An Introduction to the SAS System

Phil Spector

Statistical Computing Facility
Department of Statistics
University of California, Berkeley

What is SAS?

- Developed in the early 1970s at North Carolina State University
- Originally intended for management and analysis of agricultural field experiments
- Now the most widely used statistical software
- Used to stand for "Statistical Analysis System", now it is *not* an acronym for anything
- Pronounced "sass", not spelled out as three letters.

Overview of SAS Products

- Base SAS - data management and basic procedures
- SAS/STAT - statistical analysis
- SAS/GRAPH - presentation quality graphics
- SAS/OR - Operations research
- SAS/ETS - Econometrics and Time Series Analysis
- SAS/IML - interactive matrix language
- SAS/AF - applications facility (menus and interfaces)
- SAS/QC - quality control

There are other specialized products for spreadsheets, access to databases, connectivity between different machines running SAS, etc.

Resources: Introductory Books

Mastering the SAS System, 2nd Edition, by Jay A. Jaffe, Van Nostrand Reinhold

Quick Start to Data Analysis with SAS, by Frank C. DiIorio and Kenneth A. Hardy, Duxbury Press.

How SAS works: a comprehensive introduction to the SAS System, by P.A. Herzberg, Springer-Verlag

Applied statistics and the SAS programming language, by R.P. Cody, North-Holland, New York

SAS Institute produces several introductory guides; most are at a very elementary level. Among them are:

Introducing the SAS System, Version 6, First Edition
SAS Language and Procedures: Introduction, Version 6, First Edition
SAS Introductory Guide for Personal Computers, Release 6.03
SAS Applications Programming: A Gentle Introduction
Introduction to Market Research Using the SAS System
SAS/GRAPH Software: Introduction, Version 6, First Edition

Resources: Other SAS Institute Publications

Basic Manuals (should be available for most serious users):
SAS Language: Reference, Version 6, 1st Edition
SAS Language and Procedures: Usage, Version 6, 1st Edition
SAS Language and Procedures: Usage 2, Version 6, 1st Edition
Product Manuals (should be available for reference)
SAS/STAT User's Guide, Version 6, 4th Edition, Volumes 1 & 2
SAS/ETS User's Guide, Version 6, 2nd Edition
SAS/FSP Software: Usage and Reference, Version 6, 1st Edition
SAS/GRAPH Software: Usage, Version 6, 1st Edition
SAS/GRAPH Software: Reference, Version 6, 1st Edition, Volumes 1 & 2
SAS/AF Software: Usage and Reference, Version 6, 1st Edition
SAS/OR User's Guide, Version 6, First Edition
Syntax-only Guides (may be useful for experienced users)
SAS Language and Procedures: Syntax, Version 6, 1st Edition
SAS Screen Control Language: Syntax, Version 6, 1st Edition
SAS/ETS Software: Syntax, Version 6, 1st Edition
SAS/GRAPH Software: Syntax, Version 6, 1st Edition
SAS/STAT Software: Syntax, Version 6, 1st Edition

Resources: Other SAS Institute Publications

Some Other Potentially Useful Publications
SAS Applications Guide, 1987 Edition
SAS Companions for:
MVS, Unix, Windows, OS/2, VSE, AOS/VS and CMS
SAS Guide to the REPORT Procedure: Usage and Reference, Version 6
SAS Guide to TABULATE Processing, 2nd Edition
SAS Guide to Macro Processing, Version 6, 1st Edition
SAS/ETS Software: Applications Guides 1 & 2, Version 6, First Edition

There are many, many more SAS publications, including specialized guides for various subject matter areas, course notes and technical reports. Contact SAS Institute, or visit the web site `http://www.sas.com/service/doc/pubcat`.

Online Resources

Online help: When using the display manager, you can get help on most topics about SAS by typing `help` in the window input area.

Sample Programs: Each distribution of SAS contains several directories with sample programs. SAS Institute Home Page:
`http://www.sas.com`

SAS Institute Technical Support:
`http://www.sas.com/service/techsup/find_answer.html`

Searchable index to SAS-L, the SAS mailing list:
`http://jse.stat.ncsu.edu/othergroups/sasl`

Usenet Newsgroup (equivalent to SAS-L):
`comp.soft-sys.sas`

SAS for the Masses:
`http://faith.hypno.net/sasmass/`

Michael Friendly's Guide to SAS Resources on the Internet:
`http://www.math.yorku.ca/SCS/StatResource.html#SAS`

Basic Structure of SAS

There are two main components to most SAS programs - the data step(s) and the procedure step(s).

The data step reads data from external sources, manipulates and combines it with other data set and prints reports. The data step is used to prepare your data for use by one of the procedures (often called "procs").

SAS is very lenient about the format of its input - statements can be broken up across lines, multiple statements can appear on a single line, and blank spaces and lines can be added to make the program more readable.

The procedure steps perform analysis on the data, and produce (often huge amounts of) output.

The most effective strategy for learning SAS is to concentrate on the details of the data step, and learn the details of each procedure as you have a need for them.

Accessing SAS

There are four ways to access SAS on a UNIX system:

1. Type `sas` . This opens the SAS "display manager", which consists of three windows (program, log, and output). Some procedures must be run from the display manager.
2. Type `sas -nodms` . You will be prompted for each SAS statement, and output will scroll by on the screen.
3. Type `sas -stdio` . SAS will act like a standard UNIX program, expecting input from standard input, sending the log to standard error, and the output to standard output;
4. Type `sas filename.sas` . This is the batch mode of SAS - your program is read from `filename.sas`, the log goes to `filename.log` and the output goes to `filename.lst`.

Some Preliminary Concepts and Rules

- SAS variable names must be 8 characters or less, constructed of letters, digits and the underscore character.
- It's a good idea not to start variable names with an underscore, because special system variables are named that way.
- Data set names follow similar rules as variables, but they have a different name space.
- There are virtually no reserved keywords in SAS; it's very good at figuring things out by context.
- SAS is not case sensitive, except inside of quoted strings.
- Missing values are handled consistently in SAS, and are represented by a period (`.`).
- Each statement in SAS must end in a semicolon (`;`).

Structure of SAS programs

- Lines beginning with an asterisk (`*`) are treated as comments. Alternatively you can enclose comments between `/*` and `*/`.
- You can combine as many data and proc steps in whatever order you want.
- Data steps begin with the word `data` and procedure steps begin with the word `proc`.
- The `run;` command signals to SAS that the previous commands can be executed.
- Terminate an interactive SAS job with the `endsas;` statement.
- There are global options (like linesize and pagesize) as well as options specific to datasets and procedures.
- Informative messages are written to the SAS log - make sure you read it!

The Data Step

The data step provides a wide range of capabilities, among them reading data from external sources, reshaping and manipulating data, transforming data and producing printed reports.

The data step is actually an implied do loop whose statements will be executed for each observation either read from an external source, or accessed from a previously processed data set.

For each iteration, the data step starts with a vector of missing values for all the variables to be placed in the new observation. It then overwrites the missing value for any variables either input or defined by the data step statements. Finally, it outputs the observation to the newly created data set.

The true power of the data step is illustrated by the fact that all of these defaults may be overridden if necessary.

Data Step: Basics

Each data step begins with the word `data` and optionally one or more data set names (and associated options) followed by a semicolon. The name(s) given on the data step are the names of data sets which will be created within the data step.

If you don't include any names on the data step, SAS will create default data set names of the form `data`*n*, where *n* is an integer which starts at 1 and is incremented so that each data set created has a unique name within the current session. Since it becomes difficult to keep track of the default names, it is recommended that you always explicitly specify a data set name on the `data` statement.

When you are running a data step to simply generate a report, and don't need to create a data set, you can use the special data set name `_null_` to eliminate the output of observations.

Data Step: Inputting Data

The `input` statement of SAS is used to read data from an external source, or from lines contained in your SAS program.
The `infile` statement names an external file or fileref* from which to read the data; otherwise the `cards;` or `datalines;` statement is used to precede the data.

```
data one;
infile "input.data";
input a b c;
run;
```

Reading data from an external file

```
data one;
input a b c;
datalines;
    . . .
;
```

Reading from inline data

By default, each invocation of the input statement reads another record. This example uses free-form input, with at least one space between values.

*A fileref is a SAS name, created by the `filename` statement, which refers to an external file or other device

Data Step: `input` Statement

There are three basic forms of the input statement:

1. List input (free form) - data fields must be separated by at least one blank. List the names of the variables, follow the name with a dollar sign (`$`) for character data.
2. Column input - follow the variable name (and `$` for character) with ***`startingcolumn`*** – ***`endingcolumn`***.
3. Formatted input - Optionally precede the variable name with `@`*startingcolumn*; follow the variable name with a SAS format designation. (Examples of formats: `$10.` (10 column character), `6.` (6 column numeric))

When mixing different input styles, note that for column and formatted input, the next input directive reads from the column immediately after the previous value, while for list input, the next directive reads from the second column after the previous value.

Modifiers for the Input Statement

`+`*number*	advance *number* columns.
`#`*number*	advance to line *number*.
`/`	advance to next line.
trailing `@`	hold the line to allow further input statements in this iteration of the data step on the same data.
trailing `@@`	hold the line to allow continued reading from the line on subsequent iterations of the data step.

Note: If SAS needs to read an additional line to input all the variables referenced in the input statement it prints the following message on the log:

```
NOTE: SAS went to a new line when INPUT statement reached past
      the end of a line.
```

If you see this note, make sure you understand why it was printed!!

The `input` Statement

Variable lists can be used on the input statement. For example, the list `var1 - var4` expands to `var1 var2 var3 var4`.

You can repeat formats for variable lists by including the names and formats in parentheses: `(var1 - var4) (5.)` reads four numeric variables from 20 consecutive columns (5 columns for each variable).

You can also repeat formats using the notation *num*format*. The previous example could be replaced with `(4 * 5.)`.

A null input statement (no variables) can be used to free holding caused by trailing `@`-signs.

The `@`, `+` and `#` specifications can all be followed by a variable name instead of a number.

If you want to make sure your input data is really arranged the way you think it is, the `list;` command will display your input data with a "ruler" showing column numbers.

Options for the `infile` statement

For inline data, use the infile name `cards` or `datalines`.

`missover`	Sets values to missing if an input statement would read more than one line.
`stopover`	Like `missover`, but declares an error and stops
`lrecl=`*`num`*	Treats the input as having a logical record length of *`num`*. *Required* if input records are longer than 256 characters.
`dlm='`*`chars`*`'`	Uses the characters in *`chars`* instead of blanks and tabs as separators in list (free-form) input.
`expandtabs`	expand tabs to spaces before inputting data.
`dsd`	Read comma-separated data
`end=`*`varname`*	creates a SAS variable whose value is 1 when SAS processes the last line in the file.
`obs=`*`n`*	Limits processing of infile to *`n`* records

Reading SAS programs from external files

The `infile` statement can be used to read data which is stored in a file separate from your SAS program. When you want SAS to read your *program* from an external file you can use the `%include` statement, followed by a filename or fileref. After SAS processes a `%include` statement, it continues to read data from its original source (input file, keyboard or display manager.)

For example, suppose the SAS program statements to read a file and create a data set are in the system file `readit.sas`. To process those statements, and then print the data set, the following commands can be used:

```
%include "readit.sas";
proc print;
run;
```


Repetitive Processing of Variables

The `array` statement can be used to perform the same task on a group of variables.

