Paper 053-31

The TEMPLATE Procedure Styles: Evolution and Revolution

Kevin D. Smith, SAS Institute Inc., Cary, NC

ABSTRACT

In SAS® 9.2, PROC TEMPLATE styles will have more features and enhancements added than in any other version of SAS, including simplifying inheritance and making style overrides more intuitive. Many of these enhancements deal with the usability problem that has given PROC TEMPLATE styles their poor reputation.

In addition to the evolutionary enhancements, an entirely new species of ODS styles is being introduced: Cascading Style Sheets (CSS). Styles that are written using the World Wide Web Consortium's (W3C) CSS specification are now valid ODS styles that enable you to create and edit your ODS styles using the SAS® Enterprise Guide® style editor or third-party CSS editors.

INTRODUCTION

With the exception of adding new style attributes, PROC TEMPLATE styles have remained mostly unchanged since their inception. This would be okay except for the fact that PROC TEMPLATE styles are disliked due to their complexity and unusual inheritance model. In SAS 9.2, the PROC TEMPLATE styles have received a major "face-lift". Now, you can define a style for ODS without having to type one line of PROC TEMPLATE code!

Because the list of enhancements is so long, it is divided into three groups for discussion in this presentation. The first group explains how styles are defined and the changes that were made to style element inheritance. The second group discusses style overrides, formats, and traffic lighting. The third group introduces an entirely new method of defining styles, that is, Cascading Style Sheets (CSS).

STYLE DEFINITIONS AND STYLE ELEMENT INHERITANCE

REPLACE IS DEAD!

The REPLACE statement in PROC TEMPLATE is probably the most confusing and misunderstood part of the styles. The good news is that it no longer exists[1], because the style-element inheritance mechanism in PROC TEMPLATE has been completely re-written so that it acts more like the inheritance models in other object-oriented languages. When a style element is overridden in a style definition, any elements in the parent style definitions that inherit from the overridden style element acquire the new attributes.

For example, suppose you want to change the color of all table headers, including row headers and column headers. You look at `styles.default` using the PROC TEMPLATE SOURCE statement and see that all table headers inherit from the `header` style element. Therefore, changing the `header` style element seems to be the right thing to do.

```
define style mystyle; parent=styles.default;
    style header from header /
        foreground=red
    ;
end;
```

After running your SAS program, you notice that only the column headers have turned red. In releases prior to SAS 9.2, getting the results you want required the use of the REPLACE statement. However, the REPLACE statement has some problems. First, you must copy the style element attributes into the new style element, then you can override the attributes that you want to change. Furthermore, you can't replace an element from itself, so you have to examine `styles.default` to find which style element `header` inherited from. Luckily, in the case of `styles.default`, the `header` style element didn't have any previous attributes to be copied. Therefore, the following code accomplishes what you want.

[1] The REPLACE statement does still exist in the syntax, but it performs in the same way as the STYLE statement, which is used in its place

```
define style mystyle; parent=styles.default;
    replace header from headersandfooters /
        foreground=red
     ;
end;
```

However, in SAS 9.2, the code is as simple as the following:

```
define style mystyle; parent=styles.default;
    style header from header /
        foreground=red
    ;
end;
```

You might notice that this is *the same code as the original snippet* of code (shown under the second paragraph in this section). Therefore, in SAS 9.2, the code that you would probably try first is most likely to give you what you expect.

