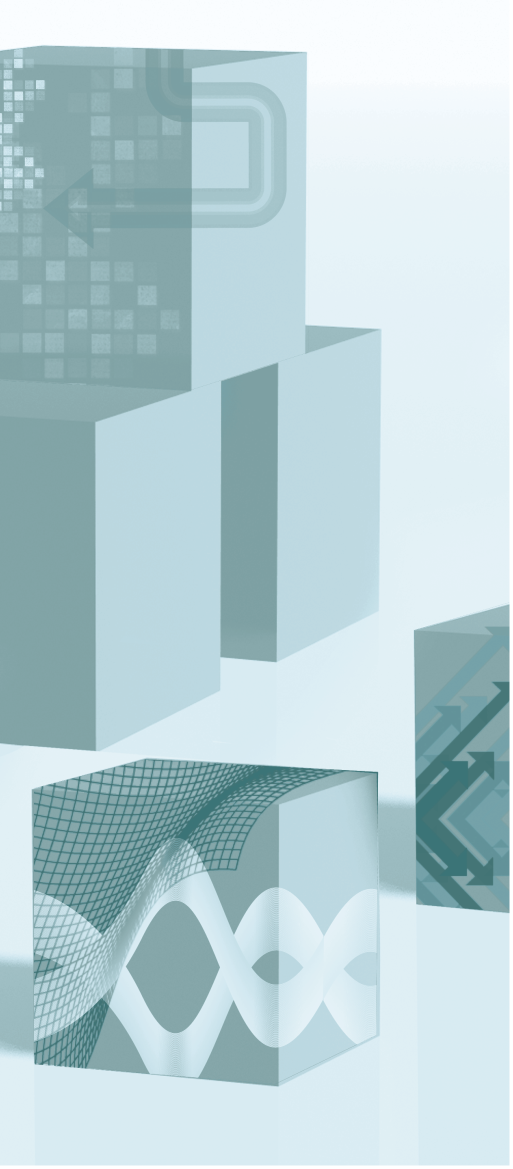


PROC TEMPLATE Made Easy

A Guide for SAS® Users

Kevin D. Smith

SAS® Press

Contents

Chapter 3: Creating and Customizing Table Templates (Partial Chapter)

Although many people don't realize it, almost all tabular output generated by SAS is created using a table template[1]. Table templates describe what columns, headers, and footers should appear in the table. They can also contain style overrides to specify extra style information. In addition, there are even ways to compute new columns based on columns in the existing data set. Because SAS generates tables using table templates, you can modify the output of procedures by editing the table templates that the procedures use. But the real power of table templates is in writing your own from scratch to create tables that aren't possible to create using any of the reporting procedures.

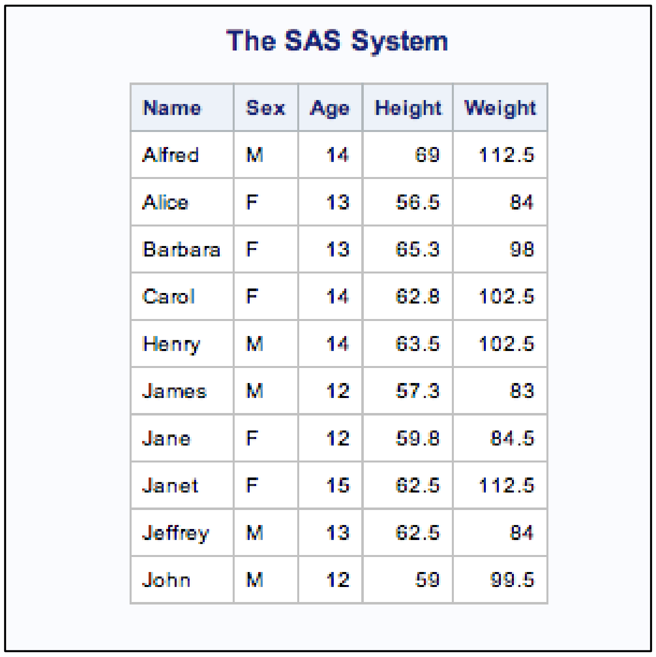
In this chapter, we show you how to create your own table templates, how to customize styles and add traffic lighting, and how to modify the table templates used by SAS procedures.

Defining a Table

You have already seen in previous chapters how to render some simple table templates, but let's recap the methods here to refresh our memories. The first method works with all versions of SAS that have ODS. It uses PROC TEMPLATE to define the table. The DATA step is then used to bind a data set to that template to create the tabular output. The code below is the simplest table template that you can create. It simply prints out all columns of the data set using all of the default attributes.

```
PROC TEMPLATE;
DEFINE TABLE mytable;
END;
RUN;

DATA _NULL_;
  SET sashelp.class(OBS = 10);
  FILE PRINT ODS = (TEMPLATE = "mytable");
  PUT _ODS_;
RUN;
```

Here is the output from the code above.



The SAS System

Name	Sex	Age	Height	Weight
Alfred	M	14	69	112.5
Alice	F	13	56.5	84
Barbara	F	13	65.3	98
Carol	F	14	62.8	102.5
Henry	M	14	63.5	102.5
James	M	12	57.3	83
Jane	F	12	59.8	84.5
Janet	F	15	62.5	112.5
Jeffrey	M	13	62.5	84
John	M	12	59	99.5

The DATA _NULL_ statement won't change in most of the examples in this chapter, so we can simplify this example using a macro. The following macro is the same one that is defined in Chapter 1. It encompasses the DATA _NULL_ statement from the code above. We will be using this macro throughout this chapter to bind data sets to table templates.

```
%MACRO render(template, dataset);
  DATA _NULL_;
    SET &dataset;
    FILE PRINT ODS=(TEMPLATE="&template");
    PUT _ODS_;
  RUN;
%MEND render;
```

Here is the example from above using the macro in place of the DATA _NULL_ statement.

```
PROC TEMPLATE;
DEFINE TABLE mytable;
END;
RUN;

%render(mytable, sashelp.class);
```

If you are using SAS 9.3, you can use the convenience procedure PROC ODSTABLE. This procedure combines the definition of the table and binding a data set to the table template in one step. The example above can be rewritten using PROC ODSTABLE as follows.

```
PROC ODSTABLE DATA = sashelp.class;
RUN;
```

