Basic Concepts #7
Flow Control and Arrays

JC Wang
Outline

1. DO Loops
   - Three Types of DO Loops
   - DO WHILE Loops
   - DO UNTIL Loops
   - Indexed DO Loops
   - DO Loops and SAS Output Data Set

2. Arrays in SAS Data Step
   - Use of Arrays
   - Declarations of Arrays
   - Examples
   - ArrayReferencing

3. Implicit Arrays
   - Implicit Arrays
Three Types of DO Loops

- **DO WHILE Loops:**
  DO WHILE (expression);
  ...SAS statements...
  END;

- **DO UNTIL Loops:**
  DO UNTIL (expression);
  ...SAS statements...
  END;

- **Indexed DO Loops:**
  DO index-variable=spec-1<,... spec-n>
  <WHILE(expression) | UNTIL(expression)>
  ...SAS statements...
  END;

Warning: Beware of infinite loop!
Three Types of DO Loops

- **DO WHILE Loops:**
  
  ```sas
  DO WHILE (expression);
  ...SAS statements...
  END;
  ```

- **DO UNTIL Loops:**
  
  ```sas
  DO UNTIL (expression);
  ...SAS statements...
  END;
  ```

- **Indexed DO Loops:**
  
  ```sas
  DO index-variable=spec-1,... spec-n>
  <WHILE(expression) | UNTIL(expression)>;
  ...SAS statements...
  END;
  ```

**Warning:** Beware of infinite loop!
Three Types of DO Loops

- **DO WHILE Loops:**
  
  ```sas
  DO WHILE (expression);
  ...SAS statements...
  END;
  ```

- **DO UNTIL Loops:**
  
  ```sas
  DO UNTIL (expression);
  ...SAS statements...
  END;
  ```

- **Indexed DO Loops:**
  
  ```sas
  DO index-variable=spec-1<,...,spec-n> <WHILE(expression) | UNTIL(expression)>;
  ...SAS statements...
  END;
  ```

**Warning:** Beware of infinite loop!
Three Types of DO Loops

- **DO WHILE Loops:**
  ```sas
  DO WHILE (expression);
  ...SAS statements...
  END;
  ```

- **DO UNTIL Loops:**
  ```sas
  DO UNTIL (expression);
  ...SAS statements...
  END;
  ```

- **Indexed DO Loops:**
  ```sas
  DO index-variable=spec-1<,... spec-n>
  <WHILE(expression) | UNTIL(expression)>
  ...SAS statements...
  END;
  ```

Warning: Beware of infinite loop!
Three Types of DO Loops

- **DO WHILE Loops:**
  
  ```sas
  DO WHILE (expression);
  ...SAS statements...
  END;
  ```

- **DO UNTIL Loops:**
  
  ```sas
  DO UNTIL (expression);
  ...SAS statements...
  END;
  ```

- **Indexed DO Loops:**
  
  ```sas
  DO index-variable=spec-1<,... spec-n> <WHILE(expression) | UNTIL(expression)>;
  ...SAS statements...
  END;
  ```

**Warning:** Beware of infinite loop!
DO WHILE Loops

Executes statements repetitively while a condition is true. The condition in *expression* is evaluated first. The SAS statements in the loop will then be executed if the condition is true. Otherwise, SAS exits the loop.
DO UNTIL Loops

Executes statements in a DO loop repetitively until a condition is true. The SAS statements in the loop are executed first, then the condition in *expression* is checked. If the condition is false, SAS starts next iteration. Otherwise, SAS exits the loop.
Indexed DO Loops

Index Variable

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

- Names a variable with values governing the execution of the loop.
- Index variable is part of the data set that you create in data step, use `DROP` statement (after loop) or `DROP=` data set option in `DATA` statement.
- The index variable can be modified within loop. However, exercise extreme caution when you do that. It is recommended to avoid the modification of index variable entirely.
Indexed DO Loops