The REPLACE statement is also commonly used to change a font in the font table. As mentioned previously, when you want to change an attribute in a style element by using the REPLACE statement, you must copy all the attributes from the style element that's being replaced, then make the changes that you want. For example, if you wanted to change the document font in the `styles.default` style definition prior to SAS 9.2, you would write the code as follows.

```
define style mystyle; parent=styles.default;
   replace fonts /
      "docFont" = (" Times Roman, serif ", 3)
      "headingFont" = ("Arial, Helvetica, sans-serif", 4, bold)
      "headingEmphasisFont" = ("Arial, Helvetica, sans-serif", 4, italic bold)
      "FixedFont" = (Courier, 2)
      "BatchFixedFont" = ("'SAS Monospace', 'Courier New', Courier, monospace", 2 )
      "FixedHeadingFont" = ("'Courier New', Courier, monospace", 2 )
      "FixedStrongFont" = ("'Courier New', Courier, monospace", 2, bold)
      "FixedEmphasisFont" = ("'Courier New', Courier, monospace", 2, italic)
      "EmphasisFont" = ("Arial, Helvetica, sans-serif", 3, italic)
      "StrongFont" = ("Arial, Helvetica, sans-serif", 4, bold)
      "TitleFont" = ("Arial, Helvetica, sans-serif", 5, italic bold)
      "TitleFont2" = ("Arial, Helvetica, sans-serif", 4, italic bold)
   ;
end;
```

Note: The highlighted code is the only change being made to the font table.

That's a lot of work just to change one font! In SAS 9.2, not only will changes to style elements propagate through to elements in the parent style definition, but you can still inherit from the style element of the same name. Therefore, in SAS 9.2, the preceding code can be replaced with the following code.

```
define style mystyle; parent=styles.default;
   style fonts from fonts /
      "docFont" = ("Times Roman, serif", 3)
   ;
end;
```

As you can see from these examples, the new inheritance algorithm makes style element inheritance significantly easier to use.

CASCADING INHERITANCE

Even with the removal of the REPLACE statement, style element inheritance can still get pretty complex. However, there is an alternative method of inheritance that is available in PROC TEMPLATE styles that mimics the cascading

behavior of CSS. If a style element name is used more than once when a style definition is defined, the attributes of both entries are combined. If a style *attribute* is specified more than once, the latter specification is used. For example, the following two style definitions are equivalent.

```
define style style1;
    style data / foreground = blue;
    style data / background = white;
end;

define style style2;
    style data /
        foreground = blue
        background = white
    ;
end;
```

In and of itself, this is not very interesting. However, when combined with the feature that's described in the next section, you'll see that it is very powerful.

DEFINING MULTIPLE STYLE ELEMENTS SIMULTANEOUSLY

Prior to SAS 9.2, if you wanted multiple style elements to have the same attributes, you either had to copy-and-paste the attributes from one element to the other, or you had to use style element inheritance. In SAS 9.2, you can specify multiple style element names, simultaneously.

```
define style mystyle;
    style data, dataempty, datafixed /
        foreground = blue
        background = white
    ;
end;
```

Specifying multiple names, as shown in the preceding code, acts like a macro expansion. In the resulting template, each name that is specified shows up as a distinct element. If you used the SOURCE statement in PROC TEMPLATE to view the previous style definition, you would see the following code.

```
define style mystyle;
    style data /
        foreground = blue
        background = white
    ;
    style dataempty /
        foreground = blue
        background = white
    ;
    style datafixed /
        foreground = blue
        background = white
    ;
end;
```

Combining this feature with the cascading inheritance, which is described earlier, creates a much cleaner style definition than using traditional style-element inheritance. Here is the same code that was presented at the beginning of this section with a cascaded attribute added for `datafixed`.

```
define style mystyle;
    style data, dataempty, datafixed /
        foreground = blue
        background = white
```

```
    ;
    style datafixed /
        fontfamily = 'Courier'
    ;
end;
```

Using the SOURCE statement again, you see that the attributes for `datafixed` include all attributes from both STYLE statements.

```
define style mystyle;
    style data /
        foreground = blue
        background = white

    ;
    style dataempty /
        foreground = blue
        background = white
    ;
    style datafixed /
        foreground = blue
        background = white
        fontfamily = 'Courier'
    ;
end;
```

