input or output device or a location that is not a physical file.

### ACTIVEMQ
specifies an access method that enables you to access an ActiveMQ messaging broker.
restriction This argument is not supported in a DATA step that runs in CAS.

Interaction If the DATA step does not recognize the access method option, the DATA step passes the option to the access method for handling.

See “FILENAME Statement: ACTIVEMQ Access Method” in Application Messaging with SAS

**CATALOG** specifies the CATALOG access method.

Restriction This argument is not supported in a DATA step that runs in CAS.

Interaction If the DATA step does not recognize the access method option, the DATA step passes the option to the access method for handling.

See For a complete list of options that are available with the CATALOG access method, see “FILENAME Statement: CATALOG Access Method” on page 98.

**CLIPBOARD** specifies the CLIPBOARD access method.

Restriction This argument is not supported in a DATA step that runs in CAS.

Interaction If the DATA step does not recognize the access method option, the DATA step passes the option to the access method for handling.

See For a complete list of options that are available with the CLIPBOARD access method, see “FILENAME Statement: CLIPBOARD Access Method” on page 98.

**DISK** specifies that the device is a disk drive.

Restriction This argument is not supported in a DATA step that runs in CAS.

**Tip** When you assign a fileref to a file on disk, you are not required to specify DISK.

**DUMMY** specifies that the output to the file is discarded.

Restriction This argument is not supported in a DATA step that runs in CAS.

Interaction If the FILEVAR= option is used in conjunction with DUMMY, no files are written to the names that are listed in the FILEVAR= option.

**Tip** Specifying DUMMY can be useful for testing.

**FTP** specifies the FTP access method.

Restriction This argument is not supported in a DATA step that runs in CAS.
### Interaction
If the DATA step does not recognize the access method option, the DATA step passes the option to the access method for handling.

### See
For a complete list of options that are available with the FTP access method, see "FILENAME Statement: FTP Access Method" on page 99.

### Example
```plaintext
infile dummy ftp user='myuid' pass='xxxx' filevar=file_to_read;
```

#### HADOOP
specifies the Hadoop access method.

**Restriction**
This argument is not supported in a DATA step that runs in CAS.

**Interaction**
If the DATA step does not recognize the access method option, the DATA step passes the option to the access method for handling.

### See
For a complete list of options that are available with the Hadoop access method, see "FILENAME Statement: Hadoop Access Method" on page 99.

#### GTERM
indicates that the output device type is a graphics device that receives graphics data.

**Restriction**
This argument is not supported in a DATA step that runs in CAS.

#### JMS
specifies a Java Message Service (JMS) destination.

**Restriction**
This argument is not supported in a DATA step that runs in CAS.

#### PIPE
specifies an unnamed pipe.

**Restriction**
This argument is not supported in a DATA step that runs in CAS.

**Operating environment**
Some operating environments do not support pipes.

#### PLOTTER
specifies an unbuffered graphics output device.

**Restriction**
This argument is not supported in a DATA step that runs in CAS.

#### PRINTER
specifies a printer or printer spool file.

**Restriction**
This argument is not supported in a DATA step that runs in CAS.

#### SFTP
specifies the SFTP access method.

**Restriction**
This argument is not supported in a DATA step that runs in CAS.
<table>
<thead>
<tr>
<th>Access Method</th>
<th>Description</th>
<th>Interaction</th>
<th>Restriction</th>
<th>Tip</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATA step</td>
<td></td>
<td>If the DATA step does not recognize the access method option, the DATA step passes the option to the access method for handling.</td>
<td>This argument is not supported in a DATA step that runs in CAS.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SFTP</td>
<td></td>
<td>For a complete list of options that are available with the SFTP access method, see “FILENAME Statement: SFTP Access Method” on page 99.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SOCKET</td>
<td>specifies the SOCKET access method.</td>
<td>If the DATA step does not recognize the access method option, the DATA step passes the option to the access method for handling.</td>
<td>This argument is not supported in a DATA step that runs in CAS.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TAPE</td>
<td>specifies a tape drive.</td>
<td>For a complete list of options that are available with the SOCKET access method, see “FILENAME Statement: SOCKET Access Method” on page 99.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TERMINAL</td>
<td>specifies the user's terminal.</td>
<td>This argument is not supported in a DATA step that runs in CAS.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UPRINTER</td>
<td>specifies a Universal Printer definition name.</td>
<td>This argument is not supported in a DATA step that runs in CAS.</td>
<td></td>
<td>If you do not specify the printer name in the FILENAME statement, the PRINTERPATH options control which Universal Printer is used and the destination of the output.</td>
<td></td>
</tr>
<tr>
<td>WEBDAV</td>
<td>specifies the WEBDAV access method.</td>
<td>For a complete list of options that are available with the WEBDAV access method, see “FILENAME Statement: WebDAV Access Method” on page 100.</td>
<td>This argument is not supported in a DATA step that runs in CAS.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Alias: DEVICE=\textit{device-type}

Default: DISK
**Requirements**

device-type or DEVICE=device-type must immediately follow file-specification in the statement.

**Operating Environment**

Additional specifications might be required when you specify some devices. See the SAS documentation for your operating environment before specifying a value other than DISK. Values in addition to the ones listed here might be available in some operating environments.

---

**Options**

**BLKSIZE=block-size**

specifies the block size of the output file.

*Default* Dependent on your operating environment. For more information, see the FILE Statement in the SAS documentation for your operating environment.

**COLUMN=variable**

specifies a variable that SAS automatically sets to the current column location of the pointer. This variable, like automatic variables, is not written to the data set.

*Alias* COL=

*See* LINE= on page 85

**DELIMITER= delimiter(s)**

specifies an alternate delimiter (other than blank) to be used for LIST output, where delimiter can be one of these items:

'list-of-delimiting-characters'

specifies one or more characters to write as delimiters.

*Requirement* Enclose the list of characters in quotation marks.

**character-variable**

specifies a character variable whose value becomes the delimiter.

*Alias* DLM=

*Default* blank space

*Restriction* Even though a character string or character variable is accepted, only the first character of the string or variable is used as the output delimiter. The FILE DLM= processing differs from INFILE DELIMITER= processing.

*Interaction* Output that contains embedded delimiters requires the delimiter sensitive data (DSD) option.

*Tips* DELIMITER= can be used with the colon (:) modifier (modified LIST output).

The delimiter is case sensitive.

*See* “DLMSTR= delimiter” on page 82 and “DSD (delimiter sensitive data)” on page 83
DLMSOPT=T' | t'
specifies a parsing option for the DLMSTR=T option that removes trailing blanks of the string delimiter.

Requirement  The DLMSOPT=T option has an effect only when used with the DLMSTR= option.

Tip        The DLMSOPT=T option is useful when you use a variable as the delimiter string.

See       DLMSTR= on page 82

DLMSTR= delimiter
specifies a character string as an alternate delimiter (other than a blank) to be used for LIST output, where delimiter can be one of these items:

'delimiting-string'
specifies a character string to write as a delimiter.

Requirement Enclose the string in quotation marks.

character-variable
specifies a character variable whose value becomes the delimiter.

Default     blank space

Interactions If you specify more than one DLMSTR= option in the FILE statement, the DLMSTR= option that is specified last is used. If you specify both the DELIMITER= and DLMSTR= options, the option that is specified last is used.

If you specify RECFM=N, ensure that the LRECL= option is large enough to hold the largest input item. Otherwise, it is possible for the delimiter to be split across the record boundary.

See       DELIMITER= on page 81, DLMSOPT= on page 82, and DSD on page 83

DROPOVER
discards data items that exceed the output line length (as specified by the LINESIZE= option or LRECL= option in the FILE statement).

By default, data that exceeds the current line length is written on a new line. When you specify DROPOVER, SAS drops (or ignores) an entire item when there is not enough space in the current line to write it. When an entire item is dropped, the column pointer remains positioned after the last value that is written in the current line. Thus, the PUT statement might write other items in the current output line if they fit in the space that remains or if the column pointer is repositioned. When a data item is dropped, the DATA step continues normal execution (_ERROR_ =0). At the end of the DATA step, a message is printed for each file from which data was lost.

Default     FLOWOVER

Tip        Use DROPOVER when you want the DATA step to continue executing if the PUT statement attempts to write past the current line length, but you do not want the data item that exceeds the line length to be written on a new line.
DSD (delimiter sensitive data)
specifies that data values that contain embedded delimiters, such as tabs or commas, be enclosed in quotation marks. The DSD option enables you to write data values that contain embedded delimiters to LIST output. This option is ignored for other types of output (for example, formatted, column, and named). Any double quotation marks that are included in the data value are repeated. When a variable value contains the delimiter and DSD is used in the FILE statement, the variable value is enclosed in double quotation marks when the output is generated. For example:

```sas
data _null_
   file log dsd;
   x="lions, tigers, and bears";
   put x ' "Oh, my!"';
run;
```

The program writes this line to the SAS log:

```plaintext
""lions, tigers, and bears"", "Oh, my!"
```

If a quoted (text) string contains the delimiter and DSD is used in the FILE statement, then the quoted string is not enclosed in double quotation marks when used in a PUT statement. For example:

```sas
data _null_
   file log dsd;
   put 'lions, tigers, and bears';
run;
```

The program writes this line to the SAS log:

```plaintext
lions, tigers, and bears
```

Interaction
If you specify DSD, the default delimiter is assumed to be the comma (,). Specify the DELIMITER= option or DLMSTR= option if you want to use a different delimiter.

Tip
By default, data values that do not contain the delimiter that you specify are not enclosed in quotation marks. However, you can use the tilde (~) modifier to force any data value, including missing values, to be enclosed in quotation marks, even if the data value contains no embedded delimiter.

See DELIMITER= on page 81 and DLMSTR= on page 82

ENCODING= 'encoding-value'
specifies the encoding to use when writing to the output file. The value for ENCODING= indicates that the output file has a different encoding from the current session encoding.

When you write data to the output file, SAS transcodes the data from the session encoding to the specified encoding.

Default SAS uses the current session encoding.

FILENAME=variable
defines a character variable, whose name you supply, that SAS sets to the value of the physical name of the file currently open for PUT statement output. The physical name is how the operating environment recognizes the file.

Tips
This variable, like automatic variables, is not written to the data set.

Use a LENGTH statement to make the variable length long enough to contain the value of the physical file name if the variable length is longer than eight bytes (the default length of a character variable).

See
FILENAME= on page 84

Example “Example 8: Specifying an Encoding When Writing to an Output File” on page 96

FILEVAR=variable
defines a variable whose change in value causes the FILE statement to close the current output file and open a new one the next time the FILE statement executes. The next PUT statement that executes writes to the new file that is specified as the value of the FILEVAR= variable.

Restriction
The value of a FILEVAR= variable is expressed as a character string that contains a physical file name.

Interaction
When you use the FILEVAR= option, the file-specification is just a placeholder, not an actual file name or a fileref that has been previously assigned to a file. SAS uses this placeholder for reporting processing information to the SAS log. The placeholder must conform to the same rules as a fileref.

Tips
This variable, like automatic variables, is not written to the data set.

If any of the physical file names are longer than eight bytes (the default length of a character variable), assign the FILEVAR= variable a longer length with another statement, such as a LENGTH statement or an INPUT statement.

See
FILENAME= on page 84

Example “Example 4: Identifying the Current Output File” on page 95

FLOWOVER
causes data that exceeds the current line length to be written on a new line. When a PUT statement attempts to write beyond the maximum allowed line length (as specified by the LINESIZE= option in the FILE statement), the current output line is written to the file and the data item that exceeds the current line length is written to a new line.

Default
FLOWOVER

Interaction
If the PUT statement contains a trailing @, the pointer is positioned after the data item on the new line and the next PUT statement writes to that line. This process continues until the end of the input.

Example “Example 5: Dynamically Changing the Current Output File” on page 95
data is reached or until a PUT statement without a trailing @ causes the current line to be written to the file.

See “DROPOVER” on page 82 and “STOPOVER” on page 90

**FOOTNOTES | NOFOOTNOTES**

controls whether currently defined footnotes are printed.

**Alias**

FOOTNOTE | NOFOOTNOTE

**Default**

NOFOOTNOTES

**Requirement**

In order to print footnotes in a DATA step report, you must set the FOOTNOTE option in the FILE statement.

**HEADER=label**

defines a statement label that identifies a group of SAS statements that you want to execute each time SAS begins a new output page.

**Restrictions**

The first statement after the label must be an executable statement. Thereafter, you can use any SAS statement.

Use the HEADER= option only when you write to print files and when you include the PRINT= option.

**Tip**

To prevent the statements in this group from executing with each iteration of the DATA step, use two RETURN statements: one statement precedes the label and the other appears as the last statement in the group.

**Example**

“Example 1: Executing Statements When Beginning a New Page” on page 93

**LINE=variable**

defines a variable whose value is the current relative line number within the group of lines available to the output pointer. You supply the variable name and SAS automatically assigns the value.

**Range**

1 to the value that is specified by the N= option or with the #n line pointer control. If neither value is specified, the LINE= variable has a value of 1.

**Tips**

This variable, like automatic variables, is not written to the data set.

The value of the LINE= variable is set at the end of PUT statement execution to the number of the next available line.

**LINESIZE=line-size**

sets the maximum number of columns per line for reports and the maximum record length for data files.

**Alias**

LS=

**Default**

The default LINESIZE= value is determined by one of two options: 1) the LINESIZE= system option when you write to a file that contains carriage-control characters or to the SAS log or 2) the LRECL= option in the FILE statement when you write to a file.
### RANGE

From 64 to the maximum logical record length that is allowed in your operating environment.

### INTERACTION

If a PUT statement tries to write a line that is longer than the value that is specified by the LINESIZE= option, the action that is taken is determined by whether FLOWOVER, DROPOVER, or STOPOVER is in effect. By default (FLOWOVER), SAS writes the line as two or more separate records.

### OPERATING ENVIRONMENT

The highest value allowed for LINESIZE= is dependent on your operating environment. For more information, see the SAS documentation for your operating environment.

### NOTE

LINESIZE= tells SAS how much of the line to use. LRECL= specifies the physical record length of the file.

### SEE

LRECL= on page 86, "DROPOVER" on page 82, "FLOWOVER" on page 84, and "STOPOVER" on page 90

### EXAMPLE

"Example 6: When the Output Line Exceeds the Line Length of the Output File" on page 95

---

**LINESLEFT=variable**

defines a variable whose value is the number of lines left on the current page. You supply the variable name and SAS assigns the value of the number of lines left on the current page to that variable. The value of the LINESLEFT= variable is set at the end of PUT statement execution.

**Alias**

LL=

**Tip**

This variable, like automatic variables, is not written to the data set.

**Example**


---

**LRECL=logical-record-length**

specifies the logical record length of the output file.

**Default**

If you omit the LRECL= option, SAS chooses a value based on the operating environment's file characteristics.

**Interaction**

Alternatively, you can specify a global logical record length by using the LRECL system option. In SAS 9.4, the default value for the global LRECL system option is 32767. If you are using fixed-length records (RECFM=F), the default value for LRECL is 256.

**Operating Environment**

Values for logical-record-length are dependent on the operating environment. For more information, see the SAS documentation for your operating environment.

**Note**

LINESIZE= tells SAS how much of the line to use; LRECL= specifies the physical line length of the file.

**See**

LINESIZE= on page 85, PAD on page 88, and PAGESIZE= on page 89
MEMVAR=variable
specifies a file to open. When a directory fileref or physical name is provided in the FILE statement, the file that is referred to in the MEMVAR= variable is opened. When the value of the MEMVAR= variable changes, the current file is closed and the new file is opened.

Restriction
The value of a MEMVAR= variable is expressed as a character string that contains a physical file name.

Tips
This variable, like automatic variables, is not written to the data set.

If any of the physical file names are longer than eight bytes (the default length of a character variable), assign the MEMVAR= variable a longer length with another statement, such as a LENGTH statement or an INPUT statement.

See
FILENAME= on page 84

MOD
writes the output lines after any existing lines in the file.
Default
OLD
Restrictions
MOD is not accepted under all operating environments. For more information, see the SAS documentation for your operating environment.

Do not use the MOD option with any ODS destination other than the Listing destination. Otherwise, you might receive unexpected output.

See
“OLD” on page 88

N=available-lines
specifies the number of lines that you want available to the output pointer in the current iteration of the DATA step. Available-lines can be expressed as a number (n) or as the keyword PAGESIZE or PS.

n specifies the number of lines that are available to the output pointer. The system can move back and forth between the number of lines that are specified while composing them before moving on to the next set.

PAGESIZE
specifies that the entire page is available to the output pointer.
Alias
PS
Restrictions
N=PAGESIZE is valid only when output is printed.

If the current output file is a file that is to be printed, available-lines must have a value of either 1 or PAGESIZE.

Interactions
In addition to using in the N= option to control the number of lines available to the output pointer, you can also use the #n line pointer control in a PUT statement.

If you omit the N= option and no # pointer controls are used, one line is available. That is, by default, N=1. If N= is not used
but there are # pointer controls, N= is assigned the highest value that is specified for a # pointer control in any PUT statement in the current DATA step.

**Tip** Setting N=PAGESIZE enables you to compose a page of multiple columns one column at a time.

**Example** “Example 3: Arranging the Contents of an Entire Page” on page 94

---

**ODS < = (ODS-suboptions) >** specifies to use the Output Delivery System to format the output from a DATA step. This option defines the structure of the data component and holds the results of the DATA step and binds that component to a table definition to produce an output object. ODS sends this object to all open ODS destinations, each of which formats the output appropriately. For information about the ODS-suboptions and the Output Delivery System, see “FILE Statement: ODS” in SAS Output Delivery System: User’s Guide.

**Defaults** If you omit the ODS suboptions, the DATA step uses a default table definition (base.datastep.table) that is stored in the SASHELP.TMPLMST template store. This definition defines two generic columns: one column for character variables and one column for numeric variables. ODS associates each variable in the DATA step with one of these columns and displays the variables in the order in which they are defined in the DATA step.

Without suboptions, the default table definition uses the variable’s label as its column heading. If no label exists, the definition uses the variable’s name as the column heading.

**Restriction** You cannot use _FILE_ _, FILEVAR=, HEADER=, or PAD with the ODS option.

**Requirement** The ODS option is valid only when you use the fileref PRINT in the FILE statement.

**Interaction** The DELIMITER= and DSD options have no effect on the ODS option. The FOOTNOTES|NOFOOTNOTES, LINESIZE, PAGESIZE, and TITLES | NOTITLES options have an effect only on the LISTING destination.

---

**OLD** replaces the previous contents of the file.

**Default** OLD

**Restriction** OLD is not accepted under all operating environments. For more information, see the SAS documentation for your operating environment.

**See** “MOD” on page 87

---

**PAD | NOPAD** controls whether records written to an external file are padded with blanks to the length that is specified in the LRECL= option.
Default  NOPAD is the default when writing to a variable-length file; PAD is the default when writing to a fixed-length file.

Tip  PAD provides a quick way to create fixed-length records in a variable-length file.

See  LRECL= on page 86

**PAGESIZE=value**

sets the number of lines per page for your reports.

<table>
<thead>
<tr>
<th>Alias</th>
<th>PS=</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default</td>
<td>The value of the PAGESIZE= system option</td>
</tr>
<tr>
<td>Range</td>
<td>The value can range from 15 to 32767.</td>
</tr>
<tr>
<td>Interaction</td>
<td>If any TITLE statements are currently defined, the lines that they occupy are included in counting the number of lines for each page.</td>
</tr>
<tr>
<td>Tips</td>
<td>After the value of the PAGESIZE= option is reached, the output pointer advances to line 1 of a new page.</td>
</tr>
<tr>
<td></td>
<td>If you specify FILE LOG, the number of lines that are written on the first page is reduced by the number of lines in the SAS start-up notes. For example, if PAGESIZE=20 and there are nine lines of SAS start-up notes, only 11 lines are available for output on the first page.</td>
</tr>
</tbody>
</table>

See  “PAGESIZE= System Option” in SAS System Options: Reference

**PRINT | NOPRINT**

controls whether carriage-control characters are placed in the output lines.

| Restriction | When you write to a file, the value of the N= option must be either 1 or PAGESIZE. |
| Operating environment | The carriage-control characters that are written to a file can be specific to the operating environment. For more information, see the SAS documentation for your operating environment. |
| Tips   | The PRINT option is not necessary if you are using fileref PRINT. |
|        | If you specify FILE PRINT in an interactive SAS session, then the Output window interprets the form-feed control characters as page breaks, and blank lines that are written before the form feed are removed from the output. Writing the results from the Output window to a flat file produces a file without page break characters. If a file needs to contain the form-feed characters, then the FILE statement should include a physical file location and the PRINT option. |

**RECFM=record-format**

specifies the record format of the output file.

| Range  | Values are dependent on the operating environment. For more information, see the SAS documentation for your operating environment. |
Interaction
In SAS 9.4, the default value for the global LRECL system option is 32767. If you are using fixed-length records (RECFM=F), the default value for LRECL is 256.

STOPOVER
stops processing the DATA step immediately if a PUT statement attempts to write a data item that exceeds the current line length. In such a case, SAS discards the data item that exceeds the current line length, writes the portion of the line that was built before the error occurred, and issues an error message.

Default FLOWOVER

See “FLOWOVER” on page 84 and “DROPOVER” on page 82

TITLES | NOTITLES
controls the printing of the current title lines on the pages of files. When NOTITLES is omitted or when TITLES is specified, SAS prints any titles that are currently defined.

Alias TITLE | NOTITLE

Default TITLES

_FILE_=variable
names a character variable that references the current output buffer of this FILE statement. You can use the variable in the same way as any other variable, even as the target of an assignment. The variable is automatically retained and initialized to blanks. Like automatic variables, the _FILE_= variable is not written to the data set.

Restriction variable cannot be a previously defined variable. Ensure that the _FILE_= specification is the first occurrence of _FILE_ in the DATA step. Do not set or change the length of _FILE_= variable with the LENGTH statement or ATTRIB statement. However, you can attach a format to this variable with the ATTRIB statement or FORMAT statement.

Interaction The maximum length of this character variable is the logical record length (LRECL) for the specified FILE statement. However, SAS does not open the file to know the LRECL until before the execution phase. Therefore, the designated size for this variable during the compilation phase is 32767 bytes.

Tips Modification of this variable directly modifies the FILE statement's current output buffer. Any subsequent PUT statement for this FILE statement writes the contents of the modified buffer. The _FILE_= variable accesses only the current output buffer of the specified FILE statement even if you use the N= option to specify multiple output buffers.

To access the contents of the output buffer in another statement without using the _FILE_= option, use the automatic variable _FILE_.

See “Updating the _FILE_ Variable” on page 92
Operating Environment Options

For descriptions of operating-environment-specific options in the FILE statement, see the SAS documentation for your operating environment.

Details

Overview

By default, PUT statement output is written to the SAS log. Use the FILE statement to route this output to either the same external file to which procedure output is written or to a different external file. You can indicate whether carriage-control characters should be added to the file. See PRINT | NOPRINT option on page 89.

You can use the FILE statement in conditional (IF-THEN) processing because it is executable. You can also use multiple FILE statements to write to more than one external file in a single DATA step.

Operating Environment Information: Using the FILE statement requires operating-environment-specific information. See the SAS documentation for your operating environment before you use this statement.

You can use the Output Delivery System with the FILE statement to write DATA step results. For more information, see “FILE Statement: ODS” in SAS Output Delivery System: User’s Guide.

Updating an External File in Place

You can use the FILE statement with the INFILE and PUT statements to update an external file in place, updating either an entire record or only selected fields within a record. Follow these guidelines:

- Always place the INFILE statement first.
- Specify the same fileref or physical file name in the INFILE and FILE statements.
- Use options that are common to both the INFILE and FILE statements in the INFILE statement. (Any such options that are used in the FILE statement are ignored.)
- Use the SHAREBUFFERS option in the INFILE statement to allow the INFILE and FILE statements to use the same buffer, which saves CPU time and enables you to update individual fields instead of entire records.

Accessing the Contents of the Output Buffer

In addition to the _FILE_= variable, you can use the automatic _FILE_ variable to reference the contents of the current output buffer for the most recent execution of the FILE statement. This character variable is automatically retained and initialized to blanks. Like other automatic variables, _FILE_ is not written to the data set.

When you specify the _FILE_= option in a FILE statement, this variable is also indirectly referenced by the automatic _FILE_ variable. If the automatic _FILE_variable is present and you omit _FILE_= in a particular FILE statement, SAS creates an internal _FILE_= variable for that FILE statement. Otherwise, SAS does not create the _FILE_= variable for a particular FILE.

During execution and at the point of reference, the maximum length of this character variable is the maximum length of the current _FILE_ variable. However, because _FILE_ references only other variables whose lengths are not known until before the execution phase, the designated length is 32,767 bytes during the compilation.
phase. For example, if you assign _FILE_ to a new variable whose length is undefined, the default length of the new variable is 32,767 bytes. You cannot use the LENGTH statement and the ATTRIB statement to set or override the length of _FILE_. You can use the FORMAT statement and the ATTRIB statement to assign a format to _FILE_.

Updating the _FILE_ Variable

Like other SAS variables, you can update the _FILE_ variable. These two methods are available:

- Use _FILE_ in an assignment statement.
- Use a PUT statement.

You can update the _FILE_ variable by using an assignment statement that has this form:

```
_FILE_ = <'string-in-quotation-marks' | character-expression>
```

The assignment statement updates the contents of the current output buffer and sets the buffer length to the length of 'string-in-quotation-marks' or character-expression. However, using an assignment statement does not affect the current column pointer of the PUT statement. The next PUT statement for this FILE statement begins to update the buffer at column 1 or at the last known location when you use the trailing @ in the PUT statement.

In this example, the assignment statement updates the contents of the current output buffer. The column pointer of the PUT statement is not affected.

```
file print;
_FILE_ = '_FILE_';
pull 'This is PUT';
```

SAS creates this output: `This is PUT`

In this example, the PUT statement updates the contents of the current output buffer beginning at column 14.

```
file print;
_FILE_ = 'This is from FILE, sir.';
pull @14 'both';
```

SAS creates this output: `This is from both, sir.`

You can also update the _FILE_ variable by using a PUT statement. The PUT statement updates the _FILE_ variable because the PUT statement formats data in the output buffer and _FILE_ points to that buffer. However, by default SAS clears the output buffers after a PUT statement executes and writes the current record (or N= block of records). Therefore, if you want to examine or further modify the contents of _FILE_ before it is written, include a trailing @ or @@ in any PUT statement (when N=1). For other values of N=, use a trailing @ or @@ in any PUT statement where the last line pointer location is on the last record of the record block.

In this example, when N=1, the first PUT statement writes out something and the trailing @ holds the line and sets _FILE_ to something. Y is then assigned something is here.

```
file ABC;
pull 'Something' @;
Y = _FILE_ || ' is here';
```
In the second FILE statement, the PUT statement writes out *Nothing* and the current output buffer is cleared. Because there is no trailing @ on the second PUT statement, Y is assigned *is here*.

Any modification of _FILE_ directly modifies the current output buffer for the current FILE statement. The execution of any subsequent PUT statements for this FILE statement writes the contents of the modified buffer.

_FILE_ accesses only the contents of the current output buffer for a FILE statement, even when you use the N= option to specify multiple buffers. You can access all the N= buffers, but you must use a PUT statement with the # line pointer control to make the desired buffer the current output buffer.

Comparisons

- The FILE statement specifies the output file for PUT statements. The INFILE statement specifies the input file for INPUT statements.
- Both the FILE and INFILE statements enable you to use options that provide SAS with additional information about the external file being used.
- In the Program Editor, Log, and Output windows, the FILE command specifies an external file and writes the contents of the window to the file.

Examples:

Example 1: Executing Statements When Beginning a New Page

This DATA step illustrates how to use the HEADER= option.

- **Write a report.** Use DATA _NULL_ to write a report rather than create a data set.
  ```sas
  data _null_; 
  set sprint; 
  by dept; 
  
  Route output to the SAS output window. Point to the header information. The PRINT fileref routes output to the same location as procedure output. HEADER= points to the label that precedes the statements that create the header for each page.
  ```
  ```sas
  file print header=newpage; 
  
  Start a new page for each department:
  ```
  ```sas
  if first.dept then put _page_; 
  put @22 salesrep @34 salesamt; 
  ```
  ```sas
  Write a header on each page. These statements execute each time a new page is begun. RETURN is necessary before the label and as the final statement in a labeled group.
  ```
  ```sas
  return; 
  newpage: 
  put @20 'Sales for 1989' / 
  @20 dept=; 
  ```
Example 2: Determining New Page by Lines Left on the Current Page

This DATA step demonstrates using the LINESLEFT= option to determine where the page break should occur, according to the number of lines left on the current page.

- **Write a report.** Use DATA _NULL_ to write a report rather than create a data set.

```sas
data _null_;  
set info;
run;
```

- **Route output to the standard SAS output window.** The PRINT fileref routes output to the same location as procedure output. LINESLEFT indicates that the variable REMAIN contains the number of lines left on the current page.

```sas
file print linesleft=remain pagesize=20;  
put @5 name @30 phone  
   @35 bldg @37 room;
```

- **Begin a new page when there are fewer than seven lines left on the current page.** Under this condition, PUT _PAGE_ begins a new page and positions the pointer at line 1.

```sas
if remain<7 then put _page_;  
run;
```

Example 3: Arranging the Contents of an Entire Page

This example shows how to use N=PAGESIZE in a DATA step to produce a two-column telephone book listing. Each column contains a name and a phone number.

- **Create a report and write it to a SAS output window.** Use DATA _NULL_ to write a report rather than create a data set. PRINT is the fileref. SAS uses carriage-control characters to write the output with the characteristics of a print file. N=PAGESIZE makes the entire page available to the output pointer.

```sas
data _null_;  
file 'external-file' print n=pagesize;
```

- **Specify the columns for the report.** This DO loop iterates twice on each DATA step iteration. The COL value is 1 on the first iteration and 40 on the second.

```sas
do col=1, 40;
```

- **Write 20 lines of data.** This DO loop iterates 20 times to write 20 lines in column 1. When finished, the outer loop sets COL equal to 40 and this DO loop iterates 20 times again, writing 20 lines of data in the second column. The values of LINE and COL, which are set and incremented by the DO statements, control where the PUT statement writes the values of NAME and PHONE on the page.

```sas
do line=1 to 20;  
set info;  
   put #line @col name $20. +1 phone 4.;  
end;
```

- **After composing two columns of data, write the page.** This END statement ends the outer DO loop. The PUT _PAGE_ writes the current page and moves the pointer to the top of a new page.

```sas
end;
```
Example 4: Identifying the Current Output File

This DATA step causes a file identification message to be printed in the SAS log and assigns the value of the current output file to the variable MYOUT. The PUT statement, demonstrating the assignment of the proper value to MYOUT, writes the value of that variable to the output file.

```sas
data _null_;
  length myout $ 200;
  file file-specification filename=myout;
  put myout=;
  stop;
run;
```

The PUT statement writes a line to the current output file that contains the physical name of the file.

```
MYOUT=your-output-file
```

Example 5: Dynamically Changing the Current Output File

This DATA step uses the FILEVAR= option to dynamically change the currently opened output file to a new physical file.

- **Write a report. Create a long character variable.** Use DATA _NULL_ to write a report rather than create a data set. The LENGTH statement creates a variable with length long enough to contain the name of an external file.

  ```sas
  data _null_;
    length name $ 200;
  run;
  ```

- **Read an in-stream data line and assign a value to the NAME variable.**

  ```sas
  input name $;
  run;
  ```

- **Close the current output file and open a new one when the NAME variable changes.** The file-specification is a placeholder; it can be any valid SAS name.

  ```sas
  file file-specification filevar=name mod;
  date = date();
  ```

- **Append a log record to the currently open output file.**

  ```sas
  put 'records updated ' date date.;
  ```

- **Supply the names of the external files.**

  ```sas
  datalines;
  external-file-1
  external-file-2
  external-file-3
  ;
  ```

Example 6: When the Output Line Exceeds the Line Length of the Output File

Because the combined lengths of the variables are longer than the output line (80 characters), this PUT statement automatically writes three separate records.
Example 7: Reading Data and Writing Text through a TCP/IP Socket

This example shows reading raw data from a file through a TCP/IP socket. The NBYTE= option is used in the INFILE statement.

```sas
/* Start this first as the server */
filename serve socket ':5205' server
  recfm=s
  lrecl=25 blocksize=2500;
data _null_;  
  nb=25;
  infile serve nbyte=nb;
  input text $char25.;
  put _all_;  
run;
```

This example shows writing text to a file through a TCP/IP socket.

```sas
/* While the server test is running, */
/* continue with this as the client. */
filename client socket "&hstname:5205"
  recfm=s
  lrecl=25 blocksize=2500;
data _null_;  
  file client;
  put 'Some text to length 25...';
run;
```

Example 8: Specifying an Encoding When Writing to an Output File

This example creates an external file from a SAS data set. The current session encoding is Wlatin1, but the external file's encoding needs to be UTF-8. By default, SAS writes the external file using the current session encoding.

To tell SAS what encoding to use when writing data to the external file, specify the ENCODING= option. When you tell SAS that the external file is to be in UTF-8 encoding, SAS then transcodes the data from Wlatin1 to the specified UTF-8 encoding when writing to the external file.

```sas
libname myfiles 'SAS-library';
filename outfile 'external-file';
data _null_;  
  set myfiles.cars;
  file outfile encoding="utf-8";
  put Make Model Year;
run;
```

Example 9: Using the FTP Access Method to Write Data to an Excel Spreadsheet

This example uses the FTP access method and the FILEVAR option to write data to several Microsoft Excel worksheets.
FILENAME Statement

 Associates a SAS fileref with an external file or an output device, disassociates a fileref and external file, or lists attributes of external files.

Note: The FILENAME Statement has moved to SAS Global Statements.
FILENAME Statement: Azure Access Method
Enables you to access data in Microsoft Azure Data Lake Storage.

Note: The FILENAME Statement: Azure Access Method has moved to SAS Global Statements.

FILENAME Statement: CATALOG Access Method
Enables you to reference a SAS catalog as an external file.

Note: The FILENAME Statement: CATALOG Access Method has moved to SAS Global Statements.

FILENAME Statement: CLIPBOARD Access Method
Enables you to read text data from and write text data to the clipboard on the host computer.

Note: The FILENAME Statement: CLIPBOARD Access Method has moved to SAS Global Statements.

FILENAME Statement: DATAURL Access Method
Enables you to read data from user-specified text.

Note: The FILENAME Statement: DATAURL Access Method has moved to SAS Global Statements.

FILENAME Statement: EMAIL (SMTP) Access Method
Enables you to send electronic mail programmatically from SAS using the SMTP (Simple Mail Transfer Protocol) email interface.

Note: The FILENAME Statement: EMAIL (SMTP) Access Method has moved to SAS Global Statements.

FILENAME Statement: FILESRCV Access Method
Enables you to store and retrieve files within the file service in the SAS Viya system.

Note: The FILENAME Statement: FILESRCV Access Method has moved to SAS Global Statements.
FILENAME Statement: FTP Access Method

Enables you to access remote files by using the FTP protocol.

Note: The FILENAME Statement: FTP Access Method has moved to SAS Global Statements.

FILENAME Statement: Hadoop Access Method

Enables you to access files on a Hadoop Distributed File System (HDFS) whose location is specified in a configuration file.

Note: The FILENAME Statement: Hadoop Access Method has moved to SAS Global Statements.

FILENAME Statement: S3 Access Method

Enables you to access Amazon S3 files.

Note: The FILENAME Statement: S3 Access Method has moved to SAS Global Statements.

FILENAME Statement: SFTP Access Method

Enables you to access remote files by using the SFTP protocol.

Note: The FILENAME Statement: SFTP Access Method has moved to SAS Global Statements.

FILENAME Statement: SOCKET Access Method

Enables you to read from or write to a TCP/IP socket.

Note: The FILENAME Statement: SOCKET Access Method has moved to SAS Global Statements.

FILENAME Statement: URL Access Method

Enables you to access remote files by using the URL access method.

Note: The FILENAME Statement: URL Access Method has moved to SAS Global Statements.
FILENAME Statement: WebDAV Access Method

Enables you to access remote files by using the WebDAV protocol.

Note: The FILENAME Statement: WebDAV Access Method has moved to SAS Global Statements.

FILENAME Statement: ZIP Access Method

Enables you to access ZIP files.

Note: The FILENAME Statement: ZIP Access Method has moved to SAS Global Statements.

FOOTNOTE Statement

Writes up to 10 lines of text at the bottom of the procedure or DATA step output.

Note: The FOOTNOTE Statement has moved to SAS Global Statements.

FORMAT Statement

Associates formats with variables.

Valid in: DATA step or PROC step
Categories: CAS
          Information
Type: Declarative

Syntax

FORMAT variable-1 <…variable-n> <format> <DEFAULT=default-format>;
FORMAT variable-1 <…variable-n> format <DEFAULT=default-format>;
FORMAT variable-1 <…variable-n> format variable-1 <…variable-n> format;

Arguments

variable
  names one or more variables for SAS to associate with a format. You must specify at least one variable.

Tip
  To disassociate a format from a variable, use the variable in a FORMAT statement without specifying a format in a DATA step or in PROC DATASETS. In a DATA step, place this FORMAT statement after the SET
FORMAT Statement

statement. See "Example 3: Removing a Format" on page 103. You can also use PROC DATASETS.

**format**  
specifies the format that is listed for writing the values of the variables.

**Tip**  
Formats that are associated with variables by using a FORMAT statement behave like formats that are used with a colon modifier in a subsequent PUT statement. For more information about using a colon modifier, see "PUT Statement: List" on page 262.

**See**  
*SAS Formats and Informats: Reference*

**DEFAULT=default-format**  
specifies a temporary default format for displaying the values of variables that are not listed in the FORMAT statement. These default formats apply only to the current DATA step; they are not permanently associated with variables in the output data set.

A DEFAULT= format specification applies to

- variables that are not named in a FORMAT or ATTRIB statement
- variables that are not permanently associated with a format within a SAS data set
- variables that are not written with the explicit use of a format.

**Default**  
If you omit DEFAULT=, SAS uses BESTw. as the default numeric format and $w. as the default character format.

**Restriction**  
Use this option only in a DATA step.

**Tip**  
A DEFAULT= specification can occur anywhere in a FORMAT statement. It can specify either a numeric default, a character default, or both.

**Example**  
"Example 1: Assigning Formats and Defaults" on page 102

**Details**

The FORMAT statement can use standard SAS formats or user-written formats that have been previously defined in PROC FORMAT. A single FORMAT statement can associate the same format with several variables, or it can associate different formats with different variables. If a variable appears in multiple FORMAT statements, SAS uses the format that is assigned last.

You use a FORMAT statement in the DATA step to permanently associate a format with a variable. SAS changes the descriptor information of the SAS data set that contains the variable. You can use a FORMAT statement in some PROC steps, but the rules are different. For more information, see *Base SAS Procedures Guide*.

**Comparisons**

Both the ATTRIB and FORMAT statements can associate formats with variables, and both statements can change the format that is associated with a variable. You can use the FORMAT statement in PROC DATASETS to change or remove the format that is associated with a variable. You can also associate, change, or
disassociate formats and variables in existing SAS data sets through the windowing environment.

Examples:

Example 1: Assigning Formats and Defaults

This example uses a FORMAT statement to assign formats and default formats for numeric and character variables. The default formats are not associated with variables in the data set but affect how the PUT statement writes the variables in the current DATA step.

```sas
data tstfmt;
  format W $char3.
  Y 10.3
  default=8.2 $char8.;
  W='Good morning.';
  X=12.1;
  Y=13.2;
  Z='Howdy-doody';
  put W/X/Y/Z;
run;
proc contents data=tstfmt;
run;
proc print data=tstfmt;
run;
```

This output shows a partial listing from PROC CONTENTS, as well as the report that PROC PRINT generates.

<table>
<thead>
<tr>
<th>Alphabetic List of Variables and Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>#</td>
</tr>
<tr>
<td>----</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
</tbody>
</table>

Output 2.4  PROC PRINT Report

<table>
<thead>
<tr>
<th>Obs</th>
<th>W</th>
<th>Y</th>
<th>X</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Goo</td>
<td>13.200</td>
<td>12.1</td>
<td>Howdy-doody</td>
</tr>
</tbody>
</table>

The default formats apply to variables X and Z while the assigned formats apply to the variables W and Y.
The PUT statement produces this result:

```
----+----1----+----2
Goo
12.10
13.200
Howdy-do
```

Example 2: Associating Multiple Variables with a Single Format

This example uses the FORMAT statement to assign a single format to multiple variables.

```sas
data report;
  input Item $ 1–6 Material $ 8–14 Investment 16–22 Profit 24–31;
  format Item Material $upcase9. Investment Profit dollar15.2;
  datalines;
  shirts cotton 2256354 83952175
  ties silk 498678 2349615
  suits silk 9482146 69839563
  belts leather 7693 14893
  shoes leather 7936712 22964
;
run;
```

```
Output 2.5 Results from Associating Multiple Variables with a Single Format
```

```
<table>
<thead>
<tr>
<th>Obs</th>
<th>Item</th>
<th>Material</th>
<th>Investment</th>
<th>Profit</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SHIRTS</td>
<td>COTTON</td>
<td>$2,256,354.00</td>
<td>$83,952,175.00</td>
</tr>
<tr>
<td>2</td>
<td>TIES</td>
<td>SILK</td>
<td>$498,678.00</td>
<td>$2,349,615.00</td>
</tr>
<tr>
<td>3</td>
<td>SUITS</td>
<td>SILK</td>
<td>$9,462,146.00</td>
<td>$69,839,563.00</td>
</tr>
<tr>
<td>4</td>
<td>BELTS</td>
<td>LEATHER</td>
<td>$7,693.00</td>
<td>$14,893.00</td>
</tr>
<tr>
<td>5</td>
<td>SHOES</td>
<td>LEATHER</td>
<td>$7,936,712.00</td>
<td>$22,964.00</td>
</tr>
</tbody>
</table>
```

Example 3: Removing a Format

This example disassociates an existing format from a variable in a SAS data set. The order of the FORMAT statement and the SET statements is important.

```sas
data rtest;
  set rtest;
  format x;
run;
```
GO TO Statement

Directs program execution immediately to the statement label that is specified and, if followed by a RETURN statement, returns execution to the beginning of the DATA step.

<table>
<thead>
<tr>
<th>Valid in</th>
<th>DATA step</th>
</tr>
</thead>
<tbody>
<tr>
<td>Categories</td>
<td>CAS</td>
</tr>
<tr>
<td></td>
<td>Control</td>
</tr>
<tr>
<td>Type</td>
<td>Executable</td>
</tr>
<tr>
<td>Alias</td>
<td>GOTO</td>
</tr>
</tbody>
</table>

Syntax

GO TO label;

Arguments

label

specifies a statement label that identifies the GO TO destination. The destination must be within the same DATA step. You must specify the label argument.

Comparisons

The GO TO statement and the LINK statement are similar. However, a GO TO statement is often used without a RETURN statement, whereas a LINK statement is usually used with an explicit RETURN statement. The action of a subsequent RETURN statement differs between the GO TO and LINK statements. A RETURN statement after a LINK statement returns execution to the statement that follows the LINK statement. A RETURN after a GO TO statement returns execution to the beginning of the DATA step (unless a LINK statement precedes the GO TO statement). In that case, execution continues with the first statement after the LINK statement.

GO TO statements can often be replaced by DO-END and IF-THEN/ELSE programming logic.

Example: Using a RETURN Statement with the GO TO Statement

Use the GO TO statement as shown here.

- In this example, if the condition is true, the GO TO statement instructs SAS to jump to a label called ADD and to continue execution from there. If the condition
is false, SAS executes the PUT statement and the statement that is associated with the GO TO label.

```sas
data info;
  input x;
  if 1<=x<=5 then go to add;
  put x=;
  add: sumx+x;
  datalines;
  7
  6
  323
;
```

Because every DATA step contains an implied RETURN at the end of the step, program execution returns to the top of the step after the sum statement is executed. Therefore, an explicit RETURN statement at the bottom of the DATA step is not necessary.

- If you do not want the Sum statement to execute for observations that do not meet the condition, rewrite the code and include an explicit return statement.

```sas
data info;
  input x;
  if 1<=x<=5 then go to add;
  put x=;
  return;
  /* SUM statement not executed */
  /* if x<1 or x>5 */
  add: sumx+x;
  datalines;
  7
  6
  323
;
```

See Also

Statements:

- “DO Statement” on page 62
- “Label: Statement” on page 187
- “LINK Statement” on page 196
- “RETURN Statement” on page 284

---

**IF Statement: Subsetting**

Continues processing only those observations that meet the condition of the specified expression.

Valid in: DATA step

Categories: Action

CAS
Using a random number function in a WHERE statement might generate a different result set from using a random number function in a subsetting IF statement. This difference can be caused by how the criteria are optimized internally by SAS and is expected behavior.

**Syntax**

**IF expression;**

**Arguments**

expression

is any SAS expression.

**Details**

**The Basics**

The subsetting IF statement causes the DATA step to continue processing only those raw data records or those observations from a SAS data set that meet the condition of the expression that is specified in the IF statement. That is, if the expression is true for the observation or record (its value is neither 0 nor missing), SAS continues to execute statements in the DATA step and includes the current observation in the data set. The resulting SAS data set or data sets contain a subset of the original external file or SAS data set.

If the expression is false (its value is 0 or missing), no further statements are processed for that observation or record, the current observation is not written to the data set, and the remaining program statements in the DATA step are not executed. SAS immediately returns to the beginning of the DATA step because the subsetting IF statement does not require additional statements to stop processing observations.

**Using the Equivalent of the CONTAINS and LIKE Operators in an IF Statement**

The LIKE operator in a WHERE clause matches patterns in words. To get the equivalent result in an IF statement, the ‘=:' operator can be used. This matches patterns that occur at the beginning of a string. Here is an example.

```sas
data test;
  input name $;
  datalines;
John
Diana
Diane
Sally
Doug
David
;  
run;

data test;
  set test;
```
The CONTAINS operator in a WHERE clause checks for a character string within a value. To get the equivalent result in an IF statement, the INDEX function can be used. For example:

```sas
data test;
  set test;
  if index(name,'ian') ge 1;
run;
```

```sas
proc print;
run;
```

**Comparisons**

- The subsetting IF statement is equivalent to this IF-THEN statement:

  ```sas
  if not (expression)
  then delete;
  ```

- When you create SAS data sets, use the subsetting IF statement when it is easier to specify a condition for including observations. When it is easier to specify a condition for excluding observations, use the DELETE statement.

- The subsetting IF and the WHERE statements are not equivalent. The two statements work differently and produce different output data sets in some cases. The most important differences are summarized as follows:
  - The subsetting IF statement selects observations that have been read into the program data vector. The WHERE statement selects observations before they are brought into the program data vector. The subsetting IF might be less efficient than the WHERE statement because it must read each observation from the input data set into the program data vector.
  - The subsetting IF statement and WHERE statement can produce different results in DATA steps that interleave, merge, or update SAS data sets.
  - When the subsetting IF statement is used with the MERGE statement, SAS selects observations after the current observations are combined. When the WHERE statement is used with the MERGE statement, SAS applies the selection criteria to each input data set before combining the current observations.
  - The subsetting IF statement can select observations from an existing SAS data set or from raw data that are read with the INPUT statement. The WHERE statement can select observations only from existing SAS data sets.
  - The subsetting IF statement is executable; the WHERE statement is not.

**Example: Limiting Observations**

- This example results in a data set that contains only those observations with the value 'F' for the variable SEX:

  ```sas
  if sex='F';
  ```
This example results in a data set that contains all observations for which the value of the variable AGE is not missing or 0:

```sas
if age;
```

See Also

- "WHERE-Expression Processing" in *SAS Language Reference: Concepts*

Data Set Options:

- "WHERE= Data Set Option" in *SAS Data Set Options: Reference*

Statements:

- "DELETE Statement" on page 58
- "IF-THEN/ELSE Statement" on page 108
- "WHERE Statement" on page 315

---

**IF-THEN/ELSE Statement**

Executes a SAS statement for observations that meet specific conditions.

**Syntax**

```sas
IF expression THEN statement;
<ELSE statement;>
```

**Arguments**

- `expression` is any SAS expression and is a required argument.
- `statement` can be any executable SAS statement or DO group.

**Details**

SAS evaluates the expression in an IF-THEN statement to produce a result that is either nonzero, zero, or missing. A nonzero and nonmissing result causes the expression to be true; a result of zero or missing causes the expression to be false.

If the conditions that are specified in the IF clause are met, the IF-THEN statement executes a SAS statement for observations that are read from a SAS data set, for records in an external file, or for computed values. An optional ELSE statement
IF-THEN/ELSE Statement

IF-THEN/ELSE Statement

gives an alternative action if the THEN clause is not executed. The ELSE statement, if used, must immediately follow the IF-THEN statement.

Using IF-THEN statements without the ELSE statement causes SAS to evaluate all IF-THEN statements. Using IF-THEN statements with the ELSE statement causes SAS to execute IF-THEN statements until it encounters the first true statement. Subsequent IF-THEN statements are not evaluated.

Note: For greater efficiency, construct your IF-THEN/ELSE statement with conditions of decreasing probability.

Comparisons

- Use a SELECT group rather than a series of IF-THEN statements when you have a long series of mutually exclusive conditions.
- Use subsetting IF statements, without a THEN clause, to continue processing only those observations or records that meet the condition that is specified in the IF clause.

Example: Different Ways of Specifying the IF-THEN/ELSE Statements

These examples show different ways of specifying the IF-THEN/ELSE statement.

- if x then delete;
- if status='OK' and type=3 then count+1;
- if age ne agecheck then delete;
- if x=0 then
  if y ne 0 then put 'X ZERO, Y NONZERO';
  else put 'X ZERO, Y ZERO';
  else put 'X NONZERO';
- if answer=9 then
  do;
    answer=.;
    put 'INVALID ANSWER FOR ' id=;
  end;
  else
    do;
      answer=answer10;
      valid+1;
    end;
- data region;
  input city $ 1-30;
  if city='New York City' or city='Miami' then
    region='ATLANTIC COAST';
  else if city='San Francisco' or city='Los Angeles' then
    region='PACIFIC COAST';
datalines;
...more data lines...
%INCLUDE Statement

Brings a SAS programming statement, data lines, or both, into a current SAS program.

Note: The %INCLUDE Statement has moved to SAS Global Statements.

INFILE Statement

Specifies an external file to read with an INPUT statement.

Valid in: DATA Step
Category: File-Handling
Type: Executable
Restrictions: This statement is not supported in a DATA step that runs in CAS. When SAS is in a locked-down state, the INFILE statement is not available for files that are not in the locked-down path list. For more information, see “SAS Processing Restrictions for Servers in a Locked-Down State” in SAS Language Reference: Concepts.

Operating environment: The INFILE statement contains operating-environment-specific material. See the SAS documentation for your operating environment before using this statement.