Index Variable

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

- Names a variable with values governing the execution of the loop.
- Index variable is part of the data set that you create in data step, use `DROP` statement (after loop) or `DROP=` data set option in `DATA` statement.
- The index variable can be modified within loop. However, exercise extreme caution when you do that. It is recommended to avoid the modification of index variable entirely.
Indexed DO Loops

Specification

Each specification \((\text{spec})\) can be

- a single item of constant (numeric or character) or expression that yields value;
- or of the form \(\text{start TO stop <BY increment>}\)

where \(\text{start, stop, and increment}\) must be numbers (numeric) or expressions that yield numbers;

Note that

1. The value of \(\text{increment}\) is evaluated first followed by the execution of the loop. Hence the modification of its value within the loop has no effect on the number of iterations.
2. All the specifications must be of the same type (numeric or character).
Indexed DO Loops

Specification

Each specification \( spec \) can be

- a single item of constant (numeric or character) or expression that yields value;
- or of the form

\[
\text{start TO stop <BY increment>}
\]

where \( \text{start, stop, and increment} \) must be numbers (numeric) or expressions that yield numbers;

Note that

1. The value of \( \text{increment} \) is evaluated first followed by the execution of the loop. Hence the modification of its value within the loop has no effect on the number of iterations.
2. All the specifications must be of the same type (numeric or character).
Indexed DO Loops
WHILE and UNTIL Clause

The **WHILE** and **UNTIL** clause apply only to the last specification. See example **DoLoop.sas**.
CONTINUE and LEAVE Statements

- The **CONTINUE** statement stops the processing of the current iteration of a loop and resumes with the next iteration. Note: It can only be used in a DO loop.
  
  \[
  \text{IF } \text{cond} \text{ THEN CONTINUE;} \]

  If the condition \((\text{cond})\) is true then SAS starts next iteration immediately.

- The **LEAVE** statement causes processing of the current loop to end. Note: It can be used in a DO loop or a SELECT group (within a DO-group following \text{WHEN} statement).
  
  \[
  \text{IF } \text{cond} \text{ THEN LEAVE;} \]

  If the condition \((\text{cond})\) is true then SAS terminates the loop immediately.
CONTINUE and LEAVE Statements

- The **CONTINUE** statement stops the processing of the current iteration of a loop and resumes with the next iteration. Note: It can only be used in a DO loop.

  ```sas
  IF cond THEN CONTINUE;
  ```

  If the condition (cond) is true then SAS starts next iteration immediately.

- The **LEAVE** statement causes processing of the current loop to end. Note: It can be used in a DO loop or a SELECT group (within a DO-group following **WHEN** statement).

  ```sas
  IF cond THEN LEAVE;
  ```

  If the condition (cond) is true then SAS terminates the loop immediately.
CONTINUE and LEAVE Statements

- The `CONTINUE` statement stops the processing of the current iteration of a loop and resumes with the next iteration. Note: It can only be used in a DO loop.
  
  ```sas
  IF cond THEN CONTINUE;
  ```

  If the condition \( cond \) is true then SAS starts next iteration immediately.

- The `LEAVE` statement causes processing of the current loop to end. Note: It can be used in a DO loop or a SELECT group (within a DO-group following `WHEN` statement).
  
  ```sas
  IF cond THEN LEAVE;
  ```

  If the condition \( cond \) is true then SAS terminates the loop immediately.
DO Loops and SAS Output Data Set

- Note: PDV (Program Data Vector) will be written, by default, to the SAS data set at the end of DATA step loop (not any of the 3 types discussed here).

- To write PDV to SAS data set in any iteration of any of the three types of DO loops, do one of the following:
  - OUTPUT; statement by itself at the end of a DO loop
  - using OUTPUT; statement in conditional execution (IF-THEN-ELSE construct or SELECT construct)
DO Loops and SAS Output Data Set

- Note: PDV (Program Data Vector) will be written, by default, to the SAS data set at the end of DATA step loop (not any of the 3 types discussed here).
- To write PDV to SAS data set in any iteration of any of the three types of DO loops, do one of the following:
  - OUTPUT; statement by itself at the end of a DO loop
  - using OUTPUT; statement in conditional execution (IF-THEN-ELSE construct or SELECT construct)
DO Loops and SAS Output Data Set

- Note: PDV (Program Data Vector) will be written, by default, to the SAS data set at the end of DATA step loop (not any of the 3 types discussed here).