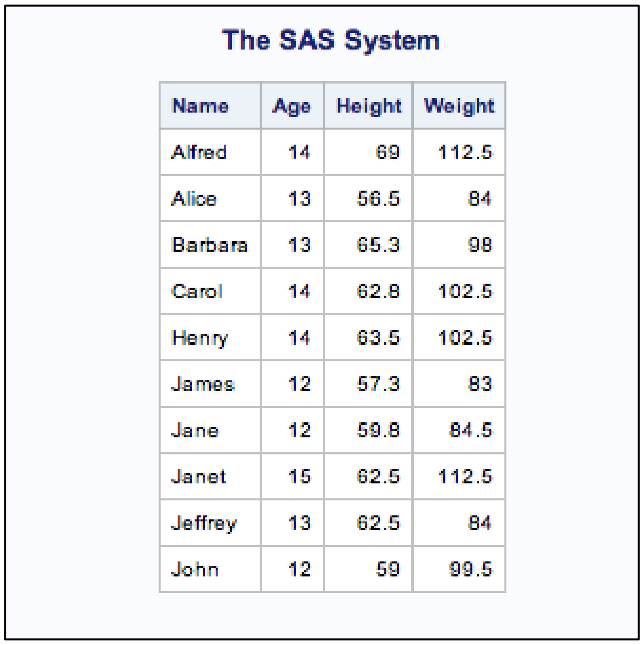
When you let PROC TEMPLATE use all defaults, it will display all columns in the data set. You can limit which columns are displayed by using the COLUMN statement. This works just like the COLUMN statement in PROC REPORT.

```
PROC TEMPLATE;
DEFINE TABLE mytable;
  COLUMN name age height weight;
END;
RUN;

%render(mytable, sashelp.class(OBS = 10));
```

For SAS 9.3, you can use the simpler PROC ODSTABLE code.

```
PROC ODSTABLE DATA = sashelp.class(OBS = 10);
  COLUMN name age height weight;
RUN;
```

Here is the output when you use the COLUMN statement.


The SAS System

Name	Age	Height	Weight
Alfred	14	69	112.5
Alice	13	56.5	84
Barbara	13	65.3	98
Carol	14	62.8	102.5
Henry	14	63.5	102.5
James	12	57.3	83
Jane	12	59.8	84.5
Janet	15	62.5	112.5
Jeffrey	13	62.5	84
John	12	59	99.5

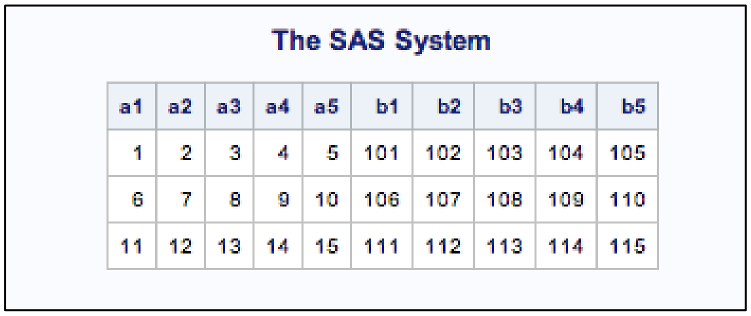
If you have a lot of columns in your data set and they can be expressed using the list notation (e.g., a1-a5), you can use the list notation in the COLUMN statement as well.

```
DATA ab;
  INPUT a1-a5 b1-b5;
DATALINES;
1 2 3 4 5 101 102 103 104 105
6 7 8 9 10 106 107 108 109 110
11 12 13 14 15 111 112 113 114 115
RUN;

PROC TEMPLATE;
DEFINE TABLE mytable;
  COLUMN a1-a5 b1-b5;
END;
RUN;

%render(mytable, ab);
```

Again, this can be simplified in SAS 9.3 as shown in the following code. For the sake of simplicity, this will be the last PROC ODSTABLE sample shown. Just keep in mind that if you have SAS 9.3, you can reduce most of the examples down by taking the content of the table template and using that within PROC ODSTABLE to get the same result. The examples that step outside of the simple compile-and-render scenario that will not work with PROC ODSTABLE will be noted.

```
PROC ODSTABLE DATA = ab;
  COLUMN a1-a5 b1-b5;
RUN;
```

Here is the output from the list notation code.



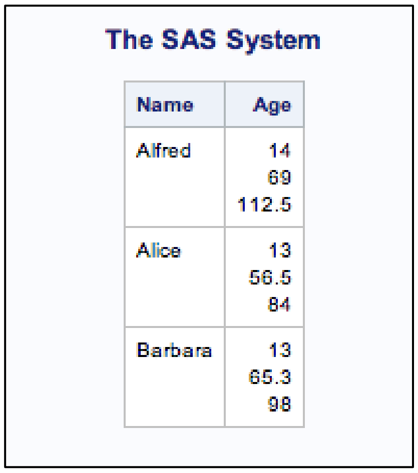
The SAS System

a1	a2	a3	a4	a5	b1	b2	b3	b4	b5
1	2	3	4	5	101	102	103	104	105
6	7	8	9	10	106	107	108	109	110
11	12	13	14	15	111	112	113	114	115

The COLUMN statement can do more advanced things as well. For example, you can stack values on top of each other within a cell by grouping the column names in parentheses. The COLUMN statement here stacks the age, height, and weight variables on top of each other in the same cell.

```
PROC TEMPLATE;
DEFINE TABLE mytable;
  COLUMN name (age height weight);
END;
RUN;

%render(mytable, sashelp.class(OBS = 3));
```

The output is shown below.



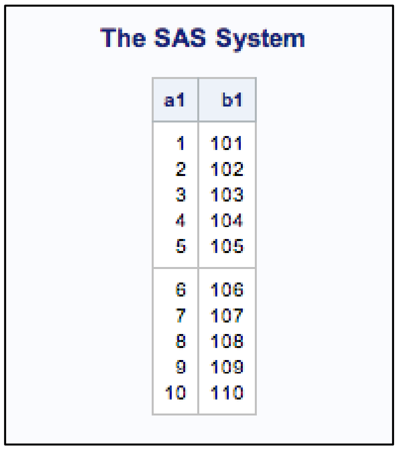
```
The SAS System

Name      Age
Alfred     14
           69
        112.5
Alice      13
         56.5
           84
Barbara    13
         65.3
           98
```

You might have noticed that the header for the stacked column contains only Age. The variable that is on the top of the stack determines what the header text is. However, you can change the text of the header manually. That technique is discussed later in this chapter.

You can combine this stacking feature with the list notation feature too.

```
PROC TEMPLATE;
DEFINE TABLE mytable;
  COLUMN (a1-a5) (b1-b5);
END;
RUN;

%render(mytable, ab(OBS = 2));
```

Here is the output from the previous code.