See: INFILE Statement under Windows, UNIX, and z/OS

Syntax

INFILE file-specification <device-type> <options> <operating-environment-options>;
INFILE DBMS-specifications;

Arguments

file-specification

Identifies the source of the input data records, which is an external file or instream data. file-specification can have these forms:

'external-file'

Specifies the physical name of an external file. The physical name is the name that the operating environment uses to access the file.

fileref

Specifies the fileref of an external file.
Range: 1 to 8 bytes

Requirement: You must have previously associated the fileref with an external file in a FILENAME statement, FILENAME function, or an appropriate operating environment command.

See: “FILENAME Statement” on page 97

fileref(file)
specifies a fileref of an aggregate storage location and the name of a file or member, enclosed in parentheses, that resides in that location.

Requirements:
- A file that is located in an aggregate storage location and has a name that is not a valid SAS name must have its name enclosed in quotation marks.
- You must have previously associated the fileref with an external file in a FILENAME statement, a FILENAME function, or an appropriate operating environment command.

Operating environment:
Different operating environments call an aggregate grouping of files by different names such as a directory, a MACLIB, or a partitioned data set. For more information about how to specify external files, see the SAS documentation for your operating environment.

See: “FILENAME Statement” on page 97

CARDS
CARDS4
for a definition, see DATALINES on page 111.

Alias: DATALINES | DATALINES4

DATALINES
DATALINES4
specifies that the input data immediately follow the DATALINES statement or DATALINES4 statement in the DATA step. Using DATALINES enables you to use the INFILE statement options to control how the INPUT statement reads instream data lines.

Alias: CARDS | CARDS4

Example: “Example 1: Changing How Delimiters Are Treated” on page 132

Tip: You can verify the existence of file-specification by using the SYSERR macro variable if the ERRORCHECK option is set to STRICT.

device-type
specifies the type of device or the access method that is used if the fileref points to an input or output device or location that is not a physical file.

ACTIVEMQ
specifies an access method that enables you to access an ActiveMQ messaging broker.
If the DATA step does not recognize the access method option, the DATA step passes the option to the access method for handling.

**See**
“FILENAME Statement: ACTIVEMQ Access Method” in Application Messaging with SAS

**CATALOG**
specifies the CATALOG access method.

If the DATA step does not recognize the access method option, the DATA step passes the option to the access method for handling.

**See**
For a complete list of options that are available with the CATALOG access method, see “FILENAME Statement: CATALOG Access Method” on page 98.

**CLIPBOARD**
specifies the CLIPBOARD access method.

If the DATA step does not recognize the access method option, the DATA step passes the option to the access method for handling.

**See**
For a complete list of options that are available with the CLIPBOARD access method, see “FILENAME Statement: CLIPBOARD Access Method” on page 98.

**DISK**
specifies that the device is a disk drive.

**Tip**
When you assign a fileref to a file on disk, you are not required to specify DISK.

**DUMMY**
specifies that the input from the external file be discarded.

**Interaction**
The DUMMY option indicates that an immediate end of file is encountered with a corresponding INPUT statement. If the FILEVAR= option is specified along with DUMMY, the file that is specified by the FILEVAR= variable is not read.

**Tip**
Specifying DUMMY can be useful for testing.

**FTP**
specifies the FTP access method.

If the DATA step does not recognize the access method option, the DATA step passes the option to the access method for handling.

**See**
For a complete list of options that are available with the FTP access method, see “FILENAME Statement: FTP Access Method” on page 99.

**Example**
infile dummy ftp user='myuid' pass='xxxx' filevar=file_to_read;
GTERM indicates that the output device type is a graphics device that receives graphics data.

HADOOP specifies the Hadoop access method.

Interaction If the DATA step does not recognize the access method option, the DATA step passes the option to the access method for handling.

See For a complete list of options that are available with the Hadoop access method, see “FILENAME Statement: Hadoop Access Method” on page 99.

JMS specifies a Java Message Service (JMS) destination.

PIPE specifies an unnamed pipe.

Note Some operating environments do not support pipes.

PLOTTER specifies an unbuffered graphics output device.

PRINTER specifies a printer or printer spool file.

SFTP specifies the SFTP access method.

Interaction If the DATA step does not recognize the access method option, the DATA step passes the option to the access method for handling.

See For a complete list of options that are available with the SFTP access method, see “FILENAME Statement: SFTP Access Method” on page 99.

SOCKET specifies the SOCKET access method.

Interaction If the DATA step does not recognize the access method option, the DATA step passes the option to the access method for handling.

See For a complete list of options that are available with the SOCKET access method, see “FILENAME Statement: SOCKET Access Method” on page 99.

TAPE specifies a tape drive.

TERMINAL specifies the user’s terminal.

UPRINTER specifies a Universal Printer definition name.
Tip If you do not specify the printer name in the FILENAME statement, the PRINTERPATH options control which Universal Printer is used and the destination of the output.

**URL**

specifies the URL access method.

**Interaction** If the DATA step does not recognize the access method option, the DATA step passes the option to the access method for handling.

**See** “STATUS=variable” on page 123. For a complete list of options that are available with the URL access method, see “FILENAME Statement: URL Access Method” on page 99.

**WEBDAV**

specifies the WEBDAV access method.

**Interaction** If the DATA step does not recognize the access method option, the DATA step passes the option to the access method for handling.

**See** For a complete list of options that are available with the WEBDAV access method, see “FILENAME Statement: WebDAV Access Method” on page 100.

### Alias

DEVICE= *device-type*

### Default

DISK

### Requirement

device-type or DEVICE= *device-type* must immediately follow file-specification in the statement.

### Operating environment

Additional specifications might be required when you specify some devices. See the SAS documentation for your operating environment before specifying a value other than DISK. Values in addition to the ones that are listed here might be available in some operating environments.

**INFILE Options**

**BLKSIZE= ** *block-size*

specifies the block size of the input file.

**Default** Dependent on the operating environment. For more information, see the SAS documentation for your operating environment.

**COLUMN= ** *variable*

names a variable that SAS uses to assign the current column location of the input pointer. As with automatic variables, the COLUMN= variable is not written to the data set.

**Alias** COL=

**See** LINE= on page 119

**Example** “Example 8: Listing the Pointer Location” on page 137
**DELMITER=** *delimiter(s)*
specifies an alternate delimiter (other than a blank) to be used for LIST input, where *delimiter(s)* can be one of these items:

*list-of-delimiting-characters*
specifies one or more characters to read as delimiters.

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Enclose the list of characters in quotation marks.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Example</td>
<td>“Example 1: Changing How Delimiters Are Treated” on page 132</td>
</tr>
</tbody>
</table>

*character-variable*
specifies a character variable whose value becomes the delimiter.

<table>
<thead>
<tr>
<th>Alias</th>
<th>DLM=</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default</td>
<td>Blank space</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tips</th>
<th>The delimiter is case sensitive.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Some common delimiters are the comma (,), verticle pipe (</td>
</tr>
<tr>
<td>See</td>
<td>“Reading Delimited Data” on page 127, DLMSTR= on page 115, and “DSD (delimiter-sensitive data)” on page 116</td>
</tr>
<tr>
<td>Example</td>
<td>“Example 1: Changing How Delimiters Are Treated” on page 132</td>
</tr>
</tbody>
</table>

**DLMSTR=** *delimiter*
specifies a character string as an alternate delimiter (other than a blank) to be used for LIST input, where *delimiter* can be one of these items:

*delimiting-string*
specifies a character string to read as a delimiter.

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Enclose the string in quotation marks.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Example</td>
<td>“Example 1: Changing How Delimiters Are Treated” on page 132</td>
</tr>
</tbody>
</table>

*character-variable*
specifies a character variable whose value becomes the delimiter.

<table>
<thead>
<tr>
<th>Default</th>
<th>Blank space</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Interactions</th>
<th>If you specify more than one DLMSTR= option in the INFILE statement, the DLMSTR= option that is specified last is used. If you specify both the DELIMITER= and DLMSTR= options, the option that is specified last is used.</th>
</tr>
</thead>
<tbody>
<tr>
<td>If you specify RECFM=N, ensure that the LRECL is large enough to hold the largest input item. Otherwise, the delimiter can be split across the record boundary.</td>
<td></td>
</tr>
<tr>
<td>Tip</td>
<td>The delimiter is case sensitive. To make the delimiter case insensitive, use the DLMSOPT='I' option.</td>
</tr>
</tbody>
</table>
See | “Reading Delimited Data” on page 127, DELIMITER= on page 115, DLMSOPT= on page 116, and DSD on page 116
---|---
Example | “Example 1: Changing How Delimiters Are Treated” on page 132

**DLMSOPT= 'options'**
specifies parsing options for the DLMSTR= option, where options can be one of these values:

| I | specifies that case-insensitive comparisons are done. |
| T | specifies that trailing blanks of the string delimiter be removed. |

**Tips**
The T option is useful when you use a variable as the delimiter string.

You can specify either I, T, or both.

**Requirement**
The DLMSOPT= option has an effect only when used with the DLMSTR= option.

---

See | DLMSTR= on page 115
---|---
Example | “Example 1: Changing How Delimiters Are Treated” on page 132

**DSD (delimiter-sensitive data)**
specifies that when data values are enclosed in quotation marks, delimiters within the value are treated as character data. The DSD option changes how SAS treats delimiters when you use LIST input and sets the default delimiter to a comma. When you specify the DSD option, SAS treats two consecutive delimiters as a missing value and removes quotation marks from character values.

**Interaction**
Use the DELIMITER= option or DLMSTR= option to change the delimiter.

**Tip**
Use the DSD option and LIST input to read a character value that contains a delimiter within a string that is enclosed in quotation marks. The INPUT statement treats the delimiter as a valid character and removes the quotation marks from the character string before the value is stored. Use the tilde (~) format modifier to retain the quotation marks.

---

See | “Reading Delimited Data” on page 127, DELIMITER= on page 115, and DLMSTR= on page 115
---|---
Examples | “Example 1: Changing How Delimiters Are Treated” on page 132

**ENCODING= 'encoding-value'**
specifies the encoding to use when reading from the external file. The value for ENCODING= indicates that the external file has a different encoding from the current session encoding.

When you read data from an external file, SAS transcodes the data from the specified encoding to the session encoding.
SAS assumes that an external file is in the same encoding as the session encoding.


“Example 11: Specifying an Encoding When Reading an External File” on page 140

END=variable

specifies a variable that SAS sets to 1 when the current input data record is the last in the input file. Until SAS processes the last data record, the END= variable is set to 0. As with automatic variables, this variable is not written to the data set.

Restriction

You cannot use the END= option with the UNBUFFERED option, the DATALINES statement, the DATALINES4 statement, or an INPUT statement that reads multiple input data records.

Tip

Use the option EOF= on page 117 when END= is invalid.

“Example 5: Reading from Multiple Input Files” on page 136

EOF=label

specifies a statement label that is the object of an implicit GO TO when the INFILE statement reaches end of file. When an INPUT statement attempts to read from a file that has no more records, SAS moves execution to the statement label indicated.

Interaction

Use EOF= instead of the END= option with the UNBUFFERED option, the DATALINES statement, the DATALINES4 statement, or an INPUT statement that reads multiple input data records.

Tip

The EOF= option is useful when you read from multiple input files sequentially.

See

END= on page 117, EOV= on page 117, and UNBUFFERED on page 124

EOV=variable

specifies a variable that SAS sets to 1 when the first record in a file in a series of concatenated files is read. The variable is set only after SAS encounters the next file. As with automatic variables, the EOV= variable is not written to the data set.

Tip

Reset the EOV= variable to 0 after SAS encounters each boundary.

See

END= on page 117 and EOF= on page 117

EXPANDTABS | NOEXPANDTABS

specifies whether to expand tab characters to the standard tab setting, which is set at 8-column intervals that start at column 9.

Default

NOEXPANDTABS

Tip

EXPANDTABS is useful when you read data that contains the tab character that is native to your operating environment.
**FILENAME=**variable

specifies a variable that SAS sets to the physical name of the currently opened input file. In a series of concatenated files, the variable is updated only after SAS encounters the next file. As with automatic variables, the FILENAME= variable is not written to the data set.

**Tip**
Use a LENGTH statement to make the variable length long enough to contain the value of the file name.

**See**
FILEVAR= on page 118

**Example**
“Example 5: Reading from Multiple Input Files” on page 136

---

**FILEVAR=**variable

specifies a variable whose change in value causes the INFILE statement to close the current input file and open a new one. When the next INPUT statement executes, it reads from the new file that the FILEVAR= variable specifies. As with automatic variables, this variable is not written to the data set.

**Restriction**
The FILEVAR= variable must contain a character string that is a physical file name.

**Interaction**
When you use the FILEVAR= option, the file-specification is a placeholder, not an actual file name or a fileref that has been previously assigned to a file. SAS uses this placeholder for reporting processing information to the SAS log. The placeholder must conform to the same rules as a fileref.

**Tips**
Use FILEVAR= to dynamically change the currently opened input file to a new physical file.

When using FILEVAR=, it is not possible to know whether the input file that is currently open is the last file. When the DATA step comes to an end-of-file marker or the end of all open data sets, it performs an orderly shutdown. In addition, if you use FILEVAR with FIRSTOBS, a file with only a header record in a series of files triggers a normal shutdown of the DATA step. The shutdown occurs because SAS reads beyond the end-of-file marker and the DATA step terminates. You can use the EOF= option to avoid the shutdown.

**See**
“Updating External Files in Place” on page 126

**Example**
“Example 5: Reading from Multiple Input Files” on page 136

---

**FIRSTOBS=**record-number

specifies a record number that SAS uses to begin reading input data records in the input file.

**Default**
1

**Tips**
Use FIRSTOBS= with OBS= to read a range of records from the middle of a file.

Use FIRSTOBS=2 to skip a header record in a file.

**Example**
This statement processes record 50 through record 100.

```sas
infile file-specification firstobs=50 obs=100;
```
FLOWOVER
causes an INPUT statement to continue to read the next input data record if it
does not find values in the current input line for all the variables in the statement.
FLOWOVER is the default behavior of the INPUT statement.

See “Reading Past the End of a Line” on page 130, MISSOVER on page 120,
STOPOVER on page 123, and TRUNCOVER on page 124

LENGTH=variable
specifies a variable that SAS sets to the length of the current input line. SAS
does not assign the variable a value until an INPUT statement executes. As with
automatic variables, the LENGTH= variable is not written to the data set.

Tip This option, in conjunction with the $VARYING informat, is useful
when the field width varies.

Examples “Example 4: Reading Files That Contain Variable-Length Records”
on page 135

“Example 7: Truncating Copied Records” on page 137

LINE=variable
specifies a variable that SAS sets to the line location of the input pointer in the
input buffer. As with automatic variables, the LINE= variable is not written to the
data set.

Range 1 to the value of the N= option

Interaction The value of the LINE= variable is the current relative line number
within the group of lines that is specified by the N= option or by the
#n line pointer control in the INPUT statement.

See COLUMN= on page 114 and N= on page 121

Example “Example 8: Listing the Pointer Location” on page 137

LINESIZE=line-size
specifies the record length that is available to the INPUT statement.

Alias LS=

Range up to 32767

Interaction If an INPUT statement attempts to read past the column that is
specified by the LINESIZE= option, the action that is taken
depends on whether the FLOWOVER, MISSOVER,
SCANOVER, STOPOVER, or TRUNCOVER option is in effect.
FLOWOVER is the default.

Operating environment Values for line-size are dependent on the operating environment
record size. For more information, see the SAS documentation
for your operating environment.

Tip Use LINESIZE= to limit the record length when you do not want
to read the entire record.

Example If your data lines contain a sequence number in columns 73
through 80, use this INFILE statement to restrict the INPUT
statement to the first 72 columns:
infile file-specification linesize=72;

| **LRECL=**<br>**logical-record-length** | specifies the logical record length. |  |
| Default | Dependent on the file characteristics of your operating environment |  |
| Restriction | LRECL is not valid when you use the DATALINES file specification. |  |
| Interaction | Alternatively, you can specify a global logical record length by using the “LRECL= System Option” in SAS System Options: Reference. The default value for the global LRECL system option is 32767. If you are using fixed-length records (RECFM=F), the default value for LRECL is 256. |  |
| Operating environment | Values for logical-record-length are dependent on the operating environment. For more information, see the SAS documentation for your operating environment. |  |
| Tip | LRECL= specifies the physical line length of the file. LINESIZE= tells the INPUT statement how much of the line to read. |  |

**MISSOVER**

prevents an INPUT statement from reading a new input data record if it does not find values in the current input line for all the variables in the statement. When an INPUT statement reaches the end of the current input data record, variables without any values assigned are set to missing.

| Tip | Use MISSOVER if the last field or fields might be missing and you want SAS to assign missing values to the corresponding variable. |  |
| See | “Reading Past the End of a Line” on page 130, FLOWOVER on page 119, SCANOVER on page 122, STOPOVER on page 123, and TRUNCOVER on page 124 |  |
| Example | “Example 2: Handling Missing Values and Short Records with List Input” on page 134 |  |

**MEMVAR=**<br>**variable**

enables you to open the individual files that are contained in a directory or contained in a directory-based file such as a ZIP file. When a directory fileref or physical name is provided in the INFILE statement, the file that is referred to in the MEMVAR= variable is opened. When the value of the MEMVAR= variable changes, the current file is closed and the new file is opened. To read all members of a directory, set the MEMVAR= variable to blanks.

| Restriction | The value of a MEMVAR= variable must contain a character string that contains a physical file name or the value must contain only blanks. When the value of the MEMVAR= variable is set to blanks, the INFILE statement closes the current file, retrieves a member name from the directory, places this name into the MEMVAR= variable, and opens the new file. |  |
| Tips | This variable, like automatic variables, is not written to the data set. |  |
If any of the physical file names is longer than 8 bytes (the default length of a character variable), assign the MEMVAR= variable a longer length with another statement such as a LENGTH statement or an INPUT statement.

See
FILENAME= on page 84

Examples
“Example 12: Reading All File Members of a Directory” on page 141
“Example 13: Reading a List of File Members” on page 141

N=available-lines
specifies the number of lines that are available to the input pointer at one time.

Default
The highest value that follows a # pointer control in any INPUT statement in the DATA step. If you omit a # pointer control, the default value is 1.

Interaction
This option affects only the number of lines that the pointer can access at a time; it has no effect on the number of lines an INPUT statement reads.

Tips
When you use # pointer controls in an INPUT statement that are less than the value of N=, you might get unexpected results. To prevent unexpected results, include a # pointer control that equals the value of the N= option. For example:

```
infile 'external file' n=5;
input #2 name : $25. #3 job : $25. #5;
```

The INPUT statement includes a #5 pointer control, even though no data is read from that record.

Example
“Example 8: Listing the Pointer Location” on page 137

NBYTE=variable
specifies the name of a variable that contains the number of bytes to read from a file when you are reading data in stream record format (RECFM=S in the FILENAME statement).

Default
The LRECL value of the file

Interaction
If the number of bytes to read is set to –1, the FTP and SOCKET access methods return the number of bytes that are currently available in the input buffer.

See
The RECFM= option in the FILENAME statement, SOCKET access method, and the RECFM= option in the FILENAME statement, FTP access method

OBS=record-number | MAX

record-number specifies the record number of the last record to read in an input file that is read sequentially.

MAX specifies the maximum number of observations to process, which is at least as large as the largest signed, 32-bit integer. The absolute maximum depends on your host operating environment.
**Default** MAX

**Tip** Use OBS= with FIRSTOBS= to read a range of records from the middle of a file.

**Example** This statement processes only the first 100 records in the file:

```
infile file-specification obs=100;
```

**PAD | NOPAD**
controls whether SAS pads the records that are read from an external file with blanks to the length that is specified in the LRECL= option.

**Default** NOPAD

**See** LRECL= option on page 120

**PRINT | NOPRINT** specifies whether the input file contains carriage-control characters.

**Tip** To read a file in a DATA step without having to remove the carriage-control characters, specify PRINT. To read the carriage-control characters as data values, specify NOPRINT.

**RECFM=record-format** specifies the record format of the input file.

**Interaction** In SAS 9.4, the default value for the global LRECL system option is 32767. If you are using fixed-length records (RECFM=F), the default value for LRECL is 256.

**Operating environment** Values for record-format are dependent on the operating environment. For more information, see the SAS documentation for your operating environment.

**RESET=variable** specifies whether to close the current input file and open a new file, or close and reopen the current input file. When the variable value is 0, the current file is closed and a new file is opened when the value of the FILEVAR= variable changes. When the variable value is 1, the current file is closed and the file that is referred to by the FILEVAR= variable is reopened.

**Default** The current file is closed and a new file is opened when the value of the FILEVAR= variable changes.

**SCANOVER** causes the INPUT statement to scan the input data records until the character string that is specified in the @'character-string' expression is found.

**Interaction** The MISSOVER, TRUNCOVER, and STOPOVER options change how the INPUT statement behaves when it scans for the @'character-string' expression and reaches the end of the record. By default (FLOWOVER option), the INPUT statement scans the next record while these other options cause scanning to stop.

**Tip** It is redundant to specify both SCANOVER and FLOWOVER.
SHAREBUFFERS

specifies that the FILE statement and the INFILE statement share the same buffer.

Alias

SHAREBUFS

Tips

Use SHAREBUFFERS with the INFILE, FILE, and PUT statements to update an external file in place. Updating an external file in place saves CPU time because the PUT statement output is written straight from the input buffer instead of the output buffer.

Use SHAREBUFFERS to update specific fields in an external file instead of an entire record.

Example

“Example 6: Updating an External File” on page 137

CAUTION

When using SHAREBUFFERS, RECFM=V, and _INFILE_, use caution if you read a record with one length and update the file with a record of a different length. The length of the record can change by modifying _INFILE_. One option to avoid this potential problem is to pad or truncate _INFILE_ so that the original record length is maintained.

START=variable

specifies a variable whose value SAS uses as the first column number of the record that the PUT _INFILE_ statement writes. As with automatic variables, the START variable is not written to the data set.

See _INFILE_ option on page 238 in the PUT statement

STATUS=variable

specifies a variable whose value contains the return status code from a URL request.

See "FILENAME Statement: URL Access Method” on page 99

Example “Example 14: Reading the Return Code from a URL Request” on page 141

STOPOVER

causes the DATA step to stop processing if an INPUT statement reaches the end of the current record without finding values for all variables in the statement. When an input line does not contain the expected number of values, SAS sets _ERROR_ to 1, stops building the data set as if a STOP statement has executed, and prints the incomplete data line.

Tip Use FLOWOVER to reset the default behavior.
TRUNCOVER
overrides the default behavior of the INPUT statement when an input data record is shorter than the INPUT statement expects. By default, the INPUT statement automatically reads the next input data record. TRUNCOVER enables you to read variable-length records when some records are shorter than the INPUT statement expects. Variables without any values assigned are set to missing.

Tip
Use TRUNCOVER to assign the contents of the input buffer to a variable when the field is shorter than expected.

UNBUFFERED
tells SAS not to perform a buffered (“look ahead”) read.

Alias
UNBUF

Interaction
When you use UNBUFFERED, SAS never sets the END= variable to 1.

Tip
When you read instream data with a DATALINES statement, UNBUFFERED is in effect.

_INFILE_=variable
specifies a character variable that references the contents of the current input buffer for this INFILE statement. You can use the variable in the same way as any other variable, even as the target of an assignment. The variable is automatically retained and initialized to blanks. As with automatic variables, the _INFILE_= variable is not written to the data set.

Restriction
variable cannot be a previously defined variable. Ensure that the _INFILE_= specification is the first occurrence of this variable in the DATA step. Do not set or change the length of the _INFILE_= variable with the LENGTH statement or ATTRIB statement. However, you can attach a format to this variable with the ATTRIB statement or FORMAT statement.

Interaction
The maximum length of this character variable is the logical record length (LRECL= on page 120) for the specified INFILE statement. However, SAS does not open the file to know the LRECL= until before the execution phase. Therefore, the designated size for this variable during the compilation phase is 32,767 bytes.

Tips
Modification of this variable directly modifies the INFILE statement's current input buffer. Any PUT _INFILE_ (when this INFILE is current) that follows the buffer modification reflects the modified
buffer contents. The _INFILE_= variable accesses only the current input buffer of the specified INFILE statement even if you use the N= option to specify multiple buffers.

To access the contents of the input buffer in another statement without using the _INFILE_= option, use the automatic variable _INFILE_.

The _INFILE_ variable does not have a fixed width. When you assign a value to the _INFILE_ variable, the length of the variable changes to the length of the value that is assigned.

See “Accessing the Contents of the Input Buffer” on page 126

Examples

“Example 9: Working with Data in the Input Buffer” on page 138

“Example 10: Accessing the Input Buffers of Multiple Files” on page 139

Operating Environment Options

**options | host-options**

Operating Environment Information: For descriptions of operating-environment-specific options in the INFILE statement, see the SAS documentation for your operating environment.

See INFILE Statement under Windows, UNIX, and z/OS

DBMS Specifications

**DBMS-Specifications**

enables you to read records from some DBMS files. You must license SAS/ACCESS software to be able to read from DBMS files. See the SAS/ACCESS documentation for the DBMS that you use.

Details

How to Use the INFILE Statement

Because the INFILE statement identifies the file to read, it must execute before the INPUT statement that reads the input data records. You can use the INFILE statement in conditional processing, such as an IF-THEN statement, because it is executable. The INFILE statement enables you to control the source of the input data records.

Usually, you use an INFILE statement to read data from an external file. When data is read from the job stream, you must use a DATALINES statement. However, to take advantage of certain data-reading options that are available only in the INFILE statement, you can use an INFILE statement with the file-specification DATALINES and a DATALINES statement in the same DATA step. For more information, see “Reading Long Instream Data Records” on page 130.

When you use more than one INFILE statement for the same file specification and you use options in each INFILE statement, the effect is additive. To avoid confusion, use all the options in the first INFILE statement for a given external file.
Reading from Multiple Input Files
You can read from multiple input files in a single iteration of the DATA step in one of two ways:

- To keep multiple files open and change which file is read, use multiple INFILE statements.
- To dynamically change the current input file within a single DATA step, use the FILEVAR= option in an INFILE statement. The FILEVAR= option enables you to read from one file, close that file, and then open another file. See “Example 5: Reading from Multiple Input Files” on page 136.

Updating External Files in Place
You can use the INFILE statement with the FILE statement to update records in an external file:

1. Specify the INFILE statement before the FILE statement.
2. Specify the same fileref or physical file name in each statement.
3. Use options that are common to both the INFILE and FILE statements in the INFILE statement instead of the FILE statement. (Any such options that are used in the FILE statement are ignored.)

See “Example 6: Updating an External File” on page 137.
To update individual fields within a record instead of the entire record, see the SHAREBUFFERS option on page 123.

Accessing the Contents of the Input Buffer
In addition to the _INFILE_ variable, you can use the automatic _INFILE_ variable to reference the contents of the current input buffer for the most recent execution of the INFILE statement. This character variable is automatically retained and initialized to blanks. As with other automatic variables, _INFILE_ is not written to the data set.

When you specify the _INFILE_= option in an INFILE statement, the _INFILE_ variable is also indirectly referenced by the automatic _INFILE_ variable. If the automatic _INFILE_ variable is present and you omit _INFILE_ in a particular INFILE statement, SAS creates an internal _INFILE_ variable for that INFILE statement. Otherwise, SAS does not create the _INFILE_ variable for a particular FILE.

During execution and at the point of reference, the maximum length of this character variable is the maximum length of the current _INFILE_ variable. However, because _INFILE_ references only other variables whose lengths are not known until before the execution phase, the designated length is 32,767 bytes during the compilation phase. For example, if you assign _INFILE_ to a new variable whose length is undefined, the default length of the new variable is 32,767 bytes. You cannot use the LENGTH statement and the ATTRIB statement to set or override the length of _INFILE_. You can use the FORMAT statement and the ATTRIB statement to assign a format to _INFILE_.

As with other SAS variables, you can update the _INFILE_ variable in an assignment statement. You can also use a format with _INFILE_ in a PUT statement. For example, this PUT statement writes the contents of the input buffer using a hexadecimal format.
put _infile_ $hex100.;

Any modification of _INFILE_ directly modifies the current input buffer for the current INFILE statement. The execution of any PUT _INFILE_ statement that follows this buffer modification reflects the contents of the modified buffer.

_INFILE_ accesses only the contents of the current input buffer for an INFILE statement, even when you use the N= option to specify multiple buffers. You can access all the N= buffers, but you must use an INPUT statement with the # line pointer control to make the desired buffer the current input buffer.

Reading Delimited Data

By default, the delimiter that is used to read input data records with list input is a blank space. The delimiter-sensitive data (DSD) option, the DELIMITER= option, the DLMSTR= option, and the DLMSOPT= option affect how list input handles delimiters. The DELIMITER= option or DLMSTR= option specifies that the INPUT statement use a character other than a blank as a delimiter for data values that are read with list input. When the DSD option is in effect, the INPUT statement uses a comma as the default delimiter.

To read a value as missing between two consecutive delimiters, use the DSD option. By default, the INPUT statement treats consecutive delimiters as a unit. When you use DSD, the INPUT statement treats consecutive delimiters separately. Therefore, a value that is missing between consecutive delimiters is read as a missing value. To change the delimiter from a comma to another value, use the DELIMITER= option or DLMSTR= option.

For example, this DATA step program uses list input to read data that is separated with commas. The second data line contains a missing value. Because SAS allows consecutive delimiters with list input, the INPUT statement cannot detect the missing value.

```sas
data scores;
  infile datalines delimiter=',';
  input test1 test2 test3;
  datalines;
  91,87,95
  97,,92
  ,1,1
;```

With the FLOWOVER option in effect, the data set SCORES contains two, not three, observations. The second observation is built incorrectly.

<table>
<thead>
<tr>
<th>OBS</th>
<th>TEST1</th>
<th>TEST2</th>
<th>TEST3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>91</td>
<td>87</td>
<td>95</td>
</tr>
<tr>
<td>2</td>
<td>97</td>
<td>92</td>
<td>1</td>
</tr>
</tbody>
</table>

To correct the problem, use the DSD option in the INFILE statement.

```sas
data scores;
  input test1 test2 test3;
  datalines;
  91,87,95
  97,,92
  ,1,1
;```
The DSD option also enables list input to read a character value that contains a delimiter within a quoted string. For example, if data is separated with commas, DSD enables you to place the character string in quotation marks and read a comma as a valid character. SAS does not store the quotation marks as part of the character value. To retain the quotation marks as part of the value, use the tilde (~) format modifier in an INPUT statement. See "Example 1: Changing How Delimiters Are Treated" on page 132.

Note: Anytime a text file originates from anywhere other than the local encoding environment, it might be necessary to specify the ENCODING= option on either the EBCDIC or ASCII environment. For example, when you read an EBCDIC text file on an ASCII platform, it is recommended that you specify the ENCODING= option in the INFILE statement. However, if you use the DSD and DLM= options or the DLMSTR= option in the INFILE statement, the ENCODING= option is a requirement because these options must contain certain characters (such as quotation marks, commas, and blanks) in the session encoding. The use of encoding-specific informats should be reserved for use with true binary files (files that contain both character and noncharacter fields).

If your data is longer than the default length, you need to use informats. For example, date or time values, names, and addresses can be longer than eight characters. In such cases, you either need to add an INFORMAT statement to the DATA step or add informats directly in the INPUT statement. If you add informats in the INPUT statement, you must add a colon (:) in front of the informat. This tells SAS to read from the first character after the current delimiter to the number of characters specified in the informat.

---

The DSD option also enables list input to read a character value that contains a delimiter within a quoted string. For example, if data is separated with commas, DSD enables you to place the character string in quotation marks and read a comma as a valid character. SAS does not store the quotation marks as part of the character value. To retain the quotation marks as part of the value, use the tilde (~) format modifier in an INPUT statement. See "Example 1: Changing How Delimiters Are Treated" on page 132.

Note: Anytime a text file originates from anywhere other than the local encoding environment, it might be necessary to specify the ENCODING= option on either the EBCDIC or ASCII environment. For example, when you read an EBCDIC text file on an ASCII platform, it is recommended that you specify the ENCODING= option in the INFILE statement. However, if you use the DSD and DLM= options or the DLMSTR= option in the INFILE statement, the ENCODING= option is a requirement because these options must contain certain characters (such as quotation marks, commas, and blanks) in the session encoding. The use of encoding-specific informats should be reserved for use with true binary files (files that contain both character and noncharacter fields).

If your data is longer than the default length, you need to use informats. For example, date or time values, names, and addresses can be longer than eight characters. In such cases, you either need to add an INFORMAT statement to the DATA step or add informats directly in the INPUT statement. If you add informats in the INPUT statement, you must add a colon (:) in front of the informat. This tells SAS to read from the first character after the current delimiter to the number of characters specified in the informat.

---

The DSD option also enables list input to read a character value that contains a delimiter within a quoted string. For example, if data is separated with commas, DSD enables you to place the character string in quotation marks and read a comma as a valid character. SAS does not store the quotation marks as part of the character value. To retain the quotation marks as part of the value, use the tilde (~) format modifier in an INPUT statement. See "Example 1: Changing How Delimiters Are Treated" on page 132.

Note: Anytime a text file originates from anywhere other than the local encoding environment, it might be necessary to specify the ENCODING= option on either the EBCDIC or ASCII environment. For example, when you read an EBCDIC text file on an ASCII platform, it is recommended that you specify the ENCODING= option in the INFILE statement. However, if you use the DSD and DLM= options or the DLMSTR= option in the INFILE statement, the ENCODING= option is a requirement because these options must contain certain characters (such as quotation marks, commas, and blanks) in the session encoding. The use of encoding-specific informats should be reserved for use with true binary files (files that contain both character and noncharacter fields).

If your data is longer than the default length, you need to use informats. For example, date or time values, names, and addresses can be longer than eight characters. In such cases, you either need to add an INFORMAT statement to the DATA step or add informats directly in the INPUT statement. If you add informats in the INPUT statement, you must add a colon (:) in front of the informat. This tells SAS to read from the first character after the current delimiter to the number of characters specified in the informat.

---

The DSD option also enables list input to read a character value that contains a delimiter within a quoted string. For example, if data is separated with commas, DSD enables you to place the character string in quotation marks and read a comma as a valid character. SAS does not store the quotation marks as part of the character value. To retain the quotation marks as part of the value, use the tilde (~) format modifier in an INPUT statement. See "Example 1: Changing How Delimiters Are Treated" on page 132.

Note: Anytime a text file originates from anywhere other than the local encoding environment, it might be necessary to specify the ENCODING= option on either the EBCDIC or ASCII environment. For example, when you read an EBCDIC text file on an ASCII platform, it is recommended that you specify the ENCODING= option in the INFILE statement. However, if you use the DSD and DLM= options or the DLMSTR= option in the INFILE statement, the ENCODING= option is a requirement because these options must contain certain characters (such as quotation marks, commas, and blanks) in the session encoding. The use of encoding-specific informats should be reserved for use with true binary files (files that contain both character and noncharacter fields).

If your data is longer than the default length, you need to use informats. For example, date or time values, names, and addresses can be longer than eight characters. In such cases, you either need to add an INFORMAT statement to the DATA step or add informats directly in the INPUT statement. If you add informats in the INPUT statement, you must add a colon (:) in front of the informat. This tells SAS to read from the first character after the current delimiter to the number of characters specified in the informat.

---

The DSD option also enables list input to read a character value that contains a delimiter within a quoted string. For example, if data is separated with commas, DSD enables you to place the character string in quotation marks and read a comma as a valid character. SAS does not store the quotation marks as part of the character value. To retain the quotation marks as part of the value, use the tilde (~) format modifier in an INPUT statement. See "Example 1: Changing How Delimiters Are Treated" on page 132.

Note: Anytime a text file originates from anywhere other than the local encoding environment, it might be necessary to specify the ENCODING= option on either the EBCDIC or ASCII environment. For example, when you read an EBCDIC text file on an ASCII platform, it is recommended that you specify the ENCODING= option in the INFILE statement. However, if you use the DSD and DLM= options or the DLMSTR= option in the INFILE statement, the ENCODING= option is a requirement because these options must contain certain characters (such as quotation marks, commas, and blanks) in the session encoding. The use of encoding-specific informats should be reserved for use with true binary files (files that contain both character and noncharacter fields).

If your data is longer than the default length, you need to use informats. For example, date or time values, names, and addresses can be longer than eight characters. In such cases, you either need to add an INFORMAT statement to the DATA step or add informats directly in the INPUT statement. If you add informats in the INPUT statement, you must add a colon (:) in front of the informat. This tells SAS to read from the first character after the current delimiter to the number of characters specified in the informat.

---

The DSD option also enables list input to read a character value that contains a delimiter within a quoted string. For example, if data is separated with commas, DSD enables you to place the character string in quotation marks and read a comma as a valid character. SAS does not store the quotation marks as part of the character value. To retain the quotation marks as part of the value, use the tilde (~) format modifier in an INPUT statement. See "Example 1: Changing How Delimiters Are Treated" on page 132.

Note: Anytime a text file originates from anywhere other than the local encoding environment, it might be necessary to specify the ENCODING= option on either the EBCDIC or ASCII environment. For example, when you read an EBCDIC text file on an ASCII platform, it is recommended that you specify the ENCODING= option in the INFILE statement. However, if you use the DSD and DLM= options or the DLMSTR= option in the INFILE statement, the ENCODING= option is a requirement because these options must contain certain characters (such as quotation marks, commas, and blanks) in the session encoding. The use of encoding-specific informats should be reserved for use with true binary files (files that contain both character and noncharacter fields).

If your data is longer than the default length, you need to use informats. For example, date or time values, names, and addresses can be longer than eight characters. In such cases, you either need to add an INFORMAT statement to the DATA step or add informats directly in the INPUT statement. If you add informats in the INPUT statement, you must add a colon (:) in front of the informat. This tells SAS to read from the first character after the current delimiter to the number of characters specified in the informat.

---

The DSD option also enables list input to read a character value that contains a delimiter within a quoted string. For example, if data is separated with commas, DSD enables you to place the character string in quotation marks and read a comma as a valid character. SAS does not store the quotation marks as part of the character value. To retain the quotation marks as part of the value, use the tilde (~) format modifier in an INPUT statement. See "Example 1: Changing How Delimiters Are Treated" on page 132.

Note: Anytime a text file originates from anywhere other than the local encoding environment, it might be necessary to specify the ENCODING= option on either the EBCDIC or ASCII environment. For example, when you read an EBCDIC text file on an ASCII platform, it is recommended that you specify the ENCODING= option in the INFILE statement. However, if you use the DSD and DLM= options or the DLMSTR= option in the INFILE statement, the ENCODING= option is a requirement because these options must contain certain characters (such as quotation marks, commas, and blanks) in the session encoding. The use of encoding-specific informats should be reserved for use with true binary files (files that contain both character and noncharacter fields).

If your data is longer than the default length, you need to use informats. For example, date or time values, names, and addresses can be longer than eight characters. In such cases, you either need to add an INFORMAT statement to the DATA step or add informats directly in the INPUT statement. If you add informats in the INPUT statement, you must add a colon (:) in front of the informat. This tells SAS to read from the first character after the current delimiter to the number of characters specified in the informat.
another operating system. For example, if you are running SAS in a Windows environment and you need to read a file that was created under UNIX, use TERMSTR=LF.

In SAS 9.4, SAS automatically normalizes imported copies of files that are created in Windows when you are reading the files in a UNIX environment. This enables easier sharing of files between the environments.

**TRUNCOVER=**
Use this option to specify that SAS use only the available input data in the current record to populate as many variables as possible. By default, if the INPUT statement reads past the end of a record without finding enough data to populate the variables listed, the statement continues to read data from the next record. Use the TRUNCOVER= option to keep SAS from reading the next record.

Here are some troubleshooting problems and solutions.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Solution</th>
</tr>
</thead>
</table>
| The SAS log contains this message and all of the data in the data set seems to be read: One or more lines were truncated. | This is most likely not a problem. This message occurs when one of these conditions is true:  
- The maximum record length is less than the active logical record length.  
- All of the data is in the data set.  
- You are using a value for the FIRSTOBS= that is higher than 1.  
The message indicates that the FIRSTOBS= option has skipped a line because the line is longer than the LRECL= value. SAS recognizes that a line is longer than the LRECL= value, but it does not recognize that the line has been skipped by the FIRSTOBS= option. |
| Your last variable is numeric and it is missing in every observation.   | Use the TERMSTR=CRLF option in your INFILE statement. In releases prior to SAS 9, either convert the file to the proper UNIX structure or add the carriage return (<CR>) to the list of delimiters. If you are reading a CSV file, specify the carriage return as DL DLM='2C0D'x. If you are reading a tab-delimited file, use DLM='090D'x. If you have a different delimiter, contact SAS Technical Support for help. |
| The most frequent cause for a missing variable is reading a Windows file in a UNIX environment, which uses only a line feed. The UNIX operating system reads the carriage return as data. Because the data is not numeric, an invalid data message is generated. | |
Problem

When you read a CSV file, the records are split inside a quoted string. For example, sometimes a long comment field abruptly stops and continues on the next row of the file that you are reading. You can see it if you open the file in a text editor.

This problem occurs most commonly in files that are created from Microsoft Excel worksheets. In Excel, the column contains soft returns to help in formatting. When you save the worksheet as a CSV file, the soft returns incorrectly appear to SAS as end-of-record markers, which causes the records to be split incorrectly.

Solution

In SAS 9 or later, under Windows, you can use the TERMSTR=CRLF option in the INFILE statement. This setting indicates that SAS accepts only the complete return and line feed as an end-of-record marker.

In releases prior to SAS 9, run the following program. The program converts any line-feed character between double quotation marks to a space to enable SAS to read the file correctly. In this program, the INFILE and FILE statements refer to the same file. The SHAREBUFFERS option enables the external file to be updated in place, removing the offending characters.

```sas
data _null_;
  infile 'c:\_today\mike.csv' recfm=n sharebuffers;
  file 'c:\_today\mike.csv' recfm=n;
  input a $char1.;
  retain open 0;
  /* toggle the open flag */
  if a="" then open=not open;
  if a='0A'x and open then put ' ';
run;
```

The program reads the file one byte at a time and replaces the line-feed character, as needed.

Reading Long Instream Data Records

You can use the INFILE statement with the DATALINES file specification to process instream data. An INPUT statement reads the data records that follow the DATALINES statement. If you use the CARDIMAGE system option, SAS processes the data lines exactly like 80-byte punched card images that are padded with blanks. The default FLOWOVER option in the INFILE statement causes the INPUT statement to read the next record if SAS does not find values in the current record for all of the variables in the statement. To ensure that your data is processed correctly, use an external file for input when record lengths are greater than 80 bytes.

**Note:** The NOCARDIMAGE system option (see the “CARDIMAGE System Option” in SAS System Options: Reference) specifies that data lines not be treated as if they were 80-byte card images. The end of a data line is always treated as the end of the last token, except for strings that are enclosed in quotation marks.

Reading Past the End of a Line

By default, if the INPUT statement tries to read past the end of the current input data record, the statement moves the input pointer to column 1 of the next record to read.
the remaining values. This default behavior is specified by the FLOWOVER option. A message is written to the SAS log.

NOTE: SAS went to a new line when INPUT statement reached past the end of a line.

Several options are available to change the INPUT statement behavior when an end of line is reached. The STOPOVER option treats this condition as an error and stops building the data set. The MISSOVER and TRUNCOVER options do not allow the input pointer to go to the next record when the current INPUT statement is not satisfied. The SCANOVER option, used with '@character-string', scans the input record until it finds the specified character-string. The FLOWOVER option restores the default behavior.

The TRUNCOVER and MISSOVER options are similar. The MISSOVER option causes the INPUT statement to set a value to missing if the statement is unable to read an entire field because the value is shorter than the field length that is specified in the INPUT statement. The TRUNCOVER option writes whatever characters are read to the appropriate variable.

For example, an external file with variable-length records contains these records:

```
---+----1----+----2
1
22
333
4444
55555
```

The following DATA step reads this data to create a SAS data set. Only one of the input records is as long as the informatted length of the variable TESTNUM.

```sas
data numbers;
  infile 'external-file';
  input testnum 5.;
run;
```

The DATA step creates the three observations from the five input records because by default the FLOWOVER option is used to read the input records.

If you use the MISSOVER option in the INFILE statement, the DATA step creates five observations. All the values that were read from records that were too short are set to missing. Use the TRUNCOVER option in the INFILE statement if you prefer to see what values were present in records that were too short to satisfy the current INPUT statement.

```sas
infile 'external-file' truncover;
```

The DATA step now reads the same input records and creates five observations. The following table compares the SAS data sets.

**Table 2.2** The Value of TESTNUM Using Different INFILE Statement Options

<table>
<thead>
<tr>
<th>OBS</th>
<th>FLOWOVER</th>
<th>MISSOVER</th>
<th>TRUNCOVER</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>22</td>
<td>.</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>4444</td>
<td>.</td>
<td>22</td>
</tr>
</tbody>
</table>
Comparisons

- The **INFILE statement** specifies the *input file* for any **INPUT statements** in the **DATA step**. The **FILE statement** specifies the *output file* for any **PUT statements** in the **DATA step**.

- An **INFILE statement** usually identifies data from an external file. A **DATALINES** statement indicates that data follows in the job stream. You can use the **INFILE** statement with the file specification **DATALINES** to take advantage of certain data-reading options that affect how the **INPUT statement** reads instream data.

Examples:

**Example 1: Changing How Delimiters Are Treated**

By default, the **INPUT statement** uses a blank as the delimiter. In this example, the **DATA step** uses a comma as the delimiter.

```
data num;
  infile datalines dsd;
  input x y z;
  datalines;
  ,2,3
  4,5,6
  7,8,9
;
```

The argument **DATALINES** in the **INFILE statement** enables you to use an **INFILE statement option** to read instream data lines. The **DSD option** sets the comma as the default delimiter. Because a comma precedes the first value in the first data line, a missing value is assigned to variable **X** in the first observation, and the value **2** is assigned to variable **Y**.

If the data uses multiple delimiters or a single delimiter other than a comma, simply specify the delimiter values with the **DELIMITER=** option. In this example, the characters **a** and **b** function as delimiters.

```
data nums;
  infile datalines dsd delimiter='ab';
  input X Y Z;
  datalines;
  laa2ab3
  4b5bab6
  7a8b9
;
```

```
proc print; run;
```
The output that PROC PRINT generates shows the resulting NUM data set. Values are missing for variables in the first and second observations because DSD causes list input to detect two consecutive delimiters. If you omit DSD, the characters a, b, aa, ab, ba, or bb function as the delimiter and no variables are assigned missing values.

**Output 2.6  The NUM Data Set**

<table>
<thead>
<tr>
<th>Obs</th>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>.2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>5</td>
<td>.</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

If you want to use a string as the delimiter, specify the delimiter values with the DLMSTR= option. In this example, the string PRD is used as the delimiter. Note that the string contains uppercase characters. By using the DLMSOPT= option, PRD, PrD, PrD, pRD, prD, P RD, and prd are all valid delimiters.

``` SAS 
data test; 	infile datalines dsd dlmstr='PRD' dlmsopt='i'; 
input X Y Z; 
datalines; 
1PRD2PRd3 
4PrD5Prd6 
7pRd8pRD9 
; 
proc print data=test; run; 
```

The output from PROC PRINT shows all the observations in the TEST data set.

**Output 2.7  The TEST Data Set**

<table>
<thead>
<tr>
<th>Obs</th>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

This DATA step uses modified list input and the DSD option to read data that is separated by commas and that might contain commas as part of a character value.

``` SAS 
data scores; 
infile datalines dsd; 
input Name : $9. Score 
     Team : $25. Div $; 
datalines; 
Mitchel,82,"Blue Bunnies, Richmond",AAA 
Sue Ellen,74,"Green Gazelles, Atlanta",AA 
```
The output that PROC PRINT generates shows the resulting SCORES data set.
The delimiter (comma) is stored as part of the value of TEAM, whereas the quotation marks are not.

**Output 2.8  The SCORES Data Set**

<table>
<thead>
<tr>
<th>Obs</th>
<th>Name</th>
<th>Score</th>
<th>Team</th>
<th>Div</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Joseph</td>
<td>76</td>
<td>Red Racers, Washington</td>
<td>AAA</td>
</tr>
<tr>
<td>2</td>
<td>Mitchel</td>
<td>82</td>
<td>Blue Bunnies, Richmond</td>
<td>AAA</td>
</tr>
<tr>
<td>3</td>
<td>Sue Ellen</td>
<td>74</td>
<td>Green Gazelles, Atlanta</td>
<td>AA</td>
</tr>
</tbody>
</table>

Example 2: Handling Missing Values and Short Records with List Input

This example shows how to prevent missing values from causing problems when you read the data with list input. Some data lines in this example contain fewer than five temperature values. Use the MISSOVER option so that these values are set to missing.

```sas
data weather;
  infile datalines missover;
  input temp1-temp5;
  datalines;
  97.9 98.1 98.3
  98.6 99.2 99.1 98.5 97.5
  96.2 97.3 98.3 97.6 96.5
;```

SAS reads the three values on the first data line as the values of TEMP1, TEMP2, and TEMP3. The MISSOVER option causes SAS to set the values of TEMP4 and TEMP5 to missing for the first observation because no values for those variables are in the current input data record.

When you omit the MISSOVER option or use FLOWOVER, SAS moves the input pointer to line 2 and reads values for TEMP4 and TEMP5. The next time the DATA step executes, SAS reads a new line which, in this case, is line 3. This message appears in the SAS log:

**NOTE:** SAS went to a new line when INPUT statement reached past the end of a line.

You can also use the STOPOVER option in the INFILE statement. Using the STOPOVER option causes the DATA step to halt execution when an INPUT statement does not find enough values in a record of raw data.

```sas
  infile datalines stopover;
```

Because SAS does not find a TEMP4 value in the first data record, it sets _ERROR_ to 1, stops building the data set, and prints data line 1.
Example 3: Scanning Variable-Length Records for a Specific Character String

This example shows how to use TRUNCOVER in combination with SCANOVER to pull phone numbers from a phone book. The phone number is always preceded by the word "phone:". Because the phone numbers include international numbers, the maximum length is 32 characters.

```sas
filename phonebk host-specific-path;
data _null_;  
file phonebk;  
input line $80.;  
put line;  
datalines;  
  Jenny's Phone Book  
  Jim Johanson phone: 619-555-9340  
  Jim wants a scarf for the holidays.  
  Jane Jovalley phone: (213) 555-4820  
  Jane started growing cabbage in her garden.  
  Her dog's name is Juniper.  
  J.R. Hauptman phone: (49)12 34-56 78-90  
  J.R. is my brother.  
;  
run;
```

Use @'phone:' to scan the lines of the file for a phone number and position the file pointer where the phone number begins. Use TRUNCOVER in combination with SCANOVER to skip the lines that do not contain 'phone:' and write only the phone numbers to the SAS log.

```sas
data _null_;  
 infile phonebk truncover scanover;  
 input @'phone:' phone $32.;  
 put phone=;  
 run;
```

The program writes these lines to the SAS log:

```
phone=619-555-9340
phone=(213) 555-4820
phone=(49)12 34-56 78-90
```

Example 4: Reading Files That Contain Variable-Length Records

This example shows how to use LENGTH=, in combination with the $VARYING. informat, to read a file that contains variable-length records.