- To write PDV to SAS data set in any iteration of any of the three types of DO loops, do one of the following:
  - `OUTPUT;` statement by itself at the end of a DO loop
  - using `OUTPUT;` statement in conditional execution (IF-THEN-ELSE construct or SELECT construct)
Note: PDV (Program Data Vector) will be written, by default, to the SAS data set at the end of DATA step loop (not any of the 3 types discussed here).

To write PDV to SAS data set in any iteration of any of the three types of DO loops, do one of the following:

- `OUTPUT;` statement by itself at the end of a DO loop
- using `OUTPUT;` statement in conditional execution (IF-THEN-ELSE construct or SELECT construct)
Outline

1. DO Loops
   - Three Types of DO Loops
   - DO WHILE Loops
   - DO UNTIL Loops
   - Indexed DO Loops
   - DO Loops and SAS Output Data Set

2. Arrays in SAS Data Step
   - Use of Arrays
   - Declarations of Arrays
   - Examples
   - Array Referencing

3. Implicit Arrays
   - Implicit Arrays
Purpose and Declarations of Arrays

Purpose: simplify SAS code in dealing with similar/same actions.

Two forms of array declarations:

1. To associate data set variables:
   ```sas
   ARRAY array-name[subscript] <=S> <Length>
   <array-elements> <= (initial-value-list)>
   ```

2. To create a temporary array:
   ```sas
   _temporary_< <= (initial-value-list)>
   ```
Purpose and Declarations of Arrays

- **Purpose**: simplify SAS code in dealing with similar/same actions.
- **Two forms of array declarations**:
  - To associate data set variables:
    ```sas
    ARRAY array-name[subscript] <$> <length> <array-elements> <(initial-value-list)>;
    ```
  - To create a temporary array:
    ```sas
    ARRAY array-name[subscript] <$> <length> _TEMPORARY_ <(initial-value-list)>;
    ```
Purpose and Declarations of Arrays

- **Purpose**: simplify SAS code in dealing with similar/same actions.
- **Two forms of array declarations**:
  1. to associate data set variables:
     ```sas
     ARRAY array-name[subscript] <$> <length> <array-elements> <(initial-value-list)>;
     ```
  2. to create a temporary array:
     ```sas
     ARRAY array-name[subscript] <$> <length> _TEMPORARY_ <(initial-value-list)>;
     ```
Purpose and Declarations of Arrays

- Purpose: simplify SAS code in dealing with similar/same actions.

- Two forms of array declarations:
  1. to associate data set variables:
     ```sas
     ARRAY array-name[subscript] <$> <length> <array-elements> <(initial-value-list)>;
     ```
  2. to create a temporary array:
     ```sas
     ARRAY array-name[subscript] <$> <length> _TEMPORARY_ <(initial-value-list)>;
     ```
Purpose and Declarations of Arrays

- **Purpose**: simplify SAS code in dealing with similar/same actions.

- **Two forms of array declarations**:
  1. **to associate data set variables**:
     ```sas
     ARRAY array-name[subscript] <$> <length> <array-elements> <(initial-value-list)>;
     ```
  2. **to create a temporary array**:
     ```sas
     ARRAY array-name[subscript] <$> <length> _TEMPORARY_ <(initial-value-list)>;
     ```
Declarations of Arrays

array-name

array-name is a valid SAS name. Avoid using

- same name as an existing variable’s name in the current data step;
- a SAS function name, otherwise, such function cannot be used in the same data step.