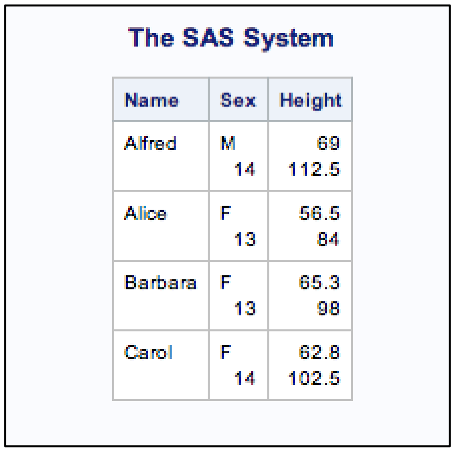


```
The SAS System

a1    b1
 1   101
 2   102
 3   103
 4   104
 5   105

 6   106
 7   107
 8   108
 9   109
10   110
```

Another way to stack is with groups of columns. In this method, you group the columns together with parentheses and then stack each group on top of each other using an asterisk (*). The following code stacks sex over age and height over weight:

```
PROC TEMPLATE;
DEFINE TABLE mytable;
  COLUMN name (sex height) * (age weight);
END;
RUN;

%render(mytable, sashelp.class(OBS = 4));
```

The output below is created by this COLUMN statement.



```
The SAS System

Name      Sex   Height
Alfred    M         69
           14    112.5

Alice     F       56.5
           13       84

Barbara   F       65.3
           13       98

Carol     F       62.8
           14    102.5
```

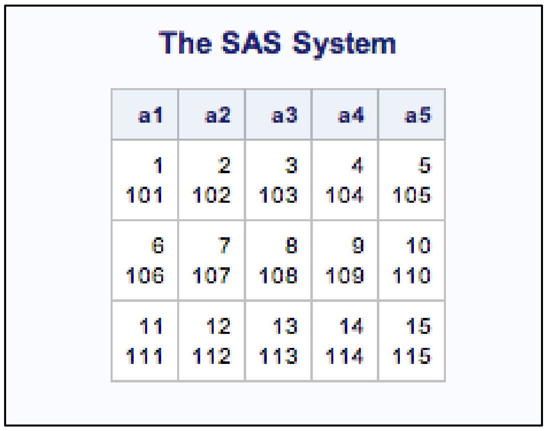
When using this method, it is sometimes easier to see what the output table will look like by putting a line break after each asterisk in the COLUMN statement. That way, your COLUMN statement looks a lot like the resulting table.

```
PROC TEMPLATE;
DEFINE TABLE mytable;
  COLUMN name (sex height) *
              (age weight);
END;
RUN;

%render(mytable, sashelp.class);
```

Of course, the list notation for variables works with this stacking method as well. Here is an example.

```
PROC TEMPLATE;
DEFINE TABLE mytable;
  COLUMN (a1-a5) *
         (b1-b5);
END;
RUN;

%render(mytable, ab);
```

The output from this code is shown below.



The SAS System

a1	a2	a3	a4	a5
1 101	2 102	3 103	4 104	5 105
6 106	7 107	8 108	9 109	10 110
11 111	12 112	13 113	14 114	15 115

Rendering columns in this manner will use the default attributes for the columns themselves. There are many attributes that you can use to change the way the columns look, and they will be discussed later in this chapter.

Defining Headers and Footers

Table templates can contain two types of headers: table headers and column headers. Luckily, both types of headers are defined the same way. This section describes how to define headers and how to apply them to tables. The next section deals with applying headers to a column.

To define a header, use the standard PROC TEMPLATE method of utilizing a DEFINE statement that has a template type and name to start the definition, and an END statement to end it. In this case, the type can be HEADER or FOOTER. Headers go at the top of the table, and footers go at the bottom of the table. The following code shows a definition of each of these types:

```
DEFINE HEADER myheader;
END;

DEFINE FOOTER myfooter;
END;
```

Defining a header or footer doesn't make much sense unless you associate some text with it. This is done by using the TEXT statement, as shown here[2].

```
DEFINE HEADER myheader;
  TEXT 'Class Information';
END;
```

The value given in the TEXT statement can be any combination of quoted strings, dynamic variables (as specified in the DYNAMIC statement), macro variables (as specified in the MVAR statement), and built-in variables.

Dynamic, macro, and built-in variables are discussed in the **Dynamic Variables** section of this chapter. The _LABEL_variable in the following code is an example of a built-in variable that inserts the data object's label.

```
DEFINE HEADER myheader;
  TEXT 'Class Information (' _LABEL_ ')';
END;
```

Now that we have a header definition, we need to attach it to a table. This can be done in one of two ways: 1) use the HEADER statement[3] or 2) define the header within a table definition. The HEADER statement is much like the simple form of the COLUMN statement because you list the names of the headers that you want to appear in the table. However, because most headers you create will be specific to a particular table template, the second method is more common. Here is an example of applying a header to a table using Method 1[4].

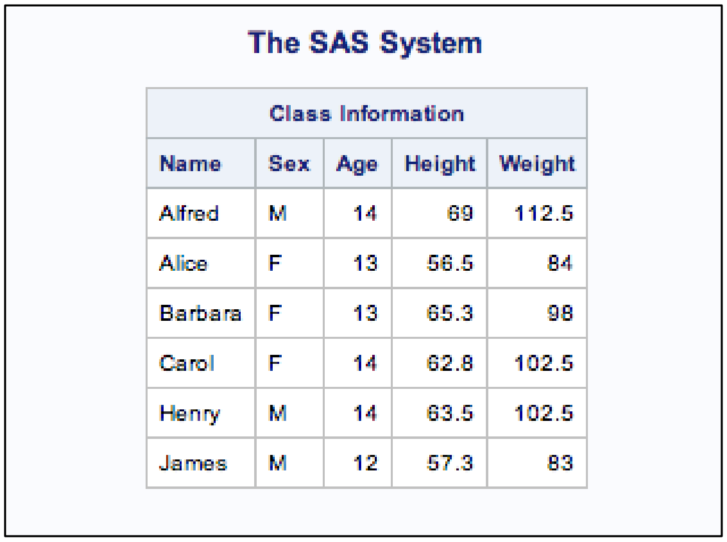
```
PROC TEMPLATE;
  DEFINE HEADER my.header;
    TEXT 'Class Information';
  END;
  DEFINE TABLE mytable;
    HEADER my.header;
  END;
RUN;

%render(mytable, sashelp.class);
```

Here is an example of applying a header to a table using Method 2:

```
PROC TEMPLATE;
DEFINE TABLE mytable;
  DEFINE HEADER myheader;
    TEXT 'Class Information';
  END;
END;
RUN;

%render(mytable, sashelp.class(OBS = 6));
```

As you can see, defining a header outside of the table is more work. In addition, this is a case where PROC ODSTABLE cannot be used to define all parts of the table. PROC ODSTABLE can only contain the statements and objects from a single table template. If you use PROC ODSTABLE, you have to use a separate PROC TEMPLATE step just to define the external header. Defining a header outside of a table is most common when you are going to use that header definition within more than one table definition. The need for this is not as common, so you might not use that particular method much.

Here is what the HTML output for either of the above code blocks looks like.