```sas
data a;  
infile file-specification length=linelen lrecl=510 pad;  
 input firstvar 1-10 @; /* assign LINELEN */  
 varlen=linelen-10; /* Calculate VARLEN */  
 input @11 secondvar $varying500. varlen;  
 run;
```

These actions occur in the preceding DATA step:

- The INFILE statement creates the variable LINELEN but does not assign a value to the variable.
When the first INPUT statement executes, SAS determines the line length of the record and assigns that value to the variable LINELEN. The single trailing @ holds the record in the input buffer for the next INPUT statement.

The assignment statement uses the two known lengths (the length of FIRSTVAR and the length of the entire record) to determine the length of VARLEN.

The second INPUT statement uses the VARLEN value with the informat $VARYING500. to read the variable SECONDVAR.

For more information, see "$VARYINGw. Informat" in SAS Formats and Informats: Reference.

Example 5: Reading from Multiple Input Files

This DATA step reads from two input files during each iteration of the DATA step. As SAS switches from one file to the next, each file remains open. The input pointer remains in place to begin reading from that location the next time an INPUT statement reads from that file.

```sas
data qtrtot(drop=jansale febsale marsale aprsale maysale junsale);
  /* identify location of 1st file */
  infile file-specification-1;
  /* read values from 1st file */
  input name $ jansale febsale marsale;
  qtr1tot=sum(jansale,febsale,marsale);
  /* identify location of 2nd file */
  infile file-specification-2;
  /* read values from 2nd file */
  input @7 aprsale maysale junsale;
  qtr2tot=sum(aprsale,maysale,junsale);
run;
```

The DATA step terminates when SAS reaches an end of file on the shortest input file.

This DATA step uses FILEVAR= to read from a different file during each iteration of the DATA step.

```sas
data allsales;
  length fileloc myinfile $ 300;
  input fileloc $ ; /* read instream data */
  /* The INFILE statement closes the current file and opens a new one if FILELOC changes value when INFILE executes */
  infile file-specification filevar=fileloc filename=myinfile end=done;
  /* DONE set to 1 when last input record read */
  do while(not done);
    /* Read all input records from the currently opened input file, write to ALLSALES */
    input name $ jansale febsale marsale;
    output;
  end;
  put 'Finished reading ' myinfile=;
  datalines;
  external-file-1
  external-file-2
  external-file-3
;```
The `FILENAME=` option assigns the name of the current input file to the variable `MYINFILE`. The `LENGTH` statement ensures that the `FILENAME=` variable and the `FILEVAR=` variable have a length that is long enough to contain the value of the file name. The `PUT` statement prints the physical name of the currently open input file to the SAS log.

Example 6: Updating an External File

This example shows how to use the `INFILE` statement with the `SHAREBUFFERS` option and the `INPUT`, `FILE`, and `PUT` statements to update an external file in place.

```sas
data _null_
  /* The INFILE and FILE statements */
  /* must specify the same file. */
  infile file-specification-1 sharebuffers;
  file file-specification-1;
  input state $ 1-2 phone $ 5-16;
  /* Replace area code for NC exchanges */
  if state= 'NC' and substr(phone,5,3)='333' then
    phone='910-'||substr(phone,5,8);
  put phone 5-16;
run;
```

Example 7: Truncating Copied Records

The `LENGTH=` option is useful when you copy the input file to another file with the `PUT _INFILE_` statement. Use `LENGTH=` to truncate the copied records. For example, these statements truncate the last 20 columns from each input data record before the input data record is written to the output file.

```sas
data _null_
  infile file-specification-1 length=a;
  input;
  a=a-20;
  file file-specification-2;
  put _infile_;
run;
```

The `START=` option is also useful when you want to truncate what the `PUT _INFILE_` statement copies. For example, if you do not want to copy the first 10 columns of each record, these statements copy from column 11 to the end of each record in the input buffer.

```sas
data _null_
  infile file-specification start=s;
  input;
  s=11;
  file file-specification-2;
  put _infile_;
run;
```

Example 8: Listing the Pointer Location

This DATA step assigns the value of the current pointer location in the input buffer to the variables `LINEPT` and `COLUMNPT`.

```sas
data _null_
  infile datalines n=2 line=Linept col=Columnpt;
  input name $ 1-15 #2 @3 id;
```
The preceding statements produce these lines for each execution of the DATA step because the input pointer is on the second line in the input buffer when the PUT statement executes.

<table>
<thead>
<tr>
<th>Linept</th>
<th>Columnpt</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
</tr>
</tbody>
</table>

Example 9: Working with Data in the Input Buffer

The _INFILE_ variable always contains the most recent record that is read from an INPUT statement. The following example illustrates the use of the _INFILE_ variable to perform these tasks:

- read an entire record that you want to parse without using the INPUT statement
- read an entire record that you want to write to the SAS log
- modify the contents of the input record before parsing the line with an INPUT statement

The example file contains phone bill information. The numeric data, minutes, and charge are enclosed in angle brackets (< >).

```
filename phonbill host-specific-filename;
data _null_;
  file phonbill;
  input line $80.;
  put line;
  datalines;
  City Number Minutes Charge
  Jackson 415-555-2384 <25> <2.45>
  Jefferson 813-555-2356 <15> <1.62>
  Joliet 913-555-3223 <65> <10.32>
run;
```

This code reads each record and parses the record to extract the minute and charge values.

```
data _null_;
  infile phonbill firstobs=2;
  input;
  city = scan(_infile_, 1, ' ');  
  char_min = scan(_infile_, 3, ' ');  
  char_min = substr(char_min, 2, length(char_min)-2);
  minutes = input(char_min, BEST12.);
  put city= minutes=;
run;
```

The program writes these lines to the SAS log:
The INPUT statement in this code reads a record from the file. The automatic _INFILE_ variable is used in the PUT statement to write the record to the SAS log.

```sas
data _null_;
  infile phonbill;
  input;
  put _infile_;
run;
```

The program writes these lines to the SAS log:

<table>
<thead>
<tr>
<th>City</th>
<th>Number</th>
<th>Minutes</th>
<th>Charge</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jackson</td>
<td>415-555-2384</td>
<td>25</td>
<td>2.45</td>
</tr>
<tr>
<td>Jefferson</td>
<td>813-555-2356</td>
<td>15</td>
<td>1.62</td>
</tr>
<tr>
<td>Joliet</td>
<td>913-555-3223</td>
<td>65</td>
<td>10.32</td>
</tr>
</tbody>
</table>

In this code, the first INPUT statement reads and holds the record in the input buffer. The _INFILE_ = option removes the angle brackets (< >) from the numeric data. The second INPUT statement parses the value in the buffer.

```sas
data _null_;
  length city number $16. minutes charge 8;
  infile phonbill firstobs=2;
  input @;
  _infile_ = compress(_infile_, '<>');
  input city number minutes charge;
  put city= number= minutes= charge=;
run;
```

The program writes these lines to the SAS log:

| city=Jackson | number=415-555-2384 | minutes=25 | charge=2.45 |
| city=Jefferson| number=813-555-2356 | minutes=15 | charge=1.62 |
| city=Joliet  | number=913-555-3223  | minutes=65 | charge=10.32|

Example 10: Accessing the Input Buffers of Multiple Files

This example uses both the _INFILE_ automatic variable and the _INFILE_ = option to read multiple files and access the input buffers for each of them. This code creates four files: three data files and one file that contains the names of all the data files. The second DATA step reads the filenames file, opens each data file, and writes the contents to the SAS log. Because the PUT statement needs _INFILE_ for the filenames file and the data file, one of the _INFILE_ variables is referenced with fname.

```sas
data _null_;
  do i = 1 to 3;
    fname = 'external-data-file' || put(i,1.) || '.dat';
    file datfiles filevar=fname;
    do j = 1 to 5;
      put i j;
    end;
    file 'external-filenames-file';
    put fname;
  end;
run;
```

```sas
Example 10: Accessing the Input Buffers of Multiple Files
139
```
The program writes these lines to the SAS log:

Example 11: Specifying an Encoding When Reading an External File

This example creates a SAS data set from an external file. The external file's encoding is in UTF-8, and the current SAS session encoding is WLatin1. By default, SAS assumes that the external file is in the same encoding as the session encoding, which causes the character data to be written to the new SAS data set incorrectly.

To tell SAS what encoding to use when reading the external file, specify the ENCODING= option. When you tell SAS that the external file is in UTF-8, SAS then transcodes the external file from UTF-8 to the current session encoding when writing to the new SAS data set. Therefore, the data is written to the new data set correctly in WLatin1.
Example 12: Reading All File Members of a Directory

This example reads all of the files from the 'c:\mydir\tmp' directory using the MEMVAR= variable set to a blank.

```sas
data _null_;  
length memname $ 1024;  
memname = ' ';
infile 'c:\mydir\tmp' memvar=memname end=done;  
put memname=;  
do while (^done);
  input x;
  put _all_;  
end;
end;
run;
```

Example 13: Reading a List of File Members

This example reads a list of files from the 'c:\mydir\tmp' directory that are assigned to the MEMVAR= option.

```sas
data _null_;  
length mname $ 32;
do mname = 'erase.dat', 'erase2.dat';
  infile 'c:\mydir\tmp' memvar=mname end=done;  
do while (^done);
  input x;
  put _all_;  
end;
end;
stop;
run;
```

Example 14: Reading the Return Code from a URL Request

This example attempts to access a web resource that does not exist. As a result, the STATUS= variable is set to the page not found code (404).

```sas
filename in url "http://www.sas.com/missing_page.html" lrecl=1000;
data _NULL_;  
infile in status=ioerr;
input;
if (ioerr = 404) then do;
  put 'File not found';
  stop;
end;
run;
```

The program writes this line to the SAS log:

```
File not found
```
See Also

Statements:
- “FILENAME Statement” on page 97
- “FILENAME Statement: ACTIVEMQ Access Method” in Application Messaging with SAS
- “FILENAME Statement: Azure Access Method” in SAS Global Statements: Reference
- “FILENAME Statement: JMS Access Method” in Application Messaging with SAS
- “FILENAME Statement: S3 Access Method” in SAS Global Statements: Reference
- “INPUT Statement” on page 146
- “LOCKDOWN Statement” in SAS Intelligence Platform: Application Server Administration Guide
- “PUT Statement” on page 236

System Options:
- “LOCKDOWN System Option in SAS Intelligence Platform: Application Server Administration Guide

**INFORMAT Statement**
Associates informats with variables.

Valid in: DATA step or PROC step
Categories: CAS Information
Type: Declarative

**Syntax**

```sas
INFORMAT variable-1 <variable-n> <informat>;
INFORMAT <variable-1> <variable-n> DEFAULT=default-informat;
INFORMAT variable-1 <variable-n> informat DEFAULT=default-informat;
```

**Arguments**

`variable`
- specifies one or more variables to associate with an informat. You must specify at least one `variable` when specifying an `informat` or when including no other
arguments. Specifying a variable is optional when using a DEFAULT= informat specification.

**Tip**
To disassociate an informat from a variable, use the variable's name in an INFORMAT statement without specifying an informat. Place the INFORMAT statement after the SET statement. See “Example 3: Removing an Informat” on page 146.

**informat**
specifies the informat for reading the values of the variables that are listed in the INFORMAT statement.

**Tip**
If an informat is associated with a variable by using the INFORMAT statement, and that same informat is not associated with that same variable in the INPUT statement, then that informat behaves like informats that you specify with a colon (:) modifier in an INPUT statement. SAS reads the variables by using list input with an informat. For example, you can use the : modifier with an informat to read character values that are longer than eight bytes, or numeric values that contain nonstandard values. For more information, see “INPUT Statement: List” on page 171.

See [SAS Formats and Informats: Reference](#)

Example “Example 2: Specifying Numeric and Character Informats” on page 145

**DEFAULT= default-informat**
specifies a temporary default informat for reading values of the variables that are listed in the INFORMAT statement. If no variable is specified, then the DEFAULT= informat specification applies a temporary default informat for reading values of all the variables of that type included in the DATA step. Numeric informats are applied to numeric variables, and character informats are applied to character variables. These default informats apply only to the current DATA step.

A DEFAULT= informat specification applies to
- variables that are not named in an INFORMAT or ATTRIB statement
- variables that are not permanently associated with an informat within a SAS data set
- variables that are not read with an explicit informat in the current DATA step.

Default If you omit DEFAULT=, SAS uses w.d as the default numeric informat and $w. as the default character informat.

Restriction Use this argument only in a DATA step.

**Tip**
A DEFAULT= specification can occur anywhere in an INFORMAT statement. It can specify either a numeric default, a character default, or both.

Example “Example 1: Specifying Default Informats” on page 145
Details

The Basics

An INFORMAT statement in a DATA step permanently associates an informat with a variable. You can specify standard SAS informats or user-written informats, previously defined in PROC FORMAT. A single INFORMAT statement can associate the same informat with several variables, or it can associate different informats with different variables. If a variable appears in multiple INFORMAT statements, SAS uses the informat that is assigned last.

CAUTION
Because an INFORMAT statement defines the length of previously undefined character variables, you can truncate the values of character variables in a DATA step if an INFORMAT statement precedes a SET statement.

How SAS Treats Variables When You Assign Informats with the INFORMAT Statement

Informats that are associated with variables by using the INFORMAT statement behave like informats that are used with modified list input. SAS reads the variables by using the scanning feature of list input, but applies the informat. In modified list input, SAS

- does not use the value of \( w \) in an informat to specify column positions or input field widths in an external file
- uses the value of \( w \) in an informat to specify the length of previously undefined character variables
- ignores the value of \( w \) in numeric informats
- uses the value of \( d \) in an informat in the same way it usually does for numeric informats
- treats blanks that are embedded as input data as delimiters unless you change their status with a DLM= or DLMSTR= option specification in an INFILE statement.

If you have coded the INPUT statement to use another style of input, such as formatted input or column input, that style of input is not used when you use the INFORMAT statement.

Comparisons

- Both the ATTRIB and INFORMAT statements can associate informats with variables, and both statements can change the informat that is associated with a variable. You can also use the INFORMAT statement in PROC DATASETS to change or remove the informat that is associated with a variable. The SAS windowing environment enables you to associate, change, or disassociate informats and variables in existing SAS data sets.
- SAS changes the descriptor information of the SAS data set that contains the variable. You can use an INFORMAT statement in some PROC steps, but the rules are different. For more information, see “FORMAT Procedure” in Base SAS Procedures Guide.
Examples:

Example 1: Specifying Default Informats
This example uses an INFORMAT statement to associate a default numeric informat:

```sas
data tstinfmt;
    informat default=3.1;
    input x;
    put x;
    datalines;
111
222
333
;
```

The PUT statement produces these results:

```
11.1
22.2
33.3
```

Example 2: Specifying Numeric and Character Informats
This example associates a character informat and a numeric informat with SAS variables. Although the character variables do not fully occupy 15 column positions, the INPUT statement reads the data records correctly by using modified list input:

```sas
data name;
    informat FirstName LastName $15. n1 6.2 n2 7.3;
    input firstname lastname n1 n2;
    datalines;
Alexander Robinson 35 11
;
```

This output shows a partial listing from PROC CONTENTS, as well as the report PROC PRINT generates.

**Output 2.9  Associating Numeric and Character Informats with SAS Variables**

<table>
<thead>
<tr>
<th>#</th>
<th>Variable</th>
<th>Type</th>
<th>Len</th>
<th>Informat</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>FirstName</td>
<td>Char</td>
<td>15</td>
<td>$15.</td>
</tr>
<tr>
<td>2</td>
<td>LastName</td>
<td>Char</td>
<td>15</td>
<td>$15.</td>
</tr>
<tr>
<td>3</td>
<td>n1</td>
<td>Num</td>
<td>8</td>
<td>6.2</td>
</tr>
<tr>
<td>4</td>
<td>n2</td>
<td>Num</td>
<td>8</td>
<td>7.3</td>
</tr>
</tbody>
</table>
Example 3: Removing an Informat

This example disassociates an existing informat. The order of the INFORMAT and SET statements is important.

data rtest;
  set rtest;
  informat x;
run;

See Also

Statements:
- “ATTRIB Statement” on page 30
- “INPUT Statement” on page 146
- “INPUT Statement: List” on page 171

INPUT Statement

Describes the arrangement of values in the input data record and assigns input values to the corresponding SAS variables.

Valid in: DATA step
Category: File-Handling
Type: Executable
Restriction: This statement is not supported in a DATA step that runs in CAS.

Syntax

```
INPUT <specification(s)> <@ | @@>;
```

Without Arguments

The INPUT statement with no arguments is called a null INPUT statement. The null INPUT statement
- brings an input data record into the input buffer without creating any SAS variables
- releases an input data record that is held by a trailing @ or a double trailing @.
For an example, see “Example 2: Using a Null INPUT Statement” on page 159.

Note: You cannot read binary files using the INPUT statement without arguments.

Arguments

**specification(s)**
can include

- **variable**
  names a variable that is assigned input values.

- **(variable-list)**
  specifies a list of variables that are assigned input values.

Requirement: The (variable-list) is followed by an (informat-list).

See: “How to Group Variables and Informats” on page 169

- **$$**
  specifies to store the variable value as a character value rather than as a numeric value.

  Tip: If the variable is previously defined as character, $ is not required.

  Example: “Example 1: Using Multiple Styles of Input in One INPUT Statement” on page 159

- **pointer-control**
  moves the input pointer to a specified line or column in the input buffer.

  See: “Column Pointer Controls” on page 148 and “Line Pointer Controls” on page 150

- **column-specifications**
  specifies the columns of the input record that contain the value to read.

  Tip: Informats are ignored. Only standard character and numeric data can be read correctly with this method.

  See: “Column Input” on page 152

  Example: “Example 1: Using Multiple Styles of Input in One INPUT Statement” on page 159

- **format-modifier**
  allows modified list input or controls the amount of information that is reported in the SAS log when an error in an input value occurs.

  Tip: Use modified list input to read data that cannot be read with simple list input.

  See: “When to Use List Input” on page 173 and “Format Modifiers for Error Reporting” on page 151

  Example: “Example 6: Positioning the Pointer with a Character Variable” on page 161

- **informat**
  specifies an informat to use to read the variable value.
### Tip
You can use modified list input to read data with informats. Modified list input is useful when the data require informats but cannot be read with formatted input because the values are not aligned in columns.

### See
- “Formatted Input” on page 153
- “List Input” on page 152

### Example
“Example 2: Using Informat Lists” on page 170

### (informat-list)
specifies a list of informats to use to read the values for the preceding list of variables.

#### Restriction
The (informat-list) must follow the (variable-list).

#### See
“How to Group Variables and Informats” on page 169

### @
holds an input record for the execution of the next INPUT statement within the same iteration of the DATA step. This line-hold specifier is called *trailing @*.

#### Restriction
The trailing @ must be the last item in the INPUT statement.

#### Tip
The trailing @ prevents the next INPUT statement from automatically releasing the current input record and reading the next record into the input buffer. It is useful when you need to read from a record multiple times.

#### See
“Using Line-Hold Specifiers” on page 155

#### Example
“Example 3: Holding a Record in the Input Buffer” on page 159

### @@
holds the input record for the execution of the next INPUT statement across iterations of the DATA step. This line-hold specifier is called *double trailing @*.

#### Restriction
The double trailing @ must be the last item in the INPUT statement.

#### Tip
The double trailing @ is useful when each input line contains values for several observations, or when a record needs to be reread on the next iteration of the DATA step.

#### See
“Using Line-Hold Specifiers” on page 155

#### Example
“Example 4: Holding a Record across Iterations of the DATA Step” on page 160

### Column Pointer Controls

#### @n
moves the pointer to column $n$.

#### Range
a positive integer

#### Tip
If $n$ is not an integer, SAS truncates the decimal value and uses only the integer value. If $n$ is zero or negative, the pointer moves to column 1.
@15 moves the pointer to column 15:
  input @15 name $10.;

"Example 7: Moving the Pointer Backward" on page 162

@numeric-variable
moves the pointer to the column given by the value of numeric-variable.

Range
  a positive integer

Tip
  If numeric-variable is not an integer, SAS truncates the decimal value and uses only the integer value. If numeric-variable is zero or negative, the pointer moves to column 1.

Example
  The value of the variable A moves the pointer to column 15:
    a=15;
    input @a name $10.;

"Example 5: Positioning the Pointer with a Numeric Variable" on page 161

@((expression))
moves the pointer to the column that is given by the value of expression.

Restriction
  Expression must result in a positive integer.

Tip
  If the value of expression is not an integer, SAS truncates the decimal value and uses only the integer value. If it is zero or negative, the pointer moves to column 1.

Example
  The result of the expression moves the pointer to column 15:
    b=5;
    input @((b*3)) name $10.;

@’character-string’
locates the specified series of characters in the input record and moves the pointer to the first column after character-string.

@character-variable
locates the series of characters in the input record that is given by the value of character-variable and moves the pointer to the first column after that series of characters.

Example
  This statement reads in the WEEKDAY character variable. The second @1 moves the pointer to the beginning of the input line. The value for SALES is read from the next non-blank column after the value of WEEKDAY:
    input @1 day 1. @5 weekday $10. @1 @weekday sales 8.2;

"Example 6: Positioning the Pointer with a Character Variable" on page 161

@((character-expression))
lodates the series of characters in the input record that is given by the value of character-expression and moves the pointer to the first column after the series.
Example 6: Positioning the Pointer with a Character Variable

+n
moves the pointer n columns.

Range a positive integer or zero

Tip If n is not an integer, SAS truncates the decimal value and uses only the integer value. If the value is greater than the length of the input buffer, the pointer moves to column 1 of the next record.

Example
This statement moves the pointer to column 23, reads a value for LENGTH from columns 23 through 26, advances the pointer five columns, and reads a value for WIDTH from columns 32 through 35:
input @23 length 4. +5 width 4.;

Example “Example 7: Moving the Pointer Backward” on page 162

+numeric-variable
moves the pointer the number of columns that is given by the value of numeric-variable.

Range a positive or negative integer or zero

Tip If numeric-variable is not an integer, SAS truncates the decimal value and uses only the integer value. If numeric-variable is negative, the pointer moves backward. If the current column position becomes less than 1, the pointer moves to column 1. If the value is zero, the pointer does not move. If the value is greater than the length of the input buffer, the pointer moves to column 1 of the next record.

Example “Example 7: Moving the Pointer Backward” on page 162

+(expression)
moves the pointer the number of columns given by expression.

Range expression must result in a positive or negative integer or zero

Tip If expression is not an integer, SAS truncates the decimal value and uses only the integer value. If expression is negative, the pointer moves backward. If the current column position becomes less than 1, the pointer moves to column 1. If the value is zero, the pointer does not move. If the value is greater than the length of the input buffer, the pointer moves to column 1 of the next record.

Line Pointer Controls

#n
moves the pointer to record n.

Range a positive integer

Interaction The N= option in the INFILE statement can affect the number of records the INPUT statement reads and the placement of the input pointer after each iteration of the DATA step. See the option N= on page 121.
Example

The #2 moves the pointer to the second record to read the value for ID from columns 3 and 4:

input name $10. #2 id 3-4;

#numeric-variable

moves the pointer to the record that is given by the value of numeric-variable.

Range  a positive integer

Tip  If the value of numeric-variable is not an integer, SAS truncates the decimal value and uses only the integer value.

#(expression)

moves the pointer to the record that is given by the value of expression.

Range  expression must result in a positive integer.

Tip  If the value of expression is not an integer, SAS truncates the decimal value and uses only the integer value.

/

advances the pointer to column 1 of the next input record.

Example

The values for NAME and AGE are read from the first input record before the pointer moves to the second record to read the value of ID from columns 3 and 4:

input name age / id 3-4;

Format Modifiers for Error Reporting

?  suppresses printing the invalid data note when SAS encounters invalid data values.

See  “How Invalid Data Is Handled” on page 157

??  suppresses printing the messages and the input lines when SAS encounters invalid data values. The automatic variable _ERROR_ is not set to 1 for the invalid observation.

See  “How Invalid Data Is Handled” on page 157

Details

When to Use INPUT

Use the INPUT statement to read raw data from an external file or in-stream data. If your data is stored in an external file, you can specify the file in an INFILE statement. The INFILE statement must execute before the INPUT statement that reads the data records. If your data is in-stream, a DATALINES statement must precede the data lines in the job stream. If your data contains semicolons, use a DATALINES4 statement before the data lines. A DATA step that reads raw data can include multiple INPUT statements.

You can also use the INFILE statement to read in-stream data by specifying a filename of DATALINES in the INFILE statement before the INPUT statement. Using
DATALINES in the INFILE statement enables you to use most of the options available in the INFILE statement with in-stream data.

To read data that is already stored in a SAS data set, use a SET statement. To read database or PC file-format data that is created by other software, use the SET statement after you access the data with the LIBNAME statement. For more information, see SAS/ACCESS for Relational Databases: Reference and SAS/ACCESS Interface to PC Files: Reference.

z/OS Specifics: LOG files that are generated under z/OS and captured with PROC PRINTTO contain an ASA control character in column 1. If you are using the INPUT statement to read a LOG file that was generated under z/OS, you must account for this character if you use column input or column pointer controls.

## Input Styles

### Overview of Input Styles

There are four ways to describe a record’s values in the INPUT statement:

- **column**
- **list (simple and modified)**
- **formatted**
- **named.**

Each variable value is read by using one of these input styles. An INPUT statement can contain any or all of the available input styles, depending on the arrangement of data values in the input records. However, once named input is used in an INPUT statement, you cannot use another input style.

### Column Input

With *column input*, the column numbers follow the variable name in the INPUT statement. These numbers indicate where the variable values are found in the input data records.

```
input name $ 1-8 age 11-12;
```

This INPUT statement can read these data records:

```
----+----1----+----2----+
Peterson 21
Morgan 17
```

Because NAME is a character variable, a $ appears between the variable name and column numbers. For more information, see “INPUT Statement: Column” on page 163.

### List Input

With *list input*, the variable names are simply listed in the INPUT statement. A $ follows the name of each character variable:

```
input name $ age;
```

This INPUT statement can read data values that are separated by blanks or aligned in columns (with at least one blank between):

```
----+----1----+----2----+
Peterson 21
Morgan 17
```
For more information, see “INPUT Statement: List” on page 171.

Formatted Input

With formatted input, an informat follows the variable name in the INPUT statement. The informat gives the data type and the field width of an input value. Informats also enable you to read data that are stored in nonstandard form, such as packed decimal, or numbers that contain special characters such as commas.

```
input name $char8. +2 income comma6.;
```

This INPUT statement reads these data records correctly:

```
----+----1----+----2----+
Peterson  21,000
Morgan    17,132
```

The pointer control of +2 moves the input pointer to the field that contains the value for the variable INCOME. For more information, see “INPUT Statement: Formatted” on page 167.

Named Input

With named input, you specify the name of the variable followed by an equal sign. SAS looks for a variable name and an equal sign in the input record:

```
input name= $ age=;
```

This INPUT statement reads these data records correctly:

```
----+----1----+----2----+
name=Peterson age=21
name=Morgan age=17
```

For more information, see “INPUT Statement: Named” on page 178.

Multiple Styles in a Single INPUT Statement

An INPUT statement can contain any or all of the different input styles:

```
input idno name $18. team $ 25-30 startwght endwght;
```

This INPUT statement reads these data records correctly:

```
----+----1----+----2----+----3----+----4----+
023 David Shaw         red    189 165
049 Amelia Serrano     yellow 189 165
```

The value of IDNO, STARTWGHT, and ENDWGHT are read with list input, the value of NAME with formatted input, and the value of TEAM with column input.

Note: Once named input is used in an INPUT statement, you cannot change input styles.

Pointer Controls

Overview of Pointers

As SAS reads values from the input data records into the input buffer, it keeps track of its position with a pointer. The INPUT statement provides three ways to control the movement of the pointer:
column pointer controls
reset the pointer's column position when the data values in the data records are read.

line pointer controls
reset the pointer's line position when the data values in the data records are read.

line-hold specifiers
hold an input record in the input buffer so that another INPUT statement can process it. By default, the INPUT statement releases the previous record and reads another record.

With column and line pointer controls, you can specify an absolute line number or column number to move the pointer or you can specify a column or line location relative to the current pointer position. This table lists the pointer controls that are available with the INPUT statement.

Table 2.3 Pointer Controls Available in the INPUT Statement

<table>
<thead>
<tr>
<th>Pointer Controls</th>
<th>Relative</th>
<th>Absolute</th>
</tr>
</thead>
<tbody>
<tr>
<td>column pointer controls</td>
<td>+n</td>
<td>@n</td>
</tr>
<tr>
<td></td>
<td>+numeric-variable</td>
<td>@numeric-variable</td>
</tr>
<tr>
<td></td>
<td>+(expression)</td>
<td>@(expression)</td>
</tr>
<tr>
<td></td>
<td>@'character-string'</td>
<td></td>
</tr>
<tr>
<td></td>
<td>@character-variable</td>
<td></td>
</tr>
<tr>
<td></td>
<td>@(character-expression)</td>
<td></td>
</tr>
<tr>
<td>line pointer controls</td>
<td>/</td>
<td>#n</td>
</tr>
<tr>
<td></td>
<td></td>
<td>#numeric-variable</td>
</tr>
<tr>
<td></td>
<td></td>
<td>#(expression)</td>
</tr>
<tr>
<td>line-hold specifiers</td>
<td>@</td>
<td>(not applicable)</td>
</tr>
<tr>
<td></td>
<td>@@</td>
<td>(not applicable)</td>
</tr>
</tbody>
</table>

Note: Always specify pointer controls before the variable to which they apply.

You can use the COLUMN= and LINE= options in the INFILE statement to determine the pointer's current column and line location.

Using Column and Line Pointer Controls
Column pointer controls indicate the column in which an input value starts.
Use line pointer controls within the INPUT statement to move to the next input record or to define the number of input records per observation. Line pointer controls specify which input record to read. To read multiple data records into the input buffer, use the N= option in the INFILE statement to specify the number of records. If you omit N=, you need to take special precautions. For more information, see “Reading More Than One Record per Observation” on page 156.

Using Line-Hold Specifiers

Line-hold specifiers keep the pointer on the current input record when

- a data record is read by more than one INPUT statement (trailing @)
- one input line has values for more than one observation (double trailing @)
- a record needs to be reread on the next iteration of the DATA step (double trailing @).

Use a single trailing @ to allow the next INPUT statement to read from the same record. Use a double trailing @ to hold a record for the next INPUT statement across iterations of the DATA step.

Normally, each INPUT statement in a DATA step reads a new data record into the input buffer. When you use a trailing @, the following occurs:

- The pointer position does not change.
- No new record is read into the input buffer.
- The next INPUT statement for the same iteration of the DATA step continues to read the same record rather than a new one.

SAS releases a record held by a trailing @ when

- a null INPUT statement executes:
  ```
  input;
  ```
- an INPUT statement without a trailing @ executes
- the next iteration of the DATA step begins.

Normally, when you use a double trailing @ (@@), the INPUT statement for the next iteration of the DATA step continues to read the same record. SAS releases the record that is held by a double trailing @

- immediately if the pointer moves past the end of the input record
- immediately if a null INPUT statement executes:
  ```
  input;
  ```
- when the next iteration of the DATA step begins if an INPUT statement with a single trailing @ executes later in the DATA step:
  ```
  input @;
  ```

Pointer Location After Reading

Understanding the location of the input pointer after a value is read is important, especially if you combine input styles in a single INPUT statement. With column and formatted input, the pointer reads the columns that are indicated in the INPUT statement and stops in the next column. With list input, however, the pointer scans data records to locate data values and reads a blank to indicate that a value has ended. After reading a value with list input, the pointer stops in the second column after the value.
For example, you can read these data records with list, column, and formatted input:

```
----+----1----+----2----+----3
REGION1    49670
REGION2    97540
REGION3    86342
```

This INPUT statement uses list input to read the data records:

```
input region $ jansales;
```

After reading a value for REGION, the pointer stops in column 9.

```
----+----1----+----2----+----3
REGION1    49670
↑
```

These INPUT statements use column and formatted input to read the data records:

- **column input**
  
  ```
  input region $ 1-7 jansales 12-16;
  ```

- **formatted input**
  
  ```
  input region $7. +4 jansales 5.;
  input region $7. @12 jansales 5.;
  ```

To read a value for the variable REGION, the INPUT statements instruct the pointer to read seven columns and stop in column 8.

```
----+----1----+----2----+----3
REGION1    49670
↑
```

**Reading More Than One Record per Observation**

**Using the # Pointer Control**

The highest number that follows the # pointer control in the INPUT statement determines how many input data records are read into the input buffer. Use the N= option in the INFILE statement to change the number of records. For example, in this statement, the highest value after the # is 3:

```
input @31 age 3. #3 id 3-4 #2 @6 name $20.;
```

Unless you use N= in the associated INFILE statement, the INPUT statement reads three input records each time the DATA step executes.

When each observation has multiple input records but values from the last record are not read, you must use a # pointer control in the INPUT statement or N= in the INFILE statement to specify the last input record. For example, if there are four records per observation, but only values from the first two input records are read, use this INPUT statement:

```
input name $ 1-10 #2 age 13-14 #4;
```

When you have advanced to the next record with the / pointer control, use the # pointer control in the INPUT statement or the N= option in the INFILE statement to set the number of records that are read into the input buffer. To move the pointer back to an earlier record, use a # pointer control. For example, this statement requires the #2 pointer control, unless the INFILE statement uses the N= option, to read two records:

```
input a / b #1 @52 c #2;
```
The INPUT statement assigns A a value from the first record. The pointer advances to the next input record to assign B a value. Then the pointer returns from the second record to column 1 of the first record and moves to column 52 to assign C a value. The #2 pointer control identifies two input records for each observation so that the pointer can return to the first record for the value of C.

If the number of input records per observation varies, use the N= option in the INFILE statement to give the maximum number of records per observation. For more information, see N= option on page 121.

Reading Past the End of a Line

When you use @ or + pointer controls with a value that moves the pointer to or past the end of the current record and the next value is to be read from the current column, SAS goes to column 1 of the next record to read it. It also writes this message to the SAS log:

NOTE: SAS went to a new line when INPUT statement reached past the end of a line.

You can alter the default behavior (the FLOWOVER option) in the INFILE statement. Use the STOPOVER option in the INFILE statement to treat this condition as an error and to stop building the data set.

Use the MISSOVER option in the INFILE statement to set the remaining INPUT statement variables to missing values if the pointer reaches the end of a record.

Use the TRUNCOVER option in the INFILE statement to read column input or formatted input when the last variable that is read by the INPUT statement contains varying-length data.

Positioning the Pointer before the Record

When a column pointer control tries to move the pointer to a position before the beginning of the record, the pointer is positioned in column 1. For example, this INPUT statement specifies that the pointer is located in column −2 after the first value is read:

```sas
data test;
  input a @(a-3) b;
  datalines;
  2
; 
```

Therefore, SAS moves the pointer to column 1 after the value of A is read. Both variables A and B contain the same value.

How Invalid Data Is Handled

When SAS encounters an invalid character in an input value for the variable indicated, it

- sets the value of the variable that is being read to missing or the value that is specified with the INVALIDDATA= system option. For more information, see “INVALIDDATA= System Option” in SAS System Options: Reference.
- prints an invalid data note in the SAS log.
prints the input line and column number that contains the invalid value in the SAS log. Unprintable characters appear in hexadecimal. To help determine column numbers, SAS prints a rule line above the input line.

sets the automatic variable _ERROR_ to 1 for the current observation.

The format modifiers for error reporting control the amount of information that is printed in the SAS log. Both the ? and ?? modifiers suppress the invalid data message. However, the ?? modifier also resets the automatic variable _ERROR_ to 0. For example, these two sets of statements are equivalent:

```
input x ?? 10-12;
input x ? 10-12;
_error_=0;
```

In either case, SAS sets invalid values of X to missing values. For information about the causes of invalid data, see *SAS Language Reference: Concepts*.

End-of-File

End-of-file occurs when an INPUT statement reaches the end of the data. If a DATA step tries to read another record after it reaches an end-of-file, then execution stops. If you want the DATA step to continue to execute, use the END= or EOF= option in the INFILE statement. Then you can write SAS program statements to detect the end-of-file, and to stop the execution of the INPUT statement but continue with the DATA step. For more information, see “INFILE Statement” on page 110.

Arrays

The INPUT statement can use array references to read input data values. You can use an array reference in a pointer control if it is enclosed in parentheses. See “Example 6: Positioning the Pointer with a Character Variable” on page 161.

Use the array subscript asterisk (*) to read in all elements of a previously defined explicit array. SAS allows single or multidimensional arrays. Enclose the subscript in braces, brackets, or parentheses. Here is the form of this statement:

```
input array-name{*};
```

You can use arrays with list, column, or formatted input. However, you cannot read in values to an array that is defined with _TEMPORARY_ and that uses the asterisk subscript. For example, these statements create variables X1 through X100 and assign data values to the variables using the 2. informat:

```
array x{100};
input x{*} 2.;
```

Comparisons

- The INPUT statement reads raw data in external files or data lines that are entered in-stream (following the DATALINES statement) that need to be described to SAS. The SET statement reads a SAS data set, which already contains descriptive information about the data values.
- The INPUT statement reads data while the PUT statement writes data values, text strings, or both to the SAS log or to an external file.
- The INPUT statement can read data from external files; the INFILE statement points to that file and has options that control how that file is read.
Examples:

Example 1: Using Multiple Styles of Input in One INPUT Statement

This example uses several input styles in a single INPUT statement:

```sas
data club1;
  input Idno Name $18.
       Team $ 25-30 Startwght Endwght;
  datalines;
  023 David Shaw         red    189 165
  049 Amelia Serrano     yellow 189 165
  ... more data lines ... 
; 
```

This table identifies the type of input styles.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type of Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>Idno, Startwght, Endwght</td>
<td>list input</td>
</tr>
<tr>
<td>Name</td>
<td>formatted input</td>
</tr>
<tr>
<td>Team</td>
<td>column input</td>
</tr>
</tbody>
</table>

Example 2: Using a Null INPUT Statement

This example uses an INPUT statement with no arguments. The DATA step copies records from the input file to the output file without creating any SAS variables:

```sas
data _null_;  
  infile file-specification-1;  
  file file-specification-2;  
  input;  
  put _infile_;  
run; 
```

Example 3: Holding a Record in the Input Buffer

This example reads a file that contains two types of input data records and creates a SAS data set from these records. One type of data record contains information about a particular college course. The second type of record contains information about the students enrolled in the course. You need two INPUT statements to read the two records and to assign the values to different variables that use different formats. Records that contain class information have a C in column 1; records that contain student information have an S in column 1, as shown here:

```
----+----1----+----2----+
C HIST101 Watson
S Williams 0459
S Flores   5423
C MATH202 Sen
S Lee      7085  
```
To know which INPUT statement to use, check each record as it is read. Use an INPUT statement that reads only the variable that tells whether the record contains class or student.

```
data schedule(drop=type);
  retain Course Professor;
  input type $1. @;
  if type='C' then
    input course $ professor $;
  else if type='S' then
    do;
      input Name $10. Id;
      output schedule;
    end;
  datalines;
  C HIST101 Watson
  S Williams 0459
  S Flores   5423
  C MATH202 Sen
  S Lee      7085
; run;
```

The first INPUT statement reads the TYPE value from column 1 of every line. Because this INPUT statement ends with a trailing @, the next INPUT statement in the DATA step reads the same line. The IF-THEN statements that follow check whether the record is a class or student line before another INPUT statement reads the rest of the line. The INPUT statements without a trailing @ release the held line. The RETAIN statement saves the values about the particular college course. The DATA step writes an observation to the SCHEDULE data set after a student record is read.

This output that PROC PRINT generates shows the resulting data set SCHEDULE.

**Output 2.11 Data Set Schedule**

<table>
<thead>
<tr>
<th>Obs</th>
<th>Course</th>
<th>Professor</th>
<th>Name</th>
<th>Id</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>HIST101</td>
<td>Watson</td>
<td>Williams</td>
<td>459</td>
</tr>
<tr>
<td>2</td>
<td>HIST101</td>
<td>Watson</td>
<td>Flores</td>
<td>5423</td>
</tr>
<tr>
<td>3</td>
<td>MATH202</td>
<td>Sen</td>
<td>Lee</td>
<td>7085</td>
</tr>
</tbody>
</table>

Example 4: Holding a Record across Iterations of the DATA Step

This example shows how to create multiple observations for each input data record. Each record contains several NAME and AGE values. The DATA step reads a NAME value and an AGE value, writes an observation, and then reads another set of NAME and AGE values to output, and so on, until all the input values in the record are processed.

```
data test;
```
input name $ age @@;
datalines;
John 13 Monica 12 Sue 15 Stephen 10
Marc 22 Lily 17
;

The INPUT statement uses the double trailing @ to control the input pointer across iterations of the DATA step. The SAS data set contains six observations.

Example 5: Positioning the Pointer with a Numeric Variable

This example uses a numeric variable to position the pointer. A raw data file contains records with the employment figures for several offices of a multinational company. The input data records are

<table>
<thead>
<tr>
<th></th>
<th>Employment</th>
<th>Region</th>
<th>Office Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>New York</td>
<td>USA</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>Cary</td>
<td>USA</td>
<td>2274</td>
</tr>
<tr>
<td>3</td>
<td>Chicago</td>
<td>USA</td>
<td>37</td>
</tr>
<tr>
<td>22</td>
<td>Tokyo</td>
<td>ASIA</td>
<td>80</td>
</tr>
<tr>
<td>5</td>
<td>Vancouver</td>
<td>CANADA</td>
<td>6</td>
</tr>
<tr>
<td>9</td>
<td>Milano</td>
<td>EUROPE</td>
<td>123</td>
</tr>
</tbody>
</table>

The first column has the column position for the office location. The next numeric column is the region category. The geographic region occurs before the number of employees in that office.

You determine the office location by combining the @numeric-variable pointer control with a trailing @. To read the records, use two INPUT statements. The first INPUT statement obtains the value for the @numeric-variable pointer control. The second INPUT statement uses this value to determine the column that the pointer moves to.

```sas
data office (drop=x);
  infile file-specification;
  input x @;
  if 1<=x<=10 then
    input @x City $9.;
  else do;
    put 'Invalid input at line ' _n_;
    delete;
  end;
run;
```

The DATA step writes only five observations to the OFFICE data set. The fourth input data record is invalid because the value of X is greater than 10. Therefore, the second INPUT statement does not execute. Instead, the PUT statement writes a message to the SAS log and the DELETE statement stops processing the observation.

Example 6: Positioning the Pointer with a Character Variable

This example uses character variables to position the pointer. The OFFICE data set, created in “Example 5: Positioning the Pointer with a Numeric Variable” on page 161, contains a character variable CITY whose values are the office locations. Suppose you discover that you need to read additional values from the raw data file. By using another DATA step, you can combine the @character-variable pointer control with a trailing @ and the @character-expression pointer control to locate the values.
If the observations in OFFICE are still in the order of the original input data records, you can use this DATA step:

```sas
data office2;
  set office;
  infile file-specification;
  array region {5} $ _temporary_ ('USA' 'CANADA' 'SA' 'EUROPE' 'ASIA');
  input @city Location : 2. @;
  input @(trim(region{location})) Population : 4.;
run;
```

The ARRAY statement assigns initial values to the temporary array elements. These elements correspond to the geographic regions of the office locations. The first INPUT statement uses an @character-variable pointer control. Each record is scanned for the series of characters in the value of CITY for that observation. Then the value of LOCATION is read from the next non-blank column. LOCATION is a numeric category for the geographic region of an office. The second INPUT statement uses an array reference in the @character-expression pointer control to determine the location POPULATION in the input records. The expression also uses the TRIM function to trim trailing blanks from the character value. In this way an exact match is found between the character string in the input data and the value of the array element.

This output that PROC PRINT generates shows the resulting data set OFFICE2.

**Output 2.12 Data Set OFFICE2**

<table>
<thead>
<tr>
<th>Obs</th>
<th>City</th>
<th>Location</th>
<th>Population</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>New York</td>
<td>1</td>
<td>14</td>
</tr>
<tr>
<td>2</td>
<td>Cary</td>
<td>1</td>
<td>2274</td>
</tr>
<tr>
<td>3</td>
<td>Chicago</td>
<td>1</td>
<td>37</td>
</tr>
<tr>
<td>4</td>
<td>Tokyo</td>
<td>5</td>
<td>80</td>
</tr>
<tr>
<td>5</td>
<td>Vancouver</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>Milano</td>
<td>4</td>
<td>123</td>
</tr>
</tbody>
</table>

Example 7: Moving the Pointer Backward

This example shows several ways to move the pointer backward.

- This INPUT statement uses the @ pointer control to read a value for BOOK starting at column 26. Then the pointer moves back to column 1 on the same line to read a value for COMPANY:

  ```sas
  input @26 book $ @1 company;
  ```

- These INPUT statements use `+numeric-variable` or `+(expression)` to move the pointer backward one column. These two sets of statements are equivalent.

  ```sas
  m=-1;
  input x 1-10 +m y 2.;
  input x 1-10 +{-1} y 2.;
  ```
INPUT Statement: Column

Reads input values from specified columns and assigns the values to the corresponding SAS variables.

Valid in: DATA step
Category: File-Handling
Type: Executable
Restriction: This statement is not supported in a DATA step that runs in CAS.

Syntax

```
INPUT variable <$> start-column<--end-column>
<.decimals> <@ | @@>;
```

Arguments

- **variable** specifies a variable that is assigned input values.
- **$** indicates that the variable has character values rather than numeric values.
  
  **Tip** If the variable is previously defined as character, $ is not required.

- **start-column** specifies the first column of the input record that contains the value to read.

- **end-column** specifies the last column of the input record that contains the value to read.
  
  **Tip** If the variable value occupies only one column, omit end-column.

Example

Because end-column is omitted, the values for the character variable GENDER occupy only column 16:

```
input name $ 1-10 pulse 11-13 waist 14-15 gender $ 16;
```

- **.decimals** specifies the power of 10 by which to divide the value. If the data contains decimal points, the .decimals value is ignored.
Tip

An explicit decimal point in the input value overrides a decimal specification in the INPUT statement.

Examples

“Example 2: Read Individual Input Data Using Decimals” on page 165
“Example 3: Read Input Records Using Decimals” on page 166

@

holds the input record for the execution of the next INPUT statement within the same iteration of the DATA step. This line-hold specifier is called trailing @.

Restriction

The trailing @ must be the last item in the INPUT statement.

Tip

The trailing @ prevents the next INPUT statement from automatically releasing the current input record and reading the next record into the input buffer. It is useful when you need to read from a record multiple times.

See

“Pointer Controls” on page 153

@@

holds the input record for the execution of the next INPUT statement across iterations of the DATA step. This line-hold specifier is called double trailing @.

Restriction

The double trailing @ must be the last item in the INPUT statement.

Tip

The double trailing @ is useful when each input line contains values for several observations.

See

“Using Line-Hold Specifiers” on page 155

Details

When to Use Column Input

With column input, the column numbers that contain the value follow a variable name in the INPUT statement. To read with column input, data values must have these attributes:

- appear in the same columns in all the input data records
- consist of standard numeric form or character form

Column input has these useful features:

- Character values can contain embedded blanks.
- Character values can be from 1 to 32,767 characters long.
- Input values can be read in any order, regardless of their position in the record.
- Values or parts of values can be read multiple times. For example, this INPUT statement reads an ID value in columns 10 through 15, and then reads a GROUP value from column 13:

  input id 10-15 group 13;

---

Both leading and trailing blanks within the field are ignored. Therefore, if numeric values contain blanks that represent 0s or if you want to retain leading and trailing blanks in character values, read the value with an informat. For more information, see “INPUT Statement: Formatted” on page 167.

Missing Values

Missing data does not require a placeholder. The INPUT statement interprets a blank field as missing and reads other values correctly. If a numeric or character field contains a single period, the variable value is set to missing.

Reading Data Lines

SAS always pads the data records that follow the DATALINES statement (in-stream data) to a fixed length in multiples of 80. The CARDIMAGE system option determines whether to read or truncate data past the 80th column.

Reading Variable-Length Records

By default, SAS uses the FLOWOVER option to read varying-length data records. If the record contains fewer values than expected, the INPUT statement reads the values from the next data record. To read varying-length data, you might need to use the TRUNCOVER option in the INFILE statement. The TRUNCOVER option is more efficient than the PAD option, which pads the records to a fixed length. For more information, see “Reading Past the End of a Line” on page 130.

Examples:

Example 1: Read Input Records with Column Input

This DATA step shows how to read input data records with column input:

```
data scores;
  input name $ 1-18 score1 25-27 score2 30-32 score3 35-37;
datalines;
Joseph                  11   32   76
Mitchel                 13   29   82
Sue Ellen               14   27   74
;
```

Example 2: Read Individual Input Data Using Decimals

This INPUT statement reads the input data for a numeric variable using two decimal places:

```
input number 1-5 .2;
```

<table>
<thead>
<tr>
<th>Input Data</th>
<th>Statement</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>2314</td>
<td>input number 1-5 .2;</td>
<td>23.14</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>.02</td>
</tr>
</tbody>
</table>
The decimal specification in the INPUT statement is overridden by the input data value.

Example 3: Read Input Records Using Decimals

This DATA step uses the .decimals argument to read values from input lines and insert a decimal place into data that does not have an explicit decimal already defined. Data that contains an explicit decimal is not changed, but the data is padded to match the greatest number of significant digits that occur in any of the output data after conversion.

```sas
data product1 noobs;
  input partcost 1-5 .2;
  datalines;
6857
21
400
3.2
12.56
;
proc print data=product1 noobs; run;
```

<table>
<thead>
<tr>
<th>Input Data</th>
<th>Statement</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>400</td>
<td></td>
<td>4.00</td>
</tr>
<tr>
<td>-140</td>
<td></td>
<td>-1.40</td>
</tr>
<tr>
<td>12.236</td>
<td></td>
<td>12.23</td>
</tr>
<tr>
<td>12.2</td>
<td></td>
<td>12.2</td>
</tr>
</tbody>
</table>

See Also

Statements:

- "INPUT Statement" on page 146
INPUT Statement: Formatted

Reads input values with specified informats and assigns them to the corresponding SAS variables.

Valid in: DATA step
Category: File-Handling
Type: Executable
Restriction: This statement is not supported in a DATA step that runs in CAS.

Syntax

```
INPUT <pointer-control> variable informat. <@ | @@>;
INPUT <pointer-control> (variable-list) (informat-list) <@ | @@>;
INPUT <pointer-control> (variable-list) (<n*> informat.) <@ | @@>;
```

Arguments

- **pointer-control**
  moves the input pointer to a specified line or column in the input buffer.

  See “Column Pointer Controls” on page 148 and “Line Pointer Controls” on page 150

- **variable**
  specifies a variable that is assigned input values.

  Requirement The (variable-list) is followed by an (informat-list).

  Example “Example 1: Formatted Input with Pointer Controls” on page 170

- **(variable-list)**
  specifies a list of variables that are assigned input values.

  See “How to Group Variables and Informats” on page 169

  Example “Example 2: Using Informat Lists” on page 170

- **informat.**
  specifies a SAS informat to use to read the variable values.

  Tip Decimal points in the actual input values override decimal specifications in a numeric informat.