`array` *`arrayname variable_list`* `<$> <(`*`startingvalues`*`)>;`

`array` *`arrayname`*{n} *`variable_list`* `<$> <(`*`startingvalues`*`)>;`

You can then use the array name with curly braces ({}) and a subscript, or in a `do over` loop:

```
array x x1-x9;                        array x x1-x9;
do i = 1 to 9;                        do over x;
   if x{i} = 9 then x{i} = .;            if x = 9 then x = .;
   end;                                  end;
```

Notes: 1. All the variables in an array must be of the same type.

2. An array can not have the same name as a variable.

3. You can use the keyword `_temporary_` instead of a variable list.

4. The statement `array x{3};` generates variables `x1`, `x2`, and `x3`.

5. The function `dim` returns the number of elements in an array.

Titles and Footnotes

SAS allows up to ten lines of text at the top (titles) and bottom (footnotes) of each page of output, specified with `title` and `footnote` statements. The form of these statements is

`title<n> text;` or `footnote<n> text;`

where *n*, if specified, can range from 1 to 10, and *text* must be surrounded by double or single quotes. If *text* is omitted, the title or footnote is deleted; otherwise it remains in effect until it is redefined. Thus, to have no titles, use:

```
title;
```

By default SAS includes the date and page number on the top of each piece of output. These can be suppressed with the `nodate` and `nopage` system options.

Missing Values

SAS handles missing values consistently throughout various procedures, generally by deleting observations which contain missing values. It is therefore very important to inspect the log and listing output, as well as paying attention to the numbers of observations used, when your data contains missing values.

For character variables, a missing value is represented by a blank (`" "` ; not a null string)
For numeric variables, a missing value is represented by a period (with no quotes). Unlike many languages, you can test for equality to missing in the usually fasion:

```
if string = " " then delete; * character variable;
if num = . then delete;      * numeric variable;

if x > 10 then x = .;        * set a variable to missing;
```

Special Missing Values

In addition to the regular missing value (`.`), you can specify one or more single alphabetic characters which will be treated as missing values when encountered in your input.

Most procedures will simply treat these special missing values in the usual way, but others (such as `freq` and `summary`) have options to tabulate each type of missing value separately. For example,

```
data one;
missing x;
input vv @@;
datalines;
12 4 5 6 x 9 . 12
;
```

The 5th and 7th observations will both be missing, but internally they are stored in different ways.

Note: When you use a special missing value, it will not be detected by a statement like `if vv = .`; in the example above, you would need to use `if vv = .x` to detect the special missing value, or to use the `is missing` operator of the `where` statement.

Variable Lists

SAS provides several different types of variable lists, which can be used in all procedures, and in some data step statements.

- Numbered List - When a set of variables have the same prefix, and the rest of the name is a consecutive set of numbers, you can use a single dash (-) to refer to an entire range:
  `x1 - x3` ⇒ `x1, x2, x3`; `x01 - x03` ⇒ `x01, x02, x03`
- Colon list - When a set of variables all begin with the same sequence of characters you can place a colon after the sequence to include them all. If variables `a`, `b`, `xheight`, and `xwidth` have been defined, then `x:` ⇒ `xwidth`, `xheight`.
- Special Lists - Three keywords refer to a list with the obvious meaning: `_numeric_` `_character_` `_all_`
  In a data step, special lists will only refer to variables which were already defined when the list is encountered.

Variable Lists (cont'd)

- Name range list - When you refer to a list of variables in the order in which they were defined in the SAS data set, you can use a double dash (`--`) to refer to the range:
  If the input statement
  `input id name $ x y z state $ salary`
  was used to create a data set, then
  `x -- salary` ⇒ `x, y, z, state, salary`
  If you only want character or numeric variables in the name range, insert the appropriate keyword between the dashes:
  `id -numeric- z` ⇒ `id, x, y, z`
  In general, variables are defined in the order they appear in the data step. If you're not sure about the order, you can check using `proc contents`.

The `set` statement

When you wish to process an already created SAS data set instead of raw data, the `set` statement is used in place of the `input` and `infile` or `lines` statements.

Each time it encounters a `set` statement, SAS inputs an observation from an existing data set, containing all the variables in the original data set along with any newly created variables.

This example creates a data set called `trans` with all the variables in the data set `orig` plus a new variable called `logx`:

```
data trans;
   set orig;
   logx = log(x);
run;
```

If you use a `set` statement without specifying a data set name, SAS will use the most recently created data set.

`drop=` and `keep=` data set options

Sometimes you don't need to use all of the variables in a data set for further processing. To restrict the variables in an input data set, the data set option `keep=` can be used with a list of variable names. For example, to process the data set `big`, but only using variables `x`, `y`, and `z`, the following statements could be used:

```
data new;
   set big(keep = x y z);
         . . .
```

Using a data set option in this way is very efficient, because it prevents all the variables from being read for each observation. If you only wanted to remove a few variables from the data set, you could use the `drop=` option to specify the variables in a similar fashion.

`drop` and `keep` statements

To control the variables which will be output to a data set, `drop` or `keep` statements can be used. (It is an error to specify both `drop` and `keep` in the same data step). Suppose we have a data set with variables representing `savings` and `income`. We wish to output only those observations for which the ratio of savings to income is greater than 0.05, but we don't need this ratio output to our final result.

```
data savers;
  set all;
  test = savings / income;
  if test > .05 then output;
  drop test;
  run;
```

As an alternative to `drop`, the statement

```
keep income savings;
```

could have been used instead.

`retain` statement

SAS' default behavior is to set all variables to missing each time a new observation is read. Sometimes it is necessary to "remember" the value of a variable from the previous observation. The `retain` statement specifies variables which will retain their values from previous observations instead of being set to missing. You can specify an initial value for `retain`ed variables by putting that value after the variable name on the `retain` statement.

Note: Make sure you understand the difference between `retain` and `keep`.

For example, suppose we have a data set which we assume is sorted by a variable called `x`. To print a message when an out-of-order observation is encountered, we could use the following code:

```
retain lastx .;     * retain lastx and initialize to missing;
if x < lastx then put 'Observation out of order, x=' x;
else lastx = x;
```


`sum` Statement

Many times the sum of a variable needs to be accumulated between observations in a data set. While a retain statement could be used, SAS provides a special way to accumulate values known as the sum statement. The format is

```
variable + expression;
```

where *`variable`* is the variable which will hold the accumulated value, and *`expression`* is a SAS expression which evaluates to a numeric value. The value of *`variable`* is automatically initialized to zero. The sum statement is equivalent to the following:

```
retain variable 0;
variable = variable + expression;
```

with one important difference. If the value of *`expression`* is missing, the sum statement treats it as a zero, whereas the normal computation will propogate the missing value.

Default Data Sets

In most situations, if you don't specify a data set name, SAS will use a default dataset, using the following rules:

- When creating data sets, SAS uses the names `data1`, `data2`, etc, if no data set name is specified. This can happen because of a `data` step, or if a procedure automatically outputs a data set which you have not named.
- When processing data sets, SAS uses the most recently created data set, which has the special name `_last_`. This can happen when you use a `set` statement with no dataset name, or invoke a procedure without a `data=` argument. To override this, you can set the value of `_last_` to a data set of your choice with the `options` statement:

```
options _last_ = mydata;
```


Temporary Data Sets

By default, the data sets you create with SAS are deleted at the end of your SAS session. During your session, they are stored in a directory with a name like `SAS_workaXXXX`, where the `X`s are used to create a unique name. By default, this directory is created within the system `/tmp` directory.
You can have the temporary data sets stored in some other directory using the `-work` option when you invoke sas, for example:

```
sas -work .
```

to use the current directory or, for example,

```
sas -work /some/other/directory
```

to specify some other directory.
Note: If SAS terminates unexpectedly, it may leave behind a work directory which may be very large. If so, it will need to be removed using operating system commands.

Permanent Data Sets

You can save your SAS data sets permanently by first specifying a directory to use with the `libname` statement, and then using a two level data set name in the data step. Any directory can be used to store SAS data sets.

```
libname project "/some/directory";
data project.one;
      . . .
```

In a later session, you could refer to the data set directly, without having to create it in a data step.

```
libname project "/some/directory";
proc reg data=project.one;
      . . .
```

Data sets created this way will have filenames of the form *`datasetname`*`.ssd` or *`datasetname`*`.ssd01`.

Operators in SAS

Arithmetic operators:

`*` multiplication `+` addition `/` division

`-` subtraction `**` exponentiation

Comparison Operators:

`=` *or* `eq` equal to `^=` *or* `ne` not equal to

`>` *or* `gt` greater than `>=` *or* `ge` greater than or equal to

`<` *or* `lt` less than `<=` *or* `le` less than or equal to

Boolean Operators:

`&` *or* `and` and `|` *or* `or` or `^` *or* `not` negation

Other Operators:

`><` minimum `<>` maximum `||` char. concatenation

The `in` operator lets you test for equality to any of several constant values. `x in (1,2,3)` is the same as `x=1 | x=2 | x=3`.

Comparison Operators

Use caution when testing two floating point numbers for equality, due to the limitations of precision of their internal representations. The `round` function can be used to alleviate this problem.

Two SAS comparison operators can be combined in a single statement to test if a variable is within a given range, without having to use any boolean operators. For example, to see if the variable `x` is in the range of 1 to 5, you can use `if 1 < x < 5 ...`.

SAS treats a numeric missing value as being less than any valid number. Comparisons involving missing values do not return missing values.

When comparing characters, if a colon is used after the comparison operator, the longer argument will be truncated for the purpose of the comparison. Thus, the expression `name =:  "R"` will be true for any value of `name` which begins with `R`.

Logical Variables

When you write expressions using comparison operators, they are processed by SAS and evaluated to `1` if the comparison is true, and `0` if the comparison is false. This allows them to be used in logical statements like an `if` statement as well as directly in numerical calculations.

For example, suppose we want to count the number of observations in a data set where the variable `age` is less than `25`. Using an `if` statement, we could write:

```
if age < 25 then count + 1;
```

(Note the use of the `sum` statement.)

With logical expressions, the same effect can be acheived as follows:

```
count + (age < 25);
```

Logical Variables (cont'd)

As a more complex example, suppose we want to create a categorical variable called `agegrp` from the continuous variable `age` where `agegrp` is `1` if age is less than 20, `2` if age is from 21 to 30, `3` if age is from 31 to 40, and `4` if age is greater than 40. To perform this transformation with `if` statements, we could use statements like the following:

```
agegrp = 1;
if 20 < age <= 30 then agegrp = 2;
if 30 < age <= 40 then agegrp = 3;
if age > 40 then agegrp = 4;
```

Using logical variables provides the following shortcut:

```
agegrp = 1 + (age > 20) + (age > 30) + (age > 40);
```


Variable Attributes

There are four attributes common to SAS variables.

- length - the number of bytes used to store the variable in a SAS data set
- informat - the format used to read the variable from raw data
- format - the format used to print the values of the variable
- label - a descriptive character label of up to 40 characters

You can set any one of these attributes by using the statement of the appropriate name, or you can set all four of them using the `attrib` statement.

Since named variable lists depend on the order in which variables are encountered in the data step, a common trick is to use a `length` or `attribute` statement, listing variables in the order you want them stored, as the first statement of your data step.