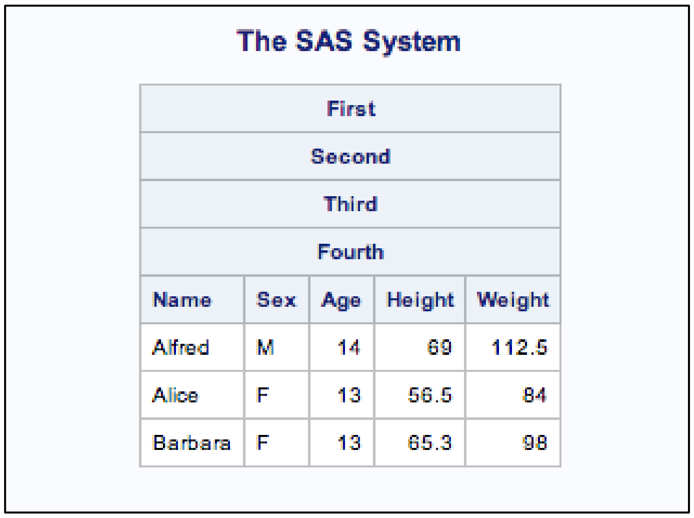

The SAS System

Class Information				
Name	Sex	Age	Height	Weight
Alfred	M	14	69	112.5
Alice	F	13	56.5	84
Barbara	F	13	65.3	98
Carol	F	14	62.8	102.5
Henry	M	14	63.5	102.5
James	M	12	57.3	83

You can add as many headers to the table as you want. They will appear in the table in the same order that they do in the table definition.

```
PROC TEMPLATE;
DEFINE TABLE mytable;
  DEFINE HEADER first;
    TEXT 'First';
  END;
  DEFINE HEADER second;
    TEXT 'Second';
  END;
  DEFINE HEADER third;
    TEXT 'Third';
  END;
  DEFINE HEADER fourth;
    TEXT 'Fourth';
  END;
END;
RUN;

%render(mytable, sashelp.class(OBS = 3));
```

This code produces the output shown below.



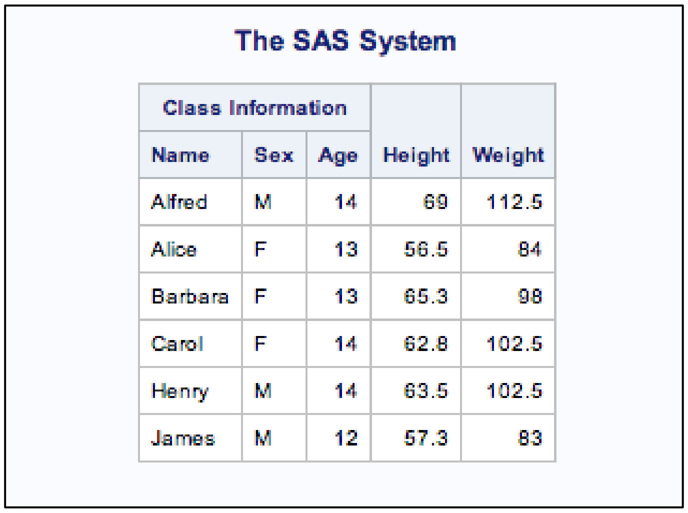
The SAS System

First				
Second				
Third				
Fourth				
Name	Sex	Age	Height	Weight
Alfred	M	14	69	112.5
Alice	F	13	56.5	84
Barbara	F	13	65.3	98

As you can see from the last two examples, table headers span the entire width of the table. You can change this by giving end points to the header using the START= and END= attributes. The values of these attributes are the names of the columns where the table header should start and end, respectively. The following code shows the same header as in the first example, but it only spans the first three columns.

```
PROC TEMPLATE;
DEFINE TABLE mytable;
  DEFINE HEADER myheader;
    TEXT 'Class Information';
    START = name;
    END = age;
  END;
END;
RUN;

%render(mytable, sashelp.class(OBS = 6));
```

The output is shown below.



The SAS System

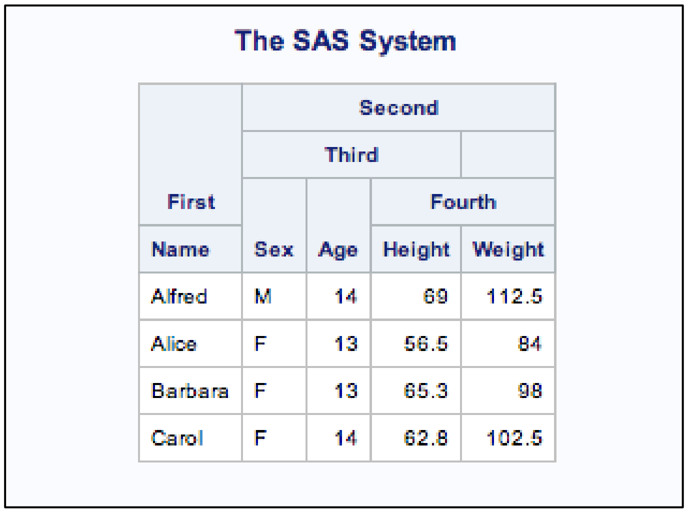
Class Information				
Name	Sex	Age	Height	Weight
Alfred	M	14	69	112.5
Alice	F	13	56.5	84
Barbara	F	13	65.3	98
Carol	F	14	62.8	102.5
Henry	M	14	63.5	102.5
James	M	12	57.3	83

Now let's combine stacking with spanning. The stacking behavior is similar to the way that the blocks in the game Tetris stack up. The headers fall and fill in empty spaces. If they encounter any other header as they fall, that is their final resting spot. ODS stretches the cells vertically to fill the empty spaces. The following code defines four headers with various spans. It's probably not something that you would do in reality, but it does show you the flexibility of table headers.

```
PROC TEMPLATE;
DEFINE TABLE mytable;
  DEFINE HEADER first;
    TEXT 'First';
    START = name;
    END = name;
  END;
  DEFINE HEADER second;
    TEXT 'Second';
    START = sex;
```

```
      END = weight;
    END;
    DEFINE HEADER third;
      TEXT 'Third';
      START = sex;
      END = height;
    END;
    DEFINE HEADER fourth;
      TEXT 'Fourth';
      START = height;
      END = weight;
    END;
  END;
  RUN;

  %render(mytable, sashelp.class(OBS = 4));
```

Here is the output.



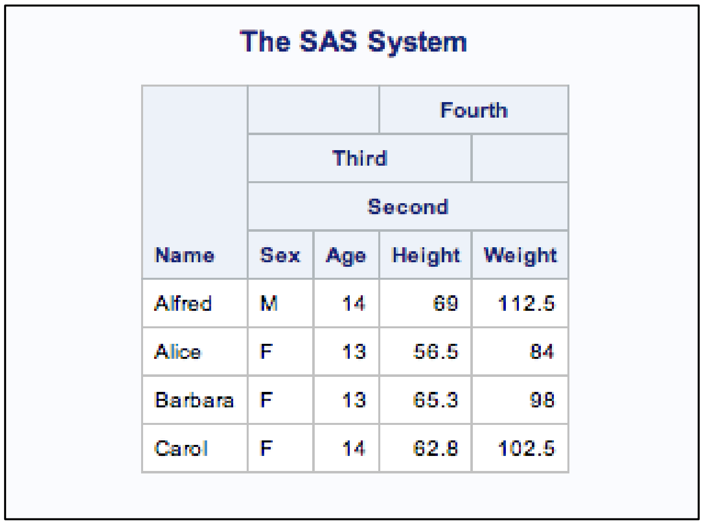
The SAS System

First	Second				
	Third				
			Fourth		
Name	Sex	Age	Height	Weight
Alfred	M	14	69	112.5
Alice	F	13	56.5	84
Barbara	F	13	65.3	98
Carol	F	14	62.8	102.5