  See SAS Informs in *SAS Formats and Informats: Reference*

  Example “Example 1: Formatted Input with Pointer Controls” on page 170
(informat-list) specifies a list of informats to use to read the values for the preceding list of variables.

In the INPUT statement, (informat-list) can include:


ingformat. specifies an informat to use to read the variable values.

pointer-control specifies one of these pointer controls to use to position a value: @, #, /, or +.

n* specifies to repeat n times the next informat in an informat list.

Example This statement uses the 7.2 informat to read GRADES1, GRADES2, and GRADES3 and the 5.2 informat to read GRADES4 and GRADES5:

```
input {grades1-grades5}(3*7.2, 2*5.2);
```

Restriction The (informat-list) must follow the (variable-list).

See "How to Group Variables and Informats" on page 169

Example "Example 2: Using Informat Lists" on page 170

@@ holds an input record for the execution of the next INPUT statement across iterations of the DATA step. This line-hold specifier is called double trailing @.

Restriction The double trailing @ must be the last item in the INPUT statement.

Tip The double trailing @ is useful when each input line contains values for several observations.

See "Using Line-Hold Specifiers" on page 155

Details When to Use Formatted Input

With formatted input, an informat follows a variable name and defines how SAS reads the values of this variable. An informat gives the data type and the field width of an input value. Informats also read data that is stored in nonstandard form, such as packed decimal, or numbers that contain special characters such as commas.
Simple formatted input requires that the variables be in the same order as their corresponding values in the input data. You can use pointer controls to read variables in any order. For more information, see “INPUT Statement” on page 146.

Missing Values

Generally, SAS represents missing values in formatted input with a single period for a numeric value and with blanks for a character value. The informat that you use with formatted input determines how SAS interprets a blank. For example, $CHAR.w reads the blanks as part of the value, whereas BZ.w converts a blank to zero.

Reading Variable-Length Records

By default, SAS uses the FLOWOVER option to read varying-length data records. If the record contains fewer values than expected, the INPUT statement reads the values from the next data record. To read varying-length data, you might need to use the TRUNCOVER option in the INFILE statement. For more information, see “Reading Past the End of a Line” on page 130.

How to Group Variables and Informats

When the input values are arranged in a pattern, you can group the informat list. A grouped informat list consists of two lists:

- the names of the variables to read enclosed in parentheses
- the corresponding informats separated by either blanks or commas and enclosed in parentheses.

Informat lists can make an INPUT statement shorter because the informat list is recycled until all variables are read and the numbered variable names can be used in abbreviated form. Using informat lists avoids listing the individual variables.

For example, if the values for the five variables SCORE1 through SCORE5 are stored as four columns per value without intervening blanks, this INPUT statement reads the values.

```
input (score1-score5) (4. 4. 4. 4. 4.);
```

However, if you specify more variables than informats, the INPUT statement reuses the informat list to read the remaining variables. Here is a shorter version of the previous statement.

```
input (score1-score5) (4.);
```

You can use as many informat lists as necessary in an INPUT statement, but do not nest the informat lists. After all the values in the variable list are read, the INPUT statement ignores any directions that remain in the informat list. For an example, see “Example 3: Including More Informat Specifications Than Necessary” on page 171.

The n* modifier in an informat list specifies to repeat the next informat n times. Here is an example.

```
input (name score1-score5) ($10. 5*4.);
```

How to Store Informats

The informats that you specify in the INPUT statement are not stored with the SAS data set. Informats that you specify with the INFORMAT or ATTRIB statement are permanently stored. Therefore, you can read a data value with a permanently stored informat in a later DATA step without having to specify the informat or use PROC FSEDIT to enter data in the correct format.

Comparisons

When a variable is read with formatted input, the pointer movement is similar to the pointer movement of column input. The pointer moves the length that the informat specifies and stops at the next column. To read data with informats that are not aligned in columns, use modified list input. Using modified list input enables you to take advantage of the scanning feature in list input. See “When to Use List Input” on page 173.

Examples:

Example 1: Formatted Input with Pointer Controls

This INPUT statement uses informats and pointer controls:

```sas
data sales;
  infile file-specification;
  input item $10. +5 jan comma5. +5 feb comma5. +5 mar comma5.;
run;
```

It can read these input data records:

```
---+----1----+----2----+----3----+----4
trucks         1,382     2,789     3,556
vans           1,265     2,543     3,987
sedans         2,391     3,011     3,658
```

The value for ITEM is read from the first 10 columns in a record. The pointer stops in column 11. The trailing blanks are discarded and the value of ITEM is written to the program data vector. Next, the pointer moves five columns to the right before the INPUT statement uses the COMMA5. informat to read the value of JAN. This informat uses five as the field width to read numeric values that contain a comma. Once again, the pointer moves five columns to the right before the INPUT statement uses the COMMA5. informat to read the values of FEB and MAR.

Example 2: Using Informat Lists

This INPUT statement uses the character informat $10. to read the values of the variable NAME and uses the numeric informat 4. to read the values of the five variables SCORE1 through SCORE5:

```sas
data scores;
  input (name score1-score5) ($10. 5*4.);
datalines;
  Whittaker 121 114 137 156 142
  Smythe    111 97 122 143 127
;```

Example 3: Including More Informat Specifications Than Necessary

This informat list includes more specifications than are necessary when this INPUT statement executes:

```sas
data test;
  input (x y z) (2.,+1);
datalines;
  2 24 36
  0 20 30
  ;
```

The INPUT statement reads the value of X with the 2. informat. Then, the +1 column pointer control moves the pointer forward one column. Next, the value of Y is read with the 2. informat. Again, the +1 column pointer moves the pointer forward one column. Then, the value of Z is read with the 2. informat. For the third iteration, the INPUT statement ignores the +1 pointer control.

See Also

**Statements:**
- “INPUT Statement” on page 146
- “INPUT Statement: List” on page 171

---

**INPUT Statement: List**

Scans the input data record for input values and assigns them to the corresponding SAS variables.

**Valid in:** DATA step

**Category:** File-Handling

**Type:** Executable

**Restriction:** This statement is not supported in a DATA step that runs in CAS.

**Syntax**

```
INPUT <pointer-control> variable <$> <&> <@ | @@>;
INPUT <pointer-control> variable <: | & | ->
  <informat.> <@ | @@>;
```

**Arguments**

`pointer-control`

moves the input pointer to a specified line or column in the input buffer.

See  “Column Pointer Controls” on page 148 and “Line Pointer Controls” on page 150
variable

specifies a variable that is assigned input values.

$ indicates to store a variable value as a character value rather than as a numeric value.

Tip If the variable is previously defined as character, $ is not required.

Example “Example 1: Reading Unaligned Data with Simple List Input” on page 176

& indicates that a character value can have one or more single embedded blanks. This format modifier reads the value from the next non-blank column until the pointer reaches two consecutive blanks, the defined length of the variable, or the end of the input line, whichever comes first.

Restriction The & modifier must follow the variable name and $ sign that it affects.

Tip If you specify an informat after the & modifier, the terminating condition for the format modifier remains two blanks.

See "Modified List Input " on page 174

Example “Example 2: Reading Character Data That Contains Embedded Blanks” on page 176

: enables you to specify an informat that the INPUT statement uses to read the variable value. For a character variable, this format modifier reads the value from the next non-blank column until the pointer reaches the next blank column, the defined length of the variable, or the end of the data line, whichever comes first. For a numeric variable, this format modifier reads the value from the next non-blank column until the pointer reaches the next blank column or the end of the data line, whichever comes first.

Tips If the length of the variable has not been previously defined, then its value is read and stored with the informat length.

The pointer continues to read until the next blank column is reached. However, if the field is longer than the formatted length, then the value is truncated to the length of variable.

See "Modified List Input " on page 174

Examples “Example 3: Reading Unaligned Data with Informats” on page 176

“Example 5: Reading Delimited Data with Modified List Input” on page 177

~ indicates to treat single quotation marks, double quotation marks, and delimiters in character values in a special way. This format modifier reads delimiters within
quoted character values as characters instead of as delimiters and retains the quotation marks when the value is written to a variable.

**Restriction**  You must use the DSD option in an INFILE statement. Otherwise, the INPUT statement ignores this option.

**See**  "Modified List Input “ on page 174

**Example**  "Example 5: Reading Delimited Data with Modified List Input" on page 177

**informat.**  specifies an informat to use to read the variable values.

**Tip**  Decimal points in the actual input values always override decimal specifications in a numeric informat.

**See**  SAS Informs in SAS Formats and Informats: Reference

**Examples**  “Example 3: Reading Unaligned Data with Informats” on page 176

"Example 5: Reading Delimited Data with Modified List Input” on page 177

@@ holds an input record for the execution of the next INPUT statement across iterations of the DATA step. This line-hold specifier is called double trailing @.

**Restriction**  The double trailing @ must be the last item in the INPUT statement.

**Tip**  The double trailing @ is useful when each input line contains values for several observations.

**See**  "Using Line-Hold Specifiers” on page 155

Details

**When to Use List Input**

List input requires that you specify the variable names in the INPUT statement in the same order that the fields appear in the input data records. SAS scans the data line to locate the next value but ignores additional intervening blanks. List input does not require that the data is located in specific columns. However, you must separate each value from the next by at least one blank unless the delimiter between values is changed. By default, the delimiter for data values is one blank space or the end of
the input record. List input does not skip over any data values to read subsequent values, but it can ignore all values after a given point in the data record. However, pointer controls enable you to change the order that the data values are read.

There are two types of list input:
- simple list input
- modified list input.

Modified list input makes the INPUT statement more versatile because you can use a format modifier to overcome several of the restrictions of simple list input. See “Modified List Input” on page 174.

Simple List Input

Simple list input places several restrictions on the type of data that the INPUT statement can read:
- By default, at least one blank must separate the input values. Use the DLM= or DLMSTR= option or the DSD option in the INFILE statement to specify a delimiter other than a blank.
- Represent each missing value with a period, not a blank, or two adjacent delimiters.
- Character input values cannot be longer than 8 bytes unless the variable is given a longer length in an earlier LENGTH, ATTRIB, or INFORMAT statement.
- Character values cannot contain embedded blanks unless you change the delimiter.
- Data must be in standard numeric or character format.

1

Modified List Input

List input is more versatile when you use format modifiers. The format modifiers are as follows:

<table>
<thead>
<tr>
<th>Format Modifier</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>reads character values that contain embedded blanks.</td>
</tr>
<tr>
<td>:</td>
<td>reads data values that need the additional instructions that informsats can provide but that are not aligned in columns.¹</td>
</tr>
<tr>
<td>~</td>
<td>reads delimiters within quoted character values as characters and retains the quotation marks.</td>
</tr>
</tbody>
</table>

¹ Use formatted input and pointer controls to quickly read data values that are aligned in columns.

For example, use the : modifier with an informat to read character values that are longer than 8 bytes or numeric values that contain nonstandard values.

Because list input interprets a blank as a delimiter, use modified list input to read values that contain blanks. The & modifier reads character values that contain single embedded blanks. However, the data values must be separated by two or more

¹ See SAS Language Reference: Concepts for the information about standard and nonstandard data values.
blanks. To read values that contain leading, trailing, or embedded blanks with list input, use the DLM= or DLMSTR= option in the INFILE statement to specify another character as the delimiter. See “Example 5: Reading Delimited Data with Modified List Input” on page 177. If your input data uses blanks as delimiters and it contains leading, trailing, or embedded blanks, you might need to use either column input or formatted input. If quotation marks surround the delimited values, you can use list input with the DSD option in the INFILE statement.

Comparisons

How Modified List Input and Formatted Input Differ

Modified list input has a scanning feature that can use informats to read data which is not aligned in columns. Formatted input causes the pointer to move like that of column input to read a variable value. The pointer moves the length that is specified in the informat and stops at the next column.

This DATA step uses modified list input to read the first data value and formatted input to read the second:

```plaintext
data jansales;
  input item : $10. amount comma5.;
datalines;
  trucks 1,382
  vans 1,235
  sedans 2,391
;```

The value of ITEM is read with modified list input. The INPUT statement stops reading when the pointer finds a blank space. The pointer then moves to the second column after the end of the field, which is the correct position to read the AMOUNT value with formatted input.

Formatted input, on the other hand, continues to read the entire width of the field. This INPUT statement uses formatted input to read both data values:

```plaintext
input item $10. +1 amount comma5.;
```

To read this data correctly with formatted input, the second data value must occur after the 10th column of the first value, as shown here:

```
----------1---------2
trucks    1,382
vans      1,235
sedans    2,391
```

Also, after the value of ITEM is read with formatted input, you must use the pointer control +1 to move the pointer to the column where the value AMOUNT begins.

When Data Contains Quotation Marks

When you use the DSD option in an INFILE statement, which sets the delimiter to a comma, the INPUT statement removes quotation marks before a value is written to a variable. When you also use the tilde (~) modifier in an INPUT statement, the INPUT statement maintains quotation marks as part of the value.
Examples:

Example 1: Reading Unaligned Data with Simple List Input

The INPUT statement in this DATA step uses simple list input to read the input data records:

```sas
data scores;
  input name $ score1 score2 score3 team $;
datalines;
Joe 11 32 76 red
Mitchel 13 29 82 blue
Susan 14 27 74 green
;
```

The next INPUT statement reads only the first four fields in the previous data lines, which demonstrates that you are not required to read all the fields in the record:

```sas
input name $ score1 score2 score3;
```

Example 2: Reading Character Data That Contains Embedded Blanks

The INPUT statement in this DATA step uses the & format modifier with list input to read character values that contain embedded blanks.

```sas
data list;
  infile file-specification;
  input name $ & score;
run;
```

It can read these input data records:

```
Joseph   11 Joergensen  red
Mitchel  13 Mc Allister  blue
Su Ellen  14 Fischer-Simon  green
```

The & modifier follows the variable that it affects in the INPUT statement. Because this format modifier follows NAME, at least two blanks must separate the NAME field from the SCORE field in the input data records.

You can also specify an informat with a format modifier, as shown here:

```sas
input name $ & +3 lastname & $15. team $;
```

In addition, this INPUT statement reads the same data to demonstrate that you are not required to read all the values in an input record. The +3 column pointer control moves the pointer past the score value in order to read the value for LASTNAME and TEAM.

Example 3: Reading Unaligned Data with Informats

This DATA step uses modified list input to read data values with an informat:

```sas
data jansales;
  input item : $10. amount;
  datalines;
  trucks 1382
  vans 1235
  sedans 2391
;
The $10. informat allows a character variable of up to ten characters to be read.

Example 4: Reading Comma-Delimited Data with List Input and an Informat

This DATA step uses the DELIMITER= option in the INFILE statement to read list input values that are separated by commas instead of blanks. The example uses an informat to read the date, and a format to write the date.

```plaintext
data scores2;
  length Team $ 14;
  infile datalines delimiter=',';
  input Name $ Score1-Score3 Team $ Final_Date:MMDDYY10.;
  format final_date weekdate17.;
  datalines;
  Mitchell,13,29,82,Blue Bunnies,4/5/2007
  Susan,14,27,74,Green Gazelles,11/13/2007
;
  proc print data=scores2;
    var Name Team Score1-Score3 Final_Date;
    title 'Soccer Player Scores';
  run;
```

Output 2.13  Output from Comma-Delimited Data

<table>
<thead>
<tr>
<th>Obs</th>
<th>Name</th>
<th>Team</th>
<th>Score1</th>
<th>Score2</th>
<th>Score3</th>
<th>Final_Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Joe</td>
<td>Red Racers</td>
<td>11</td>
<td>32</td>
<td>76</td>
<td>Sat, Feb 3, 2007</td>
</tr>
<tr>
<td>2</td>
<td>Mitchell</td>
<td>Blue Bunnies</td>
<td>13</td>
<td>29</td>
<td>82</td>
<td>Thu, Apr 5, 2007</td>
</tr>
<tr>
<td>3</td>
<td>Susan</td>
<td>Green Gazelles</td>
<td>14</td>
<td>27</td>
<td>74</td>
<td>Tue, Nov 13, 2007</td>
</tr>
</tbody>
</table>

Example 5: Reading Delimited Data with Modified List Input

This DATA step uses the DSD option in an INFILE statement and the tilde (~) format modifier in an INPUT statement to retain the quotation marks in character data and to read a character in a string that is enclosed in quotation marks as a character instead of as a delimiter.

```plaintext
data scores;
  infile datalines dsd;
  input Name : $9. Score1-Score3 Team ~ $25. Div $;
  datalines;
  Mitchell,13,29,82,"Blue Bunnies, Richmond",AAA
  Sue Ellen,14,27,74,"Green Gazelles, Atlanta",AA
;
  proc print; run;
```

The output that PROC PRINT generates shows the resulting SCORES data set. The values for TEAM contain the quotation marks.
INPUT Statement: Named

Reads data values that appear after a variable name that is followed by an equal sign and assigns the values to corresponding SAS variables.

Valid in: DATA step
Category: File-Handling
Type: Executable
Restriction: This statement is not supported in a DATA step that runs in CAS.

Syntax

\[
\text{INPUT } \langle\text{pointer-control}\rangle \text{ variable} = \langle\$\rangle \langle@ | @@\rangle; \\
\text{INPUT } \text{variable} = \langle\$\rangle \text{ start-column} \langle-\text{end-column}\rangle \langle@ | @@\rangle;
\]

Arguments

\textbf{pointer-control}

moves the input pointer to a specified line or column in the input buffer.

See "Column Pointer Controls" on page 148 and "Line Pointer Controls" on page 150

\textbf{variable=}

specifies a variable whose value is read by the INPUT statement. In the input data record, the field has this form:

\[
\text{variable}=\text{value}
\]
Tip
Use the INFORMAT statement to associate an informat with a variable.

See
SAS Informats in *SAS Formats and Informats: Reference*

Example
“Example 3: Using Named Input with Another Input Style” on page 181

$ indicates to store a variable value as a character value rather than as a numeric value.

Tip
If the variable is previously defined as character, $ is not required.

Example
“Example 3: Using Named Input with Another Input Style” on page 181

**start-column**
specifies the column that the INPUT statement uses to begin scanning in the input data records for the variable. The variable name does not have to begin here.

**end-column**
determines the default length of the variable.

@ holds an input record for the execution of the next INPUT statement within the same iteration of the DATA step. This line-hold specifier is called *trailing @*.

**Restriction**
The trailing @ must be the last item in the INPUT statement.

**Tip**
The trailing @ prevents the next INPUT statement from automatically releasing the current input record and reading the next record into the input buffer. It is useful when you need to read from a record multiple times.

**See**
“Using Line-Hold Specifiers” on page 155

@@ holds an input record for the execution of the next INPUT statement across iterations of the DATA step. This line-hold specifier is called *double trailing @*.

**Restriction**
The double trailing @ must be the last item in the INPUT statement.

**Tip**
The double trailing @ is useful when each input line contains values for several observations.

**See**
“Using Line-Hold Specifiers” on page 155

**Details**

When to Use Named Input

Named input reads the input data records that contain a variable name followed by an equal sign and a value for the variable. The INPUT statement reads the input data record at the current location of the input pointer. If the input data records contain data values at the start of the record that the INPUT statement cannot read with named input, use another input style to read the values. However, once the
INPUT statement starts to read named input, SAS expects that all the remaining values are in this form. See “Example 3: Using Named Input with Another Input Style” on page 181.

You do not have to specify the variables in the INPUT statement in the same order that they occur in the data records. Also, you do not have to specify a variable for each field in the record. However, if you do not specify a variable in the INPUT statement that another statement uses (for example, ATTRIB, FORMAT, INFORMAT, or LENGTH statement) and it occurs in the input data record, the INPUT statement automatically reads the value. SAS writes a note to the SAS log that the variable is uninitialized.

When you do not specify a variable for all the named input data values, SAS sets _ERROR_ to 1 and writes a note to the SAS log.

```
data list;
  input name=$ age=;
  datalines;
  name=John age=34 gender=M
;
```

The note that SAS writes to the SAS log states that GENDER is not defined and _ERROR_ is set to 1.

Restrictions

- After you start to read with named input, you cannot switch to another input style or use pointer controls. All the remaining values in the input data record must be in the form `variable=value`. SAS treats the values that are not in named input form as invalid data.

- If named input values continue after the end of the current input line, use a slash (/) at the end of the input line. The slash tells SAS to move the pointer to the next line and to continue to read with named input. Here is an example of an input record that is split across two input lines.

```
name=John /
age=34
```

- If you use named input to read character values that contain embedded blanks, put two blanks before and after the data value, as you would with list input. See “Example 4: Reading Character Variables with Embedded Blanks” on page 182.

- You cannot reference an array with an asterisk or an expression subscript.

Examples:

Example 1: Using List and Named Input

This DATA step uses list input with named input to read input data records.

The INPUT statement uses list input to read the ID variable. The remaining variables NAME, GENDER, AGE, and DOB are read with named input. The LENGTH statement prevents the INPUT statement from truncating the character values for the variable name to a length of 8.

```
data list;
  length name $ 20 gender $ 1;
  informat dob ddmmyy8. ;
  input id name= gender= age= dob=;
```
Example 2: Using Named Input with Variables in Random Order

Using the same data as in the previous example, this DATA step also uses list input and named input to read input data records. However, in this example, the order of the values in the input data is different for the two rows, except for the ID value, which must come first.

```sas
data list;
  length name $ 20 gender $ 1;
  informat dob ddmmyy8.;
  input id dob= name= age= gender=;
  datalines;
  4798 gender=m name=COLIN age=23 dob=16/02/75
  2653 name=MICHELE dob=17/02/73 age=46 gender=f
;  proc print data=list; run;
```

Example 3: Using Named Input with Another Input Style

This DATA step uses list input and named input to read input data records. In this example, the length and type of the NAME and GENDER variables are specified directly in the INPUT statement.

The INPUT statement uses list input to read the first variable, ID. The remaining variables NAME, GENDER, and DOB are read with named input. These variables are not read in order. The $20. informat with NAME= prevents the INPUT statement from truncating the character value to a length of 8. The INPUT statement reads the DOB= field because the INFORMAT statement refers to this variable. The statement skips the AGE= field altogether. SAS writes notes to the SAS log that DOB is uninitialized, AGE is not defined, and _ERROR_ is set to 1.

```sas
data list;
  input id name=$20. gender=$;
  informat dob ddmmyy8.;
  datalines;
  4798 gender=m name=COLIN age=23 dob=16/02/75
  2653 name=MICHELE dob=17/02/73 age=46 gender=f
;  proc print data=list; run;
```
Example 4: Reading Character Variables with Embedded Blanks

This DATA step reads character variables that contain embedded blanks with named input.

Two spaces precede and follow the value of the variable HEADER, which is `AGE=60 AND UP`. The field also contains an equal sign.

```sas
data list2;
    informat header $30. name $15.;
    input header= name=;
    datalines;
    header= age=60 AND UP name=PHILIP
;
    proc print data=list2; run;
```

See Also

**Statements:**
- “INPUT Statement” on page 146

---

### KEEP Statement

Specifies the variables to include in output SAS data sets.

**Valid in:** DATA step

**Categories:** CAS

**Type:** Declarative

**Syntax**

```
KEEP variable-list;
```

**Arguments**

`variable-list` specifies the names of the variables to write to the output data set.
Tip  List the variables in any form that SAS allows.

Details

The KEEP statement causes a DATA step to write only the variables that you specify to one or more SAS data sets. The KEEP statement applies to all SAS data sets that are created within the same DATA step and can appear anywhere in the step. If no KEEP or DROP statement appears, all data sets that are created in the DATA step contain all variables.

Note: Do not use both the KEEP and DROP statements within the same DATA step.

Comparisons

- The KEEP statement cannot be used in SAS PROC steps. The KEEP= data set option can.
- The KEEP statement applies to all output data sets that are named in the DATA statement. To write different variables to different data sets, you must use the KEEP= data set option.
- The DROP statement is a parallel statement that specifies variables to omit from the output data set.
- The KEEP and DROP statements select variables to include in or exclude from output data sets. The subsetting IF statement selects observations.
- Do not confuse the KEEP statement with the RETAIN statement. The RETAIN statement causes SAS to hold the value of a variable from one iteration of the DATA step to the next iteration. The KEEP statement does not affect the value of variables but specifies only which variables to include in any output data sets.

Examples:

Example 1: KEEP Statement Basic Usage

These examples show the correct syntax for listing variables in the KEEP statement.

```sas
keep name address city state zip phone;
keep rep1-rep5;
```

Example 2: Keeping Variables in the Data Set

This example uses the KEEP statement to include only the variables NAME and AVG in the output data set. The variables SCORE1 through SCORE20, from which AVG is calculated, are not written to the data set AVERAGE.

```sas
data average;
  keep name avg;
  infile file-specification;
  input name $ score1-score20;
  avg=mean(of score1-score20);
run;
```
See Also

**Data Set Options:**
- “KEEP= Data Set Option” in *SAS Data Set Options: Reference*

**Statements:**
- “DROP Statement” on page 71
- “IF Statement: Subsetting” on page 105
- “RETAIN Statement” on page 279

---

**LABEL Statement**

Assigns descriptive labels to variables.

**Valid in:** DATA step or PROC step

**Categories:** CAS

**Type:** Declarative

**Syntax**

\[
\text{LABEL } \text{variable-1=}\text{label-1}<\ldots\text{variable-n=}\text{label-n}>;
\]

\[
\text{LABEL } \text{variable-1=}''<\ldots\text{variable-n}=''>
\]

**Arguments**

**variable**

specifies the variable that you want to label.

**Tip**

You can specify additional pairs of labels and variables.

**label**

specifies a label of up to 256 bytes.

**Restrictions**

If the label includes a semicolon (;) or an equal sign (=), you must enclose the label in either single or double quotation marks.

If the label includes single quotation marks (‘), you must enclose the label in double quotation marks.

There is partial support for variable name substitution with the ActiveX and Java devices. When the label text for the variable exceeds the space allowed, the variable name is used instead by procedures that are labeling an axis or a legend title. With the exception of the SAS/GRAPH GPLOT procedure, the variable name is not used when ActiveX or Java devices are specified.

**Note**

The LABEL statement expects an equal sign (=) after the first variable. If any other character is found, an error occurs. The
LABEL statement considers everything after the equal sign, following the first variable, as part of the label until the statement comes to another variable followed by an equal sign, regardless of quotation marks. In this example, the label for x is **this is x y 4** =**this is y** because the next variable followed by an equal sign is z.

```plaintext
data test1;
  x=1;
  y=1;
  z=1;
  label x="this is x" y 4 ="this is y" z="This is z";
run;
```

In this LABEL statement, the x variable would have the same label:

```plaintext
code
label x='this is x'  y 4 = 'this is y'  z='This is z';
```

**Tips**

You can specify additional pairs of labels and variables.

For more information about including quotation marks as part of the label, see “Character Constants” in *SAS Language Reference: Concepts*.

removes a label from a variable. Enclose a single blank space in quotation marks to remove an existing label.

**Details**

Using a LABEL statement in a DATA step permanently associates labels with variables by affecting the descriptor information of the SAS data set that contains the variables. You can associate any number of variables with labels in a single LABEL statement.

**Comparisons**

Both the ATTRIB and LABEL statements can associate labels with variables and change a label that is associated with a variable.

Label statements can be used in a DATA step and within some PROC steps. If a label is assigned to a variable in a DATA step or in PROC DATASETS, the label is permanently assigned in the output data set descriptor. Some PROCs, such as PROC PRINT, can temporarily associate a label with a variable for use during the procedure.

This example demonstrates the use of labels during the creation of a report. By using the PROC PRINT label option, you can display labels in place of variable names in the output report.

```plaintext
 data work.cars;       /* DATA step */
  set sahelp.cars;
  label MSRP='Sticker Price'; /* Assigns a permanent label to the variable MSRP */
run;
```

```plaintext
 proc datasets library=work;       /* PROC DATASETS */
  modify cars;
```
label Invoice=Dealer Cost; /* Assigns a permanent label to the variable Invoice */ quit;

proc print data=work.cars label; /* PROC PRINT - LABEL option displays label names */ label Type=Style; /* Assigns a temporary label to the Type variable */ var Make Model MSRP Invoice Type;
run;

The output report contains the three new labels. Sticker Price is assigned with the DATA step LABEL statement. Dealer Cost is assigned with the PROC DATASETS LABEL statement, and Style is assigned with the PROC PRINT LABEL statement.

The new permanent labels Sticker Price and Dealer Cost are now assigned to the Invoice and MSRP variables. The Style label, assigned to the Type variable during the execution of the PROC PRINT statement, is not retained in the output data set.

For more information about using a LABEL statement within a PROC step, see Base SAS Procedures Guide.
`Example 1: Specifying Labels`

Here are several LABEL statements.

```sas
label compound=Type of Drug;
label date="Today's Date";
label n='Mark''s Experiment Number';
label score1="Grade on April 1 Test"
  score2="Grade on May 1 Test";
```

`Example 2: Removing a Label`

This example removes an existing label.

```sas
data rtest;
set rtest;
  label x=' ';
run;
```

See Also

Statements:

- "ATTRIB Statement" on page 30

---

**Label: Statement**

Identifies a statement that is referred to by another statement.

**Valid in:** DATA step

**Categories:** CAS

Control

**Type:** Declarative

**Syntax**

```
label: statement;
```

**Required Arguments**

`label`

specifies any SAS name. Follow the SAS name with a colon (:\).

**Requirement**

The name that you choose for your label must be followed by a colon (\:).

**Example**

data mydata2;
```sas
type="new"; num=0;
if type ="new" then
/* LINK to label calculate */
link calculate;
/* define destination (statement) for label calculate */
calculate: if num=0
then order="yes";
num=num+1;
run;
```

**statement**

specifies any executable statement, including a null statement (;

**Restrictions**

No two statements in a DATA step can have the same label.

If a statement in a DATA step is labeled, it should be referenced by a statement or option in the same step.

**Requirement**

Both the *label* and *statement* arguments are required and they must be separated by a colon (:).

**Tip**

A null statement can have a label:

ABC:;

**Example**

data test;
<other-sas-DATA-step-statements>
if x=1 then
/* Define the label-name (mylabel) */
GO TO mylabel;
/* Define the statement-name and destination for the label */
mylabel: PUT "Hello";
run;
```

**Details**

The statement label identifies the destination of one of the following language elements:

- GO TO statement
- LINK statement
- HEADER= option in a FILE statement
- EOF= option in an INFILE statement
- Null statement

**Comparisons**

- The LABEL statement assigns a descriptive label to a variable.
- The statement label identifies a statement or group of statements that are referred to in the same DATA step by another statement, such as a GO TO statement.
Examples:

Example 1: Jumping to Another Statement
In this example, if Stock=0, the GO TO statement causes SAS to jump to the statement that is labeled reorder. When Stock is not 0, execution continues to the RETURN statement and then returns to the beginning of the DATA step for the next observation.

```sas
data Inventory Order;
  input Item $ Stock @;
  /* go to label reorder: */
  if Stock=0 then goto reorder;
  output Inventory;
  return;
  /* destination of GO TO statement */
reorder: input Supplier $;
  put 'ORDER ITEM ' Item ' FROM ' Supplier;
  output Order;
  datalines;
milk 0 A
bread 3 B
```

Example 2: Diverting Program Execution
In this example, when the value of variable TYPE is `aluv`, the LINK statement diverts program execution to the statements that are associated with the label CALCUL. The program executes until it encounters the RETURN statement, which sends program execution back to the first statement that follows LINK. SAS executes the assignment statement, writes the observation, and then returns to the top of the DATA step to read the next record. When the value of TYPE is not `aluv`, SAS executes the assignment statement, writes the observation, and returns to the top of the DATA step.

```sas
data hydro;
  input type $ depth station $;
  /* link to label calcu: */
  if type ='aluv' then link calcu;
  date=today();
  /* return to top of step */
  return;
  calcu: if station='site_1'
    then elevatn=6650-depth;
  else if station='site_2'
    then elevatn=5500-depth;
  /* return to date=today(); */
  return;
  datalines;
  aluv 523 site_1
  uppa 234 site_2
  aluv 666 site_2
  ...more data lines...
```
Example 3: Using a Statement Label with Null

The Null statement is useful while you are developing a program. For example, use it after a statement label to test your program before you code the statements that follow the label.

```sas
data _null_;
set dsn;
file print header=header;
put 'report text';
...more statements...
return;
header:;
run;
```

See Also

**Statements:**
- “GO TO Statement” on page 104
- “LINK Statement” on page 196

**Statement Options:**
- “EOF=label” on page 117 in the INFILE statement
- “HEADER=label” on page 85 option in the FILE statement

---

**LEAVE Statement**

Stops processing the current loop and resumes with the next statement in the sequence.

**Valid in:** DATA step

**Categories:** CAS
             Control

**Type:** Executable

**Syntax**

```
LEAVE;
```

**Without Arguments**

The LEAVE statement stops the processing of the current DO loop or SELECT group and continues DATA step processing with the next statement following the DO loop or SELECT group.
Details
You can use the LEAVE statement to exit a DO loop or SELECT group prematurely based on a condition.

Comparisons
- The LEAVE statement causes processing of the current loop to end. The CONTINUE statement stops the processing of the current iteration of a loop and resumes with the next iteration.
- You can use the LEAVE statement in a DO loop or in a SELECT group. You can use the CONTINUE statement only in a DO loop.

Example: Stop Processing a DO Loop under a Given Condition
This DATA step demonstrates using the LEAVE statement to stop the processing of a DO loop under a given condition. In this example, the IF/THEN statement checks the value of BONUS. When the value of BONUS reaches 500, the maximum amount allowed, the LEAVE statement stops the processing of the DO loop.

```plaintext
data week;
  input name $ idno start_yr status $ dept $;
  bonus=0;
  do year= start_yr to 1991;
    if bonus ge 500 then leave;
    bonus+50;
  end;
datalines;
  Jones 9011 1990 PT PUB
  Thomas 876 1976 PT HR
  Barnes 7899 1991 FT TECH
  Harrell 1250 1975 FT HR
  Richards 1002 1990 FT DEV
  Kelly 85 1981 PT PUB
  Stone 091 1990 PT MAIT
;
```

See Also

**Statements:**
- “DO Statement” on page 62
- “SELECT Statement” on page 286

---

**LENGTH Statement**

Specifies the number of bytes for storing character and numeric variables, or the number of characters for storing VARCHAR variables.

**Valid in:** DATA step
Avoid shortening numeric variables that contain fractions. The precision of a numeric variable is closely tied to its length, especially when the variable contains fractions. You can safely shorten variables that contain integers according to the rules that are given in the SAS documentation for your operating environment, but shortening variables that contain fractions might eliminate important precision.

Syntax

**LENGTH**  
`variable-specification(s) <DEFAULT=n>`;

Arguments

*variable-specification*  
is a required argument and has this form:

`variable(s) <$length | length | VARCHAR(length) | VARCHAR(*)>`

*variable*  
specifies one or more variables that are to be assigned a length, including any variables in the DATA step. Variables that are dropped from the output data set are also included.

Restriction  
Array references are not allowed.

Tip  
If the variable is character or varchar, the length applies to the program data vector and the output data set. If the variable is numeric, the length applies only to the output data set.

*$*  
specifies that the preceding variables are character variables of type CHAR.

*length*  
specifies a numeric constant for storing variable values. For numeric and character variables, this constant is the maximum number of bytes stored in the variable.

Range  
For numeric variables, 2 to 8 bytes or 3 to 8 bytes, depending on your operating environment. For character variables, 1 to 32767 bytes under all operating environments.

Restriction  
In CAS, numeric variables shorter than 8 bytes are treated as 8 bytes.

*VARCHAR*  
specifies that the preceding variables are of type VARCHAR.

*length*  
specifies the maximum number of characters stored for VARCHAR variables.

For example, `length v varchar100;` means store up to 100 characters in the V VARCHAR variable.
### Range
1 to 536,870,911 characters for VARCHAR variables.

### Restriction
When assigning a character constant to a VARCHAR variable, the character constant is limited to 32767 bytes.

### Requirement
The CAS engine is required if you want to preserve a variable as a VARCHAR data type when reading it in or writing it out using the DATA step.

### See
For more information about automatic declaration of variables, see “Support for Implicit Declaration of Data Types” in SAS Cloud Analytic Services: User’s Guide.

For more information about how VARCHAR variables are automatically converted to character values, see “VARCHAR Support for Implicit and Explicit Data Type Conversion” in SAS Cloud Analytic Services: User’s Guide and “VARCHAR Length with Implicit Type Conversion” in SAS Cloud Analytic Services: User’s Guide.

* specifies to support the maximum length allowed: 536,870,911 characters.

**DEFAULT=n**  
changes the default number of bytes that SAS uses to store the values of any newly created numeric variables.

<table>
<thead>
<tr>
<th>Default</th>
<th>8 bytes</th>
</tr>
</thead>
</table>

### Details

#### General Information

In general, the length of a variable depends on this information:

- whether the variable is numeric or character
- how the variable was created
- whether a LENGTH statement or ATTRIB statement is present
- the SAS session encoding that is used to process your program

Subject to the rules for assigning lengths, character and numeric lengths that are assigned with the LENGTH statement can be changed in the ATTRIB statement and vice versa.

VARCHAR variables allocate memory as needed to hold their value. The maximum amount of memory used by a VARCHAR variable depends on the SAS session encoding. A variable declared as VARCHAR(100) can store up to 100 characters. In a single-byte SAS session encoding, such as LATIN1, the variable uses up to 100 bytes for its value.

For a UTF-8 SAS session encoding, where a character can take up to 4 bytes, up to 400 bytes might be used by the VARCHAR value.

For more information about assigning lengths to variables, see “SAS Variables” in SAS Language Reference: Concepts.
Operating Environment Information: Valid variable lengths depend on your operating environment. For more information, see the SAS documentation for your operating environment.

Comparisons

The ATTRIB statement can assign the length as well as other attributes of character and numeric variables.

Example

This example uses a LENGTH statement to set the length of the character variable NAME to 25 bytes. The LENGTH statement also changes the default number of bytes that SAS uses to store the values of newly created numeric variables from 8 to 4 bytes. The TRIM function removes trailing blanks from LASTNAME before it is concatenated with these items:

- a comma (,)
- a blank space
- the value of FIRSTNAME

If you omit the LENGTH statement, SAS sets the length of NAME to 32 bytes.

```sas
data testlength;
  informat FirstName LastName $15. n1 6.2;
  input firstname lastname n1 n2;
  length name $25 default=4;
  name=trim(lastname)||', '||firstname;
  datalines;
  Alexander Robinson 35 11
  ;
  proc contents data=testlength;
  run;
  proc print data=testlength;
  run;
```

This output shows a partial listing from PROC CONTENTS, as well as the report that PROC PRINT generates.

Output 2.15 Partial PROC CONTENTS for TESTLENGTH

<table>
<thead>
<tr>
<th>#</th>
<th>Variable</th>
<th>Type</th>
<th>Len</th>
<th>Informat</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>FirstName</td>
<td>Char</td>
<td>15</td>
<td>$15.</td>
</tr>
<tr>
<td>2</td>
<td>LastName</td>
<td>Char</td>
<td>15</td>
<td>$15.</td>
</tr>
<tr>
<td>3</td>
<td>n1</td>
<td>Num</td>
<td>4</td>
<td>6.2</td>
</tr>
<tr>
<td>4</td>
<td>n2</td>
<td>Num</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>name</td>
<td>Char</td>
<td>25</td>
<td></td>
</tr>
</tbody>
</table>
Output 2.16  Setting the Length of a Variable

<table>
<thead>
<tr>
<th>Obs</th>
<th>FirstName</th>
<th>LastName</th>
<th>n1</th>
<th>n2</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Alexander</td>
<td>Robinson</td>
<td>0.3500</td>
<td>11</td>
<td>Robinson, Alexander</td>
</tr>
</tbody>
</table>

See Also

- For information about the use of the LENGTH statement in PROC steps, see Base SAS Procedures Guide.

Statements:

- “ATTRIB Statement” on page 30

LIBNAME Statement

Associates or disassociates a SAS library with a libref (a shortcut name), clears one or all librefs, lists the characteristics of a SAS library, concatenates SAS libraries, or concatenates SAS catalogs.

Note: The LIBNAME Statement has moved to SAS Global Statements.

LIBNAME Statement: CVP Engine

Associates a libref for the character variable padding (CVP) engine to expand character variable lengths so that character data truncation does not occur when a file requires transcoding.

Note: The LIBNAME Statement: CVP Engine has moved to SAS Global Statements.

LIBNAME Statement: JMP Engine

Associates a libref with a JMP data table and enables you to read and write JMP data tables.

Note: The LIBNAME Statement: JMP Engine has moved to SAS Global Statements.

LIBNAME Statement: JSON Engine

Associates a SAS libref with a JSON document and ensures that the JSON data and an optional supplied JSON MAP are valid.

Note: The LIBNAME Statement: JSON engine has moved to SAS Global Statements.
LIBNAME Statement: WebDAV Server Access

 Associates a libref with a SAS library and enables access to a WebDAV (Web-based Distributed Authoring And Versioning) server.

Note: The LIBNAME Statement: WebDAV Server Access has moved to SAS Global Statements.

LINK Statement

Directs program execution immediately to the statement label that is specified and, if followed by a RETURN statement, returns execution to the statement that follows the LINK statement.

Valid in: DATA step
Categories: CAS
Control
Type: Executable

Syntax

LINK label;

Arguments

label

specifies a statement label that identifies the LINK destination. You must specify the label argument.

Details

The LINK statement tells SAS to jump immediately to the statement label that is indicated in the LINK statement and to continue executing statements from that point until a RETURN statement is executed. The RETURN statement sends program control to the statement immediately following the LINK statement.

The LINK statement and the destination must be in the same DATA step. The destination is identified by a statement label in the LINK statement.

The LINK statement can branch to a group of statements that contain another LINK statement. This arrangement is known as nesting. To avoid infinite looping, SAS has set a default number of nested LINK statements. You can have up to 10 LINK statements with no intervening RETURN statements. When more than one LINK statement has been executed, a RETURN statement tells SAS to return to the statement that follows the last LINK statement that was executed. However, you can use the /STACK option in the DATA statement to increase the number of nested LINK statements.
Comparisons

The difference between the LINK statement and the GO TO statement is in the action of a subsequent RETURN statement. A RETURN statement after a LINK statement returns execution to the statement that follows LINK. A RETURN statement after a GO TO statement returns execution to the beginning of the DATA step, unless a LINK statement precedes GO TO. In that case, execution continues with the first statement after LINK. In addition, a LINK statement is usually used with an explicit RETURN statement, whereas a GO TO statement is often used without a RETURN statement.

When your program executes a group of statements at several points in the program, using the LINK statement simplifies coding and makes program logic easier to follow. If your program executes a group of statements at only one point in the program, using DO-group logic rather than LINK-RETURN logic is simpler.

Example: Diverting Program Execution

In this example, when the value of variable TYPE is aluv, the LINK statement diverts program execution to the statements that are associated with the label CALCU. The program executes until it encounters the RETURN statement, which sends program execution back to the first statement that follows LINK. SAS executes the assignment statement, writes the observation, and then returns to the top of the DATA step to read the next record. When the value of TYPE is not aluv, SAS executes the assignment statement, writes the observation, and returns to the top of the DATA step.

data hydro;
  input type $ depth station $;
  /* link to label calcu: */
  if type ='aluv' then link calcu;
  date=today();
  /* return to top of step */
  return;
  calcu: if station='site_1'
    then elevatn=6650-depth;
  else if station='site_2'
    then elevatn=5500-depth;
  /* return to date=today(); */
  return;
  datalines;
  aluv 523 site_1
  uppa 234 site_2
  aluv 666 site_2
  ...more data lines...
;

See Also

Statements:

- “DATA Statement” on page 44
- “DO Statement” on page 62
- “GO TO Statement” on page 104
- “Label: Statement” on page 187
LIST Statement

Writes to the SAS log the input data record for the observation that is being processed.

Valid in: DATA step
Categories: Action, CAS
Type: Executable

Syntax

LIST;

Without Arguments

The LIST statement causes the input data record for the observation being processed to be written to the SAS log.

Details

The LIST statement operates only on data that is read with an INPUT statement; it has no effect on data that is read with a SET, MERGE, MODIFY, or UPDATE statement.

In the SAS log, a ruler that indicates column positions appears before the first record listed.

For variable-length records (RECFM=V), SAS writes the record length at the end of the input line. SAS does not write the length for fixed-length records (RECFM=F), unless the amount of data read does not equal the record length (LRECL).

Comparisons

This table compares the LIST and PUT statements.

<table>
<thead>
<tr>
<th>Action</th>
<th>LIST Statement</th>
<th>PUT Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Writes when</td>
<td>at the end of each iteration of the DATA step</td>
<td>immediately</td>
</tr>
<tr>
<td>Writes what</td>
<td>the input data records exactly as they appear</td>
<td>the variables or literals specified</td>
</tr>
<tr>
<td>Writes where</td>
<td>only to the SAS log</td>
<td>to the SAS log, the SAS output destination, or to any external file</td>
</tr>
</tbody>
</table>
### Action

<table>
<thead>
<tr>
<th>LIST Statement</th>
<th>PUT Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Works with INPUT statement only</td>
<td>any data-reading statement</td>
</tr>
<tr>
<td>Handles hexadecimal values</td>
<td>automatically prints a hexadecimal value if it encounters an unprintable character</td>
</tr>
<tr>
<td></td>
<td>represents characters in hexadecimal only when a hexadecimal format is given</td>
</tr>
</tbody>
</table>

### Examples:

**Example 1: Listing Records That Contain Missing Data**

This example uses the LIST statement to write to the SAS log any input records that contain missing data. Because of the #3 line pointer control in the INPUT statement, SAS reads three input records to create a single observation. Therefore, the LIST statement writes the three current input records to the SAS log each time a value for W2AMT is missing.

```sas
data employee;
  input ssn 1-9 #3 w2amt 1-6;
  if w2amt= . then list;
  datalines;
23456789
JAMES SMITH
356.79
345671234
Jeffrey Thomas
.
;
```

**Output 2.17 Log Listing of Missing Data**

```
RULE:---+-+---+---+---+---+---+---+---+---+---+---+---+---+
  9  345671234
10 Jeffrey Thomas
11  .
```

The numbers 9, 10, and 11 are line numbers in the SAS log.

**Example 2: Listing the Record Length of Variable-Length Records**

This example uses as input an external file that contains variable-length ID numbers. The RECFM=V option is specified in the INFILE statement, and the LIST statement writes the records to the SAS log. When the file has variable-length records, as indicated by the RECFM=V option in this example, SAS writes the record length at the end of each record that is listed in the SAS log.

```sas
data employee;
  infile 'your-external-file' recfm=v;
  input id $;
  list;
run;
```
Example 3: Listing Hexadecimal Values for Input Data

Beginning with SAS 9.4M6, the /HEXLISTALL DATA statement option enables the LIST statement to write all lines of input data in hexadecimal format to the SAS log. In this example, we create a file that contains a valid line of data, a line of data that contains unprintable characters, and an invalid line of data. Next, we use a default DATA statement with the LIST statement to display the input data in hexadecimal format for the unprintable characters in the SAS log. We then use the DATA statement with the /HEXLISTALL argument and the LIST statement to display all of the input data in its hexadecimal format in the SAS log.

Here, we create the file `f_out.txt` that contains the three lines of data.

```sas
filename f_out 'f_out.txt' recfm=f lrecl=30; /* assign fileref */
data;
  file f_out;
  output;
  put '123456789012345678901234567890';
  put '0102030405060708090A0B0C0D0E0F10111213141516171819202A2B2C2D'x;
  put 'Not a number                  ';
run;
```

Here, we use the default settings in the DATA statement with the LIST statement to write the contents of the file to the SAS log. By default, hexadecimal format is displayed for the unprintable characters found on line two of the input file. Notes are displayed for the invalid data found on lines two and three.

```sas
data _null_;
infile  f_out length=len;
  input num;
  list;
run;
```
Output 2.19   SAS Log of Hexadecimal Values for Unprintable Characters

```
RULE:     ----+----1----+----2----+----3----+----4----+----5----+----6----
        +----7----+----8----+---
1   123456789012345678901234567890
NOTE: Invalid data for num in line 2 1-25.
```

```
RULE:     ----+----1----+----2----+----3----+----4----+----5----+----6----
        +----7----+----8----+---
2   CHAR  ......................... *+,-
ZONE  00000000000000011111111222222
NUMR  123456789ABCDEF01234567890ABCD
len=30 num=. _ERROR_=1 _N_=2
NOTE: Invalid data for num in line 3 1-3.
3   Not a number
len=30 num=. _ERROR_=1 _N_=3
```

Here, we add the /HEXLISTALL argument to the DATA statement. The LIST statement now writes hexadecimal format for all lines of the input data in the SAS log. Notes are displayed for the invalid data found on lines two and three.

```
data _null_ /HEXLISTALL;
  infile  f_out length=len;
  input num;
  list;
run;
```

Output 2.20   SAS Log of Hexadecimal Values for All Input Data

```
RULE:     ----+----1----+----2----+----3----+----4----+----5----+----6----
        +----7----+----8----+---
1   CHAR  123456789012345678901234567890
ZONE  333333333333333333333333333333
NUMR  123456789012345678901234567890
NOTE: Invalid data for num in line 2 1-25.
2   CHAR  ......................... *+,-
ZONE  00000000000000011111111222222
NUMR  123456789ABCDEF01234567890ABCD
len=30 num=. _ERROR_=1 _N_=2
NOTE: Invalid data for num in line 3 1-3.
3   Not a number
ZONE  467262676667222222222222222222
NUMR  EF4010E5D252000000000000000000
len=30 num=. _ERROR_=1 _N_=3
```

See Also

Statements:

- “DATA Statement” on page 44
- “INPUT Statement” on page 146
- “PUT Statement” on page 236
%LIST Statement
Displays lines that are entered in the current session.

Note: The %LIST Statement has moved to SAS Global Statements.

LOCK Statement
Acquires, lists, or releases an exclusive lock on an existing SAS file.

Note: The LOCK Statement has moved to SAS Global Statements.

LOCKDOWN Statement
Secures the SAS Viya Workspace or SAS/CONNECT server on SAS Viya by restricting access from within a server process to the host operating environment.

Notes: The LOCKDOWN statement for SAS Viya has moved to SAS Viya Administration: Programming Run-Time Servers. For more information, see SAS Viya LOCKDOWN Statement.
For information about LOCKDOWN on SAS 9.4, see SAS 9.4 LOCKDOWN Statement.

LOSTCARD Statement
Resynchronizes the input data when SAS encounters a missing or invalid record in data that has multiple records per observation.

Valid in: DATA step
Categories: Action
CAS
Type: Executable

Syntax
LOSTCARD;

Without Arguments
The LOSTCARD statement prevents SAS from reading a record from the next group when the current group has a missing record.
Details

When to Use LOSTCARD

When SAS reads multiple records to create a single observation, it does not discover that a record is missing until it reaches the end of the data. If there is a missing record in your data, the values for subsequent observations in the SAS data set might be incorrect. Using LOSTCARD prevents SAS from reading a record from the next group when the current group has fewer records than SAS expected.

LOSTCARD is most useful when the input data have a fixed number of records per observation and when each record for an observation contains an identification variable that has the same value. LOSTCARD usually appears in conditional processing such as in the THEN clause of an IF-THEN statement, or in a statement in a SELECT group.

