Notes on Programming Style

Allan Glaser
Octagon Research Solutions, Inc.

Abstract

Programming style is important. A small but growing body of literature suggests that style is a critical component of programming quality, and many authors offer specific guidance about what good style should, or should not, be. Unfortunately, there is very little scientific evidence proving that any specific style really has an advantage over any other style. This paper will identify some key elements of programming style, discuss techniques to measure those elements within the context of SAS® programming, suggest ways to use the results for process improvement, and propose topics for future research. The intent is to inject clarity, rationale, and objectivity into the issue of programming style; to encourage programmers to recognize the importance of good style; and to encourage managers and organizations to implement reasonable and beneficial standards. Overall, this will enhance the art of programming.

Background and Introduction

A computer program is merely a communication. It communicates unambiguous instructions to the computer, which executes those instructions. But the program also must communicate to people who are charged with using, maintaining, enhancing, refining, explaining, and correcting the program. Those activities may involve a large number of people over an extended period of time. Programs must be written so that human readers can clearly and quickly understand what a program is trying to do and what it is actually doing. Just as style is important with everyday prose, style is an important factor in programming.

Consider the following two paragraphs. The content is identical, but the styles are different. We have a greater ability to read and understand the first paragraph than the second, and to do that more quickly.

Programming style is important. That is self-evident, when we consider that misuse of case, white space, etc., will create a document (i.e. program) that is confusing and cannot be easily read.

Programming Style is Important, that is Self-Evident, when WE consider that MisUse of case, white SPACE, ETC., will create a Document (I.E. program) that is confusing and Cannot be easily read.

Although extreme, this illustrates the fact that good programming style contributes to high quality programming, and poor style leads to poor quality programming.

It can be argued that style is subjective, i.e. each programmer communicates in his or her own way. Although true, there are certain conventions that are widely accepted in other forms of written communication such as newspapers and books, and those conventions contribute to clarity and understanding. Following conventions with SAS programming will have a similar positive impact.

Two major issues must be addressed with programming style. First, does an individual’s style conform to generally accepted conventions and standards? These may be standards that an organization has mandated or that have become de facto standards. Second, recognizing that consistency is a hallmark of quality, is an individual’s style consistent across his or her
programming products? Although an individual’s style can – and should – adapt to fit different situations, it should not be random or haphazard.

Nothing is simple, of course, and questions immediately arise. What are some attributes of programming style? How can those style attributes be measured objectively? Once the measurement results are available, how is the information interpreted and what actions are taken? What are some caveats and concerns with using this information?

**Style Attributes**

Style attributes can be divided into two distinct groups. One group consists of attributes that are purely stylistic and do not contribute to actual programming logic, and one frequently cited example of this is the use of white space. The other group includes programming constructs that are part of the programming logic, such as the use of `IF...THEN...ELSE` or `SELECT...WHEN` statements.

Pure style attributes include vertical white space or blank lines, horizontal white space or spaces within a line, indentation or blanks preceding a line, and the use of comments. In most situations, the selection of upper/lower case is also in this category. The judicious use of these attributes plays a large role in ensuring a program is readable by others. A simple illustration helps to make this point. We have all seen code like the following, and most of us have also written code like this, although we may be reluctant to admit it:

```plaintext
Data XXX;
Set YYY;
length Age_category $11;
If AGE>65 Then Age_Category='elder';
Else if AGE>55 Then AGECATEGORY='senior';
Else If AGE>40 Then Age_Category='adult';
   else if age>30 then    Age_category='young adult';
Else If age>19 Then age_Category='youngster';
else If age>   12 then age_category='teenager';
else age_category='child';
RUN;
```

But we have also seen the same code written differently and silently appreciated the author who constructed it:

```plaintext
/***  determine the age category of each person ***/
data xxx ;
  set yyy ;
  length age_category $11 ;
  if         (age > 65) then age_category = 'elder' ;
  else if (age > 55) then age_category = 'senior' ;
  else if (age > 40) then age_category = 'adult' ;
  else if (age > 30) then age_category = 'young adult' ;
  Else If (age > 19) then age_category = 'youngster' ;
  else if (age > 12) then age_category = 'teenager' ;
  else                   age_category = 'child' ;
r un ;
```
The visual appearance of the second example immediately conveys the overall structure and logical flow to the reader. The details are also clearer. Any typographical errors would be more easily detected. Any logical omission would be more readily discovered. Note the obvious differences in style, e.g. the use of indentation, embedded white space within a line, the use of a comment, a blank line, and upper / lower case. The vertical alignment of similar information obviously plays a key role in readability. Also note the use of parentheses to clearly show the expression that is being evaluated.