Note: arrays exist only for the duration of current data step.
Declarations of Arrays

array-name

array-name is a valid SAS name. Avoid using

- same name as an existing variable’s name in the current data step;
- a SAS function name, otherwise, such function cannot be used in the same data step.

Note: arrays exist only for the duration of current data step.
Declarations of Arrays

array-name

array-name is a valid SAS name. Avoid using

- same name as an existing variable’s name in the current data step;
- a SAS function name, otherwise, such function cannot be used in the same data step.

Note: arrays exist only for the duration of current data step.
Declarations of Arrays

[subscript] describes the number and arrangement of elements in the array. Use one of the forms below for subscript:

- `dimension-size<,dimension-size<,...>>` specifies dimension size(s) of the array, dimensions are separated by commas in a multi-dimensional array;
- `<lower:>>upper<,lower:>>upper<,...>>` specifies lower bound (default is 1) and upper bound for each dimension;
- `*` (an asterisk) instructs SAS to determine the dimension size (in a 1-dimensional array) by counting the variables in the array (from array-elements).

The brackets `[ ]` can be replaces by braces `{ }` or parentheses `( )`. 
Declarations of Arrays

[subscript] describes the number and arrangement of elements in the array. Use one of the forms below for subscript:

- \( \text{dimension-size}, \text{dimension-size}<, \ldots> \) specifies dimension size(s) of the array, dimensions are separated by commas in a multi-dimensional array;
- \(<\text{lower} : \text{upper}>, <\text{lower} : \text{upper}>, \ldots>\) specifies lower bound (default is 1) and upper bound for each dimension;
- * (an asterisk) instructs SAS to determine the dimension size (in a 1-dimensional array) by counting the variables in the array (from array-elements).

The brackets \([\ ]\) can be replaced by braces \(\{\}\) or parentheses \((\ )\).
Declarations of Arrays

[subscript] describes the number and arrangement of elements in the array. Use one of the forms below for subscript:

- `dimension-size<,dimension-size<,...>>` specifies dimension size(s) of the array, dimensions are separated by commas in a multi-dimensional array;

- `<lower :>upper<,lower :>upper<,...>>` specifies lower bound (default is 1) and upper bound for each dimension;

- `*` (an asterisk) instructs SAS to determine the dimension size (in a 1-dimensional array) by counting the variables in the array (from array-elements).

The brackets `[ ]` can be replaced by braces `{ }` or parentheses `()`. 
Declarations of Arrays

\textit{length} specifies the length of elements in the array that have not been previously assigned a length.
Declarations of Arrays

array-elements

array-elements names the elements (of the same types, numeric or character) that make up the array. Elements can be:

1. variables’ names;
2. _NUMERIC_ or _CHARACTER_ or _ALL_;
3. variable1-_NUMERIC_ -variable2 or variable1-_CHARACTER_ -variable2, variable1- -variable2, where variable1 and variable2 are two existing variables (of the same type for last form).

Note:

- In the first form, variables can be either existing or new.
- If array-elements is omitted, then the array associate with variables array-name1 to array-name m (m is array size).
Declarations of Arrays

array-elements

array-elements names the elements (of the same types, numeric or character) that make up the array. Elements can be

1. variables’ names;
2. _NUMERIC_ or _CHARACTER_ or _ALL_;
3. variable1-_NUMERIC_-variable2 or
   variable1-_CHARACTER_-variable2, variable1--variable2, where variable1 and variable2 are two existing variables (of the same type for last form).

Note:

- In the first form, variables can be either existing or new.
- If array-elements is omitted, then the array associate with variables array-name1 to array-name m (m is array size).
Declarations of Arrays

array-elements

array-elements names the elements (of the same types, numeric or character) that make up the array. Elements can be

1. variables’ names;
2. _NUMERIC_ or _CHARACTER_ or _ALL_;
3. variable1-_NUMERIC_~variable2 or variable1-_CHARACTER_~variable2, variable1--variable2, where variable1 and variable2 are two existing variables (of the same type for last form).