Variable Lengths: Character Values

- For character variables, SAS defaults to a length of 8 characters. If your character variables are longer than that, you'll need to use a length statement, an informat statement or supply a format on the input statement.
- When specifying a length or format for a character variable, make sure to precede the value with a dollar sign (`$`):

  ```
  attrib string length = $ 12 format = $char12.;
  ```

- The maximum length of a SAS character variable is 200.
- By default SAS removes leading blanks in character values. To retain them use the `$charw.` informat.
- By default SAS pads character values with blanks at the end. To remove them, use the `trim` function.

Variable Lengths: Numeric Values

- For numeric variables, SAS defaults to a length of 8 bytes (double precision.) For non-integers, you should probably not change from the default.
- For integers, the following chart shows the maximum value which can be stored in the available lengths:

length	Max. value	length	Max. value
3	8,192	6	137,438,953,472
4	2,097,152	7	35,184,372,088,832
5	536,870,912	8	9,007,199,254,740,992

- You can use the `default=` option of the length statement to set a default for all numeric variables produced:

  ```
  length default = 4;
  ```

- Even if a numeric variable is stored in a length less than 8, it will be promoted to double precision for all calculations.

Initialization and Termination

Although the default behavior of the data step is to automatically process each observation in an input file or existing SAS data set, it is often useful to perform specific tasks at the very beginning or end of a data step. The automatic SAS variable `_n_` counts the number of iterations of the data set. It is always available within the data step, but never output to a data set. This variable will be equal to 1 only on the first iteration of the data step, so it can be used to signal the need for initializations.

To tell when the last observation is being processed in a data step, the `end=` variable of either the `infile` or `set` statement can be used. This variable is not output to a data set, but will be equal to 1 only when the last observation of the input file or data set is being processed, and will equal 0 otherwise; thus any actions to be done at the very end of processing can be performed when this variable is equal to 1.

Flow Control: `if-then-else`

The `if-then` statement (with optional `else`) is used to conditionally execute SAS statements:

```
if x < 5 then group = "A";
```

It may be followed by a (separate) `else` statement:

```
if x < 5 then group = "A";
else group = "B";
```

To execute more than one statement (for either the `then` or the `else`, use a `do-end` block:

```
if x < 5 then do;
     group = "A";
     use = 0;
     end;
```

Flow Control: Subsetting `if`

Using an `if` statement without a corresponding `then` serves as a filter; observations which do not meet the condition will not be processed any further.

For example, the statement

```
if age < 60;
```

is equivalent to the statement

```
if age >= 60 then delete;
```

and will prevent observations where age is not less than 60 from being output to the data set. This type of `if` statement is therefore known as a *subsetting if*.
Note: You can not use an `else` statement with a subsetting `if`.

Flow Control: `goto` statement

You can use the `goto` statement to have SAS process statements in some other part of your program, by providing a label followed by a colon before the statements you wish to jump to. Label names follow the same rules as variable names, but have a different name space. When a labeled statement is encountered in normal processing, it is ignored.

Use `goto` statements with caution, since they can make program logic difficult to follow.

```
data two;
   set one;
   if x ^= . then goto out;
   x = (y + z) / 2;
   out:  if x > 20 then output;
run;
```

Flow Control: `stop`, `abort`, `return`

Although rarely necessary, it is sometimes useful to override SAS' default behavior of processing an entire set of data statements for each observation. Control within the current execution of the data step can be acheived with the `goto` statement; these statements provide more general control.

`stop`	immediately discontinue entire execution of the data step
`abort`	like stop, but set `_error_ to 1`
`error`	like abort, but prints a message to the SAS log
`return`	begin execution of next iteration of data step

For example, the following statement would stop processing the current data step and print an error message to the log:

```
if age > 99 then error "Age is too large for subject number " subjno ;
```


Do-loops

Do-loops are one of the main tools of SAS programming. They exist in several forms, always terminated by an `end;` statement

- `do;` - groups blocks of statements together
- `do over` *`arrayname`*`;` - process array elements
- `do` *`var`*`=`*`start`* `to` *`end`* `<by` *`inc`*`>;` - range of numeric values
- `do` *`var`*`=`*`list-of-values`*`;`
- `do while(`*`expression`*`);` (expression evaluated before loop)
- `do until(`*`expression`*`);` (expression evaluated after loop)

The `do until` loop is guaranteed to be executed at least once.

Some of these forms can be combined, for example

```
do i= 1 to end while (sum < 100);
```

Iterative Do-loops: Example 1

Do-loops can be nested. The following example calculates how long it would take for an investment with interest compounded monthly to double:

```
data interest;
do rate = 4,4.5,5,7,9,20;
    mrate = rate / 1200;   * convert from percentage;
    months = 0;
    start = 1;
    do while (start < 2);
         start = start * (1 + mrate);
         months + 1;
         end;
    years = months / 12;
    output;
    end;
keep rate years;
run;
```


Iterative Do-loops: Example 2

Suppose we have a record of the number of classes students take in each year of college, stored in variables `class1-class5`. We want to find out how long it takes students to take 10 classes:

```
data ten;
   set classes;
   array class class1-class5;
   total = 0;
   do i = 1 to 5 until(total >= 10);
       total = total + class{i};
       end;
   year = i;
   if total lt 10 then year = .;
   drop i total;
   run;
```

Getting out of Do-loops

There are two options for escaping a do-loop before its normal termination:

You can use a `goto` statement to jump outside the loop:

```
count = 0;
do i=1 to 10;
   if x{i} = . then count = count + 1;
   if count > 5 then goto done:
   end;
done: if count < 5 then output;
        . . .
```

You can also force termination of a do-loop by modifying the value of the index variable. *Use with caution since it can create an infinite loop.*

```
do i=1 to 10;
   if x{i} = . then count = count + 1;
   if count > 5 then i=10;
   end;
```


SAS Functions: Mathematical

Each function takes a single argument, and may return a missing value (.) if the function is not defined for that argument.

Name	Function	Name	Function
`abs`	absolute value	`arcos`	arccosine
`digamma`	digamma function	`arsin`	arcsin
`erf`	error function	`atan`	arctangent
`exp`	power of e ($2.71828\cdots$)	`cos`	cosine
`gamma`	gamma function	`cosh`	hyperbolic cosine
`lgamma`	log of `gamma`	`sin`	sine
`log`	log (base e)	`sinh`	hyperbolic sine
`log2`	log (base 2)	`tan`	tangent
`log10`	log (base 10)	`tanh`	hyperbolic tangent
`sign`	returns sign or zero		
`sqrt`	square root		

SAS Functions: Statistical Summaries

The statistical summary functions accept unlimited numbers of arguments, and ignore missing values.

Name	Function	Name	Function
`css`	corrected sum of squares	`range`	maximium − minimum
		`skewness`	skewness
`cv`	coefficient of variation	`std`	standard deviation
		`stderr`	standard error of the mean
`kurtosis`	kurtosis		
`max`	maximum	`sum`	sum
`mean`	mean	`uss`	uncorrected sum of squares
`min`	minimum		
		`var`	variance

In addition, the function `ordinal(n,...)` gives the `n`th ordered value from its list of arguments.

Using Statistical Summary Functions

You can use variable lists in all the statistical summary functions by preceding the list with the word "`of`"; for example:

```
xm = mean(of x1-x10);
vmean = mean(of thisvar -- thatvar);
```

Without the `of`, the single dash is interpreted in its usual way, that is as a minus sign or the unary minus operator; thus

```
xm = mean(of x1-x10);
```

is the same as

```
xm = mean(x1,x2,x3,x4,x5,x6,x7,x8,x9,x10);
```

but

```
xm1 = mean(x1-x10);
```

calculates the mean of x1 *minus* x10, and

```
xm2 = mean(x1--x10);
```

calculates the mean of x1 *plus* x10.

SAS Functions: Character Manipulation

`compress(target,<chars-to-remove>)`

```
expr = "one, two: three:";
new = compress(expr,",:");
```

results in `new` equal to `one two three`

With no second argument `compress` removes blanks.

`index(source,string)` - finds position of `string` in `source`

```
where = "university of california";
i = index(where,"cal");
```

results in `i` equal to `15`

`indexc(source,string)` - finds position of any character in `string` in `source`

```
where = "berkeley, ca";
i = indexc(where,"abc");
```

results in `i` equal to 1, since `b` is in position 1.

`index` and `indexc` return `0` if there is no match

SAS Functions: Character Manipulation (cont'd)

`left(string)` - returns a left-justified character variable

`length(string)` - returns number of characters in a string

`length` returns `1` if `string` is missing, `12` if `string` is uninitialized

`repeat(string,n)` - repeats a character value `n` times

`reverse(string)` - reverses the characters in a character variable

`right(string)` - returns a right-justified character variable

```
both = right(one) || left(two);
```

insures no blanks between the two concatenated variables.

`scan(string,n,<delims>)` - returns the `nth` "word" in `string`

```
field = "smith, joe";
first = scan(field,2," ,");
```

results in `first` equal to `joe`

If you omit `delims`, SAS uses most non-alphanumeric characters

SAS Functions: Character Manipulation (cont'd)

`substr(string,position,<n>)` - returns pieces of a variable

```
field = "smith, joe";
last = substr(field,1,index(field,",") - 1);
```

results in `last` equal to `smith`

`translate(string,to,from)` - changes `from` chars to `to chars`

```
word = "eXceLLent";
new = translate(word,"xl","XL");
```

results in `new` equal to `excellent`

`trim(string)` - returns `string` with leading blanks removed

`upcase(string)` - converts lowercase to uppercase

`verify(source,string)` - return position of first char. in `source` which is not in `string`

```
check = verify(val,"0123456789.");
```

results in `check` equal to 0 if `val` is a character string containing only numbers and periods.

SAS Functions for Random Number Generation

Each of the random number generators accepts a seed as its first argument. If this value is greater than 0, the generator produces a reproducible sequence of values; otherwise, it takes a seed from the system clock and produces a sequence which can not be reproduced.

The two most common random number functions are

`ranuni(seed)` - uniform variates in the range $(0, 1)$, and

`rannor(seed)` - normal variates with mean 0 and variance 1.

Other distributions include binomial (`ranbin`), Cauchy (`rancau`), exponential (`ranexp`), gamma (`rangam`), Poisson (`ranpoi`), and tabled probability functions (`rantbl`).

For more control over the output of these generators, see the documention for the corresponding `call` routines, for example `call ranuni`.

Generating Random Numbers

The following example, which uses no input data, creates a data set containing simulated data. Note the use of `ranuni` and the `int` function to produce a categorical variable (`group`) with approximately equal numbers of observations in each category.

```
data sim;
do i=1 to 100;
   group = int(5 * ranuni(12345)) + 1;
   y = rannor(12345);
   output;
   end;
keep group  y;
run;
```


Creating Multiple Data Sets

To create more than one data set in a single data step, list the names of all the data sets you wish to create on the `data` statement.

When you have multiple data set names on the `data` statement observations will be automatically output to all the data sets unless you explicitly state the name of the data set in an `output` statement.

```
data young old;
    set all;
    if age < 25 then output young;
    else output old;
run;
```

Note: If your goal is to perform identical analyses on subgroups of the data, it is usually more efficient to use a `by` statement or a `where` statement.

Subsetting Observations

Although the subsetting `if` is the simplest way to subset observations you can actively remove observations using a `delete` statement, or include observations using a `output` statement.

- `delete` statement

  ```
  if reason = 99 then delete;
  if age > 60 and sex = "F" then delete;
  ```

  No further processing is performed on the current observation when a `delete` statement is encountered.