When LOSTCARD Executes

When LOSTCARD executes, SAS takes several steps:

1. Writes three items to the SAS log: a lost card message, a ruler, and all the records that it read in its attempt to build the current observation.
2. Discards the first record in the group of records being read, does not write an observation, and returns processing to the beginning of the DATA step.
3. Does not increment the automatic variable _N_ by 1. (Normally, SAS increments _N_ by 1 at the beginning of each DATA step iteration.)
4. Attempts to build an observation by beginning with the second record in the group, and reads the number of records that the INPUT statement specifies.
5. Repeats steps 1 through 4 when the IF condition for a lost card is still true. To make the log more readable, SAS prints the message and ruler only once for a given group of records. In addition, SAS prints each record only once, even if a record is used in successive attempts to build an observation.
6. Builds an observation and writes it to the SAS data set when the IF condition for a lost card is no longer true.

Example: Resynchronizing Input Data

This example uses the LOSTCARD statement in a conditional construct to identify missing data records and to resynchronize the input data.

data inspect;
  input id 1-3 age 8-9 #2 id2 1-3 loc
    #3 id3 1-3 wt;
  if id ne id2 or id ne id3 then do;
    put 'DATA RECORD ERROR: ' id= id2= id3=;
    lostcard;
  end;
datalines;
301    32
301    61432
301    127
302    61
302    83171
The DATA step reads three input records before writing an observation. If the identification number in record 1 (variable ID) does not match the identification number in the second record (ID2) or third record (ID3), a record is incorrectly entered or omitted. The IF-THEN DO statement specifies that if an identification number is invalid, SAS prints the message that is specified in the PUT statement message and executes the LOSTCARD statement.

In this example, the third record for the second observation (ID3=400) is missing. The second record for the third observation is incorrectly entered (ID=400 while ID2=409). Therefore, the data set contains two observations with ID values 301 and 411. There are no observations for ID=302 or ID=400. The PUT and LOSTCARD statements write these statements to the SAS log when the DATA step executes.

```
DATA RECORD ERROR: id=302 id2=302 id3=400
NOTE: LOST CARD.
RULE:-----------1-----------2-----------3-----------4-----------5-----------
14   302    61
15   302    83171
16   400    46
DATA RECORD ERROR: id=302 id2=400 id3=409
NOTE: LOST CARD.
17   409    23145
DATA RECORD ERROR: id=400 id2=409 id3=400
NOTE: LOST CARD.
18   400    197
DATA RECORD ERROR: id=409 id2=400 id3=411
NOTE: LOST CARD.
19   411    53
DATA RECORD ERROR: id=400 id2=411 id3=411
NOTE: LOST CARD.
20   411    99551
```

The numbers 14, 15, 16, 17, 18, 19, and 20 are line numbers in the SAS log.

See Also

**Statements:**

- “IF-THEN/ELSE Statement” on page 108

**MERGE Statement**

Joins observations from two or more SAS data sets into a single observation.

Valid in: DATA step
Categories: CAS, File-Handling
The variables read using the MERGE statement are retained in the PDV. The data types of the variables that are read are also retained. For more information, see “Overview of DATA Step Processing” in SAS Language Reference: Concepts and the “RETAIN Statement” on page 279.

Syntax

```mergesas
MERGE SAS-data-set-1 <(data-set-options)> 
SAS-data-set-2 <(data-set-options) > 
<...SAS-data-set-n <(data-set-options)> > 
<END=variable> ;
```

Arguments

**SAS-data-set**
specifies at least two existing SAS data sets from which observations are read. You can specify individual data sets, data set lists, or a combination of both.

**data-set-options**
specifies one or more SAS data set options in parentheses after a SAS data set name.

**END=variable**
names and creates a temporary variable that contains an end-of-file indicator.

Details

Overview

The MERGE statement is flexible and has a variety of uses in SAS programming. This section describes basic uses of MERGE. Other applications include using more
than one BY variable, merging more than two data sets, and merging a few
observations with all observations in another data set.

For more information, see “How to Prepare Your Data Sets” in SAS Language
Reference: Concepts.

Using Data Set Lists with MERGE

You can use data set lists with the MERGE statement. Data set lists provide a quick
way to reference existing groups of data sets. These data set lists must be either
name prefix lists or numbered range lists.

Name prefix lists refer to all data sets that begin with a specified character string.
For example, `merge SALES1;` tells SAS to merge all data sets starting with
"SALES1" such as SALES1, SALES10, SALES11, and SALES12.

Numbered range lists require you to have a series of data sets with the same name,
except for the last character or characters, which are consecutive numbers. In a
numbered range list, you can begin with any number and end with any number. For
example, these lists refer to the same data sets:

```
sales1 sales2 sales3 sales4
sales1-sales4
```

Note: If the numeric suffix of the first data set name contains leading zeros, the
number of digits in the numeric suffix of the last data set name must be greater than
or equal to the number of digits in the first data set name. Otherwise, an error
occurs. For example, the data set lists sales001–sales99 and sales01–sales9
causes an error. The data set list sales001–sales999 is valid. If the numeric suffix of
the first data set name does not contain leading zeros, the number of digits in the
numeric suffix of the first and last data set names do not have to be equal. For
example, the data set list sales1–sales999 is valid.

Some other rules to consider when using numbered data set lists are as follows:

- You can specify groups of ranges.
  ```
  merge cost1-cost4 cost11-cost14 cost21-cost24;
  ```
- You can mix numbered range lists with name prefix lists.
  ```
  merge cost1-cost4 cost2: cost33-37;
  ```
- You can mix single data sets with data set lists.
  ```
  merge cost1 cost10-cost20 cost30;
  ```
- Quotation marks around data set lists are ignored.
  ```
  /* these two lines are the same */
  merge sales1-sales4;
  merge 'sales1'n-'sales4'n;
  ```
- Spaces in data set names are invalid. If quotation marks are used, trailing blanks
  are ignored.
  ```
  /* blanks in these statements will cause errors */
  merge sales 1-sales 4;
  merge 'sales 1'n - 'sales 4'n;
  /* trailing blanks in this statement will be ignored */
  merge 'sales1'n - 'sales4'n;
  ```
- The maximum numeric suffix is 2147483647.
/* this suffix will cause an error */
merge prod2000000000-prod2934850239;

Physical pathnames are not allowed.
/* physical pathnames will cause an error */
%let work_path = %sysfunc(pathname(WORK));
merge "&work_path\dept.sas7bdat"-"&work_path\emp.sas7bdat";

One-to-One Merging

One-to-one merging combines observations from two or more SAS data sets into a single observation in a new data set. To perform a one-to-one merge, use the MERGE statement without a BY statement. SAS combines the first observation from all data sets that are named in the MERGE statement into the first observation in the new data set, the second observation from all data sets into the second observation in the new data set, and so on. In a one-to-one merge, the number of observations in the new data set is equal to the number of observations in the largest data set named in the MERGE statement. See Example 1 for an example of a one-to-one merge. For more information, see “Reading, Combining, and Modifying SAS Data Sets” in SAS Language Reference: Concepts.

CAUTION

Use care when you combine data sets with a one-to-one merge. One-to-one merges can sometimes produce undesirable results. Test your program on representative samples of the data sets before you use this method.

Match-Merging

Match-merging combines observations from two or more SAS data sets into a single observation in a new data set according to the values of a common variable. The number of observations in the new data set is the sum of the largest number of observations in each BY group in all data sets. To perform a match-merge, use a BY statement immediately after the MERGE statement. The variables in the BY statement must be common to all data sets. Only one BY statement can accompany each MERGE statement in a DATA step. The data sets that are listed in the MERGE statement must be sorted in order of the values of the variables that are listed in the BY statement, or they must have an appropriate index. See Example 2 for an example of a match-merge. For more information, see “Reading, Combining, and Modifying SAS Data Sets” in SAS Language Reference: Concepts.

Note: The MERGE statement does not produce a Cartesian product on a many-to-many match-merge. Instead, it performs a one-to-one merge while there are observations in the BY group in at least one data set. When all observations in the BY group have been read from one data set and there are still more observations in another data set, SAS performs a one-to-many merge until all observations have been read for the BY group.

Comparisons

MERGE combines observations from two or more SAS data sets. UPDATE combines observations from exactly two SAS data sets. UPDATE changes or updates the values of selected observations in a master data set as well. UPDATE also might add observations.
Like UPDATE, MODIFY combines observations from two SAS data sets by changing or updating values of selected observations in a master data set.

The results that are obtained by reading observations using two or more SET statements are similar to the results that are obtained by using the MERGE statement with no BY statement. However, with the SET statements, SAS stops processing before all observations are read from all data sets if the number of observations are not equal. In contrast, SAS continues processing all observations in all data sets named in the MERGE statement.

Examples:

Example 1: One-to-One Merging
This example shows how to combine observations from two data sets into a single observation in a new data set.

```sas
data benefits.qtr1;
  merge benefits.jan benefits.feb;
run;
```

Example 2: Match-Merging
This example shows how to combine observations from two data sets into a single observation in a new data set according to the values of a variable that is specified in the BY statement.

```sas
data inventory;
  merge stock orders;
  by partnum;
run;
```

Example 3: Merging with a Data Set List
This example uses a data list to define the data sets that are merged.

```sas
data d008; job=3; emp=19; run;
data d009; job=3; sal=50; run;
data d010; job=4; emp=97; run;
data d011; job=4; sal=15; run;
data comb;
  merge d008-d011;
  by job;
run;
proc print data=comb;
run;
```

Example 4: Three Table Merge with BY Values and the IN= Data Set Option

```sas
DATA CAFE(KEEP=NAME PLACE CNUM);
  INPUT NAME $;
  PLACE = 'CAFE ';
  CNUM = 'C' || LEFT(PUT(_N_,2.));
DATALINES;
ANDERSON
COOPER
```
DIXON
FREDERIC
FREDERIC
PALMER
RANDALL
RANDALL
SMITH
SMITH
SMITH
;
RUN;

DATA VENDING (KEEP=NAME PLACE VNUM);
   INPUT NAME $;
   PLACE = 'VENDING ';
   VNUM = 'V' || LEFT(PUT(_N_,2.));
DATALINES;
CARTER
DANIELS
GARY
GARY
HODGE
PALMER
RANDALL
RANDALL
SMITH
SMITH
SPENCER
SPENCER
SPENCER
SPENCER
;
RUN;

DATA SNACK (KEEP=NAME PLACE SNUM);
   INPUT NAME $;
   PLACE = 'SNACK ';
   SNUM = 'S' || LEFT(PUT(_N_,2.));
DATALINES;
BARRETT
COOPER
DANIELS
DIXON
DIXON
FREDERIC
GARY
HODGE
HODGE
PALMER
RANDALL
RANDALL
SMITH
SMITH
SMITH
SMITH
SPENCER

Example 5: Two Table Merge with a BY Variable and the IN= Data Set Option

data have_a;
  input ID amount_a;
datalines;
  1 10
  3 15
  4 20
  7 15
  9 12
  10 14

procsql
  title 'MERGED DATA';
run;

proc print;
  title 'MERGED DATA';
run;
data have_b;
  input ID amount_b;
  datalines;
  2 15
  3 20
  4 10
  5 12
  7 20
  8 15
  9 10
  11 20
;

data want;
  merge have_a(in=inA) have_b(in=inb);
  by id;
  length joinType $ 2;
  joinType = cats(inA, inB);
  run;

proc print data=want;
run;
quit;

<table>
<thead>
<tr>
<th>MERGED DATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Obs</td>
</tr>
<tr>
<td>-----</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
<tr>
<td>10</td>
</tr>
</tbody>
</table>

See Also
- “Reading, Combining, and Modifying SAS Data Sets” in SAS Language Reference: Concepts

Statements:
- “BY Statement” on page 34
MISSING Statement

Assigns characters in your input data to represent special missing values for numeric data.

Note: The MISSING Statement has moved to SAS Global Statements.

MODIFY Statement

Replaces, deletes, and appends observations in an existing SAS data set in place but does not create an additional copy.

Valid in: DATA step
Category: File-Handling
Type:Executable
Restrictions: This statement is not supported in a DATA step that runs in CAS.
This statement cannot modify the descriptor portion of a SAS data set, such as adding a variable.

Notes: If you modify a password-protected data set, specify the password with the appropriate data set option (ALTER= or PW=) in the MODIFY statement, and not in the DATA statement.
The variables read using the MODIFY statement are retained in the PDV. For more information, see “Overview of DATA Step Processing” in SAS Language Reference: Concepts and the “RETAIN Statement” on page 279.

CAUTION: Damage to the SAS data set can occur if the system terminates abnormally during a DATA step that contains the MODIFY statement. Observations in native SAS data files might have incorrect data values, or the data file might become unreadable. DBMS tables that are referenced by views are not affected.

Syntax

Form 1:    MODIFY master-data-set <(data-set-options)> transaction-data-set <(data-set-options)> <CUROBS=variable> <NOBS=variable> <END=variable> <UPDATEMODE=MISSINGCHECK | NOMISSINGCHECK>; BY by-variable;

Form 2:    MODIFY master-data-set <(data-set-options)> KEY=index </ UNIQUE> <KEYRESET=variable> <NOBS=variable> <END=variable>;

Form 3:    MODIFY master-data-set <(data-set-options)> <NOBS=variable> POINT=variable;

Form 4:    MODIFY master-data-set <(data-set-options)> <NOBS=variable> <END=variable>;}
## Arguments

**master-data-set**

specifies the SAS data set that you want to modify.

### Restrictions

This data set must also appear in the DATA statement.

For sequential and matching access, the master data set can be a SAS data file, a SAS/ACCESS view, an SQL view, or a DBMS engine for the LIBNAME statement. It cannot be a DATA step view or a pass-through view.

For random access using POINT=, the master data set must be a SAS data file or an SQL view that references a SAS data file.

For direct access using KEY=, the master data set can be a SAS data file or the DBMS engine for the LIBNAME statement. If it is a SAS file, it must be indexed and the index name must be specified on the KEY= option.

For a DBMS, the KEY= is set to the keyword DBKEY and the column names to use as an index must be specified on the DBKEY= data set option. These column names are used in constructing a WHERE expression that is passed to the DBMS.

### Tip

Instead of using a data set name, you can specify the physical pathname to the file, using syntax that your operating system understands. The pathname must be enclosed in single or double quotation marks.

### (data-set-options)

specifies one or more SAS data set options in parentheses after a SAS data set name.

### Note

The data set options specify actions that SAS is to take when it reads observations into the DATA step for processing. For a list of data set options, see [SAS Data Set Options: Reference](#).

### Tip

Data set options that apply to a data set list apply to all of the data sets in the list.

**transaction-data-set**

specifies the SAS data set that provides the values for matching access. These values are the values that you want to use to update the master data set.

### Restriction

Specify this data set only when the DATA step contains a BY statement.

### Tip

Instead of using a data set name, you can specify the physical pathname to the file, using syntax that your operating system understands. The pathname must be enclosed in single or double quotation marks.

**by-variable**

specifies one or more variables by which you identify corresponding observations.
**CUROBS=**`variable`
creates and names a variable that contains the observation number that was just read from the data set.

**END=**`variable`
creates and names a temporary variable that contains an end-of-file indicator.

**Restriction**
Do not use this argument in the same MODIFY statement with the **POINT=** argument. **POINT=** indicates that MODIFY uses random access. The value of the **END=** variable is never set to 1 for random access.

**Notes**
The variable, which is initialized to zero, is set to 1 when the MODIFY statement reads the last observation of the data set being modified (for sequential access ) or the last observation of the transaction data set (for matching access ). It is also set to 1 when MODIFY cannot find a match for a **KEY=** value (random access).

This variable is not added to any data set.

**KEY=**`index`
specifies a simple or composite index of the SAS data file that is being modified. The **KEY=** argument retrieves observations from that SAS data file based on index values that are supplied by like-named variables in another source of information.

**Default**
If the **KEY=** value is not found, the automatic variable _ERROR_ is set to 1, and the automatic variable _IORC_ receives the value corresponding to the SYSRC autocall macro's mnemonic _DSENOM. See “Automatic Variable _IORC_ and the SYSRC Autocall Macro” on page 219.

**Restriction**
**KEY=** processing is different for SAS/ACCESS engines. For more information, see the SAS/ACCESS documentation.

**Tips**
Use the **KEYRESET=** option to control whether a **KEY=** search should begin at the top of the index for the data set that is being read.

Examples of sources for index values include a separate SAS data set named in a **SET** statement and an external file that is read by an **INPUT** statement.

If duplicates exist in the master file, only the first occurrence is updated unless you use a **DO-LOOP** to execute a **SET** statement for the data set that is listed on the **KEY=**option for all duplicates in the master data set.

If duplicates exist in the transaction data set, and they are consecutive, use the **UNIQUE** option to force the search for a match in the master data set to begin at the top of the index. Write an accumulation statement to add each duplicate transaction to the observation in master. Without the **UNIQUE** option, only the first duplicate transaction observation updates the master.

If the duplicates in the transaction data set are not consecutive, the search begins at the beginning of the index each time, so that each
duplicate is applied to the master. Write an accumulation statement to add each duplicate to the master.

See
"KEYRESET=variable" on page 215
UNIQUE on page 216

Examples
"Example 5: Modifying Observations Located by an Index" on page 227
"Example 6: Handling Duplicate Index Values” on page 227
"Example 7: Controlling I/O” on page 229

KEYRESET=variable
controls whether a KEY= search should begin at the top of the index for the data set that is being read. When the value of the KEYRESET variable is 1, the index lookup begins at the top of the index. When the value of the KEYRESET variable is 0, the index lookup is not reset and the lookup continues where the prior lookup ended.

Interaction
The KEYRESET= option is similar to the UNIQUE option, except the KEYRESET= option enables you to determine when the KEY= search should begin at the top of the index again.

See
“KEY=index” on page 214
“UNIQUE” on page 216

NOBS=variable
creates and names a temporary variable whose value is usually the total number of observations in the input data set. For certain SAS views and sequential engines such as the TAPE and XML engines, SAS cannot determine the number of observations. In these cases, SAS sets the value of the NOBS= variable to the largest positive integer value available in the operating environment.

Note
At compilation time, SAS reads the descriptor portion of the data set and assigns the value of the NOBS= variable automatically. Thus, you can refer to the NOBS= variable before the MODIFY statement. The variable is available in the DATA step but is not added to the new data set.

Tip
The NOBS= and POINT= options are independent of each other.

Example
“Example 4: Modifying Observations Located by Observation Number” on page 225

POINT=variable
reads SAS data sets using random (direct) access by observation number. variable names a variable whose value is the number of the observation to read. The POINT= variable is available anywhere in the DATA step, but it is not added to any SAS data set.

Restrictions
You cannot use the POINT= option with any of these items:

- BY statement
- WHERE statement
- WHERE= data set option
- transport format data sets
sequential data sets (on tape or disk)
a table from another vendor’s relational database management system

You can use POINT= with compressed data sets only if the data set was created with the POINTOBS= data set option set to YES, the default value.

You can use the random access method on compressed files only with SAS version 7 and beyond.

Requirements When using the POINT= argument, include one or both of these programming constructs:
- a STOP statement
- programming logic that checks for an invalid value of the POINT= variable

Because POINT= reads only the specified observations, SAS cannot detect an end-of-file condition as it would if the file were being read sequentially. Because detecting an end-of-file condition terminates a DATA step automatically, failure to substitute another means of terminating the DATA step when you use POINT= can cause the DATA step to go into a continuous loop.

Tip If the POINT= value does not match an observation number, SAS sets the automatic variable _ERROR_ to 1.

Example “Example 4: Modifying Observations Located by Observation Number” on page 225

**UNIQUE**
causes a KEY= search always to begin at the top of the index for the data file being modified.

Restriction UNIQUE can appear only with the KEY= option.

Tip Use UNIQUE when there are consecutive duplicate KEY= values in the transaction data set, so that the search for a match in the master data set begins at the top of the index file for each duplicate transaction. You must include an accumulation statement or the duplicate values overwrite each other causing only the last transaction value to be the result in the master observation.

See “KEYRESET=variable” on page 215

Example “Example 6: Handling Duplicate Index Values” on page 227

**UPDATEMODE=MISSINGCHECK | NOMISSINGCHECK**
specifies whether missing variable values in a transaction data set are to be allowed to replace existing variable values in a master data set.

MISSINGCHECK prevents missing variable values in a transaction data set from replacing values in a master data set.
**NOMISSINGCHECK**
allows missing variable values in a transaction data set to replace values in a master data set by preventing the check from being performed.

**Default**
MISSINGCHECK

**Requirement**
The UPDATEMODE argument must be accompanied by a BY statement that specifies the variables by which observations are matched.

**Tip**
However, special missing values are the exception and they replace values in the master data set even when MISSINGCHECK is in effect.

---

**Details**

**Matching Access (Form 1)**
The matching access method uses the BY statement to match observations from the transaction data set with observations in the master data set. The BY statement specifies a variable that is in the transaction data set and the master data set.

When the MODIFY statement reads an observation from the transaction data set, it uses dynamic WHERE processing to locate the matching observation in the master data set. The observation in the master data set can be either
- replaced in the master data set with the value from the transaction data set
- deleted from the master data set
- appended to the master data set.

"Example 3: Modifying Observations Using a Transaction Data Set" on page 224 shows the matching access method.

**Duplicate BY Values (Form 1)**
Duplicates in the master and transaction data sets affect processing.
- If duplicates exist in the master data set, only the first occurrence is updated because the generated WHERE statement always finds the first occurrence in the master.
- If duplicates exist in the transaction data set, the duplicates are applied one on top of another unless you write an accumulation statement to add all of them to the master observation. Without the accumulation statement, the values in the duplicates overwrite each other so that only the value in the last transaction is the result in the master observation.

**Direct Access by Indexed Values (Form 2)**
This method requires that you use the KEY= option in the MODIFY statement to name an indexed variable from the data set that is being modified. Use another data source (typically a SAS data set named in a SET statement or an external file read by an INPUT statement) to provide a like-named variable whose values are supplied to the index. MODIFY uses the index to locate observations in the data set that is being modified.

"Example 5: Modifying Observations Located by an Index" on page 227 shows the direct-access-by-indexed-values method.
Duplicate Index Values (*Form 2*)

- If there are duplicate values of the indexed variable in the master data set, only the first occurrence is retrieved, modified, or replaced. Use a DO LOOP to execute a SET statement with the KEY= option multiple times to update all duplicates with the transaction value.

- If there are duplicate, *nonconsecutive* values in the like-named variable in the data source, MODIFY applies each transaction cumulatively to the first observation in the master data set whose index value matches the values from the data source. Therefore, only the value in the last duplicate transaction is the result in the master observation unless you write an accumulation statement to accumulate each duplicate transaction value in the master observation.

- If there are duplicate, *consecutive* values in the variable in the data source, the values from the first observation in the data source are applied to the master data set, but the DATA step terminates with an error when it tries to locate an observation in the master data set for the second duplicate from the data source. To avoid this error, use the UNIQUE option in the MODIFY statement. The UNIQUE option causes SAS to return to the top of the master data set before retrieving a match for the index value. You must write an accumulation statement to accumulate the values from all the duplicates. If you do not, only the last one applied is the result in the master observation.

  “Example 6: Handling Duplicate Index Values” on page 227 shows how to handle duplicate index values.

- If there are duplicate index values in both data sets, you can use SQL to apply the duplicates in the transaction data set to the duplicates in the master data set in a one-to-one correspondence.

Direct (Random) Access by Observation Number (*Form 3*)

You can use the POINT= option in the MODIFY statement to name a variable from another data source (not the master data set), whose value is the number of an observation that you want to modify in the master data set. MODIFY uses the values of the POINT= variable to retrieve observations in the data set that you are modifying. (You can use POINT= on a compressed data set only if the data set was created with the POINTOBS= data set option.)

It is good programming practice to validate the value of the POINT= variable and to check the status of the automatic variable _ERROR_.

  “Example 4: Modifying Observations Located by Observation Number” on page 225 shows the direct (random) access by observation number method.

**CAUTION**

**POINT= can result in infinite looping.** Be careful when you use POINT=, as failure to terminate the DATA step can cause the DATA step to go into a continuous loop. Use a STOP statement, programming logic that checks for an invalid value of the POINT= variable, or both.

Sequential Access (*Form 4*)

The sequential access method is the simplest form of the MODIFY statement, but it provides less control than the direct access methods. With the sequential access method, you can use the NOBS= and END= options to modify a data set; you do not use the POINT= or KEY= options.
Preparing Your Data Sets Before Using MODIFY

There are a number of things that you can do to improve performance and get the results that you want when using the MODIFY statement. For more information, see “Combining SAS Data Sets: Basic Concepts” in SAS Language Reference: Concepts.

Automatic Variable _IORC_ and the SYSRC Autocall Macro

The automatic variable _IORC_ contains the return code for each I/O operation that the MODIFY statement attempts to perform. The best way to test for values of _IORC_ is with the mnemonic codes that are provided by the SYSRC autocall macro. Each mnemonic code describes one condition. The mnemonics provide an easy method for testing problems in a DATA step program. These codes are useful:

_DSENMR
  specifies that the transaction data set observation does not exist on the master data set (used only with MODIFY and BY statements). If consecutive observations with different BY values do not find a match in the master data set, both of them return _DSENMR.

_DSEMTR
  specifies that multiple transaction data set observations with a given BY value do not exist on the master data set (used only with MODIFY and BY statements). If consecutive observations with the same BY values do not find a match in the master data set, the first observation returns _DSENMR and the subsequent observations return _DSEMTR.

_DSENOM
  specifies that the data set being modified does not contain the observation that is requested by the KEY= option or the POINT= option.

_SENOCHN
  specifies that SAS is attempting to execute an OUTPUT or REPLACE statement on an observation that contains a key value that duplicates one already existing on an indexed data set that requires unique key values.

_SOK
  specifies that the observation was located.

Note: The IORCMMSG function returns a formatted error message associated with the current value of _IORC_.

"Example 7: Controlling I/O" on page 229 shows how to use the automatic variable _IORC_ and the SYSRC autocall macro.

Writing Observations When MODIFY Is Used in a DATA Step

The way SAS writes observations to a SAS data set when the DATA step contains a MODIFY statement depends on whether certain other statements are present. The possibilities are

no explicit statement
  writes the current observation to its original place in the SAS data set. The action occurs as the last action in the step (as if a REPLACE statement were the last statement in the step).
OUTPUT statement
if no data set is specified in the OUTPUT statement, writes the current
observation to the end of all data sets that are specified in the DATA step. If a
data set is specified, the statement writes the current observation to the end of
the data set that is indicated. The action occurs at the point in the DATA step
where the OUTPUT statement appears.

REPLACE <data-set-name> statement
rewrites the current observation in the specified data set or data sets, or, if no
argument is specified, rewrites the current observation in each data set specified
in the DATA statement. The action occurs at the point of the REPLACE
statement.

REMOVE <data-set-name> statement
deletes the current observation in the specified data set or data sets, or, if no
argument is specified, deletes the current observation in each data set specified
in the DATA statement. The deletion can be a physical one or a logical one,
depending on the characteristics of the engine that maintains the data set.

Remember the following as you work with these statements:

- When no OUTPUT, REPLACE, or REMOVE statement is specified, the default
  action is REPLACE.
- The OUTPUT, REPLACE, and REMOVE statements are independent of each
  other. You can code multiple OUTPUT, REPLACE, and REMOVE statements to
  apply to one observation. However, once an OUTPUT, REPLACE, or REMOVE
  statement executes, the MODIFY statement must execute again before the next
  REPLACE or REMOVE statement executes.

You can use OUTPUT and REPLACE in this example of conditional logic
because only one of the REPLACE or OUTPUT statements executes per
observation.

data master;
   modify master trans; by key;
   if _iorc_=0 then replace;
   else
      output;
run;

You should not use multiple REPLACE operations on the same observation as in
this example.

data master;
   modify master;
   x=1;
   replace;
   replace;
run;

You can code multiple OUTPUT statements per observation. However, be
careful when you use multiple OUTPUT statements. It is possible to go into an
infinite loop with just one OUTPUT statement.

data master;
   modify master;
   output;
run;
Using OUTPUT, REPLACE, or REMOVE in a DATA step overrides the default replacement of observations. If you use any one of these statements in a DATA step, you must explicitly program each action that you want to take.

If both an OUTPUT statement and a REPLACE or REMOVE statement execute on a given observation, perform the OUTPUT action last to keep the position of the observation pointer correct.

“Example 8: Replacing and Removing Observations and Writing Observations to Different SAS Data Sets” on page 231 shows how to use the OUTPUT, REMOVE, and REPLACE statements to write observations.

Missing Values and the MODIFY Statement

By default, the UPDATEMODE=MISSINGCHECK option is in effect, so missing values in the transaction data set do not replace existing values in the master data set. Therefore, if you want to update some but not all variables and if the variables that you want to update differ from one observation to the next, set to missing those variables that are not changing. If you want missing values in the transaction data set to replace existing values in the master data set, use UPDATEMODE=NOMISSINGCHECK.

Even when UPDATEMODE=MISSINGCHECK is in effect, you can replace existing values with missing values by using special missing value characters in the transaction data set. To create the transaction data set, use the MISSING statement in the DATA step. If you define one of the special missing values A through Z for the transaction data set, SAS updates numeric variables in the master data set to that value.

If you want the resulting value in the master data set to be a regular missing value, use a single underscore (_) to represent missing values in the transaction data set. The resulting value in the master data set is a period (.) for missing numeric values and a blank for missing character values.

For more information about defining and using special missing value characters, see “MISSING Statement” on page 212.

Using MODIFY with Data Set Options

If you use data set options (such as KEEP=) in your program, then use the options in the MODIFY statement for the master data set. Using data set options in the DATA statement might produce unexpected results.

Using MODIFY in a SAS/SHARE Environment

In a SAS/SHARE environment, the MODIFY statement accesses an observation in Update mode. That is, the observation is locked from the time MODIFY reads it until a REPLACE or REMOVE statement executes. At that point the observation is unlocked. It cannot be accessed until it is re-read with the MODIFY statement. The MODIFY statement opens the data set in Update mode, but the control level is based on the statement used. For example, KEY= and POINT= are member-level locking. For more information, see SAS/SHARE User’s Guide.

Comparisons

When you use a MERGE, SET, or UPDATE statement in a DATA step, SAS creates a new SAS data set. The data set descriptor of the new copy can be different from the old one (variables added or deleted, labels changed, and so
When you use a MODIFY statement in a DATA step, however, SAS does not create a new copy of the data set. As a result, the data set descriptor cannot change.

For information about DBMS replacement rules, see the SAS/ACCESS documentation.

- If you use a BY statement with a MODIFY statement, MODIFY works much like the UPDATE statement, except in these circumstances.
  - neither the master data set nor the transaction data set needs to be sorted or indexed. (The BY statement that is used with MODIFY triggers dynamic WHERE processing.)

  Note: Dynamic WHERE processing can be costly if the MODIFY statement modifies a SAS data set that is not in sorted order or has not been indexed. Having the master data set in sorted order or indexed and having the transaction data set in sorted order reduces processing overhead, especially for large files.

  - both the master data set and the transaction data set can have observations with duplicate values of the BY variables. MODIFY treats the duplicates as described in “Duplicate BY Values (Form 1)” on page 217.
  - MODIFY cannot make any changes to the descriptor information of the data set as UPDATE can. Thus, it cannot add or delete variables, change variable labels, and so on.

Examples:

Example 1: Input Data Set for Examples

The examples modify the INVTY.STOCK data set. INVTY.STOCK contains these variables:

PARTNO
  is a character variable with a unique value identifying each tool number.

DESC
  is a character variable with the text description of each tool.

INSTOCK
  is a numeric variable with a value describing how many units of each tool the company has in stock.

RECDATE
  is a numeric variable containing the SAS date value that is the day for which INSTOCK values are current.

PRICE
  is a numeric variable with a value that describes the unit price for each tool.

In addition, INVTY.STOCK contains a simple index on PARTNO. This DATA step creates INVTY.STOCK.

```sas
libname invty 'SAS-library';
data invty.stock(index=(partno));
  input PARTNO $ DESC $ INSTOCK @17 RECDATE date7. @25 PRICE;
  format recdate date7. ;
```
Example 2: Modifying All Observations

This example replaces the date on all of the records in the data set INVTY.STOCK with the current date. It also replaces the value of the variable RECDATE with the current date for all observations in INVTY.STOCK.

```sas
data invty.stock;
  modify invty.stock;
  recdate=today();
run;
proc print data=invty.stock noobs;
title 'INVTY.STOCK';
run;
```

**Output 2.21**  Results of Updating the RECDATE Field

<table>
<thead>
<tr>
<th>PARTNO</th>
<th>DESC</th>
<th>INSTOCK</th>
<th>RECDATE</th>
<th>PRICE</th>
</tr>
</thead>
<tbody>
<tr>
<td>K89R</td>
<td>seal</td>
<td>34</td>
<td>08DEC10</td>
<td>245.00</td>
</tr>
<tr>
<td>M4J7</td>
<td>sander</td>
<td>98</td>
<td>08DEC10</td>
<td>45.88</td>
</tr>
<tr>
<td>LK43</td>
<td>filter</td>
<td>121</td>
<td>08DEC10</td>
<td>10.99</td>
</tr>
<tr>
<td>MN21</td>
<td>brace</td>
<td>43</td>
<td>08DEC10</td>
<td>27.87</td>
</tr>
<tr>
<td>BC85</td>
<td>clamp</td>
<td>80</td>
<td>08DEC10</td>
<td>9.55</td>
</tr>
<tr>
<td>NCF3</td>
<td>valve</td>
<td>198</td>
<td>08DEC10</td>
<td>24.50</td>
</tr>
<tr>
<td>KJ66</td>
<td>cutter</td>
<td>6</td>
<td>08DEC10</td>
<td>19.77</td>
</tr>
<tr>
<td>UYN7</td>
<td>rod</td>
<td>211</td>
<td>08DEC10</td>
<td>11.55</td>
</tr>
<tr>
<td>JD03</td>
<td>switch</td>
<td>383</td>
<td>08DEC10</td>
<td>13.99</td>
</tr>
<tr>
<td>BV1E</td>
<td>timer</td>
<td>26</td>
<td>08DEC10</td>
<td>34.50</td>
</tr>
</tbody>
</table>

The MODIFY statement opens INVTY.STOCK for update processing. SAS reads one observation of INVTY.STOCK for each iteration of the DATA step and performs any operations that the code specifies. In this case, the code replaces the value of RECDATE with the result of the TODAY function for every iteration of the DATA
Example 3: Modifying Observations Using a Transaction Data Set

This example adds the quantity of newly received stock to its data set INVTY.STOCK as well as updating the date on which stock was received. The transaction data set ADDINV in the Work library contains the new data.

The ADDINV data set is the data set that contains the updated information. ADDINV contains these variables:

- **PARTNO** is a character variable that corresponds to the indexed variable PARTNO in INVTY.STOCK.
- **NWSTOCK** is a numeric variable that represents quantities of newly received stock for each tool.

ADDINV is the second data set in the MODIFY statement. SAS uses it as the transaction data set and reads each observation from ADDINV sequentially. Because the BY statement specifies the common variable PARTNO, MODIFY finds the first occurrence of the value of PARTNO in INVTY.STOCK that matches the value of PARTNO in ADDINV. For each observation with a matching value, the DATA step changes the value of RECDATE to today's date and replaces the value of INSTOCK with the sum of INSTOCK and NWSTOCK (from ADDINV). MODIFY does not add NWSTOCK to the INVTY.STOCK data set because that would modify the data set descriptor. Thus, it is not necessary to put NWSTOCK in a DROP statement.

This example specifies ADDINV as the transaction data set that contains information to modify INVTY.STOCK. A BY statement specifies the shared variable whose values locate the observations in INVTY.STOCK.

This DATA step creates ADDINV.

```sas
   data addinv;
      input PARTNO $ NWSTOCK;
   datalines;
   K89R 55
   MAJ7 21
   LK43 43
   MN21 73
   BC85 57
   NCF3 90
   KJ66 2
   UYN7 108
   JD03 55
   BV1E 27
   ;
```

This DATA step uses values from ADDINV to update INVTY.STOCK.

```sas
   libname invty 'SAS-library';
   data invty.stock;
      modify invty.stock addinv;
      by partno;
      RECDATE=today();
      INSTOCK=instock+ nwstock;
      if _iorc_=0 then replace;
```
run;

proc print data=invty.stock noobs;
  title 'INVTY.STOCK';
run;

Output 2.22 Results of Updating the INSTOCK and RECDATE Fields

<table>
<thead>
<tr>
<th>PARTNO</th>
<th>DESC</th>
<th>INSTOCK</th>
<th>RECDATE</th>
<th>PRICE</th>
</tr>
</thead>
<tbody>
<tr>
<td>K89R</td>
<td>seal</td>
<td>89</td>
<td>08DEC10</td>
<td>245.00</td>
</tr>
<tr>
<td>M4J7</td>
<td>sander</td>
<td>119</td>
<td>08DEC10</td>
<td>45.88</td>
</tr>
<tr>
<td>LK43</td>
<td>filter</td>
<td>164</td>
<td>08DEC10</td>
<td>10.99</td>
</tr>
<tr>
<td>MN21</td>
<td>brace</td>
<td>116</td>
<td>08DEC10</td>
<td>27.87</td>
</tr>
<tr>
<td>BC85</td>
<td>clamp</td>
<td>137</td>
<td>08DEC10</td>
<td>9.55</td>
</tr>
<tr>
<td>NCF3</td>
<td>valve</td>
<td>288</td>
<td>08DEC10</td>
<td>24.50</td>
</tr>
<tr>
<td>KJ66</td>
<td>cutter</td>
<td>8</td>
<td>08DEC10</td>
<td>19.77</td>
</tr>
<tr>
<td>UYN7</td>
<td>rod</td>
<td>319</td>
<td>08DEC10</td>
<td>11.55</td>
</tr>
<tr>
<td>JD03</td>
<td>switch</td>
<td>438</td>
<td>08DEC10</td>
<td>13.99</td>
</tr>
<tr>
<td>BV1E</td>
<td>timer</td>
<td>53</td>
<td>08DEC10</td>
<td>34.50</td>
</tr>
</tbody>
</table>

Example 4: Modifying Observations Located by Observation Number

This example reads the data set NEWP, determines which observation number in INVTY.STOCK to update based on the value of TOOL_OBS, and performs the update. This example explicitly specifies the update activity by using an assignment statement to replace the value of PRICE with the value of NEWP.

The data set NEWP contains these two variables:

<table>
<thead>
<tr>
<th>TOOL_OBS</th>
<th>NEWP</th>
</tr>
</thead>
</table>
| contains the observation number of each tool in the tool company's master data set, INVTY.STOCK.

| NEWP | contains the new price for each tool.

This DATA step creates NEWP.

data newp;
  input TOOL_OBS NEWP;
  datalines;
  1 251.00
  2 49.33
  3 12.32
  4 30.00
  5 15.00
  6 25.75
This DATA step updates INVTY.STOCK.

```
libname invty 'SAS-library';
data invty.stock;
    set newp;
    modify invty.stock point=tool_obs
        nobs=max_obs;
    if _error_=1 then
        do;
            put 'ERROR occurred for TOOL_OBS=' tool_obs /
                'during DATA step iteration' _n_ /
                'TOOL_OBS value might be out of range.';
            _error_=0;
            stop;
        end;
    PRICE=newp;
    RECDATE=today();
run;
```

proc print data=invty.stock noobs;
    title 'INVTY.STOCK';
run;

**Output 2.23  Results of Updating the RECDATE and PRICE Fields**

<table>
<thead>
<tr>
<th>PARTNO</th>
<th>DESC</th>
<th>INSTOCK</th>
<th>RECDATE</th>
<th>PRICE</th>
</tr>
</thead>
<tbody>
<tr>
<td>K89R</td>
<td>seal</td>
<td>34</td>
<td>08DEC10</td>
<td>251.00</td>
</tr>
<tr>
<td>M4J7</td>
<td>sander</td>
<td>98</td>
<td>08DEC10</td>
<td>49.33</td>
</tr>
<tr>
<td>LK43</td>
<td>filter</td>
<td>121</td>
<td>08DEC10</td>
<td>12.32</td>
</tr>
<tr>
<td>MN21</td>
<td>brace</td>
<td>43</td>
<td>08DEC10</td>
<td>30.00</td>
</tr>
<tr>
<td>BC85</td>
<td>clamp</td>
<td>80</td>
<td>08DEC10</td>
<td>15.00</td>
</tr>
<tr>
<td>NCF3</td>
<td>valve</td>
<td>198</td>
<td>08DEC10</td>
<td>25.75</td>
</tr>
<tr>
<td>KJ56</td>
<td>cutter</td>
<td>6</td>
<td>08DEC10</td>
<td>22.00</td>
</tr>
<tr>
<td>UYN7</td>
<td>rod</td>
<td>211</td>
<td>08DEC10</td>
<td>14.00</td>
</tr>
<tr>
<td>JD303</td>
<td>switch</td>
<td>363</td>
<td>08DEC10</td>
<td>14.32</td>
</tr>
<tr>
<td>BV1E</td>
<td>timer</td>
<td>26</td>
<td>08DEC10</td>
<td>35.00</td>
</tr>
</tbody>
</table>
Example 5: Modifying Observations Located by an Index

This example uses the KEY= option to identify observations to retrieve by matching the values of PARTNO from ADDINV with the indexed values of PARTNO in INVTY.STOCK. ADDINV is created in "Example 3: Modifying Observations Using a Transaction Data Set" on page 224.

KEY= supplies index values that allow MODIFY to access directly the observations to update. No dynamic WHERE processing occurs. In this example, you specify that the value of INSTOCK in the master data set INVTY.STOCK increases by the value of the variable NWSTOCK from the transaction data set ADDINV.

```
libname invty 'SAS-library';
data invty.stock;
  set addinv;
  modify invty.stock key=partno;
  INSTOCK=instock+nwstock;
  RECDATE=today();
  if _iorc_=0 then replace;
run;
```

```
proc print data=invty.stock noobs;
  title 'INVTY.STOCK';
run;
```

---

Output 2.24  Results of Updating the INSTOCK and RECDATE Fields by Using an Index

<table>
<thead>
<tr>
<th>PARTNO</th>
<th>DESC</th>
<th>INSTOCK</th>
<th>RECDATE</th>
<th>PRICE</th>
</tr>
</thead>
<tbody>
<tr>
<td>K89R</td>
<td>seal</td>
<td>89</td>
<td>08DEC10</td>
<td>245.00</td>
</tr>
<tr>
<td>M4J7</td>
<td>sander</td>
<td>119</td>
<td>08DEC10</td>
<td>45.88</td>
</tr>
<tr>
<td>LK43</td>
<td>filter</td>
<td>164</td>
<td>08DEC10</td>
<td>10.99</td>
</tr>
<tr>
<td>MN21</td>
<td>brace</td>
<td>116</td>
<td>08DEC10</td>
<td>27.87</td>
</tr>
<tr>
<td>BC85</td>
<td>clamp</td>
<td>137</td>
<td>08DEC10</td>
<td>9.55</td>
</tr>
<tr>
<td>NCF3</td>
<td>valve</td>
<td>288</td>
<td>08DEC10</td>
<td>24.50</td>
</tr>
<tr>
<td>KJ66</td>
<td>cutter</td>
<td>8</td>
<td>08DEC10</td>
<td>19.77</td>
</tr>
<tr>
<td>UYN7</td>
<td>rod</td>
<td>319</td>
<td>08DEC10</td>
<td>11.55</td>
</tr>
<tr>
<td>JD03</td>
<td>switch</td>
<td>438</td>
<td>08DEC10</td>
<td>13.99</td>
</tr>
<tr>
<td>BV1E</td>
<td>timer</td>
<td>53</td>
<td>08DEC10</td>
<td>34.50</td>
</tr>
</tbody>
</table>

Example 6: Handling Duplicate Index Values

This example shows how MODIFY handles duplicate values of the variable in the SET data set that is supplying values to the index on the master data set.

The NEWINV data set is the data set that contains the updated information. NEWINV contains these variables:
PARTNO

is a character variable that corresponds to the indexed variable PARTNO in INVTY.STOCK. The NEWINV data set contains duplicate values for PARTNO; **M4J7** appears twice.

NWSTOCK

is a numeric variable that represents quantities of newly received stock for each tool.

This DATA step creates NEWINV.

```sas
data newinv;
  input PARTNO $ NWSTOCK;
datalines;
K89R 55
M4J7 21
M4J7 26
LK43 43
MN21 73
BC85 57
NCF3 90
KJ66 2
UYN7 108
JD03 55
BV1E 27;
```

This DATA step terminates with an error when it tries to locate an observation in INVTY.STOCK to match with the second occurrence of **M4J7** in NEWINV.

```sas
libname invty 'SAS-library';
/* This DATA step terminates with an error! */
data invty.stock;
  set newinv;
  modify invty.stock key=partno;
  INSTOCK=instock+nwstock;
  RECDATE=today();
run;
```

This message appears in the SAS log.

```
ERROR: No matching observation was found in MASTER data set.
PARTNO=M4J7 NWSTOCK=26 DESC=sander INSTOCK=166 RECDATE=08DEC10 PRICE=45.88 _ERROR_=1 _IORC_=1230015 _N_=3
NOTE: The SAS System stopped processing this step because of errors.
NOTE: There were 3 observations read from the data set WORK.NEWINV.
NOTE: The data set INVTY.STOCK has been updated. There were 2 observations rewritten, 0 observations added and 0 observations deleted.
```

Adding the UNIQUE option to the MODIFY statement avoids the error in the previous DATA step. The UNIQUE option causes the DATA step to return to the top of the index each time it looks for a match for the value from the SET data set. Thus, it finds the **M4J7** in the MASTER data set for each occurrence of **M4J7** in the SET data set. The updated result for **M4J7** in the output shows that both values of NWSTOCK from NEWINV for **M4J7** are added to the value of INSTOCK for **M4J7** in INVTY.STOCK. An accumulation statement sums the values; without it, only the value of the last instance of **M4J7** would be the result in INVTY.STOCK.

```sas
data invty.stock;
  set newinv;
```
modify invty.stock key=partno / unique;
INSTOCK=instock+nwstock;
RECDATE=today();
if _iorc_=0 then replace;
run;
proc print data=invty.stock noobs;
title 'Results of Using the UNIQUE Option';
runc;

Output 2.25  Results of Updating the INSTOCK and RECDATE Fields by Using the UNIQUE Option

<table>
<thead>
<tr>
<th>PARTNO</th>
<th>DESC</th>
<th>INSTOCK</th>
<th>RECDATE</th>
<th>PRICE</th>
</tr>
</thead>
<tbody>
<tr>
<td>K89R</td>
<td>seal</td>
<td>89</td>
<td>08DEC10</td>
<td>245.00</td>
</tr>
<tr>
<td>M4J7</td>
<td>sander</td>
<td>145</td>
<td>08DEC10</td>
<td>45.88</td>
</tr>
<tr>
<td>LK43</td>
<td>filter</td>
<td>164</td>
<td>08DEC10</td>
<td>10.99</td>
</tr>
<tr>
<td>MN21</td>
<td>brace</td>
<td>116</td>
<td>08DEC10</td>
<td>27.87</td>
</tr>
<tr>
<td>BC85</td>
<td>clamp</td>
<td>137</td>
<td>08DEC10</td>
<td>9.55</td>
</tr>
<tr>
<td>NCF3</td>
<td>valve</td>
<td>288</td>
<td>08DEC10</td>
<td>24.50</td>
</tr>
<tr>
<td>KJ66</td>
<td>cutter</td>
<td>8</td>
<td>08DEC10</td>
<td>19.77</td>
</tr>
<tr>
<td>UYN7</td>
<td>rod</td>
<td>319</td>
<td>08DEC10</td>
<td>11.55</td>
</tr>
<tr>
<td>JD03</td>
<td>switch</td>
<td>438</td>
<td>08DEC10</td>
<td>13.99</td>
</tr>
<tr>
<td>BV1E</td>
<td>timer</td>
<td>53</td>
<td>08DEC10</td>
<td>34.50</td>
</tr>
</tbody>
</table>

Example 7: Controlling I/O

This example uses the SYSRC autocall macro and the _IORC_ automatic variable to control I/O condition. This technique helps prevent unexpected results that could go undetected. This example uses the direct access method with an index to update INVTY.STOCK. The data in the NEWSHIP data set updates INVTY.STOCK.

This DATA step creates NEWSHIP.

data newship;
  input PARTNO $ DESC $ NWSTOCK @17
        SHPDATE date7. @25 NWPRICE;
datalines;
K89R seal 14    14nov96 245.00
M4J7 sander 24   23aug96 47.98
LK43 filter 11   25jan97 14.99
MN21 brace 9     09jan97 27.87
BC85 clamp 12    09dec96 10.00
ME34 cutter 8    14nov96 14.50
;

Each WHEN clause in the SELECT statement specifies actions for each input/output return code that is returned by the SYSRC autocall macro:
_SOK indicates that the MODIFY statement executed successfully.

_DSENOM indicates that no matching observation was found in INVTY.STOCK. The OUTPUT statement specifies that the observation be appended to INVTY.STOCK. See the last observation in the output.

If any other code is returned by SYSRC, the DATA step terminates and the PUT statement writes the message to the SAS log.

libname invty 'SAS-library';
data invty.stock;
   set newship;
   modify invty.stock key=partno;
   select (_iorc_);
      when (%sysrc(_sok)) do;
         INSTOCK=instock+nwstock;
         RECDATE=shpdate;
         PRICE=nwprice;
         replace;
      end;
      when (%sysrc(_dsenom)) do;
         INSTOCK=nwstock;
         RECDATE=shpdate;
         PRICE=nwprice;
         output;
         _error_=0;
      end;
      otherwise do;
         put
            'An unexpected I/O error has occurred.'/
            'Check your data and your program';
         _error_=0;
         stop;
      end;
   end;
run;

proc print data=invty.stock noobs;
   title 'INVTY.STOCK Data Set';
run;
Example 8: Replacing and Removing Observations and Writing Observations to Different SAS Data Sets

This example shows that you can replace and remove (delete) observations and write observations to different data sets. Further, this example shows that if an OUTPUT, REPLACE, or REMOVE statement is present, you must specify explicitly what action to take because no default statement is generated.