Note:

- In the first form, variables can be either existing or new.
- If array-elements is omitted, then the array associate with variables array-name1 to array-name m (m is array size).
Declarations of Arrays

**initial-value-list**

`initial-value-list` gives initial values (separated by , or spaces) for the corresponding elements in the array.

- Elements and initial values are matched by positions. If more elements are specified, then the unmatched elements have missing value as initial values.
- Elements with non-missing initial values are retained until you change them.
- If the attributes (such as type or length) of array elements are not previously specified, the initial values determine their attributes.
- Can use the following short-cut for `initial-value-list`:
  
  - `<constant-iter-value>`*: `<constant-value`  
  - `<constant-iter-value>`*: `<constant-sublist>`  
  - `<low>:<high>` for increasing sequence of integers from low to high.
Declarations of Arrays

initial-value-list

initial-value-list gives initial values (separated by , or spaces) for the corresponding elements in the array.

- Elements and initial values are matched by positions. If more elements are specified, then the unmatched elements have missing value as initial values.
- Elements with non-missing initial values are retained until you change them.
- If the attributes (such as type or length) of array elements are not previously specified, the initial values determine their attributes.
- Can use the following short-cut for initial-value-list:
  - constant-iter-value*: constant-iter-value or constant-value (constant-iter-value and list)
  - low:high: for increasing sequence of integers from low to high.
Declarations of Arrays

initial-value-list

*initial-value-list* gives initial values (separated by , or spaces) for the corresponding elements in the array.

- Elements and initial values are matched by positions. If more elements are specified, then the unmatched elements have missing value as initial values.
- Elements with non-missing initial values are retained until you change them.
- If the attributes (such as type or length) of array elements are not previously specified, the initial values determine their attributes.
- Can use the following short-cut for initial-value-list:
  - `constant-iter-value*`
  - `constant-value` or `constant-sublist`
  - `low: high` for increasing sequence of integers from low to high.
Declarations of Arrays

initial-value-list

initial-value-list gives initial values (separated by , or spaces) for the corresponding elements in the array.

- Elements and initial values are matched by positions. If more elements are specified, then the unmatched elements have missing value as initial values.
- Elements with non-missing initial values are retained until you change them.
- If the attributes (such as type or length) of array elements are not previously specified, the initial values determine their attributes.

Can use the following short-cut for initial-value-list:

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

low : high for increasing sequence of integers from low to high.
Declarations of Arrays

initial-value-list
gives initial values (separated by , or spaces)
for the corresponding elements in the array.

- Elements and initial values are matched by positions. If more elements are specified, then the unmatched elements have missing value as initial values.
- Elements with non-missing initial values are retained until you change them.
- If the attributes (such as type or length) of array elements are not previously specified, the initial values determine their attributes.
- Can use the following short-cut for initial-value-list:
  - `<constant-iter-value*> <(constant-value | constant-sublist)>`
  - `low:high` for increasing sequence of integers from low to high.
Declarations of Arrays

initial-value-list
gives initial values (separated by , or spaces) for the corresponding elements in the array.

- Elements and initial values are matched by positions. If more elements are specified, then the unmatched elements have missing value as initial values.
- Elements with non-missing initial values are retained until you change them.
- If the attributes (such as type or length) of array elements are not previously specified, the initial values determine their attributes.
- Can use the following short-cut for initial-value-list:
  - `<constant-iter-value*> <(>constant-value | constant-sublist<)>`
  - `low:high` for increasing sequence of integers from low to high.
Declarations of Arrays

initial-value-list
gives initial values (separated by , or spaces) for the corresponding elements in the array.

- Elements and initial values are matched by positions. If more elements are specified, then the unmatched elements have missing value as initial values.
- Elements with non-missing initial values are retained until you change them.
- If the attributes (such as type or length) of array elements are not previously specified, the initial values determine their attributes.
- Can use the following short-cut for initial-value-list:
  - `<constant-iter-value*> <(constant-value | constant-sublist)>`
  - `low:high` for increasing sequence of integers from `low` to `high`.
Declarations of Arrays
Temporary Arrays

Declared by second form (with key word `_TEMPORARY_`).