- `output` statement

  ```
  if reason ^= 99 and age < 60 then output;
  if x > y then output;
  ```

  Subsequent statements are carried out (but not reflected in the current observation). When a data step contains one or more output statements, SAS' usual automatic outputting at the end of each data step iteration is disabled — only observations which are explicitly `output` are included in the data set.

Random Access of Observations

In the usual case, SAS automatically processes each observation in sequential order. If you know the position(s) of the observation(s) you want in the data set, you can use the `point=` option of the `set` statement to process only those observations.

The point= option of the set statement specifies the name of a temporary variable whose value will determine which observation will be read. When you use the `point=` option, SAS' default behavior of automatically looping through the data set is disabled, and you must explicitly loop through the desired observations yourself, and use the `stop` statement to terminate the data step.

The following example also makes use of the `nobs=` option of the `set` statement, which creates a temporary variable containing the number of observations contained in the data set.

Random Access of Observations: Example

The following program reads every third observation from the data set `big`:

```
data sample;
   do obsnum = 1 to total by 3;
      set big point=obsnum nobs=total;
      if _error_ then abort;
      output;
      end;
   stop;
run;
```

Note that the `set` statement is inside the do-loop. If an attempt is made to read an invalid observation, SAS will set the automatic variable `_error_` to `1`. The `stop` statement insures that SAS does not go into an infinite loop;

Application: Random Sampling I

Sometimes it is desirable to use just a subsample of your data in an analysis, and it is desired to extract a random sample, i.e. one in which each observation is just as likely to be included as each other observation. If you want a random sample where you don't control the exact number of observations in your sample, you can use the `ranuni` function in a very simple fashion. Suppose we want a random sample consisting of roughly 10% of the observations in a data set. The following program will randomly extract the sample:

```
data sample;
    set giant;
    if ranuni(12345) < .1;
run;
```

Application: Random Sampling II

Now suppose we wish to randomly extract *exactly* `n` observations from a data set. To insure randomness, we must adjust the fraction of observations chosen depending on how many observations we have already chosen. This can be done using the `nobs=` option of the `set` statement. For example, to choose exactly 15 observations from a data set `all`, the following code could be used:

```
data some;
  retain k 15 n ;
  drop k n;
  set all nobs=nn;
  if _n_ = 1 then n = nn;
  if ranuni(0) < k / n then do;
     output;
     k = k - 1;
     end;
  if k = 0 then stop;
  n = n - 1;
  run;
```


By Processing in Procedures

In procedures, the `by` statement of SAS allows you to perform identical analyses for different groups in your data. Before using a `by` statement, you must make sure that the data is sorted (or at least grouped) by the variables in the `by` statement.

The form of the `by` statement is

`by <descending>` *`variable-1`* `···` `<<descending>` *`variable-n`* `<notsorted>>;`

By default, SAS expects the `by` variables to be sorted in ascending order; the optional keyword `descending` specifies that they are in descending order.

The optional keyword `notsorted` at the end of the `by` statement informs SAS that the observations are grouped by the `by` variables, but that they are not presented in a sorted order. Any time any of the `by` variables change, SAS interprets it as a new `by` group.

Selective Processing in Procedures: `where` statement

When you wish to use only some subset of a data set in a procedure, the `where` statement can be used to select only those observations which meet some condition. There are several ways to use the `where` statement.

As a procedure statement:

```
proc reg data=old;
  where sex eq 'M';
  model y=x;
run;
```

As a data set option:

```
proc reg data=old(where = (sex eq 'M'));
   model y = x;
run;
```

In the data step:

```
data survey;
   input id q1-q10;
   where q2 is not missing and q1 < 4;

data new;
   set old(where = (group = 'control'));
```


`where` statement: Operators

Along with all the usual SAS operators, the following are available in the `where` statement:

`between/and` - specify a range of observations

```
where salary between 20000 and 50000;
```

`contains` - select based on strings contained in character variables

```
where city contains 'bay';
```

`is missing` - select based on regular or special missing value

```
where x is missing and y is not missing;
```

`like` - select based on patterns in character variables
(Use `%` for any number of characters, `_` for exactly one)

```
where name like 'S%';
```

sounds like (`=*`) - select based on soundex algorithm

```
where name =* 'smith';
```

You can use the word `not` with all of these operators to reverse the sense of the comparison.

Multiple Data Sets: Overview

One of SAS's greatest strengths is its ability to combine and process more than one data set at a time. The main tools used to do this are the `set`, `merge` and `update` statements, along with the `by` statement and `first.` and `last.` variables.

We'll look at the following situations:

- Concatenating datasets by observation
- Interleaving several datasets based on a single variable value
- One-to-one matching
- Simple Merge Matching, including table lookup
- More complex Merge Matching

Concatenating Data Sets by Observation

The simplest operation concerning multiple data sets is to concatenate data sets by rows to form one large data set from several other data sets. To do this, list the sets to be concatenated on a `set` statement; each data set will be processed in turn, creating an output data set in the usual way.

For example, suppose we wish to create a data set called `last` by concatenating the data sets `first`, `second`, and `third`.

```
data last;
   set first second third;
```

If there are variables in some of the data sets which are not in the others, those variables will be set to missing (`.` or `' '`) in observations derived from the data sets which lacked the variable in question.

Concatenating Data Sets (cont'd)

Consider two data sets `clerk` and `manager`:

Name	Store	Position	Rank
Joe	Central	Sales	5
Harry	Central	Sales	5
Sam	Mall	Stock	3

Name	Store	Position	Staff
Fred	Central	Manager	10
John	Mall	Manager	12

The SAS statements to concatenate the data sets are:

```
data both;
   set clerk manager;
run;
```

resulting in the following data set:

Name	Store	Position	Rank	Staff
Joe	Central	Sales	5	.
Harry	Central	Sales	5	.
Sam	Mall	Stock	3	.
Fred	Central	Manager	.	10
John	Mall	Manager	.	12

Note that the variable `staff` is missing for all observations from set `clerk`, and `rank` is missing for all observations from `manager`. The observations are in the same order as the input data sets.

Concatenating Data Sets with `proc append`

If the two data sets you wish to concatenate contain exactly the same variables, you can save resources by using `proc append` instead of the `set` statement, since the `set` statement must process each observation in the data sets, even though they will not be changed. Specify the "main" data set using the `base=` argument and the data set to be appended using the `new=` argument. For example, suppose we wish to append the observations in a data set called `new` to a data set called `master.enroll`. Assuming that both data sets contained the same variables, you could use `proc append` as follows:

```
proc append base=master.enroll new=new;
run;
```

The SAS System will print an error message if the variables in the two data sets are not the same.

Interleaving Datasets based on a Single Variable

If you want to combine several datasets so that observations sharing a common value are all adjacent to each other, you can list the datasets on a `set` statement, and specify the variable to be used on a `by` statement. Each of the datasets must be sorted by the variable on the `by` statement.

For example, suppose we had three data sets `A`, `B`, and `C`, and each contained information about employees at different locations:

Set A			Set B			Set C		
Loc	Name	Salary	Loc	Name	Salary	Loc	Name	Salary
NY	Harry	25000	LA	John	18000	NY	Sue	19000
NY	Fred	20000	NY	Joe	25000	NY	Jane	22000
NY	Jill	28000	SF	Bill	19000	SF	Sam	23000
SF	Bob	19000	SF	Amy	29000	SF	Lyle	22000

Notice that there are not equal numbers of observations from the different locations in each data set.

Interleaving Datasets (cont'd)

To combine the three data sets, we would use a set statement combined with a by statement.

```
data all;
set a b c;
by loc;
run;
```

which would result in the following data set:

Loc	Name	Salary	Loc	Name	Salary
LA	John	18000	NY	Jane	22000
NY	Harry	25000	SF	Bob	19000
NY	Fred	20000	SF	Bill	19000
NY	Jill	28000	SF	Amy	29000
NY	Joe	25000	SF	Sam	23000
NY	Sue	19000	SF	Lyle	22000

Similar results could be obtained through a `proc sort` on the concatenated data set, but this technique is more efficient and allows for further processing by including programming statements before the `run;`.

One-to-one matching

To combine variables from several data sets where there is a one-to-one correspondence between the observations in each of the data sets, list the data sets to be joined on a `merge` statement. The output data set created will have as many observations as the largest data set on the `merge` statement. If more than one data set has variables with the same name, the value from the rightmost data set on the `merge` statement will be used.

You can use as many data sets as you want on the `merge` statement, but remember that they will be combined in the order in which the observations occur in the data set.

Example: one-to-one matching

For example, consider the data sets `personal` and `business`:

Personal

Name	Age	Eyes
Joe	23	Blue
Fred	30	Green
Sue	24	Brown

Business

Name	Job	Salary
Joe	Clerk	20000
Fred	Manager	30000
Sue	Cook	24000

To merge the variables in `business` with those in `personal`, use

```
data both;
    merge personal business;
```

to result in data set `both`

Name	Age	Eyes	Job	Salary
Joe	23	Blue	Clerk	20000
Fred	30	Green	Manager	30000
Sue	24	Brown	Cook	24000

Note that the observations are combined in the exact order in which they were found in the input data sets.

Simple Match Merging

When there is not an exact one-to-one correspondence between data sets to be merged, the variables to use to identify matching observations can be specified on a `by` statement. The data sets being merged must be sorted by the variables specified on the `by` statement.

Notice that when there is exactly one observation with each `by` variable value in each data set, this is the same as the one-to-one merge described above. Match merging is especially useful if you're not sure exactly which observations are in which data sets.

By using the `IN=` data set option, explained later, you can determine which from data set(s) a merged observation is derived.

Simple Match Merging (cont'd)

Suppose we have data for student's grades on two tests, stored in two separate files

```
ID   Score1  Score2             ID   Score3 Score4
 7       20      18              7       19     12
 9       15      19             10       12     20
12        9      15             12       10     19
```

Clearly a one-to-one merge would not be appropriate.

Pay particular attention to ID 9, which appears only in the first data set, and ID 10 which appears only in the second set.

To merge the observations, combining those with common values of `ID` we could use the following SAS statements:

```
data both;
   merge scores1 scores2;
   by id;
run;
```

Simple Match Merging (cont'd)

Here's the result of the merge:

ID	Score1	Score2	Score3	Score4
7	20	18	19	12
9	15	19	.	.
10	.	.	12	20
12	9	15	10	19

Notes

1. All datasets must be sorted by the variables on the `by` statement.
2. If an observation was missing from one or more data sets, the values of the variables which were found only in the missing data set(s) are set to missing.
3. If there are multiple occurences of a common variable in the merged data sets, the value from the rightmost data set is used.

Table Lookup

Consider a dataset containing a patient name and a room number, and a second data set with doctors names corresponding to each of the room numbers. There are many observations with the same room number in the first data set, but exactly one observation for each room number in the second data set. Such a situation is called table lookup, and is easily handled with a merge statement combined with a by statement.

Patients

Patient	Room
Smith	215
Jones	215
Williams	215
Johnson	217
Brown	217
. . .	

Doctors

Doctor	Room
Reed	215
Ellsworth	217
. . .	

Table Lookup (cont'd)

The following statements combine the two data sets.

```
data both;
merge patients doctors;
by room;
run;
```

resulting in data set `both`

Patient	Room	Doctor
Smith	215	Reed
Jones	215	Reed
Williams	215	Reed
Johnson	217	Ellsworth
Brown	217	Ellsworth
	. . .	