When we mentioned the HEADER statement previously, it might have sounded like the only time that you would use it is to include external header definitions. While it works fine for that, you can also use the HEADER statement on internal header definitions. In fact, you can mix and match internal and external headers in the same table. Going back to our Tetris example, while in the game of Tetris you have to accept the tiles that the game chooses for you in the order that it chooses for you, when dealing with table headers, you can reorder and exclude headers from being applied to a table by using the HEADER statement. Here is the last example again, but this time we are excluding the first header and reversing the order in which the headers are applied to the table.

```
PROC TEMPLATE;
DEFINE TABLE mytable;
  DEFINE HEADER first;
    TEXT 'First';
    START = name;
    END = name;
  END;
  DEFINE HEADER second;
    TEXT 'Second';
    START = sex;
    END = weight;
  END;
  DEFINE HEADER third;
    TEXT 'Third';
    START = sex;
    END = height;
  END;
  DEFINE HEADER fourth;
    TEXT 'Fourth';
    START = height;
    END = weight;
  END;
  HEADER fourth third second;
END;
RUN;

%render(mytable, sashelp.class(OBS = 4));
```

The output for this code is shown below.



The SAS System

			Fourth	
	Third			
	Second			
Name	Sex	Age	Height	Weight
Alfred	M	14	69	112.5
Alice	F	13	56.5	84
Barbara	F	13	65.3	98
Carol	F	14	62.8	102.5

Note that the position of the HEADER statement is important when you define more headers than will appear in the table[5]. Each time you define a table header, that header is automatically added to the header list. If you were to put the HEADER statement at the beginning of this table definition, you would still have four table headers in the order: fourth, third, second, first.

Common Table Header and Footer Attributes

There are various attributes besides START= and END= that you can use to change the behavior of headers and footers. The complete list can be seen in the *SAS 9.3 Output Delivery System: User's Guide, Second Edition*. There are approximately two dozen attributes from which to choose. If you look closely in the documentation, you'll see that many of these attributes are used only by the ODS Listing output. Below is a list of the most common header and footer attributes.

FIRST_PANEL=ON | OFF
specifies whether or not a spanning header appears *only* on the first panel if the header is broken across multiple panels. This option applies only to output that supports paneling.

JUST=LEFT | CENTER | RIGHT
specifies the horizontal alignment of the text of the header.

LAST_PANEL=ON | OFF
specifies whether or not a spanning footer appears *only* on the last panel if the footer is broken across multiple panels. This option applies only to output that supports paneling.

PARENT=header-path
specifies a header definition to inherit from. Inheritance will be discussed later in this chapter.

PREFORMATTED=ON | OFF
specifies whether or not to treat the text of the header as preformatted text. Preformatted text preserves all whitespace and sets the font to a monospace font. Using this option also appends `fixed' to the default style element name used for the header or footer. For example, *header* would become *headerfixed, rowfooter* would become *rowfooterfixed*, etc.

PRINT=ON | OFF
specifies whether or not the header should be printed.

SPLIT='*character*'
specifies a character that will be treated as a newline character. Whenever the specified character is encountered in the text of the header, a line break is put in its place. Using the SPLIT= attribute isn't necessary to put line breaks in a header (though it is the recommended way). If you do not specify a character with the SPLIT= attribute and the first character of the header text isn't an alphanumeric character, a blank, an underscore (_), a hypen (-), a period (.), or a percent sign (%), the first character of the header text will automatically become the split character. For example, the header text "/Age/Height/Weight" would automatically convert "/" characters into newlines if SPLIT= was not specified.

STYLE=style-override

specifies a style override to apply to the header. Style overrides are discussed in "Style Overrides and Conditional Formatting**"** in Chapter 2.

VJUST=TOP | CENTER | BOTTOM

specifies the vertical alignment of the header text.

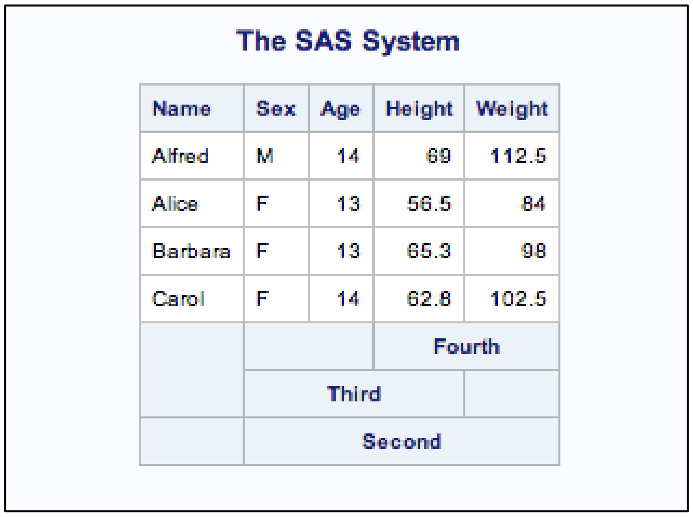
As opposed to setting values of these attributes explicitly, it is possible to set the attribute values dynamically at run time as well. This is not as useful when using the PROC ODSTABLE procedure, but it can be very handy when creating reusable table templates from PROC TEMPLATE.

One statement within both header and footer definitions is the NOTES statement. The NOTES statement is like a comment, but it is stored with the header definition so that it can be viewed later. Here is an example of using the NOTES statement based on the very first header definition that we created in this chapter.

```
PROC TEMPLATE;
DEFINE HEADER my.header;
  NOTES "Externally defined header for tables that
         use Sashelp.Class";
  TEXT "Class Information";
END;
RUN;
```

The last three statements available in headers and footers that we are going to discuss, DYNAMIC, MVAR, and NMVAR, are used to set run-time attribute values. These values come from dynamic variables set in the DATA step and macro variables[6]. Because this behavior is shared between tables, columns, headers, and footers, the explanation for this feature is can be found in "Using Variables in Table Templates."

Almost all of the discussion in this section deals with headers; we didn't say much about footers. The syntax for creating footers is identical to headers. The only difference is that they are located at the bottom of the table rather than at the top. The figure below shows the output of the last header example with all of the headers changed to footers.



The SAS System

Name	Sex	Age	Height	Weight
Alfred	M	14	69	112.5
Alice	F	13	56.5	84
Barbara	F	13	65.3	98
Carol	F	14	62.8	102.5
			Fourth	
	Third			
	Second			

So far we have covered the basics of putting together columns to form a table as well as adding headers and footers. Many more advanced operations are available for columns. Because some of these operations involve header definitions, we saved the discussion of column definitions until now.

Defining Columns

Although the COLUMN statement describes the overall structure of the table columns, the columns themselves can have their own definitions just like headers and footers. A column definition can define what the column should look like, what the column header should contain, and what data column should be used. It can even calculate the data values themselves. Putting columns in a table is just like putting headers in a table. You have two choices:

1. Use the COLUMN statement to create columns based on the data columns in a data set[7]
2. Define the column within a table definition

You can also use a combination of these methods where you define the overall structure of the table using the COLUMN statement and then create column definitions for the columns that need special treatment.