The parts that were received in 1997 are written to INVTY.STOCK97 and are removed from INVTY.STOCK. Likewise, the parts that were received in 1995 are written to INVTY.STOCK95 and are removed from INVTY.STOCK. Only the parts that were received in 1996 remain in INVTY.STOCK, and the PRICE is updated only in INVTY.STOCK.

```sas
libname invty 'SAS-library';
data invty.stock invty.stock95 invty.stock97;
  modify invty.stock;
  if recdate>'01jan97'd then do;
    output invty.stock97;
    remove invty.stock;
  end;
  else if recdate<'01jan96'd then do;
    output invty.stock95;
    remove invty.stock;
  end;
  else do;
    price=price*1.1;
    replace invty.stock;
  end;
```

**INVTY.STOCK Data Set**

<table>
<thead>
<tr>
<th>PARTNO</th>
<th>DESC</th>
<th>INSTOCK</th>
<th>RECDATA</th>
<th>PRICE</th>
</tr>
</thead>
<tbody>
<tr>
<td>K89R</td>
<td>seal</td>
<td>49</td>
<td>14NOV96</td>
<td>245.00</td>
</tr>
<tr>
<td>M4J7</td>
<td>sander</td>
<td>122</td>
<td>23AUG96</td>
<td>47.98</td>
</tr>
<tr>
<td>LK43</td>
<td>filter</td>
<td>132</td>
<td>25JAN97</td>
<td>14.99</td>
</tr>
<tr>
<td>MN21</td>
<td>brace</td>
<td>52</td>
<td>08JAN97</td>
<td>27.87</td>
</tr>
<tr>
<td>BC85</td>
<td>clamp</td>
<td>92</td>
<td>09DEC96</td>
<td>10.00</td>
</tr>
<tr>
<td>NCF3</td>
<td>valve</td>
<td>198</td>
<td>20MAR96</td>
<td>24.50</td>
</tr>
<tr>
<td>KJ66</td>
<td>cutter</td>
<td>6</td>
<td>18JUN96</td>
<td>19.77</td>
</tr>
<tr>
<td>UYN7</td>
<td>rod</td>
<td>211</td>
<td>09SEP96</td>
<td>11.55</td>
</tr>
<tr>
<td>JD03</td>
<td>switch</td>
<td>383</td>
<td>08JAN97</td>
<td>13.99</td>
</tr>
<tr>
<td>BV1E</td>
<td>timer</td>
<td>25</td>
<td>03JAN97</td>
<td>34.50</td>
</tr>
<tr>
<td>ME34</td>
<td>cutter</td>
<td>8</td>
<td>14NOV96</td>
<td>14.50</td>
</tr>
</tbody>
</table>
run;

proc print data=invty.stock noobs;
  title 'New Prices for Stock Received in 1996';
run;

Output 2.27  Output from Writing Observations to a Specific SAS Data Set

<table>
<thead>
<tr>
<th>PARTNO</th>
<th>DESC</th>
<th>INSTOCK</th>
<th>RECDATE</th>
<th>PRICE</th>
</tr>
</thead>
<tbody>
<tr>
<td>LK43</td>
<td>filter</td>
<td>121</td>
<td>19MAY96</td>
<td>12.089</td>
</tr>
<tr>
<td>MN21</td>
<td>brace</td>
<td>43</td>
<td>10AUG96</td>
<td>30.657</td>
</tr>
<tr>
<td>BC85</td>
<td>clamp</td>
<td>80</td>
<td>16AUG96</td>
<td>10.505</td>
</tr>
<tr>
<td>NCF3</td>
<td>valve</td>
<td>198</td>
<td>20MAR96</td>
<td>26.950</td>
</tr>
<tr>
<td>KJ66</td>
<td>cutter</td>
<td>6</td>
<td>18JUN96</td>
<td>21.747</td>
</tr>
<tr>
<td>UYN7</td>
<td>rod</td>
<td>211</td>
<td>09SEP96</td>
<td>12.705</td>
</tr>
</tbody>
</table>

See Also
- “Reading, Combining, and Modifying SAS Data Sets” in SAS Language Reference: Concepts
- SAS SQL Procedure User’s Guide

Statements:
- “MISSING Statement” on page 212
- “OUTPUT Statement” on page 233
- “REMOVE Statement” on page 273
- “REPLACE Statement” on page 277
- “UPDATE Statement” on page 309

Null Statement
Signals the end of data lines or acts as a placeholder.

Note: The Null Statement has moved to SAS Global Statements.

OPTIONS Statement
Specifies or changes the value of one or more SAS system options.
OUTPUT Statement

Writes the current observation to a SAS data set.

Valid in: DATA step
Categories: Action, CAS
Type: Executable

Syntax

```
OUTPUT <data-set-name(s)>;
```

Without Arguments

Using OUTPUT without arguments causes the current observation to be written to all data sets that are named in the DATA statement.

If a MODIFY statement is present, OUTPUT with no arguments writes the current observation to the end of the data set that is specified in the MODIFY statement.

Arguments

`data-set-name`

specifies the name of a data set to which SAS writes the observation.

Restriction

All names specified in the OUTPUT statement must also appear in the DATA statement.

Tips

Instead of using a data set name, you can specify the physical pathname to the file, using syntax that your operating system understands. The pathname must be enclosed in single or double quotation marks.

You can specify up to as many data sets in the OUTPUT statement as you specified in the DATA statement for that DATA step.

Details

When and Where the OUTPUT Statement Writes Observations

The OUTPUT statement tells SAS to write the current observation to a SAS data set immediately, not at the end of the DATA step. If no data set name is specified in the OUTPUT statement, the observation is written to the data set or data sets that are listed in the DATA statement.

Implicit versus Explicit Output

By default, every DATA step contains an implicit OUTPUT statement at the end of each iteration that tells SAS to write observations to the data set or data sets that
are being created. Placing an explicit OUTPUT statement in a DATA step overrides the automatic output, and SAS adds an observation to a data set only when an explicit OUTPUT statement is executed. Once you use an OUTPUT statement to write an observation to any one data set, however, there is no implicit OUTPUT statement at the end of the DATA step. In this situation, a DATA step writes an observation to a data set only when an explicit OUTPUT executes. You can use the OUTPUT statement alone or as part of an IF-THEN or SELECT statement or in DO-loop processing.

When Using the MODIFY Statement

When you use the MODIFY statement with the OUTPUT statement, the REMOVE and REPLACE statements override the implicit write action at the end of each DATA step iteration. For more information, see "Comparisons" on page 234. If both the OUTPUT statement and a REPLACE or REMOVE statement execute on a given observation, perform the output action last to keep the position of the observation pointer correct.

Comparisons

- OUTPUT writes observations to a SAS data set; PUT writes variable values or text strings to an external file or the SAS log.
- To control when an observation is written to a specified output data set, use the OUTPUT statement. To control which variables are written to a specified output data set, use the KEEP= or DROP= data set option in the DATA statement, or use the KEEP or DROP statement.
- When you use the OUTPUT statement with the MODIFY statement, these items apply:
  - Using an OUTPUT, REPLACE, or REMOVE statement overrides the default write action at the end of a DATA step. (OUTPUT is the default action; REPLACE becomes the default action when a MODIFY statement is used.) If you use any of these statements in a DATA step, you must explicitly program output for the new observations that are added to the data set.
  - The OUTPUT, REPLACE, and REMOVE statements are independent of each other. More than one statement can apply to the same observation, as long as the sequence is logical.
  - If both an OUTPUT and a REPLACE or REMOVE statement execute on a given observation, perform the OUTPUT action last to keep the position of the observation pointer correct.

Examples:

Example 1: Sample Uses of OUTPUT

These examples show how you can use an OUTPUT statement.

- This line of code writes the current observation to a SAS data set.

```sas
output;
```

- This line of code writes the current observation to a SAS data set when a specified condition is true.

```sas
if deptcode gt 2000 then output;
```
This line of code writes an observation to the data set MARKUP when the PHONE value is missing.

if phone=. then output markup;

Example 2: Creating Multiple Observations from Each Line of Input

You can create two or more observations from each line of input data. This SAS program creates three observations in the data set RESPONSE for each observation in the data set SULFA.

```sas
data response(drop=time1-time3);
  set sulfa;
  time=time1;
  output;
  time=time2;
  output;
  time=time3;
  output;
run;
```

Example 3: Creating Multiple Data Sets from a Single Input File

You can create more than one SAS data set from one input file. In this example, OUTPUT writes observations to two data sets, OZONE and OXIDES.

```sas
data ozone oxides;
  infile file-specification;
  input city $ 1-15 date date9.
    chemical $ 26-27 ppm 29-30;
  if chemical='O3' then output ozone;
  else output oxides;
run;
```

Example 4: Creating One Observation from Several Lines of Input

You can combine several input observations into one observation. In this example, OUTPUT creates one observation that totals the values of DEFECTS in the first ten observations of the input data set:

```sas
data discards;
  set gadgets;
  drop defects;
  reps+1;
  if reps=1 then total=0;
  total+defects;
  if reps=10 then do;
    output;
    stop;
  end;
run;
```

See Also

Statements:

- “DATA Statement” on page 44
PAGE Statement

Skips to a new page in the SAS log.

Note: The PAGE Statement has moved to SAS Global Statements.

PUT Statement

Writes lines to the SAS log, to the SAS output window, or to an external location that is specified in the most recent FILE statement.

Valid in: DATA step
Categories: CAS, File-Handling
Type: Executable
Interaction: When using the DSD option in the FILE statement and using the PUT statement to write to the SAS log, PRINT, or an external file, missing values might be suppressed in the output. Missing values can be displayed if an explicit line break (/) is included in the PUT statement.

Syntax

```
PUT <specification(s)> <_ODS_> <@ | @@>;
```

Without Arguments

The PUT statement without arguments is called a null PUT statement. The null PUT statement can be used to perform these tasks:

- write the current output line to the current location, even if the current output line is blank.
- release an output line that is being held with a trailing @ by a previous PUT statement.

For an example, see “Example 5: Holding and Releasing Output Lines” on page 252. For more information, see “Using Line-Hold Specifiers” on page 246.

Arguments

`specification(s)`

specifies what is written, how it is written, and where it is written. The specification can include one or more of these items:
**variable**

specifies the variable whose value is written.

---

**Note:** Beginning with Version 7, you can specify column-mapped Output Delivery System variables in the PUT statement. This functionality is described briefly here in _ODS_ on page 238. It is documented more completely in the “PUT Statement: ODS” in SAS Output Delivery System: User’s Guide.

---

**(variable-list)**

specifies a list of variables whose values are written.

**Requirement** The (format-list) must follow the (variable-list).

**See** “PUT Statement: Formatted” on page 258

---

'**character-string**'

specifies a string of text, enclosed in quotation marks, to write.

**Tips**

To write a hexadecimal string in EBCDIC or ASCII, follow the ending quotation mark with an x.

If you use single quotation marks (’) or double quotation marks (“”) together (with no space in between them) as the string of text, SAS writes a single quotation mark (’) or double quotation mark (“), respectively.

**See** “List Output” on page 243

**Example**

This statement writes HELLO when the hexadecimal string is converted to ASCII characters.

```
put '68656C6C6F'x;
```

---

*n*  
specifies to repeat n times the subsequent character string.

**Example**

This statement writes a line of 132 underscores.

```
put 132*'_';
```

**Example** “Example 4: Underlining Text” on page 252

---

**pointer-control**

moves the output pointer to a specified line or column in the output buffer.

**See** “Column Pointer Controls ” on page 239

“Line Pointer Controls ” on page 240

---

**column-specifications**  
specifies which columns of the output line the values are written.

**See** “Column Output” on page 242

**Example** “Example 2: Moving the Pointer within a Page” on page 249

---

**format.**  
specifies a format to use when the variable values are written.
See “Formatted Output” on page 243

Example “Example 1: Using Multiple Output Styles in One PUT Statement” on page 248

(format-list) specifies a list of formats to use when the values of the preceding list of variables are written.

Restriction The (format-list) must follow the (variable-list).

See “PUT Statement: Formatted” on page 258

_INFILE_ writes the last input data record that is read either from the current input file or from the data lines that follow a DATALINES statement.

Tips _INFILE_ is an automatic variable that references the current INPUT buffer. You can use this automatic variable in other SAS statements.

If the most recent INPUT statement uses line-pointer controls to read multiple input data records, PUT _INFILE_ writes only the record that the input pointer is positioned on.

Example This PUT statement writes all the values of the first input data record.
input #3 score #1 name $ 6-23;
put _infile_;

Example “Example 6: Writing the Current Input Record to the SAS Log” on page 253

_ALL_ writes the values of all variables, which includes automatic variables, that are defined in the current DATA step by using named output.

See “Named Output” on page 244

_ODS_ moves data values for all columns (as defined by the ODS option in the FILE statement) into a special buffer, from which it is eventually written to the data component. The ODS option in the FILE statement defines the structure of the data component that holds the results of the DATA step.

Restriction Use _ODS_ only if you have previously specified the ODS option in the FILE statement.

Interaction _ODS_ writes data to a specific column only if a PUT statement has not already specified a variable for that column with a column pointer. That is, a variable specification for a column overrides the _ODS_ option.

Tip You can use the _ODS_ specification in conjunction with variable specifications and column pointers, and it can appear anywhere in a PUT statement.

@ | @@
holds an output line for the execution of the next PUT statement even across iterations of the DATA step. These line-hold specifiers are called trailing @ and double trailing @.

Restriction The trailing @ or double trailing @ must be the last item in the PUT statement.

Tip Use an @ or @@ to hold the pointer at its current location. The next PUT statement that executes writes to the same output line rather than to a new output line.

See “Using Line-Hold Specifiers” on page 246

Example “Example 5: Holding and Releasing Output Lines” on page 252

Column Pointer Controls

@n
moves the pointer to column n.

Range a positive integer

Example @15 moves the pointer to column 15 before the value of NAME is written:
put @15 name $10.;

Examples “Example 2: Moving the Pointer within a Page” on page 249

@numeric-variable
moves the pointer to the column given by the value of numeric-variable.

Range a positive integer

Tip If n is not an integer, SAS truncates the decimal portion and uses only the integer value. If n is zero or negative, the pointer moves to column 1.

Example The value of the variable A moves the pointer to column 15 before the value of NAME is written:
a=15;
put @a name $10.;

Example “Example 2: Moving the Pointer within a Page” on page 249

@expression
moves the pointer to the column that is given by the value of expression.

Range a positive integer
| Tip | If the value of expression is not an integer, SAS truncates the decimal value and uses only the integer value. If it is zero, the pointer moves to column 1. |
| Example | The result of the expression moves the pointer to column 15 before the value of NAME is written:  
```
b=5;
put @(b*3) name $10.;
```

| n | moves the pointer n columns. |
| Range | a positive integer or zero |
| Tip | If n is not an integer, SAS truncates the decimal portion and uses only the integer value. |
| Example | This statement moves the pointer to column 23, writes a value of LENGTH in columns 23 through 26, advances the pointer five columns, and writes the value of WIDTH in columns 32 through 35:  
```
put @23 length 4. +5 width 4.;
```

| numeric-variable | moves the pointer the number of columns given by the value of numeric-variable. |
| Range | a positive or negative integer or zero |
| Tip | If numeric-variable is not an integer, SAS truncates the decimal value and uses only the integer value. If numeric-variable is negative, the pointer moves backward. If the current column position becomes less than 1, the pointer moves to column 1. If the value is zero, the pointer does not move. If the value is greater than the length of the output buffer, the current line is written out and the pointer moves to column 1 on the next line. |
| Example | This statement moves the pointer to column 23, writes a value of LENGTH in columns 23 through 26, advances the pointer five columns, and writes the value of WIDTH in columns 32 through 35:  
```
put @23 length 4. +5 width 4.;
```

| expression | moves the pointer the number of columns given by expression. |
| Range | expression must result in an integer |
| Tip | If expression is not an integer, SAS truncates the decimal value and uses only the integer value. If expression is negative, the pointer moves backward. If the current column position becomes less than 1, the pointer moves to column 1. If the value is zero, the pointer does not move. If the value is greater than the length of the output buffer, the current line is written out and the pointer moves to column 1 on the next line. |
| Example | "Example 2: Moving the Pointer within a Page" on page 249 |

**Line Pointer Controls**

| n | moves the pointer to line n and column 1. |
| Range | a positive integer |
Example  The #2 moves the pointer to the second line before the value of ID is written in columns 3 and 4:
put @12 name $10. #2 id 3-4;

#numeric-variable

moves the pointer to the line given by the value of numeric-variable and to column 1.

Range  a positive integer

Tip  If the value of numeric-variable is not an integer, SAS truncates the decimal value and uses only the integer value.

#(expression)

moves the pointer to the line that is given by the value of expression and to column 1.

Range  Expression must result in a positive integer.

Tip  If the value of expression is not an integer, SAS truncates the decimal value and uses only the integer value.

/ advances the pointer to column 1 of the next line.

Note  If you try to use one or more "/" line pointer controls to add blank lines to the SAS log, SAS suppresses the blank lines. For other forms of output, the blank lines are produced.

Example  The values for NAME and AGE are written on one line, and then the pointer moves to the second line to write the value of ID in columns 3 and 4:
put name age / id 3-4;

Example  "Example 3: Moving the Pointer to a New Page" on page 250

OVERPRINT
causes the values that follow the keyword OVERPRINT to be printed on the most recently written output line.

Requirement  You must direct the output to a file. Set the N= option in the FILE statement to 1 and direct the PUT statements to a file.

Tips  OVERPRINT has no effect on lines that are written to a display.

Use OVERPRINT in combination with column pointer and line pointer controls to overprint text.

Example  This statement overprints underscores, starting in column 15, which underlines the title:
put @15 'Report Title' overprint
    @15 '____________';

Example  "Example 4: Underlining Text" on page 252

_BLANKPAGE_
advances the pointer to the first line of a new page, even when the pointer is positioned on the first line and the first column of a new page.
Tip
If the current output file contains carriage-control characters, 
\_BLANKPAGE\_ produces output lines that contain the appropriate 
carriage-control character.

Example
"Example 3: Moving the Pointer to a New Page" on page 250

_\_PAGE\_ advances the pointer to the first line of a new page. SAS automatically begins a
new page when a line exceeds the current PAGESIZE= value.

Tips
If the current output file is printed, _\_PAGE\_ produces an output line
that contains the appropriate carriage-control character. _\_PAGE\_ has
no effect on a file that is not printed.

If you specify FILE PRINT in an interactive SAS session, then the
Output window interprets the form-feed control characters as page
breaks, and they are removed from the output. The resulting file is a
flat file without page break characters. If a file needs to contain the
form-feed characters, then the FILE statement should include a
physical file location and the PRINT option.

Example
"Example 3: Moving the Pointer to a New Page" on page 250

Details
When to Use PUT

Use the PUT statement to write lines to the SAS log, to the SAS output window, or
to an external location. If you do not execute a FILE statement before the PUT
statement in the current iteration of a DATA step, SAS writes the lines to the SAS
log. If you specify the PRINT option in the FILE statement, SAS writes the lines to
the SAS output window.

The PUT statement can write lines that contain variable values, character strings,
and hexadecimal character constants. With specifications in the PUT statement, you
specify what to write, where to write it, and how to format it.

Output Styles
Overview of Output Styles

There are four ways to write variable values with the PUT statement:

- column
- list (simple and modified)
- formatted
- named

A single PUT statement might contain any or all of the available output styles,
depending on how you want to write lines.

Column Output

With column output, the column numbers follow the variable in the PUT statement.
These numbers indicate where in the line to write this value:
\put name 6-15 age 17-19;
The program writes these lines to the SAS log:

Note: The ruled line is for illustrative purposes only; the PUT statement does not generate it.

```
+--------+--------+
| Peterson 21 |
| Morgan 17   |
```

The PUT statement writes values for NAME and AGE in the specified columns. For more information, see “PUT Statement: Column” on page 256.

List Output

With list output, list the variables and character strings in the PUT statement in the order in which you want to write them. For example, this PUT statement writes the values for NAME and AGE to the SAS log.

```
put name age;
```

Here is the SAS log.

```
+--------+--------+
| Peterson 21 |
| Morgan 17   |
```

Note: The ruled line is for illustrative purposes only; the PUT statement does not generate it.

For more information, see “PUT Statement: List” on page 262.

Formatted Output

With formatted output, specify a SAS format or a user-written format after the variable name. The format gives instructions on how to write the variable value. Formats enable you to write in a nonstandard form, such as packed decimal, or numbers that contain special characters such as commas. You can also override the default alignment of the formatted output by using -L, -C, or -R.

For example, this PUT statement writes the values for NAME, AGE, and DATE to the SAS log.

```
put name $char10. age 2. +1 date mmddyy10.;
```

Here is the SAS log.

```
+--------+--------+------------------+
| Peterson 21 07/18/1999 |
| Morgan 17 11/12/1999   |
```

Note: The ruled line is for illustrative purposes only; the PUT statement does not generate it.

For more information, see “PUT Statement: Formatted” on page 258.

Using a pointer control of +1 inserts a blank space between the values of AGE and DATE.
Named Output

With named output, list the variable name followed by an equal sign. For example, this PUT statement writes the values for NAME and AGE to the SAS log.

```
put name= age=;
```

Here is the SAS log.

```
----+----1----+----2----+
name=Peterson age=21
name=Morgan age=17
```

Note: The ruled line is for illustrative purposes only; the PUT statement does not generate it.

For more information, see “PUT Statement: Named” on page 267.

Using Multiple Output Styles in a Single PUT Statement

A PUT statement can combine any or all of the different output styles.

```
put name 'on ' date mmdyy8. ' weighs ' startwght +(-1) '.' idno= 40-45;
```

See “Example 1: Using Multiple Output Styles in One PUT Statement” on page 248 for an explanation of the lines written to the SAS log.

When you combine different output styles, it is important to understand the location of the output pointer after each value is written. For more information about the pointer location, see “Pointer Location After a Value Is Written” on page 246.

Avoiding a Common Error When Writing Both a Character Constant and a Variable

When using a PUT statement to write a character constant that is followed by a variable name, always put a blank space between the closing quotation mark and the variable name.

```
put 'Player:' name1 'Player:' name2 'Player:' name3;
```

Otherwise, SAS might interpret a character constant that is followed by a variable name as a special SAS constant as illustrated in this table.

<table>
<thead>
<tr>
<th>Starting Letter of Variable</th>
<th>Represents</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>bit testing constant</td>
<td>'00100000'b</td>
</tr>
<tr>
<td>d</td>
<td>date constant</td>
<td>'01jan04'd</td>
</tr>
<tr>
<td>dt</td>
<td>datetime constant</td>
<td>'18jan2003:9:27:05am'dt</td>
</tr>
<tr>
<td>n</td>
<td>name literal</td>
<td>'My Table'n</td>
</tr>
</tbody>
</table>
Table 2.5  Pointer Controls Available in the PUT Statement

<table>
<thead>
<tr>
<th>Pointer Controls</th>
<th>Relative</th>
<th>Absolute</th>
</tr>
</thead>
<tbody>
<tr>
<td>column pointer controls</td>
<td>+n</td>
<td>@n</td>
</tr>
<tr>
<td></td>
<td>+numeric-variable</td>
<td>@numeric-variable</td>
</tr>
<tr>
<td></td>
<td>+(expression)</td>
<td>@(expression)</td>
</tr>
<tr>
<td>line pointer controls</td>
<td>/ , <em>PAGE</em> ,</td>
<td>#n</td>
</tr>
<tr>
<td></td>
<td><em>BLANKPAGE</em></td>
<td>#numeric-variable</td>
</tr>
<tr>
<td></td>
<td></td>
<td>#(expression)</td>
</tr>
<tr>
<td>line-hold specifiers</td>
<td>@</td>
<td>(not applicable)</td>
</tr>
</tbody>
</table>
### Pointer Controls

<table>
<thead>
<tr>
<th>Pointer Controls</th>
<th>Relative</th>
<th>Absolute</th>
</tr>
</thead>
<tbody>
<tr>
<td>@@</td>
<td>(not applicable)</td>
<td></td>
</tr>
</tbody>
</table>

---

**Note:** Always specify pointer controls before the variable for which they apply.

For more information about how SAS determines the pointer position, see “Pointer Location After a Value Is Written” on page 246.

### Using Line-Hold Specifiers

Line-hold specifiers keep the pointer on the current output line when one of these conditions occurs:

- more than one PUT statement writes to the same output line.
- a PUT statement writes values from more than one observation to the same output line.

Without line-hold specifiers, each PUT statement in a DATA step writes a new output line.

In the PUT statement, trailing `@` and double trailing `@@` produce the same effect. Unlike the INPUT statement, the PUT statement does not automatically release a line that is held by a trailing `@` when the DATA step begins a new iteration. SAS releases the current output line that is held by a trailing `@` or double trailing `@` when it encounters one of these situations:

- a PUT statement without a trailing `@`.
- a PUT statement that uses `_BLANKPAGE_` or `_PAGE_`.
- the end of the current line (determined by the current value of the LRECL= or LINESIZE= option in the FILE statement, if specified, or the LINESIZE= system option).
- the end of the last iteration of the DATA step.

Using a trailing `@` or double trailing `@` can cause SAS to attempt to write past the current line length because the pointer value is unchanged when the next PUT statement executes. See “When the Pointer Goes Past the End of a Line” on page 247.

### Pointer Location After a Value Is Written

It is important to understand the location of the output pointer after a value is written, especially if you combine output styles in a single PUT statement. The pointer location after a value is written depends on which output style you use and whether a character string or a variable is written. With column or formatted output, the pointer is located in the first column after the end of the field that is specified in the PUT statement. These two styles write only variable values.

With list output or named output, the pointer is located in the second column after a variable value because PUT skips a column automatically after each value is written. However, when a PUT statement uses list output to write a character string, the pointer is located in the first column after the string. If you do not use a line pointer control or column output after a character string is written, add a blank space to the end of the character string to separate it from the next value.
After an _INFILE_ specification, the pointer is located in the first column after the
record is written from the current input file.

When the output pointer is in the upper left corner of a page, you can use one of
these statements to control the pointer:

- PUT _BLANKPAGE_ writes a blank page and moves the pointer to the top of the
  next page.
- PUT _PAGE_ leaves the pointer in the same location.

You can determine the current location of the pointer by examining the variables that
are specified with the COLUMN= option and the LINE= option in the FILE
statement.

When the Pointer Goes Past the End of a Line

SAS does not write an output line that is longer than the current output line length.
The line length of the current output file is determined by the value of these options:

- the value of the LINESIZE= option in the current FILE statement.
- the value of the LINESIZE= system option (for the SAS output window).
- the LRECL= option in the current FILE statement (for external files).

You can inadvertently position the pointer beyond the current line length with one or
more of these specifications:

- a + pointer control with a value that moves the pointer to a column beyond the
  current line length.
- a column range that exceeds the current line length (for example, PUT X 90 –
  100 when the current line length is 80).
- a variable value or character string that does not fit in the space that remains on
  the current output line.

By default, when PUT attempts to write past the end of the current line, SAS
withholds the entire item that overflows the current line, writes the current line, and
then writes the overflow item on a new line, starting in column 1. See the
FLOWOVER, DROPOVER, and STOPOVER options in the “FILE Statement” on
page 75.

Arrays

You can use the PUT statement to write an array element. The subscript is any SAS
expression that results in an integer when the PUT statement executes. You can
use an array reference in a numeric-variable construction with a pointer control if
you enclose the reference in parentheses, as shown here:

- @array-name(i)
- +(array-name[i])
- #(array-name[i])

Use the array subscript asterisk (*) to write all elements of a previously defined array
to an external location. SAS allows one-dimensional or multidimensional arrays, but
it does not allow a _TEMPORARY_ array. Enclose the subscript in braces, brackets,
or parentheses, and print the array using list, formatted, column, or named output.
With list output, the form of this statement is

PUT array-name{*};

With formatted output, the form of this statement is

PUT array-name{*}(format | format.list)
The format in parentheses follows the array reference.

Comparisons

- The PUT statement writes variable values and character strings to the SAS log or to an external location while the INPUT statement reads raw data in external files or data lines entered instream.

- Both the INPUT statement and the PUT statement use the trailing @ and double trailing @ line-hold specifiers to hold the current line in the input or output buffer, respectively. In an INPUT statement, a double trailing @ holds a line in the input buffer from one iteration of the DATA step to the next. In a PUT statement, however, a trailing @ has the same effect as a double trailing @; both hold a line across iterations of the DATA step.

- Both the PUT and OUTPUT statements create output in a DATA step. The PUT statement uses an output buffer and writes output lines to an external location, the SAS log, or your monitor. The OUTPUT statement uses the program data vector and writes observations to a SAS data set.

Examples:

Example 1: Using Multiple Output Styles in One PUT Statement

This example uses several output styles in a single PUT statement.

```sas
data club1;
  input idno name $ startwght date : date7. ;
  put name 'on ' date mmdyy8. ' weighs ' startwght +(-1) '.' idno= 32-40;
  datalines;
  032 David 180 25nov99
  049 Amelia 145 25nov99
  219 Alan 210 12nov99

This table shows the output style used for each variable in the example:

<table>
<thead>
<tr>
<th>Variables</th>
<th>Output Style</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAME, STARTWGHGT</td>
<td>list output</td>
</tr>
<tr>
<td>DATE</td>
<td>formatted output</td>
</tr>
<tr>
<td>IDNO</td>
<td>named output</td>
</tr>
</tbody>
</table>
```

The PUT statement also uses pointer controls and specifies both character strings and variable names. The program writes these lines to the SAS log.

```
248 Chapter 2 / Dictionary of SAS DATA Step Statements

--------1--------2--------3--------4
David on 11/25/99 weighs 180. idno=1032
Amelia on 11/25/99 weighs 145. idno=1049
Alan on 11/12/99 weighs 210. idno=1219
```
Note: The ruled line is for illustrative purposes only; the PUT statement does not generate it.

Blank spaces are inserted at the beginning and the end of the character strings to change the pointer position. These spaces separate the value of a variable from the character string. The (+-1) pointer control moves the pointer backward to remove the unwanted blank that occurs between the value of STARTWGHT and the period. For more information about how to position the pointer, see “Pointer Location After a Value Is Written” on page 246.

Example 2: Moving the Pointer within a Page

These PUT statements show how to use column and line pointer controls to position the output pointer.

To move the pointer to a specific column, use @ followed by the column number, variable, or expression whose value is that column number. For example, this statement moves the pointer to column 15 and writes the value of TOTAL SALES using list output.

```
put @15 totalsales;
```

This PUT statement moves the pointer to the value that is specified in COLUMN and writes the value of TOTALSALES with the COMMA6 format.

```
data _null_
   set carsales;
   column=15;
   put @column totalsales comma6.;
run;
```

This program shows two techniques to move the pointer backward.

```
data carsales;
   input item $10. jan : comma5.
      feb : comma5. mar : comma5.;
   saleqtr1=sum(jan,feb,mar);
   /* an expression moves pointer backward */
   put '1st qtr sales for ' item
     'is ' saleqtr1 : comma6. +(-1) ' .';
   /* a numeric variable with a negative value moves pointer backward. */
   x=-1;
   put '1st qtr sales for ' item
     'is ' saleqtr1 : comma5. +x ' .';
datalines;
trucks         1,382     2,789     3,556
vans           1,265     2,543     3,987
sedans         2,391     3,011     3,658
;
```

Because the value of SALEQTR1 is written with modified list output, the pointer moves automatically two spaces. For more information, see “How Modified List Output and Formatted Output Differ” on page 265. To remove the unwanted blank that occurs between the value and the period, move the pointer backward by one space.

The program writes these lines to the SAS log.

```
------1--------2--------3--------4
```
st qtr sales for trucks is 7,727.
st qtr sales for trucks is 7,727.
st qtr sales for vans is 7,795.
st qtr sales for vans is 7,795.
st qtr sales for sedans is 9,060.
st qtr sales for sedans is 9,060.

Note: The ruled line is for illustrative purposes only; the PUT statement does not generate it.

This program uses a PUT statement with the / line pointer control to advance to the next output line:

```sas
data _null_;  
  set carsales end=lastrec;  
  totalsales+saleqtr1;  
  if lastrec then  
    put @2 'Total Sales for 1st Qtr' / totalsales 10-15;  
run;
```

After the DATA step calculates TOTALSALES using all the observations in the CARSALES data set, the PUT statement executes. It writes a character string beginning in column 2 and moves to the next line to write the value of TOTALSALES in columns 10 through 15.

```
--------1--------2--------3
Total Sales for 1st Qtr
  24582
```

Note: The ruled line is for illustrative purposes only; the PUT statement does not generate it.

Example 3: Moving the Pointer to a New Page

This example creates a data set called STATEPOP, which contains information from the 1990 U.S. census about the population of metropolitan and non-metropolitan areas. It executes the FORMAT procedure to group the 50 states and the District of Columbia into four regions. It then uses the IF and PUT statements to control the printed output.

```sas
title1;

data statepop;
  input state $ cityp90 ncityp90 region @@;
  label cityp90='1990 metropolitan population (million)'
    ncityp90='1990 nonmetropolitan population (million)'
    region='Geographic region';
  datalines;
  ME   .443   .785   1   NH   .659   .450   1
  VT   .152   .411   1   MA   5.788   .229   1
  RI   .938   .065   1   CT   3.148   .140   2
  NY  16.515  1.475   1   NJ   7.730   .A   1
  PA  10.083  1.799   1   DE   .553   .113   2
  MD   4.439   .343   2   DC   .607   .   2
```

250 Chapter 2 / Dictionary of SAS DATA Step Statements
Example 3: Moving the Pointer to a New Page

proc format;
  value regfmt 1='Northeast'
         2='South'
         3='Midwest'
         4='West';
run;

data _null_;  
set statepop;  
by region;  
pop90=sum(cityp90,ncityp90);  
file print;  
put state 1-2 @5 pop90 7.3 ' million';  
if first.region then  
  regioncitypop=0;  /* new region */  
regioncitypop+cityp90;  
if last.region then  
  do;  
    put // '1990 US CENSUS for ' region regfmt.  
      /'Total Urban Population: '  
        regioncitypop ' million' _page_;  
  end;  
run;
PUT _PAGE_ advances the pointer to line 1 of the new page when the value of LAST.REGION is 1. The example prints a footer message before exiting the page.

Example 4: Underlining Text

This example uses OVERPRINT to underscore a value written by a previous PUT statement.

```sas
data _null_;  
  input idno name $ startwght;  
  file file-specification print;  
  put name 1-10 @15 startwght 3.;  
  if startwght > 200 then  
    put overprint @15 '___';  
  datalines;  
  032 David 180  
  049 Amelia 145  
  219 Alan 210  
;  
```

The second PUT statement underlines weights above 200 on the output line the first PUT statement prints.

This PUT statement uses OVERPRINT with both a column pointer control and a line pointer control.

```sas
  put @5 name $8. overprint @5 8*'_'  
    / @20 address;  
```

The PUT statement writes a NAME value, underlines it by overprinting eight underscores, and moves the output pointer to the next line to write an ADDRESS value.

Example 5: Holding and Releasing Output Lines

This DATA step demonstrates how to hold and release an output line with a PUT statement.

```sas
data _null_;  
  input idno name $ startwght 3.;  
  put name @;  
  if startwght ne . then  
    put @15 startwght;  
  else put;  
  datalines;  
  032 David 180  
;  
```
Example 6: Writing the Current Input Record to the SAS Log

When a value for ID is less than 1000, PUT _INFILE_ executes and writes the current input record to the SAS log. The DELETE statement prevents the DATA step from writing the observation to the TEAM data set.

```sas
data team;
  input id team $ score1 score2;
  if id le 1000 then
    do;
      put _infile_;
      delete;
    end;
  datalines;
  032 red 180 165
  049 yellow 145 124
  219 red 210 192
;
```

The program writes this line to the SAS log.

```
--------1--------2
219 red 210 192
```

Note: The ruled line is for illustrative purposes only; the PUT statement does not generate it.
Example 7: Avoiding a Common Error When Writing a Character Constant Followed by a Variable

This example illustrates how to use a PUT statement to write character constants and variable values without causing them to be misinterpreted as SAS name literals. A SAS name literal is a name token that is expressed as a string within quotation marks, followed by the letter n. For more information about SAS name literals, see SAS Language Reference: Concepts.

In the following program, the PUT statement writes the constant 'n' followed by the value of the variable NVAR1, and then writes another constant 'n':

```sas
data _null_
  n=5;
  nvar1=1;
  var1=7;
  put @1 'n' nvar1 'n';
run;
```

The program writes this line to the SAS log.

```
----+----1----+----2
n1 n
```

**Note:** The ruled line is for illustrative purposes only; the PUT statement does not generate it.

If all the spaces between the constants and the variables are removed from the previous PUT statement, SAS interprets 'n' as a name literal instead of reading 'n' as a constant. The next variable is read as VAR1 instead of NVAR1. The final 'n' constant is interpreted correctly.

```sas
put @1 'n'nvar1'n';
```

The PUT statement writes this line to the SAS log.

```
----+----1----+----2
5 7 n
```

To print character constants and variable values without intervening spaces, and without potential misinterpretation, you can add spaces between them and use pointer controls where necessary. For example, this PUT statement uses a pointer control to write the correct character constants and variable values but does not insert blank spaces. Note that +(-1) moves the PUT statement pointer backward by one space.

```sas
put @1 'n' nvar1 +(1) 'n';
```

The PUT statement writes this line to the SAS log.

```
----+----1----+----2
n1n
```
Example 8: Creating Multi-Column Output

This example uses the #n and @n column and pointer controls to create multi-column output.

```sas
/*
 * Use #i and @j to position name and weight information into
 * four columns in column-major order. That is print down column 1
 * first, then print down column 2, etc.
 * This example highlights the need to specify # before @ because
 * # sets the column pointer to 1.
 */
data _null_;   
file print n=ps notitles header=hd;
  do i = 1 to 80 by 20;
    do j = 1 to ceil(num_students/4);
      set sashelp.class nobs=num_students;
      put #(j+3) @i name $8. '-' +1 weight 5.1;
    end;
  end;
  stop;
hd:
  put @26 'Student Weight in Pounds' / @26 24**'-';
  return;
run;
```

The program creates this output.

<table>
<thead>
<tr>
<th>Student Weight in Pounds</th>
</tr>
</thead>
<tbody>
<tr>
<td>-----------------------------------------</td>
</tr>
<tr>
<td>Alfred  - 112.5  James - 83.0  Joyce - 50.5  Robert - 128.0</td>
</tr>
<tr>
<td>Alice - 84.0  Jane - 84.5  Judy - 90.0  Ronald - 133.0</td>
</tr>
<tr>
<td>Barbara - 98.0  Janet - 112.5  Louise - 77.0  Thomas - 85.0</td>
</tr>
<tr>
<td>Carol - 102.5  Jeffrey - 84.0  Mary - 112.0  William - 112.0</td>
</tr>
<tr>
<td>Henry - 102.5  John - 99.5  Philip - 150.0</td>
</tr>
</tbody>
</table>

See Also

Statements:
- "FILE Statement" on page 75
- "PUT Statement: Column" on page 256
- "PUT Statement: Formatted" on page 258
- "PUT Statement: List" on page 262
- "PUT Statement: Named" on page 267

System Options:
- "LINESIZE= System Option" in SAS System Options: Reference
PUT Statement: Column

Writes variable values in the specified columns in the output line.

Valid in: DATA step
Categories: CAS
File-Handling
Type: Executable

Syntax

```
PUT variable start-column <- end-column>
<decimal-places> <@ | @@>
```

Arguments

- **variable** specifies the variable whose value is written.
- **start-column** specifies the first column of the field where the value is written in the output line.
- **end-column** specifies the last column of the field for the value.

Tip If the value occupies only one column in the output line, omit `end-column`.

Example Because `end-column` is omitted, the values for the character variable GENDER occupy only column 16:

```
put name 1-10 gender 16;
```

- **decimal-places** specifies the number of digits to the right of the decimal point in a numeric value.

Range positive integer

Tip If you specify 0 for `d` or omit `d`, the value is written without a decimal point.

Example "Example: Using Column Output in the PUT Statement" on page 257

- **@ | @@** holds an output line for the execution of the next PUT statement even across iterations of the DATA step. These line-hold specifiers are called `trailing @` and `double trailing @`.

Requirement The trailing @ or double trailing @ must be the last item in the PUT statement.

See "Using Line-Hold Specifiers" on page 246
Details

With column output, the column numbers indicate the position that each variable value occupies in the output line. If a value requires fewer columns than specified, a character variable is left-aligned in the specified columns, and a numeric variable is right-aligned in the specified columns.

There is no limit to the number of column specifications that you can make in a single PUT statement. You can write anywhere in the output line, even if a value overwrites columns that were written earlier in the same statement. You can combine column output with any of the other output styles in a single PUT statement. For more information, see "Using Multiple Output Styles in a Single PUT Statement" on page 244.

Example: Using Column Output in the PUT Statement

Use column output in the PUT statement as shown here.

- This PUT statement uses column output.
  ```sas
  data _null_;
  input name $ 1-18 score1 score2 score3;
  put name 1-20 score1 23-25 score2 28-30 score3 33-35;
  datalines;
  Joseph                  11   32   76
  Mitchel                 13   29   82
  Sue Ellen               14   27   74
  ;
  ```

  The program writes these lines to the SAS log.

  ----+----1----+----2----+----3----+----4
  Joseph                 11   32   76
  Mitchel                13   29   82
  Sue Ellen              14   27   74

  Note: The ruled line is for illustrative purposes only; the PUT statement does not generate it.

  The values for the character variable NAME begin in column 1, the left boundary of the specified field (columns 1 through 20). The values for the numeric variables SCORE1 through SCORE3 appear flush with the right boundary of their field.

- This statement produces the same output lines, but writes the SCORE1 value first and the NAME value last.
  ```sas
  put score1 23-25 score2 28-30 score3 33-35 name $ 1-20;
  ```

- This DATA step specifies decimal points with column output.
  ```sas
  data _null_;
  x=11;
  y=15;
  put x 10-18 .1 y 20-28 .1;
  run;
  ```

  The program writes this line to the SAS log.
PUT Statement: Formatted

Writes variable values with the specified format in the output line.

Valid in: DATA step

Categories: CAS
File-Handling

Type: Executable

Syntax

PUT <pointer-control> variable format. @ | @@;
PUT <pointer-control> (variable-list) (format-list) @ | @@;

Arguments

pointer-control
moves the output pointer to a specified line or column.

variable
specifies the variable whose value is written.

(variable-list)
specifies a list of variables whose values are written.

format.
specifies a format to use when the variable values are written. To override the default alignment, you can add an alignment specification to a format:

See Also

Statements:

- “PUT Statement” on page 236

Example

“Example 1: Writing a Character between Formatted Values” on page 261
Tip
Ensure that the format width provides enough space to write the value and any commas, dollar signs, decimal points, or other special characters that the format includes.

Examples
This PUT statement uses the format dollar7.2 to write the value of X:
put x dollar7.2;

When X is 100, the formatted value uses seven columns:
$100.00

Example
"Example 2: Overriding the Default Alignment of Formatted Values" on page 261

(format-list)
specifies a list of formats to use when the values of the preceding list of variables are written. In a PUT statement, a format-list can include

format.
specifies the format to use to write the variable values.

Tip
You can specify either a SAS format or a user-written format.

See
SAS Formats and Informats: Reference

pointer-control
specifies one of these pointer controls to use to position a value: @, #, /, +, and OVERPRINT.

Example
"Example 1: Writing a Character between Formatted Values" on page 261

character-string
specifies one or more characters to place between formatted values.

Example
This statement places a hyphen between the formatted values of CODE1, CODE2, and CODE3:
put bldg $ (code1 code2 code3) (3. '-');

Example
"Example 1: Writing a Character between Formatted Values" on page 261

n*
specifies to repeat n times the next format in a format list.

Restriction
The (format-list) must follow (variable-list).

See
"How to Group Variables and Formats" on page 260

Example
This statement uses the 7.2 format to write GRADES1, GRADES2, and GRADES3 and the 5.2 format to write GRADES4 and GRADES5:
put (grades1-grades5) (3*7.2, 2*5.2);
@ | @@
holds an output line for the execution of the next PUT statement even across iterations of the DATA step. These line-hold specifiers are called trailing @ and double trailing @.

Restriction The trailing @ or double trailing @ must be the last item in the PUT statement.

See “Using Line-Hold Specifiers” on page 246

Details

Using Formatted Output
The Formatted output describes the output lines by listing the variable names and the formats to use to write the values. You can use a SAS format or a user-written format to control how SAS prints the variable values. For a complete description of the SAS formats, see “Definition of Formats” in SAS Formats and Informats: Reference.

With formatted output, the PUT statement uses the format that follows the variable name to write each value. SAS does not automatically add blanks between values. If the value uses fewer columns than specified, character values are left-aligned and numeric values are right-aligned in the field that is specified by the format width.

Formatted output, combined with pointer controls, makes it possible to specify the exact line and column location to write each variable. For example, this PUT statement uses the dollar7.2 format and centers the value of X starting at column 12:

put @12 x dollar7.2-c;

How to Group Variables and Formats
When you want to write values in a pattern on the output lines, use format lists to shorten your coding time. A format list consists of the corresponding formats separated by either blanks or commas and enclosed in parentheses. It must follow the names of the variables enclosed in parentheses.

For example, this statement uses a format list to write the five variables SCORE1 through SCORE5, one after another, using four columns for each value with no blanks in between:

put (score1-score5) (4. 4. 4. 4. 4.);

A shorter version of the previous statement is

put (score1-score5) (4.);

You can include any of the pointer controls (@, #, /, +, and OVERPRINT) in the list of formats, as well as n*, and a character string. You can use as many format lists as necessary in a PUT statement, but do not nest the format lists. After all the values in the variable list are written, the PUT statement ignores any directions that remain in the format list. For an example, see “Example 3: Including More Format Specifications Than Necessary” on page 261.

You can also specify a reference to all elements in an array as (array-name *), followed by a list of formats. However, you cannot specify the elements in a _TEMPORARY_ array in this way. This PUT statement specifies an array name and a format list:
put (array1{*}) (4.);

For more information about how to reference an array, see “Arrays” on page 247.

Examples:

Example 1: Writing a Character between Formatted Values
This example formats some values and writes a - (hyphen) between the values of variables BLDG and ROOM:

data _null_;  
  input name & $15. bldg $ room;  
  put name @20 (bldg room) ($1. "-" 3.);  
  datalines;  
Bill Perkins  J 126  
Sydney Riley  C 219  
;

The program writes these lines to the SAS log:

Bill Perkins       J-126  
Sydney Riley       C-219

Example 2: Overriding the Default Alignment of Formatted Values
This example includes an alignment specification in the format:

data _null_;  
  input name $ 1-12 score1 score2 score3;  
  put name $12.-r +3 score1 3. score2 3. score3 4.;  
  datalines;  
Joseph           11   32   76  
Mitchel          13   29   82  
Sue Ellen        14   27   74  
;

The program writes these lines to the SAS log:  

---+----1----+----2----+----3----+----4
Joseph    11 32  76  
Mitchel    13 29  82  
Sue Ellen    14 27  74

The value of the character variable NAME is right-aligned in the formatted field. (Left alignment is the default for character variables.)

Example 3: Including More Format Specifications Than Necessary
This format list includes more specifications than are necessary when the PUT statement executes:

data _null_;  
  input x y z;  
  put (x y z) (2.,+1);  
  datalines;

1. The ruled line is for illustrative purposes only; the PUT statement does not generate it.
The PUT statement writes the value of X using the 2. format. Then, the +1 column pointer control moves the pointer forward one column. Next, the value of Y is written with the 2. format. Again, the +1 column pointer moves the pointer forward one column. Then, the value of Z is written with the 2. format. For the third iteration, the PUT statement ignores the +1 pointer control.

The program writes these lines to the SAS log:

```
2 24 36
0 20 30
```

See Also

**Statements:**

- "PUT Statement" on page 236

## PUT Statement: List

Writes variable values and the specified character strings in the output line.

### Syntax

```
PUT <pointer-control> variable <@ | @@>;
PUT <pointer-control> <n*> 'character-string' <@ | @@>;
PUT <pointer-control> variable < : | ->> format.<@ | @@>;
```

### Arguments

- **pointer-control**
  moves the output pointer to a specified line or column.

### See

- "Column Pointer Controls " on page 239
- "Line Pointer Controls " on page 240

### Example

- "Example 2: Writing Character Strings and Variable Values" on page 266

---

1. The ruled line is for illustrative purposes only; the PUT statement does not generate it.
variable
specifies the variable whose value is written.

Example  “Example 1: Writing Values with List Output” on page 265

n*
specifies to repeat n times the subsequent character string.

Example  This statement writes a line of 132 underscores:
put 132*'_';

'character-string'
specifies a string of text, enclosed in quotation marks, to write.

Interaction  When insufficient space remains on the current line to write the entire text string, SAS withholding the entire string and writes the current line. Then it writes the text string on a new line, starting in column 1. For more information, see “When the Pointer Goes Past the End of a Line” on page 247.

Tips  To avoid misinterpretation, always put a space after a closing quotation mark in a PUT statement.

If you follow a quotation mark with X, SAS interprets the text string as a hexadecimal constant.

If you use single quotation (’) or double quotation marks (") together (with no space in between them) as the string of text, SAS writes a single quotation mark (’) or double quotation mark ("), respectively.

See  “How List Output Is Spaced” on page 264

Example  “Example 2: Writing Character Strings and Variable Values” on page 266

:  enables you to specify a format that the PUT statement uses to write the variable value. All leading and trailing blanks are deleted, and each value is followed by a single blank.

Requirement  You must specify a format.

See  “How Modified List Output and Formatted Output Differ” on page 265

Example  “Example 3: Writing Values with Modified List Output (:)” on page 266

~  enables you to specify a format that the PUT statement uses to write the variable value. SAS displays the formatted value in quotation marks even if the formatted value does not contain the delimiter. SAS deletes all leading and trailing blanks, and each value is followed by a single blank. Missing values for character variables are written as a blank (" ") and, by default, missing values for numeric variables are written as a period (".").

Requirement  You must specify the DSD option in the FILE statement.
Example 4: Writing Values with Modified List Output and ~

format.
specifies a format to use when the data values are written.

Tip
You can specify either a SAS format or a user-written format. See SAS Formats and Informats: Reference

Example 3: Writing Values with Modified List Output (~)

@ | @@
holds an output line for the execution of the next PUT statement even across iterations of the DATA step. These line-hold specifiers are called trailing @ and double trailing @.

Restriction
The trailing @ or double-trailing @ must be the last item in the PUT statement.

See
“Using Line-Hold Specifiers” on page 246

Details
Using List Output

With list output, you list the names of the variables whose values you want written, or you specify a character string in quotation marks. The PUT statement writes a variable value, inserts a single blank, and then writes the next value. Missing values for numeric variables are written as a single period. Character values are left-aligned in the field; leading and trailing blanks are removed. To include blanks (in addition to the blank inserted after each value), use formatted or column output instead of list output.

There are two types of list output:

- simple list output
- modified list output.

Modified list output increases the versatility of the PUT statement because you can specify a format to control how the variable values are written. See “Example 3: Writing Values with Modified List Output (~)” on page 266.

How List Output Is Spaced

List output uses different spacing methods when it writes variable values and character strings. When a variable is written with list output, SAS automatically inserts a blank space. The output pointer stops at the second column that follows the variable value. However, when a character string is written, SAS does not automatically insert a blank space. The output pointer stops at the column that immediately follows the last character in the string.

To avoid spacing problems when both character strings and variable values are written, you might want to use a blank space as the last character in a character string. When a character string that provides punctuation follows a variable value, you need to move the output pointer backward. Moving the output pointer backward prevents an unwanted space from appearing in the output line. See “Example 2: Writing Character Strings and Variable Values” on page 266.
How Modified List Output and Formatted Output Differ

List output and formatted output use different methods to determine how far to move the pointer after a variable value is written. Therefore, modified list output, which uses formats, and formatted output produce different results in the output lines. Modified list output writes the value, inserts a blank space, and moves the pointer to the next column. Formatted output moves the pointer the length of the format, even if the value does not fill that length. The pointer moves to the next column; an intervening blank is not inserted.