Notes:

- As always, both data sets must be sorted by the variables on the `by` list.
- The data set with one observation per `by` variable must be the second dataset on the `merge` statement.

Updating Data Sets

When you're combining exactly two data sets with the goal of updating some or all of the first data set's values with values from the second data set, you can use the `update` statement.

An `update` statement accepts exactly two data sets, and must be followed by a `by` statement. An `update` statement is similar to a `merge` statement except that

- the `update` statement will not overwrite non-missing values in data set `one` with missing values from data set `two`, and
- the `update` statement doesn't create any observations until all the observations for a `by` group are processed. Thus, the first data set should contain exactly one observation for each `by` group, while the second data set can contain multiple observations, with later observations supplementing or overriding earlier ones.

Example: `update` statement

Set `orig`

ID	Account	Balance
1	2443	274.40
2	4432	79.95
3	5002	615.00

Set `upd`

ID	Account	Balance
1	.	699.00
2	2232	.
2	.	189.95
3	6100	200.00

Data set `orig` can be updated with the values in `upd` using the following statements:

```
data orig;
  update orig upd;
  by id;
```

resulting in the updated data set:

ID	Account	Balance
1	2443	699.00
2	2232	189.95
3	6100	200.00

More Complex Merging

Keep the following in mind when performing more complex merges:

- If the merged data sets have variables in common (in addition to the variables on the `by` statement), and the values differ, the values from the rightmost data set in the `merge` statement are used.
- If there are multiple occurences of observations with the same value for the `by` variable(s) in the data sets being merged, they are combined one-to-one. If there are unequal numbers of these observations in any of the data sets being merged, the last value from the data set with fewer observations is reused for all the other observations with matching values.
- Various problems arise if variables in data sets being merged have different attributes. Try to resolve these issues before merging the data.

More Complex Merging (cont'd)

The following example, although artificial, illustrates some of the points about complex merging:

one

a	b	c
1	3	20
1	3	19
1	7	22
2	9	18
2	3	22

two

a	b	d
1	3	17
1	5	12
2	9	21
2	3	15
2	6	31

$\Longrightarrow$

three

a	b	c	d
1	3	20	17
1	5	19	12
1	7	22	12
2	9	18	21
2	3	22	15
2	6	22	31

The data sets were merged with the following statements:

```
data three;
merge one two;
by a;
```


`in=` data set option

When creating observations from multiple data sets, it is often helpful to know which data set an observation should come from. It should be clear that when merging large data sets, additional tools will be necessary to determine exactly how observations are created.

The `in=` data set option is one such tool. The option is associated with one or more of the data sets on a `merge` statement, and specifies the name of a temporary variable which will have a value of 1 if that data set contributed to the current observation and a 0 otherwise.

Two common uses of the `in=` variable are to make sure that only complete records are output, and to create a data set of problem observations which were missing from one of the merged data sets.

The next slide provides an example of these ideas, using the test scores data from a previous example.

Example of `in=` data set option

```
data both problem;
merge scores1(in=one) scores2(in=two);
by id;
if one and two then output both;
else output problem;
run;
```

The resulting data sets are shown below; note that the `in=` variables are not output to the data sets which are created.

Data set `both`

Obs	Id	Score1	Score2	Score3	Score4
1	7	20	18	19	12
2	12	9	15	10	19

Data set `problem`

Obs	Id	Score1	Score2	Score3	Score4
1	9	15	19	.	.
2	10	.	.	12	20

Programming with `by` statements

The power of the `by` statement coupled with the `merge` and `set` statements is enough to solve most problems, but occasionally more control is needed.

Internally, SAS creates two temporary variables for each ***`variable`*** on the `by` statement of a data step. `first.`***`variable`*** is equal to 1 if the current observation is the first occurence of this value of ***`variable`*** and 0 otherwise. Similarly, `last.`***`variable`*** is equal to 1 if the current observation is the last occurence of the value of ***`variable`*** and 0 otherwise.

When there are several `by` variables, remember that a new `by`-group begins when the value of the rightmost variable on the `by` statement changes.

Application: Finding Duplicate Observations I

Many data sets are arranged so that there should be exactly one observation for each unique combination of variable values. In the simplest case, there may be an identifier like a social security or student identification number, and we want to check to make sure there are not multiple observations with the same value for that variable.

If the data set is sorted by the identifier variable (say, `ID`), code like the following will identify the duplicates:

```
data check;
   set old;
   by id;
   if first.id and ^last.id;
run;
```

The duplicates can now be found in data set `check`

Example of `first.` and `last.` variables

Suppose we have a data set called `grp` with several observations for each value of a variable called `group`. We wish to output one observation for each group containing the three highest values of the variable `x` encountered for that group.

```
data max;
   set grp;
   by group;
   retain x1-x3;                        * preserve values btwn obs;
   if first.group then x1 = x2 = x3 = .;   * initialize;
   if x >= x1 then do;
        x3 = x2; x2 = x1; x1 = x; end;
   else if x >= x2 then do;
        x3 = x2; x2 = x; end;
   else if x >= x3 then x3 = x;
   if last.group then output;           * output one obs per group;
   keep group x1-x3;
run;
```

Example of `first.` and `last.` variables (cont'd)

Here are the results of a simple example of the previous program:

Set `grp`

Group	X
1	16
1	12
1	19
1	15
1	18
1	17
2	10
2	20
2	8
2	14
2	30
3	59
3	45
3	2
3	18

$\Longrightarrow$

Set `max`

Group	X1	X2	X3
1	19	18	17
2	30	20	14
3	59	45	18

Sorting datasets

For procedures or data steps which need a data set to be sorted by one or more variables, there are three options available:

1. You can use `proc sort;` to sort your data. Note that SAS stores information about the sort in the dataset header, so that it will not resort already sorted data.
2. You can enter your data in sorted order. If you choose this option, make sure that you use the correct sorting order.* To prevent `proc sort` from resorting your data, use the `sortedby=` data set option when you create the data set.
3. You can create an index for one or more combinations of variables which will stored along with the data set.

* EBCDIC Sorting Sequence (IBM mainframes):
`blank .<(+|\&!$*);^-/,%_>?‘:#@’="abcdefghijklmnopqr~stuvwxyz{ABCDEFGHI}JKLMNOOPQR\STUVWXYZ0123456789`
ASCII Sorting Sequence (most other computers):
`blank !"#$%&’()* +,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_‘abcdefghijklmnopqrstuvwxyz{|}~`

Indexed Data Sets

If you will be processing a data set using `by` statements, or subsetting your data based on the value(s) of one or more variables, you may want to store an index based on those variables to speed future processing.

`proc datasets` is used to create an index for a SAS data set. Suppose we have a data set called `company.employ`, with variables for `branch` and `empno`. To store a simple index (based on a single variable), statements like the following are used:

```
proc datasets library=company;
    modify employ;
    index create branch;
run;
```

More than one index for a data set can be specified by including multiple `index` statements. The index will be used whenever a `by` statement with the indexed variable is encountered.

Indexed Data Sets (cont'd)

To create a composite index (based on more than one variable), you need to provide a label for the index. (This label need *not* be specified when you access the data set.) You can have multiple composite indices within the same data set:

```
proc datasets library=company;
    modify employ;
    index create brnum = (branch idnum);
run;
```

In the previous example, the composite index would mean the data set is also indexed for `branch`, but *not* for `idnum`.

Note: If you are moving or copying an indexed data set, be sure to use a SAS procedure like `proc copy`, `datasets`, or `cport` rather than system utilities, to insure that the index gets correctly copied.

Formats and Informats

Formats are used for controlling the appearance of variable values in many procedures as well as for printing variable values using the `put` statement of the data step. In addition, many procedures use the format statement to group values together for analysis.

Informats are used in conjunction with the `input` statement to specify the way that variables are to be read if they are not in the usual numeric or character format.

SAS provides a large number of predefined formats, as well as the ability to write your own formats, for input as well as output.

You can permanently associate a format with a variable by including a `format` or `attribute` statement in the data step when the data set is created, or temporarily by using a format statement within a procedure.

Basic Formats

Numeric formats are of the form `w.` or `w.d`, representing a field width of `w`, and containing `d` decimal places.

```
put x 6.;          *write x with field width of 6;
format price 8.2; *use field width of 8 and 2 d.p. for price;
```

The `bestw.` format can be used if you're not sure about the number of decimals. (For example, `best6.` or `best8.`.)

Simple character formats are of the form `$w.`, where `w` is the desired field width. (Don't forget the period.)

```
put name $20.;       * write name with field width of 20;
format city $50.;    * use field width of 50 for name;
```

You can also use formats with the `put` function to create character variables formatted to your specifications:

```
x = 8;
charx = put(x,8.4);
```

creates a character variable called `charx` equal to `8.0000`

Informats

The basic informats are the same as the basic formats, namely `w.d` for numeric values, and `$w.` for character variables.

By default, leading blanks are stripped from character values. To retain them, use the `$charw.` format.

When you specify a character informat wider than the default of 8 columns, SAS automatically will make sure the variable is big enough to hold your input values.

Some Other SAS Informats

Name	Description	Name	Description
`hexw.`	numeric hexadecimal	`$hexw.`	character hexadecimal
`octalw.`	numeric octal	`$octalw.`	character octal
`bzw.d`	treat blanks as zeroes	`ew.d`	scientific notation
`rbw.d`	floating point binary	`ibw.d`	integer binary
`pdw.d`	packed decimal	`$ebcdicw.`	EBCDIC to ASCII

Writing User-defined formats using `proc format`

- You can specify several different `value` statements within a single invocation of `proc format`.
- Each value must be given a name (which does not end in a number), followed by a set of range/value pairs.
- The keywords `low`, `high`, and `other` can be used to construct ranges.
- If you wish a range of values to be formatted using some other format, enclose its name (including the period) in square brackets (`[ ]`) as the value for that range.
- Character values should be enclosed in quotes; the name of a character format must begin with a dollar sign (`$`).
- If a variable falls outside of the specified ranges, it is formatted using the usual defaults

User-defined Format: Examples

For a variable with values from 1 to 5, the format `qval.` displays 1 and 2 as `low`, 3 as `medium` and 4 and 5 as `high`.

The format `mf.` displays values of 1 as `male`, 2 as `female` and all other values as `invalid`.

The format `tt.` display values below .001 as `undetected`, and all other values in the usual way.

```
proc format;
value qval 1-2='low' 3='medium' 4-5='high';
value mf 1='male' 2='female' other='invalid';
value tt low-.001='undetected';
run;
```


Recoding Values using Formats

Since many SAS procedures use formatted values to produce groups, you can often recode variables by simply changing their formats. This is more efficient than processing an entire data set, and leaves the original variable unchanged for future use.

Suppose we have a survey which asks people how long they've been using computers (measured in years), and how happy they are with the computer they are using (measured on a scale of 1 to 5). We wish to produce a cross tabulation of these results, that is a table where rows represent levels of one variable, columns represent the level of another variable, and the entries represent the number of observations which fall into the row/column categories. Often the unformatted table will have many empty cells - it is common practice to group categories or values to create more meaningful tables.