Let's go back to our first table that used the COLUMN statement.

```
PROC TEMPLATE;
DEFINE TABLE mytable;
  COLUMN name age height weight;
END;
RUN;
```

```
%render(mytable, sashelp.class);
```

The following table template is equivalent but is not nearly as convenient.

```
PROC TEMPLATE;
DEFINE TABLE mytable;
  DEFINE COLUMN name;
  END;
  DEFINE COLUMN age;
  END;
  DEFINE COLUMN height;
  END;
  DEFINE COLUMN weight;
  END;
END;
RUN;

%render(mytable, sashelp.class);
```

Using a hybrid approach, you can use the COLUMN statement to define the overall structure, and then use column definitions to apply specific behavior. This would look something like the following.

```
PROC TEMPLATE;
DEFINE TABLE mytable;
  COLUMN name age height weight;
  DEFINE height;
  END;
  DEFINE weight;
  END;
END;
RUN;

%render(mytable, sashelp.class);
```

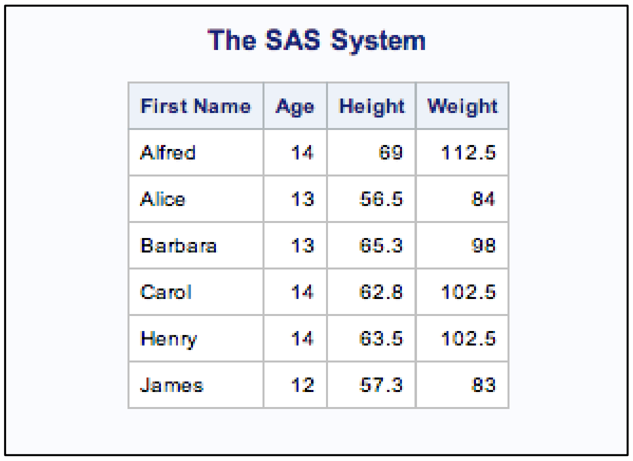
Note that when you use both a COLUMN statement and column definitions, specifying the template type in the column definition is optional. PROC TEMPLATE already knows that it is a column because of the information in the COLUMN statement[8]. Also, just as with headers, the position of the COLUMN statement is important. In the example above, if you were to define more column templates than there are columns listed in the COLUMN statement, they would automatically be appended to the end of the column list. To explicitly limit which columns go into a table, put the COLUMN statement at the end of the table definition[9].

All of the column templates in that last code snippet were kind of nonsensical because they didn't do anything other than the default behavior, which can be gotten simply by listing the columns in the COLUMN statement. However, defining column templates is necessary when you start to add customization. Let's say that we wanted to change the column header for the column Name to

‘First Name’. The simplest way would be to add a column definition for that column and set the HEADER= attribute.

```
PROC TEMPLATE;
DEFINE TABLE mytable;
  COLUMN name age height weight;
  DEFINE name;
    HEADER = 'First Name';
  END;
END;
RUN;

%render(mytable, sashelp.class(OBS=6));
```

The resulting output is shown below.



The SAS System

First Name	Age	Height	Weight
Alfred	14	69	112.5
Alice	13	56.5	84
Barbara	13	65.3	98
Carol	14	62.8	102.5
Henry	14	63.5	102.5
James	12	57.3	83

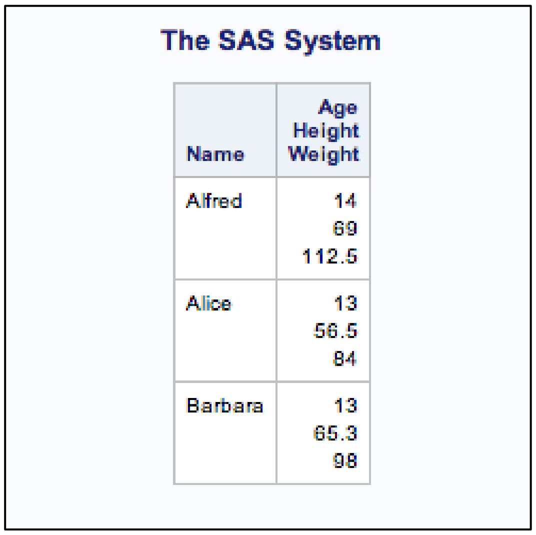
While this works for simple headers, you might want to use a full header definition instead of just a string. The following form enables you to add styles and other visual effects to headers as we will see later. In the case of column headers, you must use the HEADER= attribute to associate a header definition with a column. It is not automatically applied as is the case in a table header. The output for this code is identical to the output from the previous example:

```
PROC TEMPLATE;
DEFINE TABLE mytable;
  COLUMN name age height weight;
  DEFINE name;
    DEFINE HEADER nameheader;
      TEXT 'First Name';
    END;
    HEADER = nameheader;
  END;
END;
RUN;

%render(mytable, sashelp.class);
```

Let’s go back to a problem that we had in the column stacking example. When stacking columns, the header for the resulting column is always the header from the column that is on the top of the stack. Let’s put a header on that column to include the headers for all of the columns in the stack. We also use this opportunity to show off the SPLIT= header attribute described in the previous section.

```
PROC TEMPLATE;
DEFINE TABLE mytable;
  COLUMN name (age height weight);
  DEFINE age;
    DEFINE HEADER ageheightweight;
      TEXT 'Age*Height*Weight';
      SPLIT = '*';
    END;
    HEADER = ageheightweight;
  END;
END;
RUN;

%render(mytable, sashelp.class(OBS = 3));
```

The output for this example is shown below.

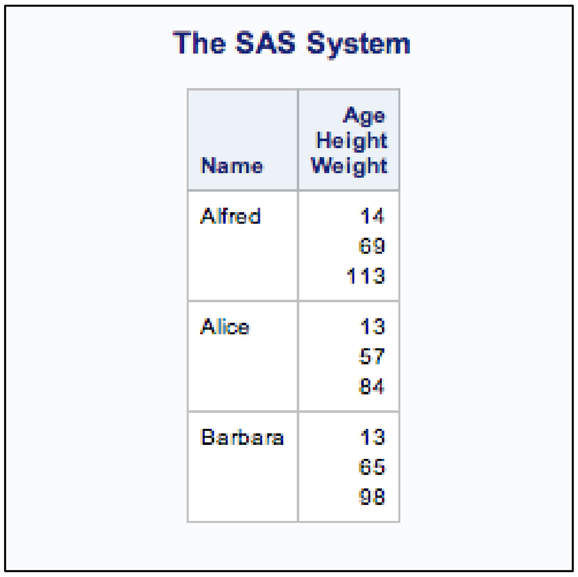


```
The SAS System

         Age
      Height
Name  Weight
Alfred    14
          69
       112.5
Alice     13
        56.5
          84
Barbara   13
        65.3
          98
```

Another common thing to do with a column template is to change the data format. This is done using the FORMAT= attribute. The value is simply a SAS format. Let's continue to use the definition from the last example to try out the FORMAT= attribute. We want all of the numbers in our Age/Height/Weight column to line up. You can do this by formatting all of the numbers the same way. You can make them all integers with the following code.

```
PROC TEMPLATE;
DEFINE TABLE mytable;
  COLUMN name (age height weight);
  DEFINE age;
    DEFINE HEADER ageheightweight;
      TEXT 'Age*Height*Weight';
      SPLIT = '*';
    END;
    HEADER = ageheightweight;
    FORMAT = 3.;
  END;
  DEFINE height;
    FORMAT = 3.;
  END;
  DEFINE weight;
    FORMAT = 3.;
  END;
END;
RUN;
%render(mytable, sashelp.class(OBS = 3));
```

Here is the output from the code above.