This method of inheritance does have a limiting condition. If the style definition has a parent, the style elements that are defined do not inherit the attributes from the elements in the parent style definition (there is no FROM clause). Also, if there are multiple names in the STYLE statement and only one location is specified in the FROM clause, it appears as though all the elements that are named must inherit from the same style element. To remedy this, the special keyword `_self_` was added. If the FROM clause in a STYLE statement contains the keyword `_self_`, each style element that is listed inherits from the style element of the same name. For example, the following two style definitions are equivalent.

```
define style style1; parent=styles.default;
   style data, dataempty, datafixed from _self_ / ... ;
end;

define style style2; parent=styles.default;
   style data from data / ... ;
   style dataempty from dataempty / ... ;
   style datafixed from datafixed / ... ;
end;
```

If you usually use the cascading style of inheritance when defining styles, you almost *always* want to inherit from `_self_`. A shortcut is described in the next section.

CLASS STATEMENT

When using cascading inheritance, it is so common to use the keyword `_self_` in the FROM clause that a shortcut, the CLASS statement,[2] is available. The CLASS statement is equivalent to the STYLE statement, but the CLASS statement includes an implicit `FROM _self_`. The equivalent style definitions in the following examples show the difference when using the CLASS statement.

```
define style style1; parent=styles.default;
   class data, dataempty, datafixed / ... ;
end;
```

[2] The term "class" was borrowed from CSS terminology.

```
define style style2; parent=styles.default;
   style data, dataempty, datafixed from _self_ / ... ;
end;
```

STYLE OVERRIDES, FORMATS, AND TRAFFIC LIGHTING

SAS FORMATS IN STYLE ELEMENTS

Prior to SAS 9.2, specifying a data format for a style attribute value was limited to being used in style overrides. This limitation made it difficult to share traffic lighting styles across table and column templates. In SAS 9.2, you can use data formats in style elements and in style overrides. The following example shows the use of a data format in a style element. This element can then be used in the STYLE= option or the CELLSTYLE-AS statement for a table or a column to get the traffic lighting effects.

```
proc format;
   value traffic 0-3 = 'red'
                 4-7 = 'yellow'
                 8-10 = 'green';
run;

proc template;
define style mystyle; parent=styles.default;
   style trafficlight from data /
      background = traffic.
      color = white
;
end;
run;
```

Now, instead of hard-coding your traffic lighting into an override, you can use your style element name. This method centralizes all the traffic lighting attributes, which makes it easier to make global changes. The following example shows the old way and the new way of applying formats for traffic lighting. You'll see that, when using the old way, you have to repeat the COLOR=WHITE attribute. If these attributes occurred in many places or across reports, your chances of missing some of them are greatly increased.

```
* Hard-coding the attributes used for traffic lighting;
define table mytable;
   define column one;
      style= {background=traffic. color=white} ;
   end;
   define column two;
      style= {background=traffic. color=white} ;
   end;
end;


* Using a style element for traffic lighting;
define table mytable;
   define column one;
      style= trafficlight ;
   end;
   define column two;
      style= trafficlight ;
   end;
end;
```

NEW VARIABLES IN THE CELLSTYLE-AS STATEMENT

While most of the following variables were made available in the CELLSTYLE-AS statement in SAS 9.1, they haven't received much publicity. These new variables are listed here for those who might not be familiar with them.

col	is the column number, which starts at 1
dataname	is the name of the data column that is associated with the output column
label	is the label that is associated with the data column
row	is the row number, which starts at 1
style	is the style element name that is used to render the column (this is new in SAS 9.2)

Of particular note is the _row_ variable. Using this variable in a CELLSTYLE-AS statement for a table enables you to create striped tables[3]. In addition, using the _col_ and _row_ variables enables you to address and modify any cell in a table. These techniques are discussed in the following section.