Recoding Values using Formats (cont'd)

If the two variables in our survey are called years and happy, the following program would produce a cross tabulation:

```
proc freq;tables years*happy/nocol norow nocum nopct;
```

TABLE OF YEARS BY HAPPY

YEARS HAPPY

Frequency	1	2	3	4	5	Total
0.5	1	2	1	2	0	6
1	0	0	2	2	0	4
1.5	0	0	0	1	0	1
2	0	0	2	2	3	7
3	1	1	0	1	1	4
8	0	0	0	0	1	1
10	0	1	0	0	0	1
12	0	0	0	1	0	1
Total	2	4	5	9	5	25

Recoding Values using Formats (cont'd)

To make the table more useful, we define the following formats:

```
proc format;
    value yy 0-1='<=1yr' 1.5-3='1-3yr' 3.5-high='>3yr';
    value hh 1-2='Low' 3='Medium' 4-5='High';
run;
proc freq;
    tables years*happy/nocol norow nocum nopct;
    format years yy. happy hh.;
run;
```

TABLE OF YEARS BY HAPPY

YEARS HAPPY

Frequency	Low	Medium	High	Total
<=1yr	3	3	4	10
1-3yr	2	2	8	12
>3yr	1	0	2	3
Total	6	5	14	25

SAS Date and Time Values

There are three types of date and time values which SAS can handle, shown with their internal representation in parentheses:

- Time values (number of seconds since midnight)
- Date values (number of days since January 1, 1970)
- Datetime values (number of seconds since January 1, 1970)

You can specify a date and/or time as a constant in a SAS program by surrounding the value in quotes, and following it with a `t`, a `d` or a `dt`. The following examples show the correct format to use:

3PM ⇒ `'3:00p't` or `'15:00't` or `'15:00:00't`
January 4, 1937 ⇒ `'4jan37'd`
9:15AM November 3, 1995 ⇒
`'3nov95:9:15'dt` or `'3nov95:9:15:00'dt`

Date and Time Informats and Formats

Name	Width(default)	Examples
`date`*w*`.`	5-9 (7)	`26jun96`
`datetime`*w*`.`	7-40 (16)	`4JUL96:01:30p`
`ddmmyy`*w*`.`	2-8 (8)	`15/9/96 15-9-96 150996`
`julian`*w*`.`	5-7 (5)	`96200 1995001`
`mmddyy`*w*`.`	2-8 (8)	`9/15/96 9-15-96 091596`
`monyy`*w*`.`	5-7 (5)	`jan95 mar1996`
`time`*w*`.`	2-20 (8)	`6:00:00 3:15:00p`
`yymmdd`*w*`.`	2-8(8)	`19961225 950704`

The above formats are valid for input and output. In addition, the `worddate`*w*`.` format can be used to display dates like `July 4, 1996`, and the `monname`*w*`.` and `year`*w*`.` formats display parts of dates.

Date and Time Functions

`datepart` – Extract date from datetime value

```
dateonly = datepart(fulldate);
```

`day,month year` – Extract part of a date value

```
day = day(date);
```

`dhms` – Construct value from date, hour, minute and second

```
dtval = dhms(date,9,15,0);
```

`mdy` – Construct date from month, day and year

```
date = mdy(mon,day,1996);
```

`time` – Returns the current time of day

```
now = time();
```

`today` – Returns the current date

```
datenow = today();
```

`intck` – Returns the number of intervals between two values

```
days = intck('day',then,today());
```

`intnx` – Increments a value by a number of intervals

```
tomrw = intnx('day',today(),1);
```


Application: Blue Moons

When there are two full moons in a single month, the second is known as a blue moon. Given that July 9, 1998 is a full moon, and there are 29 days between full moons, in what months will the next few blue moons occur?

First, we create a data set which has the dates of all the full moons, by starting with July 9, 1998, and incrementing the date by 29 days.

```
data fullmoon;
   date = '09jul98'd;
   do i=1 to 500;
      month = month(date);
      year = year(date);
      output;
      date = intnx('day',date,29);
      end;
run;
```

Application: Blue Moons (cont'd)

Now we can use the `put` function to create a variable with the full month name and the year.

```
data bluemoon;
   set fullmoon;
   by year month;
   if last.month and not first.month then do;
      when = put(date,monname.) || ", " || put(date,year.);
      output;
   end;
run;
proc print data=bluemoon noobs;
   var when;
run;
```

The results look like this:

```
        December, 1998
          August, 2000
             May, 2002
             . . .
```


Customized Output: `put` statement

The `put` statement is the reverse of the `input` statement, but in addition to variable names, formats and pointer control, you can also print text. Most of the features of the `input` statement work in a similar fashion in the `put` statement. For example, to print a message containing the value of of a variable called `x`, you could use a `put` statement like:

```
put 'the value of x is ' x;
```

To print the values of the variables `x` and `y` on one line and `name` and `address` on a second line, you could use:

```
put x 8.5 y best10. / name $20 @30 address ;
```

Note the use of (optional) formats and pointer control.

By default, the `put` statement writes to the SAS log; you can override this by specifying a filename or fileref on a `file` statement.

Additional Features of the `put` statement

By default, the `put` statement puts a newline after the last item processed. To prevent this (for example to build a single line with multiple `put` statements, use a trailing `@` at the end of the `put` statement.

The *`n*`* operator repeats a string *`n`* times. Thus

```
put 80*"-";
```

prints a line full of dashes.

Following a variable name with an equal sign causes `the` put statement to include the variable's name in the output. For example, the statements

```
x = 8;
put x=;
```

results in `X=8` being printed to the current output file. The keyword `_all_` on the `put` statement prints out the values of all the variables in the data set in this named format.

Headers with `put` statements

You can print headings at the top of each page by specifying a `header=` specification on the `file` statement with the label of a set of statements to be executed. For example, to print a table containing names and addresses, with column headings at the top of each page, you could use statements like the following:

```
options ps=50;
 data _null_;
    set address;
    file "output" header = top print;
    put @1 name $20. @25 address $30.;
    return;
 top:
    put @1 'Name' @25 'Address' / @1 20*'-' @25 30*'-';
    return;
 run;
```

Note the use of the two `return` statements. The `print` option is required when using the `header=` option on the file statement.

SAS System Options

SAS provides a large number of options for fine tuning the way the program behaves. Many of these are system dependent, and are documented online and/or in the appropriate SAS Companion.

You can specify options in three ways:

1. On the command line when invoking SAS, for example

   ```
   sas -nocenter -nodms -pagesize 20
   ```

2. In the system wide `config.sas` file, or in a local `config.sas` file (see the SAS Companion for details).
3. Using the options statement:

   ```
   options nocenter pagesize=20;
   ```

Note that you can precede the name of options which do not take arguments with `no` to shut off the option. You can display the value of all the current options by running `proc options`.

Some Common Options

Option	Argument	Description
		Options which are useful when invoking SAS
`dms`	-	Use display manager windows
`stdio`	-	Obey UNIX standard input and output
`config`	*`filename`*	Use *`filename`* as configuration file
		Options which control output appearance
`center`	-	Center output on the page
`date`	-	Include today's date on each page
`number`	-	Include page numbers on output
`linesize`	*`number`*	Print in a width of *`number`* columns
`pagesize`	*`number`*	Go to a new page after *`number`* lines
`ovp`	-	Show emphasis by overprinting
		Options which control data set processing
`obs`	*`number`*	Process a maximum of *`number`* obs.
`firstobs`	*`number`*	Skip first *`number`* observations
`replace`	-	Replace permanent data sets?

Application: Rescanning Input

Suppose we have an input file which has a county name on one line followed by one or more lines containing x and y coordinates of the boundaries of the county. We wish to create a separate observation, including the county name, for each set of coordinates.

A segment of the file might look like this:

```
alameda
-121.55   37.82 -121.55   37.78 -121.55   37.54 -121.50   37.53
-121.49   37.51 -121.48   37.48
amador
-121.55   37.82 -121.59   37.81 -121.98   37.71 -121.99   37.76
-122.05   37.79 -122.12   37.79 -122.13   37.82 -122.18   37.82
-122.20   37.87 -122.25   37.89 -122.27   37.90
calaveras
-121.95   37.48 -121.95   37.48 -121.95   37.49 -122.00   37.51
-122.05   37.53 -122.07   37.55 -122.09   37.59 -122.11   37.65
-122.14   37.70 -122.19   37.74 -122.24   37.76 -122.27   37.79
-122.27   37.83 -122.27   37.85 -122.27   37.87 -122.27   37.90
                                  . . .
```


Application: Rescanning Input (cont'd)

Note that we don't know how many observations (or data lines) belong to each county.

```
data counties;
length county $ 12 name $ 12;
infile "counties.dat";
retain county;
input name $ @;        * hold current line for rescanning;
if indexc(name,'0123456789') = 0 then do;
   county = name;
   delete;             * save county name, but don't output;
   end;
else do;
   x = input(name,12.) * do numeric conversion;
   input y @@;         * hold the line to read more x/y pairs;
   end;
drop name;
run;
```

Application: Reshaping a Data Set I

Since SAS procedures are fairly rigid about the organization of their input, it is often necessary to use the data step to change the shape of a data set. For example, repeated measurements on a single subject may be on several observations, and we may want them all on the same observation. In essence, we want to perform the following transformation:

Subj	Time	X
1	1	10
1	2	12
	...	
1	n	8
2	1	19
2	2	7
	...	
2	n	21

$\Longrightarrow$

Subj	X1	X2	...	Xn
1	10	12	...	8
2	19	7	...	21

Application: Reshaping a Data Set I(cont'd)

Since we will be processing multiple input observations to produce a single output observation, a `retain` statement must be used to remember the values from previous inputs. The combination of a `by` statement and `first.` and `last.` variables allows us to create the output observations at the appropriate time, even if there are incomplete records.

```
data two;
   set one;
   by subj;
   array xx x1-xn;
   retain x1-xn;
   if first.subj then do i=1 to n; xx{i} = .;end;
   xx{time} = x;
   if last.subj then output;
   drop time x;
   run;
```

Application: Reshaping a Data Set II

A similar problem to the last is the case where the data for several observations is contained on a single line, and it is necessary to convert each of the lines of input into several observations. Suppose we have test scores for three different tests, for each of several subjects; we wish to create three separate observations from each of these sets of test scores:

```
    data scores;
 * assume set three contains id, group and score1-score3;
       set three;
       array sss score1-score3;
       do time = 1 to 3;
           score = sss{time};
           output;
           end;
    drop score1-score3;
    run;
```


Output Data Sets

Many SAS procedures produce output data sets containing data summaries (`means`, `summary`, `univariate`), information about statistical analyses (`reg`, `glm`, `nlin`, `tree`) or transformed variables (`standard`, `score`, `cancorr`, `rank`); some procedures can produce multiple output data sets. These data sets can be manipulated just like any other SAS data sets.

Recall that the statistical functions like `mean()` and `std()` can calculate statistical summaries for variables within an observation; output data sets are used to calculate summaries of variables over the whole data set.

When you find that you are looping through an entire data set to calculate a single quantity which you then pass on to another data step, consider using an output data set instead.

Output Data Sets: Example I

It is often useful to have summary information about a data set available when the data set is being processed. Suppose we have a data set called `new`, with a variable `x`, and we wish to calculate a variable `px` equal to `x` divided by the maximum value of `x`.

```
proc summary data=new;
   var x;
   output out=sumnew max=maxx;
run;
data final;
   if _n_ = 1 then set sumnew(keep=maxx);
   set new;
   px = x / maxx;
run;
```

The automatic variable `_n_` will be 1 for the first observation only; the single observation in `sumnew` gets read at this time. The `set new;` statement then reads in the original data.