- Arrays of temporary elements are useful when the only purpose for creating an array is to perform a calculation.
- They can be used to improve performance time.
- To preserve the result of the calculation from a temporary array element, assign it to a DATA step variable.
Declarations of Arrays
Temporary Arrays

Declared by second form (with key word `_TEMPORARY_`).

- Arrays of temporary elements are useful when the only purpose for creating an array is to perform a calculation.
- They can be used to improve performance time.
- To preserve the result of the calculation from a temporary array element, assign it to a DATA step variable.
Declarations of Arrays
Temporary Arrays (continued)

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

- They do not have names. Refer to temporary array elements by the array name and dimension.
- They do not appear in the output data set.
- The special subscript asterisk (\(\ast\)) cannot be used to refer to all the elements.
- Temporary array element values are always automatically retained, rather than being reset to missing at the beginning of the next DATA step iteration.
Declarations of Arrays
Temporary Arrays (continued)

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

- They do not have names. Refer to temporary array elements by the array name and dimension.
- They do not appear in the output data set.
- The special subscript asterisk (*) cannot be used to refer to all the elements.
- Temporary array element values are always automatically retained, rather than being reset to missing at the beginning of the next DATA step iteration.
Declarations of Arrays

Examples

1. ARRAY color[3] red green blue;
2. ARRAY A[2,3] score1-score6;
3. ARRAY q[2:3,3]; (Note: row-wise)
4. ARRAY num[*] _NUMERIC_;
5. ARRAY char[*] $ a__CHARACTER__final;
6. ARRAY tmp[3] $ 1 _TEMPORARY_ ('a','b','c');
7. ARRAY expense[*] grocery--travel;
8. ARRAY h[6] (1:3 3*0);

See example Array.sas;
Referencing Array Elements

array-name[subscript]; describes elements in array to be processed. subscript specifies subscript in either form:

- variable-1<,variable-2...> indicates variable/variable list that is usually used with DO-loop processing.
- expression-1<,expression-2...> indicates SAS expressions that evaluate to subscript values when the statement that contains the array reference executes.

* forces SAS to treat the elements in the array as a variable list which can be used in INPUT and PUT statements and some SAS functions.

Warning: * cannot be used to reference temporary array element.

See example Array.sas
Referencing Array Elements

array-name[subscript]; describes elements in array to be processed. subscript specifies subscript in either form:

- variable-1<,variable-2...> indicates variable/variable list that is usually used with DO-loop processing.

- expression-1<,expression-2...> indicates SAS expressions that evaluate to subscript values when the statement that contains the array reference executes.

* forces SAS to treat the elements in the array as a variable list which can be used in INPUT and PUT statements and some SAS functions.

Warning: * cannot be used to reference temporary array element.

See example Array.sas
Referencing Array Elements

array-name[subscript]; describes elements in array to be processed. subscript specifies subscript in either form:

- variable-1<,variable-2...> indicates variable/variable list that is usually used with DO-loop processing.
- expression-1<,expression-2...> indicates SAS expressions that evaluate to subscript values when the statement that contains the array reference executes.

* forces SAS to treat the elements in the array as a variable list which can be used in INPUT and PUT statements and some SAS functions.

Warning: * cannot be used to reference temporary array element.