The SAS System

Name	Age Height Weight
Alfred	14 69 113
Alice	13 57 84
Barbara	13 65 98

Common Table Column Attributes

There are many other column attributes that can be used to affect the behavior of columns. Just as with headers, many of the attributes are used only for ODS Listing output. The Listing-only attributes have been left out of this list. For a full list of column attributes and descriptions, see the *SAS 9.3 Output Delivery System: User's Guide, Second Edition*.

BLANK_DUPS=ON | OFF
specifies whether or not to leave a cell empty if the value in the previous row was the same.

DATA_FORMAT_OVERRIDE=ON | OFF
specifies whether or not the format from the data column should take precedence over the format in the template.

DATANAME=column-name
specifies the name of the data column to use rather than using the name of the data column that matches the column name.

DROP=ON | OFF
specifies whether or not to include the column in an output data set. This option applies only to the ODS Output destination.

FORMAT=SAS-format
specifies a SAS format to use for the data in the column.

FORMAT_WIDTH=positive-integer
specifies the format width for the column. This attribute is mainly intended for procedure writers. In most cases, you should use the FORMAT= attribute.

FORMAT_NDEC=positive-integer
specifies the number of decimals for the column. This attribute is mainly intended for procedure writers. In most cases, you should use the FORMAT= attribute.

FUZZ=*number*
specifies a numeric value that is used as a minimum value for data in the cells. Any value that is less than the value given in FUZZ= is treated as zero.

GENERIC=ON | OFF
specifies whether or not the column template can be used by multiple columns in the table. The details about generic columns will be discussed later in this chapter.

HEADER=header-specification
specifies the header for the column. The value of *header-specification* can be either a quoted string or the name of a header template.

ID=ON | OFF
specifies whether or not the column should be repeated on each data panel. This applies only to output types that support paneling.

JUST=LEFT | CENTER | RIGHT
specifies the horizontal alignment of the column's content. The default behavior is to left align character content and right align numeric content. Because of some details in the way that alignments are evaluated, setting the alignment using the TEXTALIGN= style attribute in a style override is preferred over this method.

LABEL='*text*'
specifies a label to use instead of the label supplied in the data set.

MERGE=ON | OFF
specifies whether or not the content of the current data cell should be merged with the content of the following column's data.

PARENT=column-path
specifies a column definition to inherit from. Inheritance will be discussed later in this chapter.

PREFORMATTED=ON | OFF
specifies whether or not to treat the text of the column as preformatted text. Preformatted text preserves all whitespace and sets the font to a monospace font. Using this option also sets the style element to *datafixed*.

PRE_MERGE=ON | OFF
specifies whether or not the content of the current data cell should be merged with the content of the previous column's data.

PRINT=ON | OFF
specifies whether or not the column should be printed.

PRINT_HEADERS=ON | OFF
specifies whether or not the column header should be printed.

STYLE=style-override
specifies a style override to apply to the header. Style overrides are discussed in the "Style Overrides and Conditional Formatting" section in Chapter 2.

TEXT_SPLIT='*character*'
specifies a character that will be treated as a newline character. Whenever the specified character is encountered in the text of the column, a line break is put in its place.

VARNAME=variable-name
specifies the name to use for the variable in an output data set.

VJUST=TOP | CENTER | BOTTOM
specifies the vertical alignment of the column's content.

Just as with headers, columns also have an attribute-like statement called NOTES that embeds a comment into the template that can be viewed later. Also, like headers, the attribute values can be set dynamically at run time as discussed in “Reusable Table Templates and DATA _NULL_” later in this chapter.

So far we have been using only data that is explicitly defined in a data set, but you can create columns whose data is based on combinations of data in a data set. This type of column is discussed in the following section.

Using Computed Column Values as Column Data

Although you do need a data set to render a table template, you do not necessarily need a data column for each column in the table. You can create columns that are computed from other data in the table. These are called “computed columns.” The COMPUTE AS statement determines whether a column is associated with a data column or an expression. This statement takes a standard expression as its argument. The expression is evaluated for every observation in the input data set to get the resulting output data value. Here is the general format of the COMPUTE AS statement:

```
COMPUTE AS expr;
```

Since we have been using the SASHELP.CLASS data set, let’s continue that trend and use that information to compute something useful. One thing that can be calculated using that information is the body mass index (BMI). This is one measure to determine if someone is underweight, normal weight, or overweight. It is computed using the following formula[10].

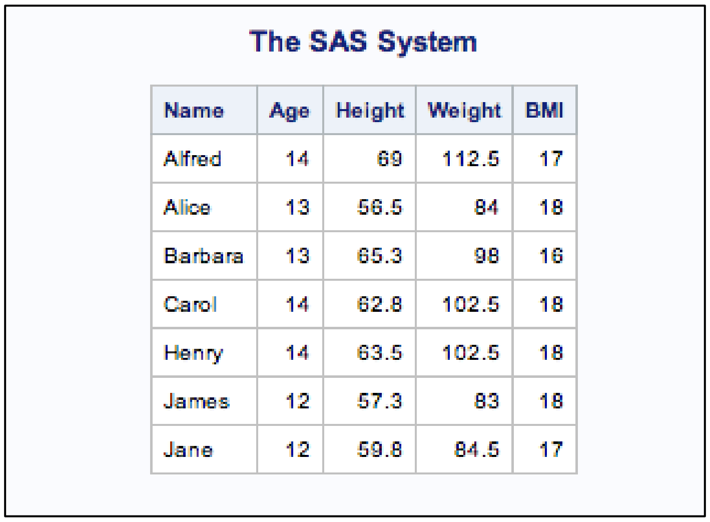
where *weight* is expressed in pounds and *height* is in inches.

$$BMI = \frac{weight \cdot 703}{height^2}$$

Now that we have all of the information, let’s add a new column to the table that contains each individual’s BMI.

```
PROC ODSTABLE DATA = sashelp.class(OBS = 7);
  COLUMN name age height weight bmi;
  DEFINE bmi;
    COMPUTE AS (weight * 703) / (height * height);
    HEADER = 'BMI';
    FORMAT = 2.;
  END;
RUN;
```

The resulting output is shown below.