Output Data Sets: Example II

Now suppose we have two classification variables called `group` and `trtmnt`, and we wish to use the maximum value for each `group`/`trtmnt` combination in the transformation. If the data set had already been sorted, the following statements could be used:

```
proc summary nway data=new;
  class group trtmnt;
  var x;
  output out=sumnew max=maxx;
run;
data final;
  merge new sumnew(keep=maxx);
  by group trtmnt;
  px = x / maxx;
run;
```

The `nway` option limits the output data set to contain observations for each unique combination of the variables given in the `class` statement.

Output Data Sets: Example III

Suppose we have a data set called `hdata`, consisting of three variables: `hospital`, `time` and `score`, representing the score of some medical exam taken at three different times at three different hospitals, and we'd like to produce a plot with three lines: one for the means of each of the three hospitals over time. The following statements could be used:

```
proc means noprint nway data=hdata;
   class hospital time;
   var score;
   output out=hmeans mean=mscore;
run;
```

The `noprint` option suppresses the usual printing which is the default for `proc means`. You could acheive similar results using a `by` statement instead of a `class` statement, but the data set would need to be sorted.

Output Data Sets: Example III(cont'd)

The transformation which the previous program produced can be thought of as the following:

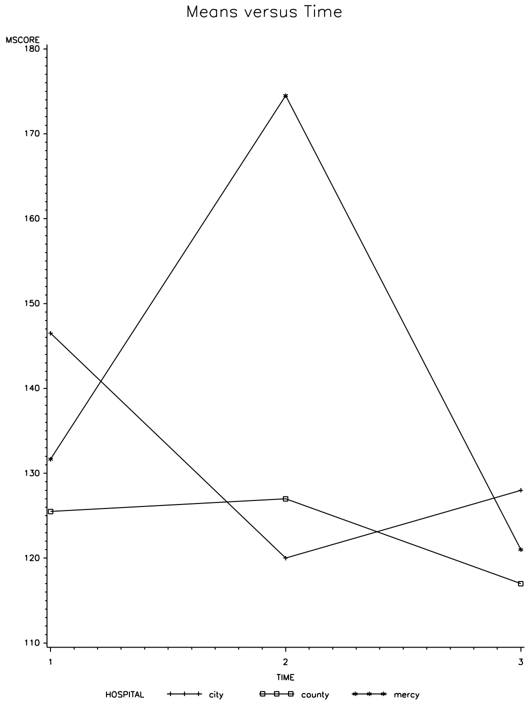
```
hospital   time   score
 mercy       1     132
 mercy       2     125
           . . .            hospital  time  _type_  _freq_   mscore
 mercy       1     144       city       1      3       2    146.500
 mercy       2     224       city       2      3       2    120.000
 county      1     119       city       3      3       2    128.000
 county      2     125   ⟹  county     1      3       2    125.500
           . . .             county     2      3       2    127.000
 county      2     129       county     3      3       2    117.000
 county      3     113       mercy      1      3       3    131.667
 city        1     144       mercy      2      3       2    174.500
 city        2     121       mercy      3      3       1    121.000
           . . .
 city        1     149
 city        3     122
```

The `_type_` variable indicates the observations are for a unique combination of levels of two `class` variables, while the `_freq_` variable is the number of observations which were used for the computation of that statistic.

Plotting the Means

The following program produces the graph shown on the right:

```
symbol1 interpol=join
        value=plus;
symbol2 interpol=join
        value=square;
symbol3 interpol=join
        value=star;
title "Means versus Time";
proc gplot data=hmeans;
plot mscore*time=hospital;
run;
```



Application: Finding Duplicate Observations II

In a previous example, duplicates were found by using `by` processing and `first.` and `last.` variables. If the data set were very large, or not already sorted by `id`, that program would not be very efficient. In this case, an output data set from `proc freq` might be more useful. Once again, assume the identifier variable is called `id`. The following statements will produce a data set with the `id` values of the duplicate observations.

```
proc freq data=old noprint;
   tables id/ out=counts(rename = (count = n) keep=id count);
run;
data check;
   set counts;
   if n > 1;
run;
```

Note that, even though `count` is renamed to `n`, the original variable name (`count`) is used on the `keep` statement.

SAS Macro Language: Overview

At it's simplest level, the SAS Macro language allows you to assign a string value to a variable and to use the variable anywhere in a SAS program:

```
%let header = "Here is my title";
            . . .
proc print ;
   var x y z;
   title &header;
   run;
```

This would produce exactly the same result as if you typed the string `"Here is my title"` in place of `&header` in the program. Notice that the substitution is very simple - the text of the macro variable is substituted for the macro symbol in the program.

SAS Macro Language: Overview (cont'd)

The macro facility can be used to replace pieces of actual programs by creating named macros:

```
%macro readsome;
data one;
  infile "myfile";
  input x y z;
  if
%mend;

%readsome x > 1; run;
```

The final statement is equivalent to typing

```
data one;
  infile "myfile";
  input x y z;
  if x > 1; run;
```

Once again, all that is performed is simple text replacement.

SAS Macro Language: Overview (cont'd)

A large part of the macro facility's utility comes from the macro programming statements which are all preceded by a percent sign (%). For example, suppose we need to create 5 data sets, named `sales1`, `sales2`, etc., each reading from a corresponding data file `insales1`, `insales2`, etc.

```
%macro dosales;
  %do i=1 %to 5;
     data sales&i;
     infile "insales&i";
     input dept $ sales;
     run;
  %end;
%mend dosales;

%dosales;
```

Note that, until the last line is entered, no actual SAS statements are carried out; the macro is only compiled.

Defining and Accessing Macro Variables in the Data Step

You can set a macro variable to a value in the data step using the `call symput` function. The format is

```
call symput(name,value);
```

where `name` is a string or character variable containing the name of the macro variable to be created, and `value` is the value the macro variable will have.

To access a macro variable in a data step, you can use the `symget` function.

```
value = symget(name);
```

`name` is a string or character variable containing the name of the macro variable to be accessed.

`call symput`: Example

Suppose we want to put the maximum value of a variable in a title. The following program shows how.

```
data new;
  retain max -1e20;
  set salary end = last;
  if salary > max then max = salary;
  if last then call symput("maxsal",max);
  drop max;
  run;

title "Salaries of employees (Maximum = &maxsal)";
proc print data=salary;
run;
```

Note that no ampersand is used in `call symput`, but you must use an ampersand to reference the macro variable later in your program.

An Alternative to the macro facility

As an alternative to the previous example, we can use the `put` statement to write a SAS program, and then use the `%include` statement to execute the program. Using this technique, the following statements recreate the previous example:

```
proc means data=salary noprint;
  var salary;
  output out=next max=maxsal;
data _null_;
  set next;
  file "tmpprog.sas";
  put 'title "Salaries of employees (Maximum =' maxsal ')";';
  put 'proc print data=salary;';
  put 'run;';
run;
%include "tmpprog.sas";
```

Pay special attention to quotes and semicolons in the generated program.

Another Alternative to the Macro Facility

In addition to writing SAS statements to a file, SAS provides the `call execute` function. This function takes a quoted string, a character variable, or a SAS expression which resolves to a character variable and then executes its input when the current data step is completed.

For example, suppose we have a data set called `new` which contains a variable called `maxsal`. We could generate a title statement containing this value with statements like the following.

```
data _null_;
set new;
call execute('title
   "Salaries of employees (Maximum = '|| put(maxsal,6.) || ')";');
run;
```


`call execute` Example

As a larger example of the use of the `call execute function`, consider the problem of reading a list of filenames from a SAS data set and constructing the corresponding data steps to read the files. The following program performs the same function as the earlier macro example.

```
data _null_;
   set files;
   call execute('data ' || name || ';');
   call execute('infile "'|| trim(name) || '";');
   call execute('input x y;');
   call execute('run;');
run;
```

Be careful with single and double quotes, and make sure the generated statements follow the rules of SAS syntax.

Macros with Arguments (cont'd)

To add arguments to a macro, simply replace the parts of the program in question with macro variables (beginning with `&`), and list the variables in the argument list (without the `&`).

```
%macro prntdups(data,by);
proc summary data=&data nway ;
  class &by;
  output out=next(keep = &by  _freq_ rename=(_freq_ = count));
run;
data dups;
  merge &data next;
  by &by;
  if count > 1;
run;
proc print data=dups;
run;
%mend prntdups;
```

Then the program on the previous slide would be replaced by:

```
%prntdups(one,a b);
```


Application: Reading a Series of Files (cont'd)

Now we can write a macro to loop over the previously defined file names and create the data sets.

```
%macro readem;
   %do i=1 %to &num;
      data &&file&i;
      infile "&&file&i";
      input x y;
      run;
   %end;
%mend;
%readem;
```

Notice that the macro variable is refered to as `&&file&i`, to force the macro substitution to be scanned twice. If we used just a single ampersand, SAS would look for a macro variable called `&file`.

Macros with Arguments

Consider the following program to print duplicate cases with common values of the variables `a` and `b` in data set `one`:

```
data one;
  input a b y @@;
datalines;
1 1 12 1 1 14 1 2 15 1 2 19 1 3 15 2 1 19 2 4 16 2 4 12 2 8 18 3 1 19
proc summary data=one nway ;
  class a b;
  output out=next(keep = a b _freq_ rename=(_freq_ = count));
data dups;
  merge one next;
  by a b;
  if count > 1;
proc print data=dups;
run;
```

If we had simple way of changing the input data set name and the list of variables on the `by` statement, we could write a general macro for printing duplicates in a data set.

Application: Reading a Series of Files

Suppose we have a data set containing the names of files to be read, and we wish to create data sets of the same name from the data in those files. First, we use the `call symput` function in the data step to create a series of macro variables containing the file names

```
data _null_;
  set files end=last;
  n + 1;
  call symput("file"||left(n),trim(name));
  if last then call symput("num",n);
  run;
```

Since macros work by simple text substitution, it is important that there are no blanks in either the macro name or value, thus the use of `left` and `trim`

Accessing Operating System Commands

If you need to run an operating system command from inside a SAS program, you can use the `x` command. Enclose the command in quotes after the `x`, and end the statement with a semicolon. The command will be executed as soon as it is encountered.

For example, in an earlier program, a file called `tmpprog.sas` was created to hold program statements which were later executed. To remove the file after the statements were executed (on a UNIX system) you could use the SAS statement:

```
x 'rm tmpprog.sas';
```

Other interfaces to the operating system may be available. For example, on UNIX systems the `pipe` keyword can be used on a `filename` statement to have SAS read from or write to a process instead of a file. See the SAS Companion for your operating system for more details.

Transporting Data Sets

It is sometimes necessary to move a SAS data set from one computer to another. The internal format of SAS data sets is *not* the same on all computers, so to make it possible to transfer data sets from one computer to another, SAS provides what is known as a *transport format*. Whenever you move a SAS data set from one computer to another, you must first convert it into transport format.

Keep in mind that SAS data sets are in general readable only by SAS. Thus, an alternative (or perhaps a backup) method for transporting data sets is to write them in a human-readable way, using, for example, put statements. Human-readable files can be processed by SAS (or some other program), and are generally easier to move around than SAS transport data sets.

SAS/CONNECT

SAS also provides a product called SAS/CONNECT which lets you initiate a SAS job on a remote computer from a local SAS display manager session. It also provides two procedures, `proc upload` and `proc download` to simplify transporting data sets. If SAS/CONNECT is available on the machines between which the data set needs to be moved, it may be the easiest way to move the data set.