This DATA step uses modified list output to write each output line:

data _null_;  
  input x y;  
  put x : comma10.2 y : 7.2;  
datalines;  
2353.20 7.10  
6231 121  
;

The program writes these lines to the SAS log:

----+----1----+----2----+----3----+----4
2,353.20 7.10  
6,231.00 121.00

In comparison, this example uses formatted output:

put x comma10.2 y 7.2;

The program writes these lines to the SAS log with the values aligned in columns:

----+----1----+----2
2,353.20   7.10  
6,231.00 121.00

Examples:

Example 1: Writing Values with List Output

This DATA step uses a PUT statement with list output to write variable values to the SAS log:

data _null_;  
  input name $ 1-10 sex $ 12 age 15-16;  
  put name sex age;  
datalines;  
Joseph M 13  
Mitchel M 14  
Sue Ellen F 11  
;

The program writes these lines to the SAS log:

----+----1----+----2----+----3----+----4----+----5----+----6
Joseph M 13  
Mitchel M 14  
Sue Ellen F 11

By default, the values of the character variable NAME are left-aligned in the field.
Example 2: Writing Character Strings and Variable Values

This PUT statement adds a space to the end of a character string and moves the output pointer backward to prevent an unwanted space from appearing in the output line after the variable STARTWGHT:

```sas
data _null_;  
  input idno name $ startwght;  
  put name ' weighs ' startwght +-(-1) '.';  
  datalines;  
032 David 180  
049 Amelia 145  
219 Alan 210  
  ;  

The program writes these lines to the SAS log:

David weighs 180.  
Amelia weighs 145.  
Alan weighs 210.
```

The blank space at the end of the character string changes the pointer position. This space separates the character string from the value of the variable that follows. The +-(-1) pointer control moves the pointer backward to remove the unwanted blank that occurs between the value of STARTWGHT and the period.

Example 3: Writing Values with Modified List Output (:)

This DATA step uses modified list output to write several variable values in the output line using the : argument:

```sas
data _null_;  
  input salesrep : $10. tot : comma6. date : date9.;  
  put ' Week of ' date : worddate15. salesrep : $12. ' sales were ' tot : dollar9. + (-1) '.';  
  datalines;  
Wong 15,300 12OCT2004  
Hoffman 9,600 12OCT2004  
  ;  

The program writes these lines to the SAS log:

Week of Oct 12, 2004 Wong sales were $15,300.  
Week of Oct 12, 2004 Hoffman sales were $9,600.
```

Example 4: Writing Values with Modified List Output and ~

This DATA step uses modified list output to write several variable values in the output line using the ~ argument:

```sas
data _null_;  
  input salesrep : $10. tot : comma6. date : date9.;  
  file log delimiter=" " dsd;  
  put ' Week of ' date ~ worddate15. salesrep ~ $12. ' sales were ' tot ~ dollar9. + (-1) '.';  
  datalines;  
Wong 15,300 12OCT2004  
  ;  

Wong 15,300 12OCT2004
```
The program writes these lines to the SAS log:

Week of "Oct 12, 2004" "Wong" sales were "$15,300".
Week of "Oct 12, 2004" "Hoffman" sales were "$9,600".

See Also

Statements:

- "PUT Statement" on page 236
- "PUT Statement: Formatted" on page 258

PUT Statement: Named

Writes variable values after the variable name and an equal sign.

Valid in: DATA step
Categories: CAS
            File-Handling
Type: Executable

Syntax

```
PUT <pointer-control> variable= <format> [@ | @@];
PUT variable= start-column <end-column> <decimal-places> [@ | @@];
```

Arguments

**pointer-control**

moves the output pointer to a specified line or column in the output buffer.

See "Column Pointer Controls " on page 239
            "Line Pointer Controls " on page 240

**variable**

specifies the variable whose value is written by the PUT statement in the form

```
variable=value
```

**format.**

specifies a format to use when the variable values are written.

Tip

Ensure that the format width provides enough space to write the value and any commas, dollar signs, decimal points, or other special characters that the format includes.

See "Formatting Named Output" on page 268
Examples

This PUT statement uses the format DOLLAR7.2 to write the value of X:

```
put x= dollar7.2;
```

When X=100, the formatted value uses seven columns:
```
X=$100.00
```

**start-column**

specifies the first column of the field where the variable name, equal sign, and value are to be written in the output line.

**end-column**

determines the last column of the field for the value.

**Tip**

If the variable name, equal sign, and value require more space than the columns specified, PUT writes past the end column rather than truncate the value. You must leave enough space before beginning the next value.

**.decimal-places**

specifies the number of digits to the right of the decimal point in a numeric value. If you specify 0 for $d$ or omit $d$, the value is written without a decimal point.

<table>
<thead>
<tr>
<th><strong>Range</strong></th>
<th>positive integer</th>
</tr>
</thead>
</table>

**@ | @@**

holds an output line for the execution of the next PUT statement even across iterations of the DATA step. These line-hold specifiers are called **trailing @** and **double trailing @**.

**Restriction**

The trailing @ or double trailing @ must be the last item in the PUT statement.

**See**

“Using Line-Hold Specifiers” on page 246

---

**Details**

**Using Named Output**

With named output, follow the variable name with an equal sign in the PUT statement. You can use either list output, column output, or formatted output specifications to indicate how to position the variable name and values. To insert a blank space between each variable value automatically, use list output. To align the output in columns, use pointer controls or column specifications.

**Formatting Named Output**

You can specify either a SAS format or a user-written format to control how SAS prints the variable values. The width of the format does not include the columns required by the variable name and equal sign. To align a formatted value, SAS deletes leading blanks and writes the variable value immediately after the equal sign. SAS does not align on the right side of the formatted length, as in unnamed formatted output.

For a complete description of the SAS formats, see “Definition of Formats” in SAS Formats and Informats: Reference.
Example: Using Named Output in the PUT Statement

Use named output in the PUT statement as shown here.

- This PUT combines named output with column pointer controls to align the output:

```sas
data _null_
   input name $ 1-18 score1 score2 score3;
   put name = @20 score1= score3= ;
datalines;
Joseph                  11   32   76
Mitchel                 13   29   82
Sue Ellen               14   27   74
;
```

The program writes these lines to the SAS log:

```
----+----1----+----2----+----3----+----4
NAME=Joseph        SCORE1=11 SCORE3=76
NAME=Mitchel       SCORE1=13 SCORE3=82
NAME=Sue Ellen     SCORE1=14 SCORE3=74
```

- This example specifies an output format for the variable AMOUNT:

```sas
put item= @25 amount= dollar12.2;
```

When the value of ITEM is binders and the value of AMOUNT is 153.25, this output line is produced:

```
----+----1----+----2----+----3----+----4
ITEM=binders            AMOUNT=$153.25
```

See Also

**Statements:**

- "PUT Statement" on page 236

**PUTLOG Statement**

Writes a message to the SAS log.

**Valid in:** DATA step

**Categories:** Action  
CAS

**Type:** Executable

**Syntax**

`PUTLOG 'message';`
Arguments

**message**
specifies the message that you want to write to the SAS log. *Message* can include character literals (enclosed in quotation marks), variable names, formats, and pointer controls.

**Tip**
You can precede your message text with WARNING, MESSAGE, or NOTE to better identify the output in the SAS log.

Details

The PUTLOG statement writes a message that you specify to the SAS log. The PUTLOG statement is also helpful when you use macro-generated code because you can send output to the SAS log without affecting the current file destination.

Comparisons

The PUTLOG statement is similar to the ERROR statement, except that PUTLOG does not set _ERROR_ to 1.

Example: Writing Messages to the SAS Log Using the PUTLOG Statement

This program creates the computeAverage92 macro, which computes the average score, validates input data, and uses the PUTLOG statement to write error messages to the SAS log. The DATA step uses the PUTLOG statement to write a warning message to the SAS log.

```sas
data ExamScores;
  input Name $ 1-16 Score1 Score2 Score3;
datalines;
Sullivan, James 86 92 88
Martinez, Maria 95 91 92
Guzik, Eugene 99 98 .
Schultz, John 90 87 93
van Dyke, Sylvia 98 . 91
Tan, Carol 93 85 85
;
filename outfile 'path-to-your-output-file';
/* Create a macro that computes the average score, validates  */
/* input data, and uses PUTLOG to write error messages to the */
/* SAS log.                                                   */
%macro computeAverage92(s1, s2, s3, avg);
  if &s1 < 0 or &s2 < 0 or &s3 < 0 then
    do;
      putlog 'ERROR: Invalid score data ' &s1= &s2= &s3=;
      &avg = .;
    end;
  else
    &avg = mean(&s1, &s2, &s3);
%mend;
data _null_; 
set ExamScores;
```
file outfile;
%computeAverage92(Score1, Score2, Score3, AverageScore);
put name Score1 Score2 Score3 AverageScore;
    /* Use PUTLOG to write a warning message to the SAS log. */
if AverageScore < 92 then
    putlog 'WARNING: Score below the minimum ' name=
    AverageScore= 5.2;
run;

proc print;
run;

The program writes these lines to the SAS log:

WARNING: Score below the minimum Name=Sullivan, James AverageScore=88.67
ERROR: Invalid score data Score1=99 Score2=98 Score3=.
WARNING: Score below the minimum Name=Guzik, Eugene AverageScore=.
WARNING: Score below the minimum Name=Schultz, John AverageScore=90.00
ERROR: Invalid score data Score1=98 Score2=. Score3=91
WARNING: Score below the minimum Name=van Dyke, Sylvia AverageScore=.
WARNING: Score below the minimum Name=Tan, Carol AverageScore=87.67

SAS creates this output file.

Output 2.29  Individual Examination Scores

<table>
<thead>
<tr>
<th>Obs</th>
<th>Name</th>
<th>Score1</th>
<th>Score2</th>
<th>Score3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Sullivan, James</td>
<td>86</td>
<td>92</td>
<td>88</td>
</tr>
<tr>
<td>2</td>
<td>Martinez, Maria</td>
<td>95</td>
<td>91</td>
<td>92</td>
</tr>
<tr>
<td>3</td>
<td>Guzik, Eugene</td>
<td>99</td>
<td>98</td>
<td>.</td>
</tr>
<tr>
<td>4</td>
<td>Schultz, John</td>
<td>90</td>
<td>87</td>
<td>93</td>
</tr>
<tr>
<td>5</td>
<td>van Dyke, Sylvia</td>
<td>98</td>
<td>.</td>
<td>91</td>
</tr>
<tr>
<td>6</td>
<td>Tan, Carol</td>
<td>93</td>
<td>85</td>
<td>85</td>
</tr>
</tbody>
</table>

See Also

Statements:
- “ERROR Statement” on page 73

REDIRECT Statement

Points to different input or output SAS data sets when you execute a stored program.

Valid in: DATA step
Category: Action
Type: Executable
Restriction: This statement is not supported in a DATA step that runs in CAS.
Requirement: You must specify the PGM= option in the DATA statement.

Syntax
REDIRECT INPUT | OUTPUT old-name-1=new-name-1 <...old-name-n=new-name-n>;

Arguments
INPUT | OUTPUT
specifies whether to redirect input or output data sets. When you specify INPUT, the REDIRECT statement associates the name of the input data set in the source program with the name of another SAS data set. When you specify OUTPUT, the REDIRECT statement associates the name of the output data set with the name of another SAS data set.

old-name
specifies the name of the input or output data set in the source program.

new-name
specifies the name of the input or output data set that you want SAS to process for the current execution.

Details
The REDIRECT statement is available only when you execute a stored program. For more information about stored programs, see “Stored Compiled DATA Step Programs” in SAS Language Reference: Concepts.

CAUTION
Use care when you redirect input data sets. The number and attributes of variables in the input data sets that you read with the REDIRECT statement should match the number and attributes of variables in the input data sets in the MERGE, SET, MODIFY, or UPDATE statements of the source code. If the variable type attributes differ, the stored program stops processing and an appropriate error message is written to the SAS log. If the variable length attributes differ, the length of the variable in the source code data set determines the length of the variable in the redirected data set. Extra variables in the redirected data sets cause the stored program to stop processing and an error message is written to the SAS log.

TIP The DROP or KEEP data set options can be added in the stored program if the input data set that you read with the REDIRECT statement has more variables than are in the data set used to compile the program.

Comparisons
The REDIRECT statement applies only to SAS data sets. To redirect input and output stored in external files, include a FILENAME statement to associate the fileref in the source program with different external files.
Example: Executing a Stored Program

This example executes the stored program called STORED.SAMPLE. The REDIRECT statement specifies the source of the input data as BASE.SAMPLE. The output data set from this execution of the program is redirected and stored in a data set named SUMS.SAMPLE.

```
libname stored 'SAS-library';
libname base 'SAS-library';
libname sums 'SAS-library';
data pgm=stored.sample;
  redirect input in.sample=base.sample;
  redirect output out.sample=sums.sample;
run;
```

See Also
- “Stored Compiled DATA Step Programs” in SAS Language Reference: Concepts

Statements:
- “DATA Statement” on page 44

REMOVE Statement

Deletes an observation from a SAS data set.

Valid in: DATA step
Category: Action
Type: Executable
Restrictions: This statement is not supported in a DATA step that runs in CAS.
Use only with a MODIFY statement.

Syntax

```
REMOVE <data-set-name(s)>;
```

Without Arguments

If you specify no argument, the REMOVE statement deletes the current observation from all data sets that are named in the DATA statement.

Arguments

- `data-set-name`
  - specifies the data set in which the observation is deleted.

  Restriction The data set name must also appear in the DATA statement and in one or more MODIFY statements.
Tip

Instead of using a data set name, you can specify the physical pathname to the file, using syntax that your operating system understands. The pathname must be enclosed in single or double quotation marks.

Details

The deletion of an observation can be physical or logical, depending on the engine that maintains the data set. Using REMOVE overrides the default replacement of observations. If a DATA step contains a REMOVE statement, you must explicitly program all output for the step.

Comparisons

- Using an OUTPUT, REPLACE, or REMOVE statement overrides the default write action at the end of a DATA step. (OUTPUT is the default action; REPLACE becomes the default action when a MODIFY statement is used.) If you use any of these statements in a DATA step, you must explicitly program all output for new observations.
- The OUTPUT, REPLACE, and REMOVE statements are independent of each other. More than one statement can apply to the same observation, as long as the sequence is logical.
- If both an OUTPUT and a REPLACE or REMOVE statement execute on a given observation, perform the OUTPUT action last to keep the position of the observation pointer correct.
- Because the REMOVE statement can perform a physical or a logical deletion, REMOVE is available with the MODIFY statement for all SAS data set engines. Both the DELETE and subsetting IF statements perform only physical deletions. Therefore, they are not available with the MODIFY statement for certain engines.

Example: Removing an Observation from a Data Set

This example removes one observation from a SAS data set.

```sas
libname perm 'SAS-library';
data perm.accounts;
   input AcctNumber Credit;
datalines;
1001 1500
1002 4900
1003 3000
;
data perm.accounts;
   modify perm.accounts;
   if AcctNumber=1002 then remove;
run;
proc print data=perm.accounts;
   title 'Edited Data Set';
run;
```

Here are the results of the PROC PRINT statement:
RENAME Statement

Specifies new names for variables in output SAS data sets.

Valid in: DATA step
Categories: CAS Information
Type: Declarative

Syntax

RENAME old-name-1=new-name-1 <…old-name-n=new-name-n>;

Arguments

old-name
specifies the name of a variable or variable list as it appears in the input data set, or in the current DATA step for newly created variables.

new-name
specifies the name or list to use in the output data set.

Details

The RENAME statement enables you to change the names of one or more variables, variables in a list, or a combination of variables and variable lists. The
new variable names are written to the output data set only. Use the old variable names in programming statements for the current DATA step. RENAME applies to all output data sets.

**Note:** The RENAME statement has an effect on data sets opened in output mode only.

### Comparisons

- **RENAME** cannot be used in PROC steps, but the RENAME= data set option can.
- The RENAME= data set option enables you to specify the variables that you want to rename for each input or output data set. Use it in input data sets to rename variables before processing.
- If you use the RENAME= data set option in an output data set, you must continue to use the old variable names in programming statements for the current DATA step. After your output data is created, you can use the new variable names.
- The RENAME= data set option in the SET statement renames variables in the input data set. You can use the new names in programming statements for the current DATA step.
- To rename variables as a file management task, use the DATASETS procedure or access the variables through the SAS windowing interface. These methods are simpler and do not require DATA step processing.

### Example: Renaming Data Set Variables

- These examples show the correct syntax for renaming variables using the RENAME statement:

```
rename street=address;
rename time1=temp1 time2=temp2 time3=temp3;
rename name=Firstname score1-score3=Newscore1-Newscore3;
```

- This example uses the old name of the variable in program statements. The variable Olddept is named Newdept in the output data set, and the variable Oldaccount is named Newaccount.

```
rename Olddept=Newdept Oldaccount=Newaccount;
if Oldaccount>5000;
keep Olddept Oldaccount items volume;
```

- This example uses the old name OLDACCNT in the program statements. However, the new name NEWACCNT is used in the DATA statement because SAS applies the RENAME statement before it applies the KEEP= data set option.

```
data market(keep=newdept newaccnt items volume);
   rename olddept=newdept oldaccnt=newaccnt;
   set sales;
   if oldaccnt>5000;
run;
```

- This example uses both a variable and a variable list to rename variables. New variable names appear in the output data set.
data temp;
  input (score1-score3) {2.,+1} name $;
  rename name=Firstname
       score1-score3=Newscore1-Newscore3;
  datalines;
  12 24 36 Lisa
  22 44 66 Fran
;

See Also

Data Set Options:

- “RENAME= Data Set Option” in SAS Data Set Options: Reference

REPLACE Statement

Replaces an observation in the same location.

Valid in: DATA step
Category: Action
Type: Executable
Restrictions: This statement is not supported in a DATA step that runs in CAS.
Use only with a MODIFY statement.

Syntax

REPLACE <data-set-name-1> <...data-set-name-n>;

Without Arguments

If you specify no argument, the REPLACE statement writes the current observation to the same physical location from which it was read in all data sets that are named in the DATA statement.

Arguments

- **data-set-name** specifies the data set to which the observation is written.

  Requirement The data set name must also appear in the DATA statement and in one or more MODIFY statements.

  Tip Instead of using a data set name, you can specify the physical pathname to the file, using syntax that your operating system understands. The pathname must be enclosed in single or double quotation marks.
Details

Using an explicit REPLACE statement overrides the default replacement of observations. If a DATA step contains a REPLACE statement, explicitly program all output for the step.

Comparisons

- Using an OUTPUT, REPLACE, or REMOVE statement overrides the default write action at the end of a DATA step. (OUTPUT is the default action; REPLACE becomes the default action when a MODIFY statement is used.) If you use any of these statements in a DATA step, you must explicitly program output of a new observation for the step.

- The OUTPUT, REPLACE, and REMOVE statements are independent of each other. More than one statement can apply to the same observation, as long as the sequence is logical.

- If both an OUTPUT and a REPLACE or REMOVE statement execute on a given observation, perform the OUTPUT action last to keep the position of the observation pointer correct.

- REPLACE writes the observation to the same physical location. OUTPUT writes a new observation to the end of the data set.

- REPLACE can appear only in a DATA step that contains a MODIFY statement. You can use OUTPUT with or without MODIFY.

Example: Replacing Observations

This example updates phone numbers in data set MASTER with values in data set TRANS. It also adds one new observation at the end of data set MASTER. The SYSRC autocall macro tests the value of _IORC_ for each attempted retrieval from MASTER. (SYSRC is part of the SAS autocall macro library.) The resulting SAS data set appears after the code:

```
data master;
  input FirstName $ id $ PhoneNumber;
  datalines;
  Kevin ABCjkh 904
  Sandi defns 905
  Terry ghitDP 951
  Jason jklJWM 962
  ;
data trans;
  input FirstName $ id $ PhoneNumber;
  datalines;
  ABCjkh 2904
  defns 2905
  Madeline mnombt 2983
  ;
data master;
  modify master trans;
  by id;
  /* obs found in master   */
  /* change info, replace */
  if _iorc_ = %sysrc(_sok) then replace;
  /* obs not in master     */
```
else if _iorc_ = %sysrc(_dsenmr) then
do;
   /* reset _error_ */
   _error_=0;
   /* reset _iorc_ */
   _iorc_=0;
   /* output obs to master */
   output;
end;
run;
proc print data=master;
   title 'MASTER with New Phone Numbers';
run;

Output 2.31  Data Set with Replaced Observations

MASTER with New Phone Numbers

<table>
<thead>
<tr>
<th>Obs</th>
<th>FirstName</th>
<th>id</th>
<th>PhoneNumber</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Kevin</td>
<td>ABCjkh</td>
<td>2904</td>
</tr>
<tr>
<td>2</td>
<td>Sandi</td>
<td>defsn</td>
<td>2905</td>
</tr>
<tr>
<td>3</td>
<td>Tery</td>
<td>ghitDP</td>
<td>951</td>
</tr>
<tr>
<td>4</td>
<td>Jason</td>
<td>jkJWM</td>
<td>962</td>
</tr>
<tr>
<td>5</td>
<td>Madeline</td>
<td>mnombt</td>
<td>2983</td>
</tr>
</tbody>
</table>

See Also

Statements:
- “MODIFY Statement” on page 212
- “OUTPUT Statement” on page 233
- “REMOVE Statement” on page 273

RESETLINE Statement

Restarts the program line numbers in the SAS log to 1.

Note: The RESETLINE Statement has moved to SAS Global Statements.

RETAIN Statement

Causes a variable that is created by an INPUT or assignment statement to retain its value from one iteration of the DATA step to the next.
Syntax

```
RETAIN <element-list(s) <initial-value(s) | (initial-value-1) | (initial-value-list-1)>
<… element-list-n <initial-value-n | (initial-value-n) | (initial-value-list-n)> >> ;
```

Without Arguments

If you do not specify an argument, the RETAIN statement causes the values of all variables that are created with INPUT or assignment statements to be retained from one iteration of the DATA step to the next.

Arguments

**element-list**

specifies variable names, variable lists, or array names whose values you want retained.

**Tips**

If you specify _ALL_, _CHAR_, or _NUMERIC_, only the variables that are defined before the RETAIN statement are affected.

---

If a variable name is specified only in the RETAIN statement and you do not specify an initial value, the variable is not written to the data set, and a note stating that the variable is uninitialized is written to the SAS log. If you specify an initial value, the variable is written to the data set.

**initial-value**

specifies an initial value, numeric or character, for one or more of the preceding elements.

**Tip**

If you omit *initial-value*, the initial value is missing. *Initial-value* is assigned to all the elements that precede it in the list. All members of a variable list, therefore, are given the same initial value.

**See**

*(initial-value) and (initial-value-list)*

---

**(initial-value)**

specifies an initial value, numeric or character, for a single preceding element or for the first in a list of preceding elements.

**(initial-value-list)**

specifies an initial value, numeric or character, for individual elements in the preceding list. SAS matches the first value in the list with the first variable in the list of elements, the second value with the second variable, and so on.

Element values are enclosed in quotation marks. To specify one or more initial values directly, use this format:

```
(initial-value(s))
```

To specify an iteration factor and nested sublists for the initial values, use this format:
Restriction
If you specify both an initial-value-list and an element-list, then element-list must be listed before initial-value-list in the RETAIN statement.

Tips
You can separate initial values by blank spaces or commas.

You can also use a shorthand notation for specifying a range of sequential integers. The increment is always +1.

You can assign initial values to both variables and temporary data elements.

If there are more variables than initial values, the remaining variables are assigned an initial value of missing and SAS issues a warning message.

Details

Default DATA Step Behavior
Without a RETAIN statement, SAS automatically sets variables that are assigned values by an INPUT or assignment statement to missing before each iteration of the DATA step.

Assigning Initial Values
Use a RETAIN statement to specify initial values for individual variables, a list of variables, or members of an array. If a value appears in a RETAIN statement, variables that appear before it in the list are set to that value initially. (If you assign different initial values to the same variable by naming it more than once in a RETAIN statement, SAS uses the last value.) You can also use RETAIN to assign an initial value other than the default value of 0 to a variable whose value is assigned by a sum statement.

Redundancy
It is redundant to name any of these items in a RETAIN statement, because their values are automatically retained from one iteration of the DATA step to the next:
- variables that are read with a SET, MERGE, MODIFY or UPDATE statement
- a variable whose value is assigned in a sum statement
- the automatic variables _N_, _ERROR_, _I_, _CMD_, and _MSG_
- variables that are created by the END= or IN= option in the SET, MERGE, MODIFY, or UPDATE statement or by options that create variables in the FILE and INFILE statements
- data elements that are specified in a temporary array
- array elements that are initialized in the ARRAY statement
- elements of an array that have assigned initial values to any or all of the elements in the ARRAY statement.

However, you can use a RETAIN statement to assign an initial value to any of the previous items, with the exception of _N_ and _ERROR_.

<constant-iter-value>* <(constant value | constant-sublist)>
Comparisons

The RETAIN statement specifies variables whose values are not set to missing at the beginning of each iteration of the DATA step. The KEEP statement specifies variables that are to be included in any data set that is being created.

Examples:

Example 1: Basic Usage

- This RETAIN statement retains the values of variables MONTH1 through MONTH5 from one iteration of the DATA step to the next:
  ```
  retain month1-month5;
  ```

- This RETAIN statement retains the values of nine variables and sets their initial values:
  ```
  retain month1-month5 1 year 0 a b c 'XYZ';
  ```
  The values of MONTH1 through MONTH5 are set initially to 1; YEAR is set to 0; variables A, B, and C are each set to the character value XYZ.

- This RETAIN statement assigns the initial value 1 to the variable MONTH1 only:
  ```
  retain month1-month5 (1);
  ```
  Variables MONTH2 through MONTH5 are set to missing initially.

- This RETAIN statement retains the values of all variables that are defined earlier in the DATA step but not the values that are defined afterwards:
  ```
  retain _all_;
  ```

- All of these statements assign initial values of 1 through 4 to VAR1 through VAR4:
  ```
  retain var1-var4 (1 2 3 4);
  retain var1-var4 (1,2,3,4);
  retain var1-var4(1:4);
  ```

Example 2: Overview of the RETAIN Operation

This example shows how to use variable names and array names as elements in the RETAIN statement and shows assignment of initial values with and without parentheses:

```sas
data _null_
array City{3} $ City1-City3;
array cp{3} Citypop1-Citypop3;
retain Year Taxyear 1999 City ' ' cp (10000,50000,100000);
file file-specification print;
put 'Values at beginning of DATA step:' / @3 _all_ /;
input Gain;
do i=1 to 3;
cp{i}=cp{i}+Gain;
end;
put 'Values after adding Gain to city populations:'
```


Here are the initial values assigned by RETAIN:

- Year and Taxyear are assigned the initial value 1999.
- City1, City2, and City3 are assigned missing values.
- Citypop1 is assigned the value 10000.
- Citypop2 is assigned 50000.
- Citypop3 is assigned 100000.

Here are the lines written by the PUT statements:

```
Values at beginning of DATA step:
  City1=  City2=  City3=  Citypop1=10000
  Citypop2=50000 Citypop3=100000
Year=1999 Taxyear=1999 Gain=. i=.
_ERROR_=0 _N_=1

Values after adding GAIN to city populations:
  City1=  City2=  City3=  Citypop1=15000
  Citypop2=55000 Citypop3=105000
Year=1999 Taxyear=1999 Gain=5000 i=4
_ERROR_=0 _N_=1

Values at beginning of DATA step:
  City1=  City2=  City3=  Citypop1=15000
  Citypop2=55000 Citypop3=105000
Year=1999 Taxyear=1999 Gain=. i=.
_ERROR_=0 _N_=2

Values after adding GAIN to city populations:
  City1=  City2=  City3=  Citypop1=25000
  Citypop2=65000 Citypop3=115000
Year=1999 Taxyear=1999 Gain=10000 i=4
_ERROR_=0 _N_=2

Values at beginning of DATA step:
  City1=  City2=  City3=  Citypop1=25000
  Citypop2=65000 Citypop3=115000
Year=1999 Taxyear=1999 Gain=. i=.
_ERROR_=0 _N_=3
```

The first PUT statement is executed three times, whereas the second PUT statement is executed only twice. The DATA step ceases execution when the INPUT statement executes for the third time and reaches the end of the file.

Example 3: Selecting One Value from a Series of Observations

In this example, the data set ALLSCORES contains several observations for each identification number and variable ID. Different observations for a particular ID value might have different values of the variable GRADE. This example creates a new data set, CLASS.BESTSCORES, which contains one observation for each ID value. The observation must have the highest GRADE value of all observations for that ID in BESTSCORES.

```sas
libname class 'SAS-library';
proc sort data=class.allscores;
   by id;
run;
data class.bestscores;
```
drop grade;
set class.allscores;
by id;
    /* Prevents HIGHEST from being reset*/
    /* to missing for each iteration. */
retain highest;
    /* Sets HIGHEST to missing for each */
    /* different ID value. */
if first.id then highest=.;
    /* Compares HIGHEST to GRADE in */
    /* current iteration and resets */
    /* value if GRADE is higher. */
highest=max(highest,grade);
if last.id then output;
run;

See Also

Statements:
- "Assignment Statement" on page 28
- "BY Statement" on page 34
- "INPUT Statement" on page 146

Other Documentation
- "Inter-row Dependencies" in SAS Cloud Analytic Services: DATA Step Programming
- "Sum a Variable across an Entire Table" in SAS Cloud Analytic Services: DATA Step Programming

RETURN Statement

Stops executing statements at the current point in the DATA step and returns to a predetermined point in the step.

Valid in: DATA step
Categories: CAS, Control
Type: Executable

Syntax
RETURN;

Without Arguments

The RETURN statement causes execution to stop at the current point in the DATA step, and returns control to a previous DATA step statement.
Details

The point to which SAS returns depends on the order in which statements are executed in the DATA step.

The RETURN statement is often used with the

- GO TO statement
- HEADER= option in the FILE statement
- LINK statement.

When RETURN causes a return to the beginning of the DATA step, an implicit OUTPUT statement writes the current observation to any new data sets (unless the DATA step contains an explicit OUTPUT statement, or REMOVE or REPLACE statements with MODIFY statements). Every DATA step has an implied RETURN as its last executable statement.

Example: Basic Usage

In this example, when the values of X and Y are the same, SAS executes the RETURN statement and adds the observation to the data set. When the values of X and Y are not equal, SAS executes the remaining statements and then adds the observation to the data set.

```sas
data survey;
  input x y;
  if x=y then return;
  put x= y=;
datalines;
21 25
20 20
7 17
;```

See Also

**Statements:**

- “FILE Statement” on page 75
- “GO TO Statement” on page 104
- “LINK Statement” on page 196

---

**RUN Statement**

Executes the previously entered SAS statements.

Note: The RUN Statement has moved to SAS Global Statements.
%RUN Statement

Ends source statements following a %INCLUDE * statement.

Note: The %RUN Statement has moved to SAS Global Statements.

SASFILE Statement

Opens a SAS data set and allocates enough buffers to hold the entire file in memory.

Note: The SASFILE Statement has moved to SAS Global Statements.

SELECT Statement

Executes one of several statements or groups of statements.

Valid in: DATA step
Categories: CAS
            Control
Type: Executable

Syntax

```
SELECT <(select-expression)> ;
   WHEN-1 (when-expression-1 <…, when-expression-n> ) statement;
   <… WHEN-n (when-expression-1 <…, when-expression-n> ) statement;>
   < OTHERWISE statement;>
END;
```

Arguments

(select-expression)

specifies any SAS expression that evaluates to a single value.

Restriction

If a variable is specified in the select-expression, an operator cannot be used in a when-expression. For more information, see “Example 6: Using Variables in the Select Statement” on page 289.

See

"Evaluating the when-expression When a select-expression Is Included" on page 287

(when-expression)

specifies any SAS expression, including a compound expression. SELECT requires you to specify at least one when-expression.
Restriction  An operator can be used in a when-expression as long as a variable does not appear in the select-expression. For more information, see "Example 6: Using Variables in the Select Statement" on page 289.

Tips  Separating multiple when-expressions with a comma is equivalent to separating them with the logical operator OR.

The way a when-expression is used depends on whether a select-expression is present.

See  "Evaluating the when-expression When a select-expression Is Not Included" on page 287

statement  can be any executable SAS statement, including DO, SELECT, and null statements. You must specify the statement argument.

Details
Using WHEN Statements in a SELECT Group
The SELECT statement begins a SELECT group. SELECT groups contain WHEN statements that identify SAS statements that are executed when a particular condition is true. Use at least one WHEN statement in a SELECT group. An optional OTHERWISE statement specifies a statement to be executed if no WHEN condition is met. An END statement ends a SELECT group.

Null statements that are used in WHEN statements cause SAS to recognize a condition as true without taking further action. Null statements that are used in OTHERWISE statements prevent SAS from issuing an error message when all WHEN conditions are false.

Evaluating the when-expression When a select-expression Is Included
If the select-expression is present, SAS evaluates the select-expression and when-expression. SAS compares the two for equality and returns a value of true or false. If the comparison is true, statement is executed. If the comparison is false, execution proceeds either to the next when-expression in the current WHEN statement, or to the next WHEN statement if no more expressions are present. If no WHEN statements remain, execution proceeds to the OTHERWISE statement, if one is present. If the result of all SELECT-WHEN comparisons is false and no OTHERWISE statement is present, SAS issues an error message. If more than one WHEN statement has a true when-expression evaluates to true, SAS proceeds to the OTHERWISE statement if it is present. If the result of all when-expressions is false and no OTHERWISE statement is present, SAS issues an error message. If more than one WHEN statement has a true when-expression

Evaluating the when-expression When a select-expression Is Not Included
If no select-expression is present, the when-expression is evaluated to produce a result of true or false. If the result is true, statement is executed. If the result is false, SAS proceeds to the next when-expression in the current WHEN statement or to the next WHEN statement if no more expressions are present. (That is, SAS performs the action that is indicated in the first true WHEN statement.) If no when-expression evaluates to true, SAS proceeds to the OTHERWISE statement if it is present. If the result of all when-expressions is false and no OTHERWISE statement is present, SAS issues an error message. If more than one WHEN statement has a true when-expression.
expression, only the first WHEN statement is used. After a when-expression is true, no other when-expressions are evaluated.

Processing Large Amounts of Data with %INCLUDE Files

One way to process large amounts of data is to use %INCLUDE statements in your DATA step. Using %INCLUDE statements enables you to perform complex processing while keeping your main program manageable. The %INCLUDE files that you use in your main program can contain WHEN statements and other SAS statements to process your data. See “Example 5: Processing Large Amounts of Data” on page 289 for an example.

Comparisons

Use IF-THEN/ELSE statements for programs with few statements. Use subsetting IF statements without a THEN clause to continue processing only those observations or records that meet the condition that is specified in the IF clause.

The SELECT statement works much like the CASE statement in the SQL procedure.

Examples:

Example 1: Using Statements

```
select (a);
  when (1) x=x*10;
  when (2);
  when (3,4,5) x=x*100;
  otherwise;
end;
```

Example 2: Using DO Groups

```
select (payclass);
  when ('monthly') amt=salary;
  when ('hourly')
    do;
      amt=hrlywage*min(hrs,40);
      if hrs>40 then put 'CHECK TIMECARD';
    end; /* end of do */
  otherwise put 'PROBLEM OBSERVATION';
end; /* end of select */
```

Example 3: Using a Compound Expression

```
select;
  when (mon in ('JUN', 'JUL', 'AUG')
    and temp>70) put 'SUMMER ' mon=;
  when (mon in ('MAR', 'APR', 'MAY'))
    put 'SPRING ' mon=;
  otherwise put 'FALL OR WINTER ' mon=;
end;
```
Example 4: Making Comparisons for Equality

```plaintext
/* INCORRECT usage to select value of 2 */
select (x);
/* evaluates T/F and compares for */
/* equality with x */
when (x=2) put 'two';
end;
/* correct usage */
select(x);
/* compares 2 to x for equality */
when (2) put 'two';
end;
/* correct usage */
select;
/* compares 2 to x for equality */
when (x=2) put 'two';
end;
```

Example 5: Processing Large Amounts of Data

In this example, the %INCLUDE statements contain code that includes WHEN statements to process new and old items in the inventory. The main program shows the overall logic of the DATA step.

```plaintext
data test (keep=ItemNumber);
set ItemList;
select;
%include NewItems;
%include OldItems;
otherwise put 'Item ' ItemNumber ' is not in the inventory.';
end;
run;
```

Example 6: Using Variables in the Select Statement

A variable can be used in an operation in the `when-expression` if the variable is not specified in the SELECT statement. For example:

```plaintext
data _null_
    h1=hour(datetime());
select;
    when (h1<12) greetp1 = "Morning ";
end;
put greetp1;
run;
```

This example produces an error because the less than (<) operator that is specified in the `when-expression` is attempting to operate on the h1 variable that is specified in the `select-expression`.

```plaintext
data _null_
    h1=hour(datetime());
select (h1);
    when (<12) greetp1 = "Morning ";
end;
prompt greetp1;
```
Here is the SAS log that is produced at the point of the less than (<) operator:

```
Syntax error, expecting one of the following: a name, a quoted string, a numeric constant, a datetime constant, a missing value, INPUT, PUT
```

See Also

**Statements:**
- "DO Statement" on page 62
- "IF Statement: Subsetting" on page 105
- "IF-THEN/ELSE Statement" on page 108

---

**SET Statement**

Reads an observation from one or more SAS data sets.

**Valid in:** DATA step

**Categories:** CAS  
File-Handling

**Type:** Executable

**Note:** The variables read using the SET statement are retained in the PDV. For more information, see “Overview of DATA Step Processing” in SAS Language Reference: Concepts and the “RETAIN Statement” on page 279.

**Syntax**

```
SET<
SAS-data-set(s) <(data-set-options(s))> > <options>;
```

**Without Arguments**

When you do not specify an argument, the SET statement reads an observation from the most recently created data set.

**Arguments**

**SAS-data-set (s)**  
specifies a one-level name, a two-level name, or one of the special SAS data set names.

**Tips**  
You can specify data set lists. For more information, see “Using Data Set Lists with SET” on page 296.

Instead of using a data set name, you can specify the physical pathname to the file, using syntax that your operating system
understands. The pathname must be enclosed in single or double quotation marks.

See “SAS Data Sets” in SAS Language Reference: Concepts for a description of the levels of SAS data set names and when to use each level.

Example “Example 13: Using Data Set Lists” on page 302

\( \text{(data-set-options)} \)

specifies actions SAS is to take when it reads variables or observations into the program data vector for processing.

Tip Data set options that apply to a data set list apply to all of the data sets in the list.

See For more information, see “Definition of Data Set Options” in SAS Data Set Options: Reference for a list of the data set options to use with input data sets.

SET Options

\( \text{CUROBS=} \text{variable} \)

creates and names a variable that contains the observation number that was just read from the data set.

Example “Example 14: Finding the Current Observation Number” on page 304

\( \text{END=} \text{variable} \)

creates and names a temporary variable that contains an end-of-file indicator. The variable, which is initialized to zero, is set to 1 when SET reads the last observation of the last data set listed. This variable is not added to any new data set.

Restriction END= cannot be used with POINT=. When random access is used, the END= variable is never set to 1.

Interaction If you use a BY statement, END= is set to 1 when the SET statement reads the last observation of the interleaved data set. For more information, see “BY-Group Processing with SET” on page 297.

Example “Example 11: Writing an Observation Only After All Observations Have Been Read” on page 301

\( \text{INDSNAME=} \text{variable} \)

creates and names a variable that stores the name of the SAS data set from which the current observation is read. The stored name can be a data set name or a physical name. The physical name is the name by which the operating environment recognizes the file.

Tips For data set names, SAS adds the library name to the variable value (for example, WORK.PRICE) and converts the two-level name to uppercase.

Unless previously defined, the length of the variable is set to 41 bytes. Use a LENGTH statement to make the variable length long enough to
contain the value of the physical file name if the file name is longer than 41 bytes.

If the variable is previously defined as a character variable with a specific length, that length is not changed. If the value that is placed into the INDSNAME variable is longer than that length, the value is truncated.

If the variable is previously defined as a numeric variable, an error occurs.

The variable is available in the DATA step, but the variable is not added to any output data set.

Example
"Example 12: Retrieving the Name of the Data Set from Which the Current Observation Is Read" on page 301

KEY=</UNIQUE>
provides nonsequential access to observations in a SAS data set, which are based on the value of an index variable or a key.

Range
Specify the name of a simple or composite index of the data set that is being read.

Restriction
KEY= cannot be used with POINT=.

Tips
Using the _IORC_ automatic variable in conjunction with the SYSRC autocall macro provides you with more error-handling information than was previously available. When you use the SET statement with the KEY= option, the new automatic variable _IORC_ is created. This automatic variable is set to a return code that shows the status of the most recent I/O operation that is performed on an observation in a SAS data set. If the KEY= value is not found, the _IORC_ variable returns a value that corresponds to the SYSRC autocall macro’s mnemonic _DSENOM and the automatic variable _ERROR_ is set to 1.

When using the SET statement with the KEY= option and a non-unique index, it is often desirable to force the SET statement to start reading again with the first observation that matches the key value. Use the KEYRESET= option to control whether a KEY= search should begin at the top of the index for the data set that is being read.

See
For more information, see the description of the autocall macro SYSRC in SAS Macro Language: Reference.

"KEYRESET=variable" on page 293

UNIQUE option on page 295

Examples
"Example 7: Performing a Table Lookup" on page 300

"Example 8: Performing a Table Lookup When the Master File Contains Duplicate Observations" on page 300

CAUTION
Continuous loops can occur when you use the KEY= option. If you use the KEY= option without specifying the primary data set, you
must include either a STOP statement to stop DATA step processing or programming logic that uses the _IORC_ automatic variable in conjunction with the SYSRC autocall macro and checks for an invalid value of the _IORC_ variable, or both.

**KEYRESET=variable**
controls whether a KEY= search should begin at the top of the index for the data set that is being read. When the value of the KEYRESET variable is 1, the index lookup begins at the top of the index. When the value of the KEYRESET variable is 0, the index lookup is not reset and the lookup continues where the prior lookup ended.

**Interaction**
The KEYRESET= option is similar to the UNIQUE option, except the KEYRESET= option enables you to determine when the KEY= search should begin at the top of the index again.

**See**
"KEY=index/<UNIQUE>" on page 292

"UNIQUE" on page 295

**Example**
“Example 15: Using the KEYRESET Option” on page 304

**NOBS=variable**
creates and names a temporary variable whose value is usually the total number of observations in the input data set or data sets. If more than one data set is listed in the SET statement, the value of the NOBS= variable equals the total number of observations in the data sets that are listed. The number of observations includes those observations that are marked for deletion but are not yet deleted.

**Restriction**
For certain SAS views and sequential engines such as the TAPE and XML engines, SAS cannot determine the number of observations. In these cases, SAS sets the value of the NOBS= variable to the largest positive integer value that is available in your operating environment.

**Interaction**
The NOBS= and POINT= options are independent of each other.

**Tip**
At compilation time, SAS reads the descriptor portion of each data set and assigns the value of the NOBS= variable automatically. Thus, you can refer to the NOBS= variable before the SET statement. The variable is available in the DATA step but is not added to any output data set.

**Example**
“Example 10: Performing a Function until the Last Observation Is Reached” on page 301

**OPEN=(| DEFER)**
enables you to delay the opening of any concatenated SAS data sets until they are ready to be processed.

**IMMEDIATE**
during the compilation phase, opens all data sets that are listed in the SET statement.

**Restriction**
When you use the IMMEDIATE option, KEY=, POINT=, and BY statement processing are mutually exclusive.
Tip

If a variable on a subsequent data set is of a different type (for example, character versus numeric) from the type of the same-named variable on the first data set, the DATA step stops processing and produces an error message.

DEFER

opens the first data set during the compilation phase, and opens subsequent data sets during the execution phase. When the DATA step reads and processes all observations in a data set, it closes the data set and opens the next data set in the list.

Restriction

When you specify the DEFER option, you cannot use the KEY= statement option, the POINT= statement option, or the BY statement. These constructs imply either random processing or interleaving of observations from the data sets, which is not possible unless all data sets are open.

Requirement

You can use the DROP=, KEEP=, or RENAME= data set options to process a set of variables, but the set of variables that are processed for each data set must be identical. In most cases, if the set of variables defined by any subsequent data set differ from the variables defined by the first data set, SAS prints a warning message to the log but does not stop execution.

- If a variable on a subsequent data set is of a different type (for example, character versus numeric) from the type of the same-named variable on the first data set, the DATA step stops processing and produces an error message.
- If a variable on a subsequent data set was not defined by the first data set in the SET statement, but was defined previously in the DATA step program, the DATA step stops processing and produces an error message. In this case, the value of the variable in previous iterations might be incorrect because the semantic behavior of SET requires this variable to be set to missing when processing the first observation of the first data set.

Default

IMMEDIATE

POINT=variable

specifies a temporary variable whose numeric value determines which observation is read. POINT= causes the SET statement to use random (direct) access to read a SAS data set.
### Restrictions
You cannot use POINT= with a BY statement, a WHERE statement, or a WHERE= data set option. In addition, you cannot use POINT= with transport format data sets, data sets in sequential format on tape or disk, and SAS/ACCESS views or the SQL procedure views that read data from external files.

You cannot use POINT= with KEY=.

### Requirement
a STOP statement

### Note
Remember that _N_ is an iteration count and not the observation number of the last observation that was read.

### Tips
You must supply the values of the POINT= variable. For example, you can use the POINT= variable as the index variable in some form of the DO statement.

The POINT= variable is available anywhere in the DATA step, but it is not added to any new SAS data set.

### Examples
“Example 6: Combining One Observation with Many” on page 300

“Example 9: Reading a Subset by Using Direct Access” on page 300

### CAUTION
**Continuous loops can occur when you use the POINT= option.** When you use the POINT= option, you must include a STOP statement to stop DATA step processing, or programming logic that checks for an invalid value of the POINT= variable, or both. Because POINT= reads only those observations that are specified in the DO statement, SAS cannot read an end-of-file indicator as it would if the file were being read sequentially. Because reading an end-of-file indicator ends a DATA step automatically, failure to substitute another means of ending the DATA step when you use POINT= can cause the DATA step to go into a continuous loop. If SAS reads an invalid value of the POINT= variable, it sets the automatic variable _ERROR_ to 1. Use this information to check for conditions that cause continuous DO-loop processing, or include a STOP statement at the end of the DATA step, or both.

### UNIQUE
causes a KEY= search always to begin at the top of the index for the data set that is being read.

**TIP** For examples, see *Combining and Modifying SAS Data Sets: Examples, Second Edition*.

### Restriction
UNIQUE can appear only with the KEY= argument and must be preceded by a slash.

### Notes
By default, SET begins searching at the top of the index only when the KEY= value changes.

If the KEY= value does not change on successive executions of the SET statement, the search begins by following the most recently
retrieved observation. In other words, when consecutive duplicate KEY= values appear, the SET statement attempts a one-to-one match with duplicate indexed values in the data set that is being read. If more consecutive duplicate KEY= values are specified than exist in the data set that is being read, the extra duplicates are treated as not found.

When KEY= is a unique value, only the first attempt to read an observation with that key value succeeds; subsequent attempts to read the observation with that value of the key fail. The _IORC_ variable returns a value that corresponds to the SYSRC autocall macro’s mnemonic _DSENOM. If you add the /UNIQUE option, subsequent attempts to read the observation with the unique KEY= value succeed. The _IORC_ variable returns a 0.

See

“KEYRESET=variable” on page 293

Example

“Example 8: Performing a Table Lookup When the Master File Contains Duplicate Observations” on page 300

Details

What SET Does

Each time the SET statement is executed, SAS reads one observation into the program data vector. SET reads all variables and all observations from the input data sets unless you tell SAS to do otherwise. A SET statement can contain multiple data sets; a DATA step can contain multiple SET statements.

Note: When the DATA step comes to an end-of-file indicator or the end of all open data sets, it performs an orderly shutdown. For example, if you use SET with FIRSTOBS, a file with only a header record in a series of files triggers a normal shutdown of the DATA step. The shutdown occurs because SAS reads beyond the end-of-file indicator and the DATA step terminates. You can use the END= option to avoid the shutdown.

Uses

The SET statement is flexible and has a variety of uses in SAS programming. These uses are determined by the options and statements that you use with the SET statement:

- reading observations and variables from existing SAS data sets for further processing in the DATA step
- concatenating and interleaving data sets, and performing one-to-one reading of data sets
- reading SAS data sets by using direct access methods

Using Data Set Lists with SET

You can use data set lists with the SET statement. Data set lists provide a quick way to reference existing groups of data sets. These data set lists must either be name prefix lists or numbered range lists.
Name prefix lists refer to all data sets that begin with a specified character string. For example, set SALES1:; tells SAS to read all data sets that start with "SALES1" such as SALES1, SALES10, SALES11, and SALES12.

Numbered range lists require you to have a series of data sets with the same name, except for the last character or characters, which are consecutive numbers. In a numbered range list, you can begin with any number and end with any number. For example, these lists refer to the same data sets: sales1 sales2 sales3 sales4 sales1-sales4

---

**Note:** If the numeric suffix of the first data set name contains leading zeros, the number of digits in the numeric suffix of the last data set name must be greater than or equal to the number of digits in the first data set name. Otherwise, an error occurs. For example, the data set lists sales001–sales99 and sales01–sales9 cause an error. The data set list sales001–sales999 is valid. If the numeric suffix of the first data set name does not contain leading zeros, the number of digits in the numeric suffix of the first and last data set names do not have to be equal. For example, the data set list sales1–sales999 is valid.

---

Here are some other rules to consider when using numbered data set lists:

- You can specify groups of ranges.
  ```
  set cost1-cost4 cost11-cost14 cost21-cost24;
  ```

- You can combine numbered range lists with name prefix lists.
  ```
  set cost1-cost4 cost2: cost33-37;
  ```

- You can mix single data sets with data set lists.
  ```
  set cost1 cost10-cost20 cost30;
  ```

- Quotation marks around data set lists are ignored.
  ```
  /* these two lines are the same */
  set sales1 - sales4;
  set 'sales1'n - 'sales4'n;
  ```

- Spaces in data set names are invalid. If quotation marks are used, trailing blanks are ignored.
  ```
  /* blanks in these statements will cause errors */
  set sales 1 - sales 4;
  set 'sales 1'n - 'sales 4'n;
  /* trailing blanks in this statement will be ignored */
  set 'sales 1'n - 'sales 4'n;
  ```

- The maximum numeric suffix is 2147483647.
  ```
  /* this suffix will cause an error */
  set prod2000000000-prod2934850239;
  ```

---

BY-Group Processing with SET

Only one BY statement can accompany each SET statement in a DATA step. The BY statement should immediately follow the SET statement to which it applies. The data sets that are listed in the SET statement must be sorted by the values of the variables that are listed in the BY statement, or they must have an appropriate index. SET, when it is used with a BY statement, interleaves data sets. The observations in the new data set are arranged by the values of the BY variable or variables, and within each BY group, by the order of the data sets in which they
occur. For an example of BY-group processing with the SET statement, see “Example 2: Interleaving SAS Data Sets” on page 299.

Combining SAS Data Sets

Use a single SET statement with multiple data sets to concatenate the specified data sets. The number of observations in the new data set is the sum of the number of observations in the original data sets, and the order of the observations is all the observations from the first data set followed by all the observations from the second data set, and so on. For an example of concatenating data sets, see “Example 1: Concatenating SAS Data Sets” on page 298.

Use a single SET statement with a BY statement to interleave the specified data sets. The observations in the new data set are arranged by the values of the BY variable or variables, and within each BY group, by the order of the data sets in which they occur. For an example of interleaving data sets, see “Example 2: Interleaving SAS Data Sets” on page 299.

Use multiple SET statements to perform one-to-one reading (also called one-to-one matching) of the specified data sets. The new data set contains all the variables from all the input data sets. The number of observations in the new data set is the number of observations in the smallest original data set. If the data sets contain common variables, the values that are read in from the last data set replace the values that were read in from earlier data sets. For examples of one-to-one reading of data sets, see

- “Example 6: Combining One Observation with Many” on page 300
- “Example 7: Performing a Table Lookup” on page 300
- “Example 8: Performing a Table Lookup When the Master File Contains Duplicate Observations” on page 300

**TIP** For more information, see *Combining and Modifying SAS Data Sets: Examples, Second Edition*.

For more information about how to prepare your data sets, see “Combining SAS Data Sets: Basic Concepts” in *SAS Language Reference: Concepts*.

Comparisons

- SET reads an observation from an existing SAS data set. INPUT reads raw data from an external file or from in-stream data lines in order to create SAS variables and observations.
- Using the KEY= option with SET enables you to access observations nonsequentially in a SAS data set according to a value. Using the POINT= option with SET enables you to access observations nonsequentially in a SAS data set according to the observation number.

Examples:

Example 1: Concatenating SAS Data Sets
If more than one data set name appears in the SET statement, the resulting output data set is a concatenation of all the data sets that are listed. SAS reads all observations from the first data set, then all observations from the second data set, and so on, until all observations from all the data sets have been read. This example concatenates the three SAS data sets into one output data set named FITNESS.