MERGING STYLE OVERRIDES

Style overrides can be made in the following ways: by using the STYLE= option in the table or the column template, by specifying the overrides in the CELLSTYLE-AS statement for the table or the column, and by including data formats in style elements. Prior to SAS 9.2, only one of these overrides could be applied at any one time. For example, if you specified a STYLE= option for both a table and a column in the table, only the value for the STYLE= option for the column would be applied. The precedence of these overrides, from high to low, was: the column's CELLSTYLE-AS statement, the table's CELLSTYLE-AS statement, the column's STYLE= option, and the table's STYLE= option. Because they were mutually exclusive, you would get some strange behavior when using the CELLSTYLE-AS statement on both the table and a column.

In SAS 9.2, the attributes from each of these override locations are merged to create the final result. The precedence for the styles remains the same as before. However, this new behavior makes it easier to create striped tables. Although striped tables have been easy to create since SAS 9.1 by the addition of the _row_ variable to the CELLSTYLE-AS statement, the stripes from the table couldn't be combined with the STYLE= option or the CELLSTYLE-AS statement for the column. Now that these overrides are merged, tables like the following are possible.[4]

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

Table 1. Example of Striped Table Created by Using the CELLSTYLE-AS Statement on Both the Table Template and Column Template

IMPLICIT PARENT TEMPLATES

While it is not technically an enhancement to PROC TEMPLATE styles, a new feature available in templates enables you to do things never before possible with styles. Now, table, column, header, and footer templates have implicit parents. That is, in the process of expanding the parentage of a template, when there are no more PARENT= statements, an implicit built-in template name is substituted. Following are the built-in template names:

Base.Template.Table	is the implicit parent for tables
Base.Template.Column	is the implicit parent for columns
Base.Template.Header	is the implicit parent for table headers
Base.Template.Footer	is the implicit parent for table footers

[3] Striped tables are tables that have alternating row colors.
[4] The code for creating this table is included in the Appendix of this paper.

By default, these templates are not defined. However, they will be used if they are defined by a user. So what does this have to do with styles? These implicit parents enable you to apply a STYLE= option or a CELLSTYLE-AS statement to all the templates at one time. For example, you can stripe the tables for all table template-based procedures by using the following table definition.

```
define table base.template.table;
   cellstyle mod(_row_, 2) as {background=#e0e0e0},
             1 as {background=#c3c3c3};
end;
```

CASCADING STYLE SHEETS

The ability to read CSS is entirely new in SAS 9.2. Even though many people think that CSS is used only by Web browsers, CSS works equally as well with media other than HTML and XML. In general, CSS styles are not that dissimilar to PROC TEMPLATE styles. In fact, when you look at ODS HTML output, you can see that most of the PROC TEMPLATE style elements and attributes map one-to-one with CSS classes and properties. So adding a CSS parser front-end to PROC TEMPLATE is a natural extension. The primary benefits are: 1) styles can use a documented, standard syntax; 2) style designers can use existing CSS tools; and 3) you can now use the style editor in SAS Enterprise Guide to create ODS styles.

Although ODS can use CSS styles, only a subset of the behaviors that are outlined in the CSS 2.1 specification is available. For example, you must only use CSS class names for selectors; no selectors that use IDs, element names, attributes, or combinations of selectors are accepted at this time. In addition, CSS class names must match the existing ODS style element names. Essentially, the CSS that you see when you generate HTML from ODS is the same type of CSS that PROC TEMPLATE expects as input. Following is a sample fragment of CSS that is usable by ODS:

```
Body
{
  font-family: Arial, Helvetica, sans-serif;
  font-size: small;
  color: #002288;
  background-color: #E0E0E0;
}
.Data
{
  font-family: Arial, Helvetica, sans-serif;
  font-size: small;
  color: #000000;
  background-color: #D3D3D3;
}
```

In addition to the preceding rules, not all CSS properties are supported, but this is true of all CSS implementations. Because support for new properties is always being added, the supported properties are not listed here. However, the general rule is that, if there is a PROC TEMPLATE attribute that maps one-to-one with a CSS property, that property is very likely to be supported.