See example Array.sas
Outline

1. **DO Loops**
   - Three Types of DO Loops
   - DO WHILE Loops
   - DO UNTIL Loops
   - Indexed DO Loops
   - DO Loops and SAS Output Data Set

2. **Arrays in SAS Data Step**
   - Use of Arrays
   - Declarations of Arrays
   - Examples
   - Array Referencing

3. **Implicit Arrays**
   - Implicit Arrays
About Implicit Arrays

- For version 5 or earlier, they are alternatives to explicit arrays.
- The use of implicit arrays is still supported by later versions but may go away in version 10.
- SAS recommended the use of explicit arrays and the documentations of implicit arrays were not available for later versions (7+).
About Implicit Arrays

- For version 5 or earlier, they are alternatives to explicit arrays.
- The use of implicit arrays is still supported by later versions but may go away in version 10.
- SAS recommended the use of explicit arrays and the documentations of implicit arrays were not available for later versions (7+)
About Implicit Arrays

- For version 5 or earlier, they are alternatives to explicit arrays.
- The use of implicit arrays is still supported by later versions but may go away in version 10.
- SAS recommended the use of explicit arrays and the documentations of implicit arrays were not available for later versions (7+)
About Implicit Arrays

- For version 5 or earlier, they are alternatives to explicit arrays.
- The use of implicit arrays is still supported by later versions but may go away in version 10.
- SAS recommended the use of explicit arrays and the documentations of implicit arrays were not available for later versions (7+).
Declaration of Implicit Arrays

ARRAY array-name<(index-variable)> <$> <length>
array-elements<(initial-value-list)>;

- The declaration of *index-variable* is optional, but must be enclosed within parentheses when provided. *Index-variable*, when declared, becomes part of the data set.
- *array-elements* list is required
- do not use existing variable name or function name for *array-name*
Declaration of Implicit Arrays

ARRAY array-name<(index-variable)> <$> <length> array-elements <(initial-value-list)>;

- The declaration of index-variable is optional, but must be enclosed within parentheses when provided. Index-variable, when declared, becomes part of the data set.
- array-elements list is required
- do not use existing variable name or function name for array-name
Declaration of Implicit Arrays

ARRAY array-name<(index-variable)> <$> <length>
array-elements <(initial-value-list)>;

- The declaration of *index-variable* is optional, but must be enclosed within parentheses when provided. *Index-variable*, when declared, becomes part of the data set.
- *array-elements* list is required
- do not use existing variable name or function name for *array-name*
Declaration of Implicit Arrays

 ARRAY array-name<(index-variable)> <$> <length> array-elements <(initial-value-list)>;

- The declaration of index-variable is optional, but must be enclosed within parentheses when provided. Index-variable, when declared, becomes part of the data set.
- array-elements list is required
- do not use existing variable name or function name for array-name
Referencing Implicit Arrays

- If *index-variable* is not declared in array declaration, the implicit automatic index-variable *_I_* can be used and it’s not part of data set.
- With DO OVER statement, can use implicit array name without index-variable in referencing.
- Example use:

  ```
  ARRAY x var1-var7;
  DO OVER x;
      IF x=999 THEN x=.;  /* or this;
      * IF x[_i_]=999 THEN x[_i_]=.;
  END;
  ```
Referencing Implicit Arrays

- If *index-variable* is not declared in array declaration, the implicit automatic index-variable `_I_` can be used and it's not part of data set.

- With DO OVER statement, can use implicit array name without index-variable in referencing.

- Example use:

```sas
ARRAY x var1-var7;
DO OVER x;
  IF x=999 THEN x=.;  * or this;
  * IF x[_i_] =999 THEN x[_i_] =.;
END;
```
Referencing Implicit Arrays

- If *index-variable* is not declared in array declaration, the implicit automatic index-variable _I_ can be used and it’s not part of data set.
- With DO OVER statement, can use implicit array name without index-variable in referencing.

Example use:

```sas
ARRAY x var1-var7;
DO OVER x;
   IF x=999 THEN x=.;  * or this;
   * IF x[_i_]=999 THEN x[_i_]=.;
END;
```
Referencing Implicit Arrays

- If *index-variable* is not declared in array declaration, the implicit automatic index-variable `_I_` can be used and it's not part of data set.

- With DO OVER statement, can use implicit array name without index-variable in referencing.

Example use:

```sas
ARRAY x var1-var7;
DO OVER x;
   IF x=999 THEN x=.; * or this;
   * IF x[_i__]=999 THEN x[_i__]=.;
END;
```