```sas
data fitness;
   set health exercise well;
run;
```

**Example 2: Interleaving SAS Data Sets**

To interleave two or more SAS data sets, use a BY statement after the SET statement.

```sas
data april;
   set payable recvable;
   by account;
run;
```

**Example 3: Reading a SAS Data Set**

In this DATA step, each observation in the data set NC.MEMBERS is read into the program data vector. Only those observations whose value of CITY is Raleigh are written to the new data set RALEIGH.MEMBERS.

```sas
data raleigh.members;
   set nc.members;
   if city='Raleigh';
run;
```

**Example 4: Merging a Single Observation with All Observations in a SAS Data Set**

An observation to be merged into an existing data set can be created by a SAS procedure or another DATA step. In this example, the data set AVGSALES has only one observation.

```sas
data national;
   if _n_=1 then set avgsales;
   set totsales;
run;
```

**Example 5: Reading from the Same Data Set More Than Once**

In this example, SAS treats each SET statement independently. That is, it reads from one data set as if it were reading from two separate data sets.

```sas
data drugxyz;
   set trial5(keep=sample);
   if sample>2;
   set trial5;
run;
```

For each iteration of the DATA step, the first SET statement reads one observation. The next time the first SET statement is executed, it reads the next observation. Each SET statement can read different observations with the same iteration of the DATA step.
Example 6: Combining One Observation with Many

You can subset observations from one data set and combine them with observations from another data set by using direct access methods.

data south;
   set revenue;
   if region=4;
   set expense point=_n_;
runto;

Example 7: Performing a Table Lookup

This example illustrates using the KEY= option to perform a table lookup. The DATA step reads a primary data set that is named INVTORY and a lookup data set that is named PARTCODE. The DATA step uses the index PARTNO to read PARTCODE nonsequentially, by looking for a match between the PARTNO value in each data set. The purpose is to obtain the appropriate description, which is available only in the variable DESC in the lookup data set, for each part that is listed in the primary data set.

data combine;
   set invtory(keep=partno instock price);
   set partcode(keep=partno desc) key=partno;
runto;

Example 8: Performing a Table Lookup When the Master File Contains Duplicate Observations

This example uses the KEY= option to perform a table lookup. The DATA step reads a primary data set that is named INVTORY, which is indexed on PARTNO, and a lookup data set named PARTCODE. PARTCODE contains quantities of new stock (variable NEW_STK). The UNIQUE option ensures that, if there are any duplicate observations in INVTORY, values of NEW_STK are added only to the first observation of the group.

data combine;
   set partcode(keep=partno new_stk);
   set invtory(keep=partno instock price)
key=partno/unique;
   instock=instock+new_stk;
runto;

Example 9: Reading a Subset by Using Direct Access

These statements select a subset of 50 observations from the data set DRUGTEST by using the POINT= option to access observations directly by number.

data sample;
   do obsnum=1 to 100 by 2;
      set drugtest point=obsnum;
      if _error_ then abort;
      output;
   end;
stop;
runto;
Example 10: Performing a Function until the Last Observation Is Reached

These statements use NOBS= to set the termination value for DO-loop processing. The value of the temporary variable LAST is the sum of the observations in SURVEY1 and SURVEY2.

```sas
do obsnum=1 to last by 100;
    set survey1 survey2 point=obsnum nobs=last;
    output;
end;
stop;
```

Example 11: Writing an Observation Only After All Observations Have Been Read

This example uses the END= variable LAST to tell SAS to assign a value to the variable REVENUE and write an observation only after the last observation of RENTAL has been read.

```sas
set rental end=last;
totdays + days;
if last then
    do;
        revenue=totdays*65.78;
        output;
    end;
```

Example 12: Retrieving the Name of the Data Set from Which the Current Observation Is Read

This example creates three data sets and stores the data set name in a variable named `dsn`. The name is split into three parts and the example prints the results.

```sas
/* Create some data sets to read */
data gas_price_option; value=395; run;
data gas_rbid_option; value=840; run;
data gas_price_forward; value=275; run;
/* Create a data set D */
data d;
    set gas_price_option gas_rbid_option gas_price_forward indsnname=dsn;
    /* split the data set names into 3 parts */
    commodity = scan (dsn, 2, "._");
    type = scan (dsn, 3, "._");
    instrument = scan (dsn, 4, "._");
    run;
proc print data=d;
run;
```
Example 13: Using Data Set Lists

This example uses a numbered range list to read in the data sets.

```sas
   data dept008; emp=13; run;
   data dept009; emp=9; run;
   data dept010; emp=4; run;
   data dept011; emp=33; run;
   data _null_;
      set dept008-dept010;
      put _all_;
   run;
```

The program writes these lines to the SAS log.
### Example 2.2 Using a Data Set List with the SET Statement

```sas
1  data dept008; emp=13; run;
NOTE: The data set WORK.DEPT008 has 1 observations and 1 variables.
NOTE: DATA statement used (Total process time):
   real time       0.06 seconds
   cpu time        0.03 seconds

2  data dept009; emp=9; run;
NOTE: The data set WORK.DEPT009 has 1 observations and 1 variables.
NOTE: DATA statement used (Total process time):
   real time       0.00 seconds
   cpu time        0.00 seconds

3  data dept010; emp=4; run;
NOTE: The data set WORK.DEPT010 has 1 observations and 1 variables.
NOTE: DATA statement used (Total process time):
   real time       0.00 seconds
   cpu time        0.00 seconds

4  data dept011; emp=33; run;
NOTE: The data set WORK.DEPT011 has 1 observations and 1 variables.
NOTE: DATA statement used (Total process time):
   real time       0.00 seconds
   cpu time        0.00 seconds

5
6  data _null_;  
7  set dept008-dept010;
8  put _all_;  
9  run;  
emp=13 _ERROR_=0 _N_=1
emp=9 _ERROR_=0 _N_=2
emp=4 _ERROR_=0 _N_=3
NOTE: There were 1 observations read from the data set WORK.DEPT008.
NOTE: There were 1 observations read from the data set WORK.DEPT009.
NOTE: There were 1 observations read from the data set WORK.DEPT010.
NOTE: DATA statement used (Total process time):
   real time       0.00 seconds
   cpu time        0.00 seconds
```

In addition, you could use data set lists to find missing data sets. This example uses a numbered range list to locate the missing data sets. An error occurs for each data set that does not exist. When you know which data sets are missing, you can correct the SET statement to reflect the data sets that actually exist.

```sas
data dept008; emp=13; run;  
data dept009; emp=9; run;  
data dept011; emp=4; run;  
data dept014; emp=33; run;  
data _null_;  
   set dept008-dept014;  
   put _all_;  
run;
```

The program writes these lines to the SAS log.
Example Code 2.3  Finding Missing Data Sets Using the SET Statement

Example 14: Finding the Current Observation Number
This example uses the CUROBS option to return the number of the current observation.

data women;
  set sashelp.class curobs=cobs;
  where sex = 'F';
  orig_obs = cobs;
run;

Example 15: Using the KEYRESET Option
This example uses the KEYRESET= option to look up all the values when I=3 two times.

data a(index=(i));
  do i = 1,2,3,3,3,4,5;
    j=ranuni(4);
    output;
  end;
run;
data _null_; input i;
reset = 1;
do while (_iorc_ = 0);
   set a key=i keyreset=reset;
   put _all_;
end;
_error_ = 0; _iorc_ = 0;
datalines;
3
3
;
Syntax

STOP;

Details

The STOP statement causes SAS to stop processing the current DATA step immediately and resume processing statements after the end of the current DATA step.

SAS writes a data set for the current DATA step. However, the observation being processed when STOP executes is not added. The STOP statement can be used alone or in an IF-THEN statement or SELECT group.

Use STOP with any features that read SAS data sets using random access methods, such as the POINT= option in the SET statement. Because SAS does not detect an end-of-file with this access method, you must include program statements to prevent continuous processing of the DATA step.

Comparisons

- When you use a windowing environment or other interactive methods of operation, the ABORT statement and the STOP statement both stop processing. The ABORT statement sets the value of the automatic variable _ERROR_ to 1, but the STOP statement does not.
- In batch or noninteractive mode, the two statements also have different effects. Use the STOP statement in batch or noninteractive mode to continue processing with the next DATA or PROC step.

Examples:

Example 1: Basic Usage

- stop;
- if idcode=9999 then stop;
- select {a};
  - when (0) output;
  - otherwise stop;
- end;

Example 2: Avoiding an Infinite Loop

This example shows how to use STOP to avoid an infinite loop within a DATA step when you are using random access methods:

data sample;
  do sampleobs=1 to totalobs by 10;
    set master.research point=sampleobs nobs=totalobs;
    output;
  end;
  stop;
run;
Sum Statement

Adds the result of an expression to an accumulator variable.

Valid in: DATA step
Categories: Action, CAS
Type: Executable

Example:
```sas
data Table2;
  set Table1;
  total + x;
run;
```

Syntax

```sas
variable + expression;
```

Arguments

**variable**

specifies the name of the accumulator variable, which contains a numeric value.

**expression**

is any SAS expression.

Comparisons

The sum statement is equivalent to using the SUM function and the RETAIN statement, as shown here:

```sas
retain variable 0;
```
variable=sum(variable,expression);

Example

Here are examples of sum statements that illustrate various expressions. The accumulator variable is highlighted in each example:

- data Table2;
  set Table;
  Total + x;
  run;

- data AccountBal;
  set Account;
  retain Balance 1000;
  Balance + (-Debit);
  run;

- data Table2;
  set Table;
  SumXsq + x * x;
  run;

- data Table2;
  set Table;
  nx + (x ne .);
  run;

- data AccountBal2;
  set AccountBal;
  if Balance <=500 then Alert + 1;
  run;

See Also

Functions:
- “SUM Function” in SAS Functions and CALL Routines: Reference

Statements:
- “RETAIN Statement” on page 279

Other Documentation
- “Inter-row Dependencies” in SAS Cloud Analytic Services: DATA Step Programming
- “Sum a Variable across an Entire Table” in SAS Cloud Analytic Services: DATA Step Programming

SYSECHO Statement

Sends a global statement complete event and passes a text string back to the IOM client.

Note: The SYSECHO Statement has moved to SAS Global Statements.
TITLE Statement
Specifies title lines for SAS output.

Note: The TITLE Statement has moved to SAS Global Statements.

UPDATE Statement
Updates a master file by applying transactions.

Valid in: DATA step
Category: File-Handling
Type: Executable
Restriction: This statement is not supported in a DATA step that runs in CAS.

Note: The values read using the UPDATE statement are retained in the PDV. For more information, see “Overview of DATA Step Processing” in SAS Language Reference: Concepts and the “RETAIN Statement” on page 279.

CAUTION: If you add an OUTPUT statement when using an UPDATE statement, the results that are generated are predictable but can be undesired.

Syntax
<END=variable>
<UPDATEMODE=MISSINGCHECK | NOMISSINGCHECK>;
BY by-variable;

Arguments

master-data-set
specifies the SAS data set used as the master file.

Range
The name can be a one-level name (for example, FITNESS), a two-level name (for example, IN.FITNESS), or one of the special SAS data set names.

Tip
Instead of using a data set name, you can specify the physical pathname to the file, using syntax that your operating system understands. The pathname must be enclosed in single or double quotation marks.

See “Rules for Words and Names in the SAS Language” in SAS Language Reference: Concepts
(data-set-options)  
specifies actions SAS is to take when it reads variables into the DATA step for processing.

**Requirement**  
Data-set-options must appear within parentheses and follow a SAS data set name.

**Tip**  
Dropping, keeping, and renaming variables is often useful when you update a data set. Renaming like-named variables prevents the second value that is read from over-writing the first one. By renaming one variable, you make the values of both of them available for processing, such as comparing.

**See**  
A list of data set options to use with input data sets in *SAS Data Set Options: Reference*  

**Example**  
“Example 2: Updating By Renaming Variables” on page 312

---

transaction-data-set  
specifies the SAS data set that contains the changes to be applied to the master data set.

**Range**  
The name can be a one-level name (for example, HEALTH), a two-level name (for example, IN.HEALTH), or one of the special SAS data set names.

**Tip**  
Instead of using a data set name, you can specify the physical pathname to the file, using syntax that your operating system understands. The pathname must be enclosed in single or double quotation marks.

---

END=variable  
creates and names a temporary variable that contains an end-of-file indicator. This variable is initialized to 0 and is set to 1 when UPDATE processes the last observation. This variable is not added to any data set.

**UPDATEMODE=MISSINGCHECK**  
**UPDATEMODE=NOMISSINGCHECK**  
specifies whether missing variable values in a transaction data set are to be allowed to replace existing variable values in a master data set.

**MISSINGCHECK**  
prevents missing variable values in a transaction data set from replacing values in a master data set.

**NOMISSINGCHECK**  
allows missing variable values in a transaction data set to replace values in a master data set.

**Default**  
MISSINGCHECK

**Tip**  
Special missing values, however, are the exception and replace values in the master data set even when MISSINGCHECK (the default) is in effect.
Details

Requirements

- The UPDATE statement must be accompanied by a BY statement that specifies the variables by which observations are matched.
- The BY statement should immediately follow the UPDATE statement to which it applies.
- The data sets listed in the UPDATE statement must be sorted by the values of the variables listed in the BY statement, or they must have an appropriate index.
- Each observation in the master data set should have a unique value of the BY variable or BY variables. If there are multiple values for the BY variable, only the first observation with that value is updated. The transaction data set can contain more than one observation with the same BY value. (Multiple transaction observations are all applied to the master observation before it is written to the output file.)

For more information, see "How to Prepare Your Data Sets" in SAS Language Reference: Concepts.

Transaction Data Sets

Usually, the master data set and the transaction data set contain the same variables. However, to reduce processing time, you can create a transaction data set that contains only those variables that are being updated. The transaction data set can also contain new variables to be added to the output data set.

The output data set contains one observation for each observation in the master data set. If any transaction observations do not match master observations, they become new observations in the output data set. Observations that are not to be updated can be omitted from the transaction data set. See “BY-Group Processing in the DATA Step” in SAS Language Reference: Concepts.

Missing Values

By default the UPDATEMODE=MISSINGCHECK option is in effect, so missing values in the transaction data set do not replace existing values in the master data set. Therefore, if you want to update some but not all variables and if the variables that you want to update differ from one observation to the next, set to missing those variables that are not changing. If you want missing values in the transaction data set to replace existing values in the master data set, use UPDATEMODE=NOMISSINGCHECK.

Even when UPDATEMODE=MISSINGCHECK is in effect, you can replace existing values with missing values by using special missing value characters in the transaction data set. To create the transaction data set, use the MISSING statement in the DATA step. If you define one of the special missing values A through Z for the transaction data set, SAS updates numeric variables in the master data set to that value.

If you want the resulting value in the master data set to be a regular missing value, use a single underscore (_) to represent missing values in the transaction data set. The resulting value in the master data set is a period (.) for missing numeric values and a blank for missing character values.

For more information about defining and using special missing value characters, see “MISSING Statement” on page 212.
Comparisons

- Both UPDATE and MERGE can update observations in a SAS data set.
- MERGE automatically replaces existing values in the first data set with missing values in the second data set. UPDATE, however, does not do so by default. To cause UPDATE to overwrite existing values in the master data set with missing ones in the transaction data set, you must use UPDATEMODE=NOMISSINGCHECK.
- UPDATE changes or updates the values of selected observations in a master file by applying transactions. UPDATE can also add new observations.

Examples:

Example 1: Basic Updating

These program statements create a new data set (OHIO.QTR1) by applying transactions to a master data set (OHIO.JAN). The BY variable STORE must appear in both OHIO.JAN and OHIO.WEEK4, and its values in the master data set should be unique:

```sas
data ohio.qtr1;
  update ohio.jan ohio.week4;
  by store;
run;
```

Example 2: Updating By Renaming Variables

This example shows renaming a variable in the FITNESS data set so that it does not overwrite the value of the same variable in the program data vector. Also, the WEIGHT variable is renamed in each data set and a new WEIGHT variable is calculated. The master data set and the transaction data set are listed before the code that performs the update:

Master Data Set

```
Master Data Set

<table>
<thead>
<tr>
<th>OBS</th>
<th>ID</th>
<th>NAME</th>
<th>TEAM</th>
<th>WEIGHT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1114</td>
<td>sally</td>
<td>blue</td>
<td>125</td>
</tr>
<tr>
<td>2</td>
<td>1441</td>
<td>sue</td>
<td>green</td>
<td>145</td>
</tr>
<tr>
<td>3</td>
<td>1750</td>
<td>joey</td>
<td>red</td>
<td>189</td>
</tr>
<tr>
<td>4</td>
<td>1994</td>
<td>mark</td>
<td>yellow</td>
<td>165</td>
</tr>
<tr>
<td>5</td>
<td>2304</td>
<td>joe</td>
<td>red</td>
<td>170</td>
</tr>
</tbody>
</table>
```

Transaction Data Set

```
Transaction Data Set

<table>
<thead>
<tr>
<th>OBS</th>
<th>ID</th>
<th>NAME</th>
<th>TEAM</th>
<th>WEIGHT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1114</td>
<td>sally</td>
<td>blue</td>
<td>119</td>
</tr>
<tr>
<td>2</td>
<td>1994</td>
<td>mark</td>
<td>yellow</td>
<td>174</td>
</tr>
<tr>
<td>3</td>
<td>2304</td>
<td>joe</td>
<td>red</td>
<td>170</td>
</tr>
</tbody>
</table>
```

```sas
/************************************************************************/

data health;
  input ID NAME $ TEAM $ WEIGHT;
  length team $ 6;
  cards;
1114 sally blue 125
```
1441 sue green   145
1750 joey red    189
1994 mark yellow 165
2304 joe red     170
;
data fitness;
   input ID NAME $ TEAM $ WEIGHT;
   length team $ 6;
cards;
1114 sally blue 119
1994 mark yellow 174
2304 joe red     170
;
   /* Sort both data sets by ID */
   proc sort data=health;
      by id;
   run;
   proc sort data=fitness;
      by id;
   run;
   /* Update Master with Transaction */
   data health2;
      length STATUS $11;
      update health(rename=(weight=ORIG) in=a)
         fitness(drop=name team in=b);
      by id;
      if a and b then
         do;
            CHANGE=abs(orig - weight);
            if weight<orig then status='loss';
            else if weight>orig then status='gain';
            else status='same';
            end;
            else status='no weigh in';
      run;
   proc print data=health2;
      title 'Weekly Weigh-in Report';
   run;
Example 3: Updating with Missing Values

This example illustrates the DATA steps used to create a master data set PAYROLL and a transaction data set INCREASE that contains regular and special missing values. Note the following after the update is made:

- The salary for ID 1026 remains the same.
- The salary for ID 1034 is a special missing value.
- The salary for ID 1057 is a regular missing value.

/* Create the Master Data Set */
data payroll;
  input ID SALARY;
datalines;
1011 245
1026 269
1028 374
1034 333
1057 582
;
/* Create the Transaction Data Set */
data increase;
  input ID SALARY;
  missing A _;
datalines;
1011 376
1026 .
1028 374
1034 A
1057 _
;
/* Update Master with Transaction */
data newpay;
  update payroll increase;
  by id;
run;
proc print data=newpay;
  title 'Updating with Missing Values';
run;
WHERE Statement

Selects observations from SAS data sets that meet a particular condition.

Valid in: DATA step or PROC step

Categories: Action

CAS

Type: Declarative

Note: Using a random number function in a WHERE statement might generate a different result set from using a random number function in a subsetting IF statement. This difference can be caused by how the criteria are optimized internally by SAS and is expected behavior.

Tip: The SAS Tutorial video Filtering a SAS Table in a DATA Step shows you how to filter data using the WHERE statement.
Syntax

**WHERE** where-expression-1
< logical-operator where-expression-n>;

Arguments

*where-expression*  
is an arithmetic or logical expression that generally consists of a sequence of operands and operators.

Tips  
The operands and operators described in the next several sections are also valid for the `WHERE=` data set option.

You can specify multiple *where-expressions*.

*logical-operator*  
can be AND, AND NOT, OR, or OR NOT.

Details

The Basics

Using the WHERE statement might improve the efficiency of your SAS programs because SAS is not required to read all observations from the input data set.

The WHERE statement cannot be executed conditionally. That is, you cannot use it as part of an IF-THEN statement.

WHERE statements can contain multiple WHERE expressions that are joined by logical operators.

Note: Using indexed SAS data sets can significantly improve performance when you use WHERE expressions to access a subset of the observations in a SAS data set. See “Understanding SAS Indexes” in [SAS Language Reference: Concepts](#) for a complete discussion of WHERE-expression processing with indexed data sets and a list of guidelines to consider before you index your SAS data sets.

In DATA Steps

The WHERE statement applies to all data sets in the preceding SET, MERGE, MODIFY, or UPDATE statement, and variables that are used in the WHERE statement must appear in all of those data sets. You cannot use the WHERE statement with the POINT= option in the SET and MODIFY statements.

You can apply OBS= and FIRSTOBS= processing to WHERE processing. For more information, see “Processing a Segment of Data That Is Conditionally Selected” in [SAS Language Reference: Concepts](#).

You cannot use the WHERE statement to select records from an external file that contains raw data, nor can you use the WHERE statement within the same DATA step in which you read in-stream data with a DATALINES statement.

For each iteration of the DATA step, the first operation SAS performs in each execution of a SET, MERGE, MODIFY, or UPDATE statement is to determine
whether the observation in the input data set meets the condition of the WHERE statement. The WHERE statement takes effect immediately after the input data set options are applied and before any other statement in the DATA step is executed. If a DATA step combines observations using a WHERE statement with a MERGE, MODIFY, or UPDATE statement, SAS selects observations from each input data set before it combines them.

WHERE and BY in a DATA Step
If a DATA step contains both a WHERE statement and a BY statement, the WHERE statement executes before BY groups are created. Therefore, BY groups reflect groups of observations in the subset of observations that are selected by the WHERE statement, not the actual BY groups of observations in the original input data set.

For a complete discussion of BY-group processing, see “BY-Group Processing in the DATA Step” in SAS Language Reference: Concepts.

In PROC Steps
You can use the WHERE statement with any SAS procedure that reads a SAS data set. The WHERE statement is useful in order to subset the original data set for processing by the procedure. The Base SAS Procedures Guide documents the action of the WHERE statement only in those procedures for which you can specify more than one data set. In all other cases, the WHERE statement performs as documented here.

Use of Indexes
A DATA or PROC step attempts to use an available index to optimize the selection of data when an indexed variable is used in combination with one of these operators and functions:
- the BETWEEN-AND operator
- the comparison operators, with or without the colon modifier
- the CONTAINS operator
- the IS NULL and IS NOT NULL operators
- the LIKE operator
- the TRIM function
- the SUBSTR function, in some cases

SUBSTR requires these arguments:
where substr(variable,position,length) = 'character-string';

An index is used in processing when the arguments of the SUBSTR function meet all of these conditions:
- position is equal to 1
- length is less than or equal to the length of variable
- length is equal to the length of character-string
Operands Used in WHERE Expressions

Operands in WHERE expressions can contain these values:

- constants
- time and date values
- values of variables that are obtained from the SAS data sets
- values created within the WHERE expression itself

You cannot use variables that are created within the DATA step (for example, FIRST._variable_, LAST._variable_, _N_, or variables that are created in assignment statements) in a WHERE expression because the WHERE statement is executed before SAS brings observations into the DATA or PROC step. When WHERE expressions contain comparisons, the unformatted values of variables are compared.

Here are some examples of using operands in WHERE expressions:

- where score>50;
- where date>='01jan1999'd and time>='9:00't;
- where state='Mississippi';

As in other SAS expressions, the names of numeric variables can stand alone. SAS treats values of 0 or missing as false; other values are true. These examples are WHERE expressions that contain the numeric variables EMPNUM and SSN:

- where empnum;
- where empnum and ssn;

Character literals or the names of character variables can also stand alone in WHERE expressions. If you use the name of a character variable by itself as a WHERE expression, SAS selects observations where the value of the character variable is not blank.

Operators Used in the WHERE Expression

You can include both SAS operators and special WHERE-expression operators in the WHERE statement. For a complete list of the operators, see Table 2.8 on page 318. For the rules that SAS follows when it evaluates WHERE expressions, see “WHERE-Expression Processing” in SAS Language Reference: Concepts.

This table lists the operators for the WHERE statement.

<table>
<thead>
<tr>
<th>Operator Type</th>
<th>Symbol or Mnemonic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td></td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>multiplication</td>
<td></td>
</tr>
<tr>
<td>/</td>
<td>division</td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>addition</td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>subtraction</td>
<td></td>
</tr>
<tr>
<td>Operator Type</td>
<td>Symbol or Mnemonic</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>-------------------</td>
<td>----------------------</td>
</tr>
<tr>
<td>**</td>
<td></td>
<td>exponentiation</td>
</tr>
<tr>
<td>Comparison</td>
<td></td>
<td></td>
</tr>
<tr>
<td>= or EQ</td>
<td></td>
<td>equal to</td>
</tr>
<tr>
<td>^=, -=, ~=, or NE¹</td>
<td></td>
<td>not equal to</td>
</tr>
<tr>
<td>&gt; or GT</td>
<td></td>
<td>greater than</td>
</tr>
<tr>
<td>&lt; or LT</td>
<td></td>
<td>less than</td>
</tr>
<tr>
<td>&gt;= or GE</td>
<td></td>
<td>greater than or equal to</td>
</tr>
<tr>
<td>&lt;= or LE</td>
<td></td>
<td>less than or equal to</td>
</tr>
<tr>
<td>IN</td>
<td></td>
<td>equal to one of a list</td>
</tr>
<tr>
<td>Logical (Boolean)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&amp; or AND</td>
<td></td>
<td>logical and</td>
</tr>
<tr>
<td></td>
<td>or OR²</td>
<td></td>
</tr>
<tr>
<td>¬, ^, ¬, or NOT¹</td>
<td></td>
<td>logical not</td>
</tr>
<tr>
<td>Other</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>¹</td>
</tr>
<tr>
<td>(</td>
<td></td>
<td>indicate order of evaluation</td>
</tr>
<tr>
<td>+ prefix</td>
<td></td>
<td>positive number</td>
</tr>
<tr>
<td>- prefix</td>
<td></td>
<td>negative number</td>
</tr>
</tbody>
</table>

**WHERE Expression Only**

- BETWEEN–AND: an inclusive range
- ? or CONTAINS: a character string
- IS NULL or IS MISSING: missing values
- LIKE: match patterns
<table>
<thead>
<tr>
<th>Operator Type</th>
<th>Symbol or Mnemonic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>=*</td>
<td>sounds-like</td>
<td></td>
</tr>
<tr>
<td>SAME-AND</td>
<td></td>
<td>add clauses to an existing WHERE statement without retyping original one</td>
</tr>
</tbody>
</table>

1. The caret (^), tilde (~), and the not sign (~) all indicate a logical not. Use the character available on your keyboard, or use the mnemonic equivalent.
2. The OR symbol ( | ), broken vertical bar ( | ), and exclamation point (!) all indicate a logical or. Use the character available on your keyboard, or use the mnemonic equivalent.
3. Two OR symbols (| | ), two broken vertical bars ( | | ), or two exclamation points (!! ) indicate concatenation. Use the character available on your keyboard.
4. You can use the colon modifier (:) with any of the comparison operators in order to compare only a specified prefix of a character string.

Comparisons

- You can use the WHERE command in SAS/FSP software to subset data for editing and browsing. You can use both the WHERE statement and WHERE= data set option in windowing procedures and in conjunction with the WHERE command.

- To select observations from individual data sets when a SET, MERGE, MODIFY, or UPDATE statement specifies more than one data set, apply a WHERE= data set option to each data set. In the DATA step, if a WHERE statement and a WHERE= data set option apply to the same data set, SAS uses the data set option and ignores the statement for that data set. Other data sets without a WHERE data set option use the statement.

- The most important differences between the WHERE statement in the DATA step and the subsetting IF statement are as follows:
  - The WHERE statement selects observations before they are brought into the program data vector, making it a more efficient programming technique. The subsetting IF statement works on observations after they are read into the program data vector.
  - The WHERE statement can produce a different data set from the subsetting IF when a BY statement accompanies a SET, MERGE, or UPDATE statement. The different data set occurs because SAS creates BY groups before the subsetting IF statement selects but after the WHERE statement selects.
  - The WHERE statement cannot be executed conditionally as part of an IF statement, but the subsetting IF statement can.
  - The WHERE statement selects observations in SAS data sets only, whereas the subsetting IF statement selects observations from an existing SAS data set or from observations that are created with an INPUT statement.
  - The subsetting IF statement cannot be used in SAS windowing procedures to subset observations for browsing or editing.

- Do not confuse the WHERE statement with the DROP or KEEP statement. The DROP and KEEP statements select variables for processing. The WHERE statement selects observations.
Examples:

Example 1: Specify the WHERE Statement in a SAS DATA Step

This DATA step produces a SAS data set that contains only observations from data set `customer` in which the value for `name` begins with `Mac` and the value for `city` is Charleston or Atlanta.

```sas
data testmacs;
set customer;
where substr(name,1,3)='Mac' and (city='Charleston' or city='Atlanta');
run;
```

Example 2: Specify the WHERE Statement in a CAS DATA Step

This example produces a CAS table that contains only the observations from the `Sashelp.Class` data set in which the values for `Age` are greater than 14.

```sas
cas casauto sessopts=(caslib='casuser'); /*1*/
caslib _all_ assign;
libname mycas cas;

data mycas.class; /*2*/
set sashelp.class;
run;

data mycas.class_out;
set mycas.class;
where Age>14; /*3*/
run;
```

1 Connect to CAS. To run the DATA step in CAS, you must first connect to a CAS server and start a CAS session. See “Set Up Code for Examples” in SAS Cloud Analytic Services: DATA Step Programming for information.

2 Load some sample data from the Sashelp library to CAS. Specify a CAS engine libref on the output table. ¹

3 Run the DATA step in CAS. Specify the WHERE statement to filter the loaded CAS table `mycas.class`. The DATA step writes the filtered output from `mycas.class` to the output table `mycas.class_out`.

See “Running the DATA Step in CAS” in SAS Cloud Analytic Services: DATA Step Programming for more information about the DATA step and SAS Cloud Analytic Services.

Example 3: Using Operators Available Only in the WHERE Statement

- Using BETWEEN-AND:
  ```sas
  where empnum between 500 and 1000;
  ```

- Using CONTAINS:

¹ When a DATA step is used to load a SAS data set to CAS, the DATA step does not actually execute in the CAS server. For the DATA step to run in CAS, both the input and output tables must exist as in-memory CAS tables and the input and output tables must both contain a CAS engine libref.
where company ? 'bay';
where company contains 'bay';

- Using IS NULL and IS MISSING:
  where name is null;
  where name is missing;

- Using LIKE to select all names that start with the letter D:
  where name like 'D%';

- Using LIKE to match patterns from a list of these names:
  Diana
  Diane
  Dianna
  Dianthus
  Dyan

<table>
<thead>
<tr>
<th>WHERE Statement</th>
<th>Name Selected</th>
</tr>
</thead>
<tbody>
<tr>
<td>where name like 'D_an';</td>
<td>Dyan</td>
</tr>
<tr>
<td>where name like 'D_an_';</td>
<td>Diana, Diane</td>
</tr>
<tr>
<td>where name like 'D_an__';</td>
<td>Dianna</td>
</tr>
<tr>
<td>where name like 'D_an%';</td>
<td>all names from list</td>
</tr>
</tbody>
</table>

- Using the Sounds-like Operator to select names that sound like “Smith”:
  where lastname=*'Smith';

- Using SAME-AND:
  where year>1991;
  ...more SAS statements...
  where same and year<1999;

  In this example, the second WHERE statement is equivalent to the following WHERE statement:
  where year>1991 and year<1999;

See Also

- Base SAS Procedures Guide
- “BY-Group Processing in the DATA Step” in SAS Language Reference: Concepts
- SAS SQL Query Window User’s Guide
- SAS/IML User’s Guide
- “Understanding SAS Indexes” in SAS Language Reference: Concepts
- “WHERE-Expression Processing” in SAS Language Reference: Concepts
WINDOW Statement

Creates customized windows for your applications.

Valid in: DATA step
Category: Window Display
Type: Declarative
Restriction: This statement is not supported in a DATA step that runs in CAS.

Syntax

WINDOW window <window-options> field-definition(s);
WINDOW window <window-options> group-definition(s);

Arguments

window
specifies the window name.

Restriction Window names must conform to SAS naming conventions.

window-options
specifies characteristics of the window as a whole. Specify these window-options before any field or GROUP= specifications:

COLOR=color
specifies the color of the window background for operating environments that have this capability. In other operating environments, this option affects the color of the window border. These colors are available:

BLACK  MAGENTA
BLUE    ORANGE
BROWN   PINK
CYAN    RED
GRAY    WHITE
GREEN   YELLOW

Default If you do not specify a color with the COLOR= option, the window's background color is device-dependent instead of black, and the color of a field is device-dependent instead of white.

Tip The representation of colors might vary, depending on the monitor being used. COLOR= has no effect on monochrome monitors.
**COLUMNS=columns**
specifies the number of columns in the window.

Default  The window fills all remaining columns on the monitor; the number of columns that are available depends on the type of monitor that is being used.

**ICOLUMN=column**
specifies the initial column within the monitor at which the window is displayed.

Default  SAS displays the window at column 1.

**IROW=row**
specifies the initial row (or line) within the monitor at which the window is displayed.

Default  SAS displays the window at row 1.

**KEYS=<<libref.>catalog.>keys-entry**
specifies the name of a KEYS entry that contains the function key definitions for the window.

Default  SAS uses the current function key settings that are defined in the KEYS window.

Tips  If you specify only an entry name, SAS looks in the SASUSER.PROFILE catalog for a KEYS entry of the name that is specified. You can also specify the three-level name of a KEYS entry, in the form

```
libref.catalog.keys-entry
```

To create a set of function key definitions for a window, use the KEYS window. Define the keys as you want, and use the SAVE command to save the definitions in the SASUSER.PROFILE catalog or in a SAS library and catalog that you specify.

**MENU=<<libref.>catalog.>pmenu-entry**
specifies the name of a menu (pmenu) you have built with the PMENU procedure.

Tip  If you specify only an entry name, SAS looks in the SASUSER.PROFILE catalog for a PMENU entry of the name specified. You can also specify the three-level name of a PMENU entry in the form

```
libref.catalog.pmenu-entry
```

**ROWS=rows**
specifies the number of rows (or lines) in the window.

Default  The window fills all remaining rows on the monitor.

Tip  The number of rows that are available depends on the type of monitor that is being used.

**field-definition(s)**
specifies and describes a variable or character string to be displayed in a window or within a group of related fields.
Tips  A window or group can contain any number of fields, and you can define the same field in several groups or windows.

You can specify multiple field-definitions.

See  “Field Definitions” on page 326

group-definition(s)  specifies a group and defines all fields within a group. A group definition consists of two parts: the GROUP= option and one or more field definitions.

GROUP=group  specifies a group of related fields.

Default  A window contains one unnamed group of fields.

Restriction  group must be a SAS name.

Tips  When you refer to a group in a DISPLAY statement, write the name as window.group.

A group contains all fields in a window that you want to display at the same time. Display various groups of fields within the same window at different times by naming each group. Choose the group to appear by specifying window.group in the DISPLAY statement.

Specifying several groups within a window prevents repetition of window options that do not change and helps you keep track of related displays. For example, if you are defining a window to check data values, arrange the display of variables and messages for most data values in the data set in a group that is named STANDARD. Arrange the display of different messages in a group that is named CHECKIT that appears when data values meet the conditions that you want to check.

Details

The Basics

Operating Environment Information:  The WINDOW statement has some functionality that might be specific to your operating environment. For more information, see the SAS documentation for your operating environment.

You can use the WINDOW statement in the SAS windowing environment, in interactive line mode, or in noninteractive mode to create customized windows for your applications. 1 Windows that you create can display text and accept input; they have command and message lines. The window name appears at the top of the window. Use commands and function keys with windows that you create. A window definition remains in effect only for the DATA step that contains the WINDOW statement.

Define a window before you display it. Use the DISPLAY statement to display windows that are created with the WINDOW statement. For more information, see “DISPLAY Statement” on page 60.

---

1. You cannot use the WINDOW statement in batch mode because no computer is connected to a batch executing process.
Field Definitions

Use a field definition to identify a variable or a character string to be displayed, its position, and its attributes. Enclose character strings in quotation marks. The position of an item is its beginning row (or line) and column. Attributes include color, whether you can enter a value into the field, and characteristics such as highlighting.

You can define a field to contain a variable value or a character string, but not both. The form of a field definition for a variable value is

<row column> variable <format> options

The form for a character string is

<row column> 'character-string' options

The elements of a field definition are described here.

row column
specifies the position of the variable or character string.

SAS keeps track of its position in the window with a pointer. For example, when you tell SAS to write a variable's value in the third column of the second row of a window, the pointer moves to row 2, column 3 to write the value. Use the pointer controls that are listed here to move the pointer to the appropriate position for a field.

In a field definition, row can be one of these row pointer controls:

#n
specifies row n within the window.
Range n must be a positive integer.

#numeric-variable
specifies the row within the window that is given by the value of numeric-variable.
Restriction #numeric-variable must be a positive integer. If the value is not an integer, the decimal portion is truncated and only the integer is used.

#(expression)
specifies the row within the window that is given by the value of expression.
Restrictions expression can contain array references and must evaluate to a positive integer.
Enclose expression in parentheses.

/ moves the pointer to column 1 of the next row.

In a field definition, column can be one of these column pointer controls:

@n
specifies column n within the window.
Restriction n must be a positive integer.

@numeric-variable
specifies the column within the window that is given by the value of numeric-variable.
Restriction  numeric-variable must be a positive integer. If the value is not an integer, the decimal portion is truncated and only the integer is used.

@ (expression) specifies the column within the window that is given by the value of expression.

Restrictions  expression can contain array references and must evaluate to a positive integer.

Enclose expression in parentheses.

+n moves the pointer \( n \) columns.

Range  \( n \) must be a positive integer.

+n numeric-variable moves the pointer the number of columns that is given by the numeric-variable.

Restriction  + numeric-variable must be a positive or negative integer. If the value is not an integer, the decimal portion is truncated and only the integer is used.

Default  If you omit row in the first field of a window or group, SAS uses the first row of the window. If you omit row in a later field specification, SAS continues on the row that contains the previous field. If you omit column, SAS uses column 1 (the left border of the window).

Tip  Although you can specify either row or column first, the examples in this documentation show the row first.

variable specifies a variable to be displayed or to be assigned the value that you enter at that position when the window is displayed.

Tips  variable can be the name of a variable or of an array reference.

To allow a variable value in a field to be displayed but not changed by the user, use the PROTECT= option (described later in this section). You can also protect an entire window or group for the current execution of the DISPLAY statement by specifying the NOINPUT option in the DISPLAY statement.

If a field definition contains the name of a new variable, that variable is added to the data set that is being created (unless you use a KEEP or DROP specification).

format gives the format for the variable.

Default  If you omit format, SAS uses an informat and format that are specified elsewhere (for example, in an ATTRIB, INFORMAT, or FORMAT statement or permanently stored with the data set) or a SAS default informat and format.
Tips

If a field displays a variable that cannot be changed (that is, you use the PROTECT=YES option), format can be any SAS format or a format that you define with the FORMAT procedure.

If a field can both display a variable and accept input, you must either specify the informat in an INFORMAT or ATTRIB statement or use a SAS format such as $CHAR. or TIME. that has a corresponding informat.

If a format is specified, the corresponding informat is assigned automatically to fields that can accept input.

A format and an informat in a WINDOW statement override an informat and a format that are specified elsewhere.

'character-string'
contains the text of a character string to be displayed.

Restrictions
The character string must be enclosed in quotation marks.

You cannot enter a value in a field that contains a character string.

options
Specify field definition attributes:

ATTR=highlighting-attribute
controls these highlighting attributes of the field:

BLINK causes the field to blink.
HIGHLIGHT displays the field at high intensity.
REV_VIDEO displays the field in reverse video.
UNDERLINE underlines the field.

Alias A=

Tips
To specify more than one highlighting attribute, use the form
ATTR=(highlighting-attribute-1,...)

The highlighting attributes that are available depend on the type of monitor that you use.

AUTOSKIP=YES | NO
controls whether the cursor moves to the next unprotected field of the current window or group when you have entered data in all positions of a field.

YES specifies that the cursor moves automatically to the next unprotected field.

NO specifies that the cursor does not move automatically.

Alias AUTO=

Default NO

COLOR=color
specifies a color for the variable or character string. You can specify one of these colors:
BLACK  MAGENTA
BLUE  ORANGE
BROWN  PINK
CYAN  RED
GRAY  WHITE
GREEN  YELLOW

Alias  C=

Default  WHITE

Tips  The representation of colors might vary, depending on the monitor that you use.

COLOR= has no effect on monochrome monitors.

DISPLAY=YES | NO
controls whether the contents of a field are displayed.

YES  specifies that SAS displays characters in a field as you enter them in.

NO  specifies that the entered characters are not displayed.

Default  YES

PERSIST=YES | NO
controls whether a field is displayed by all executions of a DISPLAY statement in the same iteration of the DATA step until the DISPLAY statement contains the BLANK option.

YES  specifies that each execution of the DISPLAY statement displays all previously displayed contents of the field as well as the contents that are scheduled for display by the current DISPLAY statement. If the new contents overlap persisting contents, the persisting contents are no longer displayed.

NO  specifies that each execution of a DISPLAY statement displays only the current contents of the field.

Default  NO

Tip  PERSIST= is most useful when the position of a field changes in each execution of a DISPLAY statement.

Example  “Example 3: Persisting and Nonpersisting Fields” on page 333

PROTECT=YES | NO
controls whether information can be entered into a field.

YES  specifies that you cannot enter information.

NO  specifies that you can enter information.

Alias  P=

Default  No
Tip Use PROTECT= only for fields that contain variables; fields that contain text are automatically protected.

**REQUIRED=YES | NO**
controls whether a field can be left blank.

- **NO** specifies that you can leave the field blank.
- **YES** specifies that you must enter a value in the field.

Default NO

Tip If you try to leave a field blank that was defined with REQUIRED=YES, SAS does not allow you to enter values in any subsequent fields in the window.

### Automatic Variables

The WINDOW statement creates two automatic SAS variables: _CMD_ and _MSG_.

- **_CMD_**
  - contains the last command from the window's command line that was not recognized by the window.
  
  Tip _CMD_ is a character variable with a length of 80 bytes; its value is set to ' ' (blank) before each execution of a DISPLAY statement.

  Example “Example 4: Sending a Message” on page 333

- **_MSG_**
  - contains a message that you specify to be displayed in the message area of the window.
  
  Tip _MSG_ is a character variable with a length of 80 bytes; its value is set to ' ' (blank) after each execution of a DISPLAY statement.

  Example “Example 4: Sending a Message” on page 333

### Displaying Windows

The DISPLAY statement enables you to display windows. Once you display a window, the window remains visible until you display another window over it or until the end of the DATA step. When you display a window that contains fields into which you can enter values, either enter a value or press Enter at each unprotected field to cause SAS to proceed to the next display. While a window is being displayed, you can use commands and function keys to view other windows, change the size of the current window, and so on. The execution proceeds to the next display only after you have pressed Enter in all unprotected fields.

A DATA step that contains a DISPLAY statement continues execution until one of these events occurs:

- the last observation that is read by a SET, MERGE, MODIFY, UPDATE, or INPUT statement has been processed
- a STOP or ABORT statement is executed
- an END command executes.
Comparisons

- The WINDOW statement creates a window, and the DISPLAY statement displays it.
- The %WINDOW and %DISPLAY statements in the macro language create and display windows that are controlled by the macro facility.

Examples:

Example 1: Creating a Single Window

This DATA step creates a window with a single group of fields:

```sas
data _null_;  
  window start  
    #9 @26 'WELCOME TO THE SAS SYSTEM'  
    color=black  
    #12 @19 'THIS PROGRAM CREATES'  
    #12 @40 'TWO SAS DATA SETS'  
    #14 @26 'AND USES THREE PROCEDURES'  
    #18 @27 'Press ENTER to continue';  
  display start;  
  stop;  
run;
```

The START window fills the entire monitor. The first line of text is black. The other three lines are the default for your operating environment. The text begins in the column that you specified in your program. The START window does not require you to enter any values. However, to exit the window take one of these actions:

- Press Enter to cause DATA step execution to proceed to the STOP statement.
- Issue the END command.

If you omit the STOP statement from this program, the DATA step executes endlessly until you execute END from the window, either with a function key or from the command line. (Because this DATA step does not read any observations, SAS cannot detect an end-of-file to end DATA step execution.)
Example 2: Displaying Two Windows Simultaneously

These statements assign news articles to reporters. The list of article topics is stored as variable art in SAS data set category.article. This application enables you to assign each topic to a writer and to view the accumulating assignments. The program creates a new SAS data set named Assignment.

```
libname category 'SAS-library';
data Assignment;
set category.article end=final;
drop a b j s o;
window Assignment irow=1 rows=12 color=white
   #3 @10 'Article:' +1 art protect=yes
   'Name:' +1 name $14.;
window Showtotal irow=20 rows=12 color=white
   group=subtotal
   #1 @10 'Adams has' +1 a
   #2 @10 'Brown has' +1 b
   #3 @10 'Johnson has' +1 j
   #4 @10 'Smith has' +1 s
   #5 @10 'Other has' +1 o
   group=lastmessage
   #8 @10
   'ALL ARTICLES ASSIGNED.'
   Press ENTER to stop processing.';
display Assignment blank;
if name='Adams' then a+1;
else if name='Brown' then b+1;
else if name='Johnson' then j+1;
else if name='Smith' then s+1;
else o+1;
display Showtotal.subtotal blank noinput;
if final then display Showtotal.lastmessage;
run;
```

When you execute the DATA step, these windows appear.

In the Assignment window (located at the top of the monitor), you see the name of the article and a field into which you enter a reporter’s name. After you enter a name and press Enter, SAS displays the Showtotal window (located at the bottom of the monitor) that shows the number of articles that are assigned to each reporter.
(including the assignment that you just made). As you continue to make assignments, the values in the Showtotal window are updated. During the last iteration of the DATA step, SAS displays the message that all articles are assigned, and instructs you to press Enter to stop processing.

Example 3: Persisting and Nonpersisting Fields

This example demonstrates the PERSIST= option. You move from one window to the other by positioning the cursor in the current window and pressing Enter.

```sas
data _null_;  
array row{3} r1-r3;  
array col{3} c1-c3;  
input row{*} col{*};  
window One  
  rows=20 columns=36  
  #1 @14 'PERSIST=YES' color=black  
  #{row[i]} @(col[i]) 'Hello'  
  color=black persist=yes;  
window Two  
  icolumn=43 rows=20 columns=36  
  #1 @14 'PERSIST=NO' color=black  
  #{row[i]} @(col[i]) 'Hello'  
  color=black persist=no;  
  do i=1 to 3;  
    display One;  
    display Two;  
  end;  
datalines;  
5 10 15 5 10 15 ;
```

These windows show the results of this DATA step after its third iteration.

Note that window One shows Hello in all three positions in which it was displayed. Window Two shows only the third and final position in which Hello was displayed.

Example 4: Sending a Message

This example uses the _CMD_ and _MSG_ automatic variables to send a message when you execute an erroneous windowing command in a window that is defined with the WINDOW statement:

```sas
if _cmd_ ne ' ' then  
  _msg_='CAUTION: UNRECOGNIZED COMMAND' || _cmd_;
When you enter a command that contains an error, SAS sets the value of _CMD_ to the text of the erroneous command. Because the value of _CMD_ is no longer blank, the IF statement is true. The THEN statement assigns to _MSG_ the value that is created by concatenating CAUTION: UNRECOGNIZED COMMAND and the value of _CMD_ (up to a total of 80 bytes). The next time a DISPLAY statement displays that window, the message line of the window displays this message:

CAUTION: UNRECOGNIZED COMMAND

*Command* is the erroneous windowing command.

Example 5: Creating a SAS Data Set

These statements create a SAS data set by using input from the WINDOW statement.

```sas
data new;
  length name $20;
  window start
    #3 @20 'Type the variable name'
    #4 @20 'and press the Enter key.'
    #7 'Name:' +1 name attr=underline
    #11 'When you are finished entering variable names, type "end"'
    #12 'at the command line.';
  display start;
run;
proc print;
run;
```

![Start window](image)

See Also

- “How Many Characters Can I Use When I Measure SAS Name Lengths in Bytes?” in *SAS Language Reference: Concepts*
- “PMENU Procedure” in *Base SAS Procedures Guide*

Statements:

- “DISPLAY Statement” on page 60
X Statement

Issues an operating-environment command from within a SAS session.

Note: The X Statement has moved to SAS Global Statements.
SAS_FTP_AUTHTLS Environment Variable

Enables Transport Layer Security (TLS) authentication.

Note: The SAS_FTP_AUTHTLS Environment Variable has moved to SAS Global Statements.