SAS/CONNECT must be run from the display manager. When you connect with the other system, you will be prompted for a login name and a password (if appropriate). Once you're connected, the `rsubmit` display manager command will submit jobs to the remote host, even though the log and output will be managed by the local host.

Creating a Dataset in transport format

`proc copy` can be used to create a transport format file, but the critical step is to use the `xport` keyword in the `libname` statement. The specified libname is then the name of the transport format file which SAS will create, not a directory as is usually the case.

Suppose we wanted to create a SAS transport data file named `move.survey` from a SAS data set named `save.results`.

```
libname save "/my/sas/dir";
libname move xport "move.survey";
proc copy in=save out=move;
  select results;
  run;
```

If you transfer the transport data set using a program like ftp, **make sure that you use binary (image) mode to transfer the file**.

`proc transpose`

Occasionally it is useful to switch the roles of variables and observations in a data set. Since the data step doesn't "know" the names of the variables in the data step, this can be tricky to do through data step commands. The `proc transpose` program takes care of this task.

As an example, suppose we wish to write a program which will print a line like:

```
The mean of variable x is 35.3
```

for each variable in a data set. We can create an output data set from `proc means`, transpose it so that the variables become observations, and then print the desired information in a data step. The variable created by `proc transpose` which contains the variable names is labelled `_name_` and the variables representing the observations are labelled `col1`, `col2`, etc.

Using `proc transpose`

The following statements will create a data set called `means` containing the means of the variables in the data set `one`. Then data set `means` is transposed, and a report is printed.

```
proc means noprint data=one;
  output out=means(drop = _type_ _freq_)  mean=;
run;
proc transpose data=means out=use;
run;
data _null_;
  set use;
  put 'The mean of variable ' _name_ 'is ' col1;
run;
```

The output looks like this:

```
The mean of variable X is 0.04095916
The mean of variable Y is -0.067345303
The mean of variable Z is -0.080134777
```

What `proc transpose` does

To understand the workings of `proc transpose`, consider a data set with four observations and three variables (`x`, `y` and `z`). Here's the transformation `proc transpose` performs:

Original data

X	Y	Z
12	19	14
21	15	19
33	27	82
14	32	99

$\Longrightarrow$

Transposed data

NAME	COL1	COL2	COL3	COL4
X	12	21	33	14
Y	19	15	27	32
Z	14	19	82	99

Many more complex transformations can also be performed using `proc transpose`, especially with the use of a `by` statement, because the structure of the `by` variables is maintained in the output data set. See the SAS manual or the sample library for examples.

`proc contents`

Since SAS data sets cannot be read like normal files, it is important to have tools which provide information about data sets. `proc print` can show what's in a data set, but it not always be appropriate. The `var` and `libname` windows of the display manager are other useful tools, but to get printed information or to manipulate that information, you should use `proc contents`.

Among other information, `proc contents` provides the name, type, length, format, informat and label for each variable in the data set, as well as the creation date and time and the number of observations. To use `proc contents`, specify the data set name as the `data=` argument of the `proc contents` statement; to see the contents of all the data sets in a directory, define an appropriate libname for the directory, and provide a data set name of the form *`libname`*`._all_`.

Options for `proc contents`

The `short` option limits the output to just a list of variable names.

The `position` option orders the variables by their position in the data set, instead of the default alphabetical order. This can be useful when working with double dashed lists.

The `directory` option provides a list of all the data sets in the library that the specified data set comes from, along with the usual output for the specified data set.

The `nods` option, when used in conjunction with a data set of the form *`libname`*`._all_`, limits the output to a list of the data sets in the specified *`libname`*, with no output for the individual data sets.

The `out=` option specifies the name of an output data set to contain the information about the variables in the specified data set(s). The program on the next slide uses this data set to write a human readable version of a SAS data set.

Using the output data set from `proc contents`

```
%macro putdat(indata,outfile);
proc contents noprint data=&indata out=fcon; run;
data _null_;
   file "tmpprog.sas";
   set fcon end=last;
   length form $ 8;
   if _n_ = 1 then do;
      put "data _null_;"/"set &indata;";
      put 'file "&outfile";' / "put " @;
      end;
   put name @;
   if type = 2 then
      form = "$"||trim(left(put(length,3.)))||".";
   else form = "best12.";
   put form @;
   if ^last then put "+1 " @;
   else put ";" / "run;" ;
run;
%include "tmpprog.sas";
x 'rm tmpprog.sas';
%mend putdat;
```

The Display Manager

When SAS is invoked, it displays three windows to help you interact with your programs and output:

- Program Window - for editing and submitting SAS statements
- Log Window - for viewing and saving log output
- Output Window - for viewing and saving output

Some commands which open other useful windows include:

- `assist` - menu driven version of SAS
- `dir` - shows data sets in a library
- `var` - shows variables in a data set
- `notepad` - simple text window
- `options` - view and change system options
- `filename` - view current filename assignments
- `help` - interactive help system
- `libname` - view current libname assignments

Appearance of Display Manager

```
OUTPUT-------------------------------------------------------------------------+
|Command ===>                                                                  |
|                                                                              |
|                                                                              |
|                                                                              |
|                                                                              |
|                                                                              |
|                                                                              |
|                                                                              |
|                                                                              |
|                                                                              |
|                                                                              |
LOG----------------------------------------------------------------------------+
|File Edit View Locals Globals Help                                            |
|                                                                              |
|This is version 6.11 of the SAS System.                                       |
|                                                                              |
|NOTE: AUTOEXEC processing beginning; file is /app/sas611/autoexec.sas.        |
|                                                                              |
|NOTE: SAS initialization used:                                                |
|      real time           0.567 seconds                                       |
|      cpu time            0.421 seconds                                       |
|                                                                              |
PROGRAM EDITOR-----------------------------------------------------------------+
|Command ===>                                                                  |
|                                                                              |
|00001                                                                         |
|00002                                                                         |
|00003                                                                         |
|00004                                                                         |
|00005                                                                         |
|00006                                                                         |
|00007                                                                         |
|00008                                                                         |
|00009                                                                         |
|00010                                                                         |
+------------------------------------------------------------------------------+
```

Entering Display Manager Commands

You can type display manager commands on the command line of any display manager window. (To switch from menu bar to command line select `Globals -> Options -> Command line`; to switch back to menu bar, enter the `command` command.)

You can also enter display manager commands from the program editor by surrounding them in quotes, and preceding them by `dm`, provided that the display manager is active.

Some useful display manager commands which work in any window include:

- `clear` - clear the contents of the window
- `end` - close the window
- `endsas` - end the sas session
- `file "`*`filename`*`"` - save contents of the window to *`filename`*
- `prevcmd` - recall previous display manager command

The Program Editor

There are a number of special display manager commands available in the program editor.

- `submit` - submit all the lines in the editor to SAS
- `subtop` - submit the first line in the editor to SAS
- `recall` - place submitted statements back in the editor
- `include "`*`file`*`"` - place the contents of *`file`* in the editor
- `hostedit` - on UNIX systems, invoke the editor described in the `editcmd` system option

Typing control-`x` when the cursor is in the program editor toggles between insert and overwrite mode.

You can close the program window with the display manager command `program off`.

Using the Program Editor

There are two types of commands which can be used with the program editor

- Command line commands are entered in the `Command ===>` prompt, or are typed into a window when menus are in effect.
- Line commands are entered by typing over the numbered lines on the left hand side of the editor window. Many of the line commands allow you to operate on multiple selected lines of text.

In addition, any of the editor or other display manager commands can be assigned to a function or control key, as will be explained later.

Note: The `undo` command can be used to reverse the effect of editing commands issued in the display manager.

Editor Line Commands

Commands followed by `<n>` optionally accept a number to act on multiple lines.

Inserting Lines		Deleting Lines	
`i<n>`	insert after current line	`d<n>`	delete lines
`ib<n>`	insert before current line	`dd`	block delete
Moving Lines		**Copying Lines**	
`m<n>`	move lines	`c<n>`	copy lines
`mm`	block move	`cc`	block copy
Other Commands			
`>><n>`	indent lines	`<<<n>`	remove indentation
`tc`	connect text	`ts`	split text

Type block commands on the starting and ending lines of the block and use a `b` or `a` command to specify the a line before which or after which the block should be placed.

Defining Function and Control Keys

You can define function keys, control keys, and possibly other keys depending on your operating system, through the `keys` window of the display manager.

To define a function key to execute a display manager command, enter the name of the command in the right hand field next to the key you wish to define.

To define a function key to execute an editor line command, enter the letter(s) corresponding to the command preceded by a colon (`:`) in the right hand field.

To define a function key to insert text into the editor, precede the text to be inserted with a tilda (`~`) in the right hand field.

Some display manager commands only make sense when defined through keys. For example the command `home` puts the cursor on the command line of a display manager window.

More on Function Keys

You can define a function key to perform several commands by separating the commands with a semicolon (`;`).

Function keys defining display manager commands are executed *before* any text which is on the command line. Thus, you can enter arguments to display manager commands before hitting the appropriate function key.

To set function keys without using the `keys` window, use the display manager `keydef` command, for example:

```
keydef f2 rfind
```

Keys set in this way will only be in effect for the current session.

Cutting and Pasting

If block moves and/or copies do not satisfy your editing needs, you can cut and paste non-rectangular blocks of text. Using these commands generally requires that keys have been defined for the display manager commands `mark`, `cut`, and `paste`, or `home`.

To define a range of text, issue the `mark` command at the beginning and end of the text you want cut or pasted. Then issue the `cut` (destructive) or `store` (non-destructive) command. Finally, place the cursor at the point to which you want the text moved, and issue the `paste` command.

When using `cut` or `store`, you can optionally use the `append` option, which allows you to build up the contents of the paste buffer with several distinct cuts or copies, or the `buffer=`*`name`* option, to create or use a named buffer.

Searching and Changing Text

The display manager provides the following commands for searching and, in the program editor or notepad, changing text. These commands include:

- `find` *`string`* - search forward for *`string`*
- `bfind` *`string`* - search backward for *`string`*
- `rfind` - repeat previous find command
- `change` *`old new`* - change *`old`* to *`new`*
- `rchange` - repeat previous change command

Each of these commands takes a variety of options:
Scope of Search: `next`, `first`, `last`, `prev`, `all`
Component of Search: `word`, `prefix`, `suffix`
Case Independence: `icase`

If there are blanks or special symbols in any of the strings, surround the entire string in single or double quotes.

Using the Find and Change Commands

- To change every occurence of the string "sam" to "fred", ignoring the case of the first string, enter

  ```
  change sam fred all icase
  ```

- To selectively change the word `cat` to `dog`, use

  ```
  change cat dog word
  ```

  followed by repeated use of `rfind`, to find the next occurence of the word, and `rchange` if a change is desired.

- To count the number of occurences of the word `fish`, use

  ```
  find fish all
  ```

  and the count will be displayed under the command line.

If an area of text is marked (using the display manager `mark` command), then search and/or find commands apply only to the marked region.

Customizing the Display Manager

Key definitions entered through the `keys` window are automatically stored from session to session.

The `color` display manager command allows you to customize the colors of various components of SAS windows. The `sascolor` window allows you to change colors through a window.

To save the geometry and location of display manager windows, issue the display manager command `wsave` from within a given window, or `wsave all` to save for all windows.