There are two ways to use a CSS file as an ODS style. These methods are described in the following sections.

IMPORT STATEMENT

The new IMPORT statement in PROC TEMPLATE serves almost the same purpose as the `@import` rule in CSS. The IMPORT statement loads CSS content from an external file or a URL. The IMPORT statement can be used multiple times within a style definition. When PROC TEMPLATE code that contains the IMPORT statement is executed, the CSS content is read and translated into PROC TEMPLATE style elements. Here are some examples that use the IMPORT statement.

```
proc template;
define style mystyle;
   import 'mycssfile.css'; /* Local file */
   import 'http://www.mycompany.com/style.css'; /* URL */
```

```
   import mycss; /* Fileref */
end;
run;
```

In addition to the path to the file, you can limit the CSS content to specific media types, which enables you to put on-screen and print styles in the same file. Again, this usage of the IMPORT statement is very similar to its CSS counterpart. The following example illustrates how to load the CSS content that is associated with the PRINT media type.

```
proc template;
define style mystyle;
   import 'mycssfile.css' print;
end;
run;
```

You can specify as many media types as you want when you use the IMPORT statement. Media types must be separated by commas. In addition, because media type names are arbitrary, you can create your own media types that are not outlined by the CSS specification.

CSSSTYLE= OPTION IN ODS STATEMENT

The preceding section describes how to convert a CSS file into a PROC TEMPLATE style definition. However, if you are using CSS exclusively, there is no need to use PROC TEMPLATE. The CSSSTYLE= option in the ODS statement enables you to use CSS files directly without having to save the files as PROC TEMPLATE style definitions. The rules are the same as for the IMPORT statement, but the syntax is slightly different. Here are some examples that use the CSSSTYLE= option.

```
ods html cssstyle='mycssfile.css'; /* Local file */
ods pdf cssstyle='http://www.mycompany.com/style.css'; /* URL */
ods rtf cssstyle=mycss; /* fileref */
```

The CSSSTYLE= option also supports media types, which makes it easier to put the online and print CSS styles in the same file.

```
ods html cssstyle='mycssfile.css'(screen);
ods pdf cssstyle='mycssfile.css'(print);
```

You can specify multiple media types when you use the CSSSTYLE= option. However, in this case, media types must be separated by blanks or white spaces.

STYLE EDITOR IN SAS ENTERPRISE GUIDE

The style editor in SAS Enterprise Guide was originally designed to create ODS-compatible CSS files. Therefore, the files that are created by this style editor are now readable by either the IMPORT statement in PROC TEMPLATE or the CSSSTYLE= option in the ODS statement.

CONCLUSION

The extensive changes made to PROC TEMPLATE are worth the cost of upgrading to SAS 9.2. Even if you are suffering from PTSD (PROC TEMPLATE Style Disdain) from a previous experience, you should re-visit PROC TEMPLATE styles. You will be pleasantly surprised.

APPENDIX

CREATING A STRIPED TABLE USING THE CELLSTYLE-AS STATEMENT FOR COLUMNS

```
proc template;
define table mytable;
   cellstyle mod(_row_, 2) as {background=#e0e0e0},
             1 as {background=#c3c3c3};
   column name age sex height weight;
   define column weight;
```

```
      cellstyle _val_ >= 110 as {foreground=red},
                _val_ >= 90 as {foreground=yellow},
                1 as {foreground=green};
   end;
end;
run;

ods pdf file='striped.pdf';
data _null_;
   set sashelp.class;
   file print ods=(template='mytable');
   put _ods_;
run;
ods pdf close;
```

RECOMMENDED READING

Bos, Bert, et. al. 2005. "Cascading Style Sheets, Level 2 Revision 1, CSS 2.1 Specification." Available www.w3.org/TR/CSS21/.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author:

Kevin D. Smith
SAS Institute Inc.
SAS Campus Drive
Cary, NC 27513
Kevin.Smith@sas.com