The SAS System

Name	Age	Height	Weight	BMI
Alfred	14	69	112.5	17
Alice	13	56.5	84	18
Barbara	13	65.3	98	16
Carol	14	62.8	102.5	18
Henry	14	63.5	102.5	18
James	12	57.3	83	18
Jane	12	59.8	84.5	17

A method to create new data values in a column is by using the TRANSLATE INTO statement. This type of column doesn't generate a new column; rather it simply takes the data in a column and converts it into a different representation. It is kind of like using a SAS format, but it gives you the full power of an expression in both the matching of a value and the generating of a value. In addition, the value of the column can be based on the values of any number of columns, just like in the case of the COMPUTE AS statement. The general form of the TRANSLATE INTO statement is as follows.

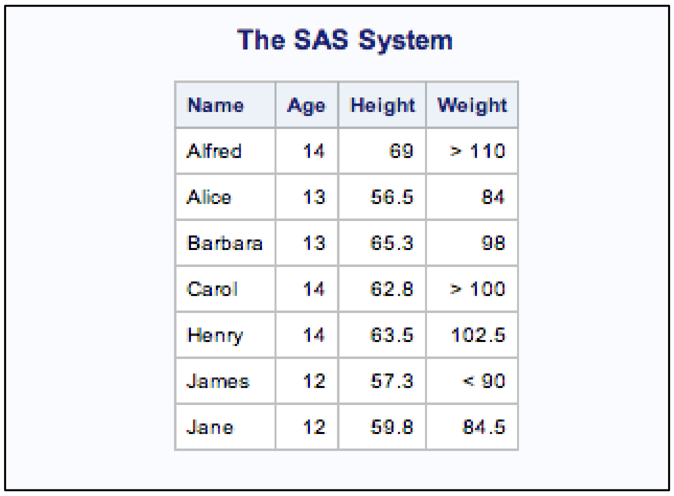
```
TRANSLATE expr-1a INTO expr-1b,
               ...
               expr-na INTO expr-nb;
```

The data in the column is tested against the expressions on the left side of the INTO keyword. If the expression is true, then the data that is printed in the output is generated by the expression on the right side of the INTO keyword. Let's look at an example.

We do not care about the exact weight values for the individuals in the SASHELP.CLASS data set if they are above or below a certain value. However, those values are different for the male and female students. For male students, we only care if the weight is between 90 and 120. Any values outside that range can be represented by '< 90' and '> 110', respectively. For females, we want to do the same thing, except the values are 70 and 100. We could not accomplish this with a SAS format because it requires the data from two columns.

```
PROC TEMPLATE;
DEFINE TABLE mytable;
  COLUMN name sex age height weight;
  DEFINE sex;
    PRINT = OFF;
  END;
  DEFINE weight;
    TRANSLATE (sex = 'M' AND weight > 110) INTO '> 110',
              (sex = 'M' AND weight < 90)  INTO '< 90',
              (sex = 'F' AND weight > 100)  INTO '> 100',
              (sex = 'F' AND weight < 70)  INTO '< 70';
    STYLE = {TEXTALIGN = RIGHT};
  END;
END;
RUN;

%render(mytable, sashelp.class(OBS = 7));
```

In this case, because we are using the *sex* variable in the calculation, we need to include it in the COLUMN statement as well. Tturn it off by using PRINT=OFF in the column definition. The value of the weight column is referenced as 'weight' in the expression. However, you could also use the keyword _VAL_ which refers to the data value of the current column. This does not make any difference when the TRANSLATE INTO statement is on a column; however, the same TRANSLATE INTO statement can be used in the table context to apply to all of the columns in a table. In that case, the _VAL_ keyword would correspond to whichever column the expression was currently being applied to. Here is the output from the code above.



The SAS System

Name	Age	Height	Weight
Alfred	14	69	> 110
Alice	13	56.5	84
Barbara	13	65.3	98
Carol	14	62.8	> 100
Henry	14	63.5	102.5
James	12	57.3	< 90
Jane	12	59.8	84.5

The TRANSLATE INTO statement can be used on the table itself. There are attributes, just like in columns, headers, and footers, that can be used on tables to affect their behaviors as well. Some of these attributes don't make sense until after you know some things about columns and headers, so we saved the discussion of them until now.

Table Attributes

Table attributes are used to affect the behavior of the table itself, as well as some header and column behaviors. Just like with columns and headers, many of the attributes are ODS Listing-specific and will not be discussed here. A complete list of attributes can be found in the *SAS 9.3 Output Delivery System: User's Guide, Second Edition*.

BYLINE=ON | OFF
specifies whether or not to print the byline before the table.

CLASSLEVELS=ON | OFF
specifies whether or not to enable the BLANK_DUPS= attribute on a column if the previous column also has BLANK_DUPS= enabled and its value changed in the current row.

CONTENTS=ON | OFF
specifies whether or not to put the table in the table of contents.

CONTENTS_LABEL='*string*'
specifies the label to use for the table in the table of contents, the Results window, and the trace record.

DATA_FORMAT_OVERRIDE=ON | OFF
specifies whether or not the format from the data column should take precedence over the format in the template.

LABEL='*text*'
specifies a label for the table. If no label is provided, the label from the data set will be used. If it does not have a label, the text of the first table header is used.

NEWPAGE=ON | OFF
specifies whether or not to force a page break before the table.

ORDER_DATA=ON | OFF
specifies whether or not the order of the columns in the data set should override the order of the columns specified in the template.

PARENT=table-path
specifies a table definition to inherit from. Inheritance will be discussed later in this chapter.

PRINT_FOOTERS=ON | OFF
specifies whether or not to print the table footers.

PRINT_HEADERS=ON | OFF
specifies whether or not to print the table headers.

STYLE=style-override
specifies a style override to apply to the header. Style overrides are discussed in "Style Overrides and Conditional Formatting" in Chapter 2.

USE_FORMAT_DEFAULTS=ON | OFF
specifies whether or not the width and number of decimals should be taken from the default values for the format name rather than the template.

USE_NAME=ON | OFF
specifies the column name as the column header if neither the column definition nor the data set specifies one.

Just as with columns, headers, and footers, tables also support a NOTES statement that enables you to store a comment with the table that describes what the table is for.

While we have only shown each attribute with explicit values here, it is possible to have attribute values dynamically set at run time rather than at compile time. This is not as useful when you are using the PROC ODSTABLE convenience procedure to generate tables, but it can be quite powerful when you are creating reusable table templates using PROC TEMPLATE.

While the attributes described for columns, headers, footers, and tables have enabled us to change various aspects of behavior, the look and feel defined by the overall style remains. It is possible to override style attributes within the table family of templates, and even do traffic lighting based on the data in the table. These techniques are discussed in the following section.

Style Overrides and Traffic Lighting

About The Author

Kevin D. Smith has been a software developer at SAS since 1997. He has supported PROC TEMPLATE and other underlying ODS technologies for most of that time. Kevin has spoken at numerous SAS Global Forums as well as at regional and local SAS users groups with the “From Scratch” series of presentations that were created to help users of any level master various ODS technologies. Kevin is also the author of many of the ODS tip sheets available on the SAS support Web site.

Learn more about this author by visiting his author page at http://support.sas.com/smithk. There you can download free chapters, access example code and data, read the latest reviews, get updates, and more.