Rigidly adhering to a consistent style should not be done when it impacts the overall clarity of the program. For example, the last ELSE statement in the sample code above could be rewritten as follows, but this is not a common convention and may be misunderstood or misinterpreted:

```
else if ( 1 ) then age_category = 'child' ;
```

**Measurement of Style Attributes**

Although the reader can immediately evaluate the above code samples subjectively, we need to evaluate them objectively. This can be aided by a simple SAS program that reads source code, measures the various attributes, and reports the results in a meaningful way. It is useful to make some reasonable assumptions, e.g. a horizontal tab character is equivalent to three spaces. The overall function and flow of the program can be very simple, as this outline illustrates:

```
data ;
  infile <filename> truncover ;
  input @1 code_line $char200. ;
  <determine if line is blank>
  <count comments>
  <count characters>
  <compress and recount to determine horizontal whitespace>
  <check first character(s):  comment?  certain keyword?>
  <count upper/lower case>
  <additional counting, etc.>
  ...
  proc means sum mean max ;
```

Although this approach yields a very workable tool, there are limitations. Most obvious is the fact that in the target code that is being evaluated, a one-to-one correspondence is required between non-blank physical lines and SAS statements. In other words, no more than one SAS statement can appear on a line, and a statement cannot wrap across multiple lines. In practice, this is a very commonly used convention and presents little difficulty, and greatly simplifies the coding.

As in other programming situations, the devil is in the details. The INFILE statement should use the TRUNCOVER option. Determining the horizontal white space requires counting the characters on a line in multiple ways. The LENGTH function gets the overall line length, the composition of the LENGTH and LEFT functions, i.e. LENGTH(LEFT()), will determine the non-indented line length, and LENGTH(COMPRESS()) will yield the number of non-blank characters. Together, these can be used to determine the total amount of horizontal white space.

There are limitations with this simple approach. Spaces that are included within quoted strings should not typically be regarded as white space, but properly recognizing those spaces greatly complicates the code. The error resulting from taking the simple approach is likely acceptable in most situations.
Examining some specific coding techniques, the TRANWRD function can be used to convert horizontal tab characters (hexadecimal 09) into a predetermined number of spaces:

```plaintext
code_line = tranwrds(code_line, '09'x, repeat(' ', 2)) ;
```

and the COUNTC function can be used to count the number of upper case characters:

```plaintext
n_upper = countc(code_line, 'ABCDEFGHIJKLMNOPQRSTUVWXYZ') ;
```

Different programmers and different organizations may want to examine specific style attributes, but adapting the programming approach outlined above is straightforward and should suffice in most situations.

**More About Indentation**

There seems to be universal agreement among programmers that indentation is important, regardless of the chosen programming language. In ordinary prose, indentation denotes the beginning of a section of related information, e.g. the start of a paragraph. Indentation within a computer program, however, follows outline rules, i.e. subordinate information is indented. In other words, high level programming logic is leftmost, secondary logic is indented, tertiary logic is indented further, and so on.

At the highest level, this is easy to measure – lines between a DATA or PROC statement and the subsequent RUN or QUIT statement could be examined. This does not give the complete picture, because we also want to know if a group of statements within a DO...END block are indented, and there are other similar complications.

Context is also important when we examine indentation, but that cannot be easily determined programmatically. For example, the bulk of one program may be a lengthy PROC FORMAT, in which all of the subordinate VALUE statements and conversion values are indented. If the shop standard is to indent subordinate statements three spaces, then the measured average indentation will be very close to, and slightly lower than, 6. On the other hand, a program with a large number of separate short DATA and PROC steps may be nicely indented, but the measured average indentation would be lower than 3.

Providing a complete and meaningful measure of indentation is beyond the scope of this work.

**Measurement of Alignment**

Vertical alignment is related to indentation, and is another important component of programming style. The initial approach to measuring alignment was to remove null and comment lines and then compare each line with its predecessor. Matching characters were counted and that number was divided by the average number of characters on the two lines. Indentation and embedded white space were preserved in these comparisons. Higher ratios indicated both code similarity and alignment.

One variation of that approach looked only at the similarity between consecutive lines with regard to the placement of white space and some symbols such as parentheses and the equals sign. Yet another, and simpler, approach looked only at the variability of indentation between consecutive lines.
A single robust technique to measure vertical alignment has not been found, and further refinements of these approaches have not been explored. As with indentation, alignment is also related to context. Additional research is needed.

**Measurement Results and Interpretation**

The output generated by `PROC MEANS` in the programming approach outlined above is simple but adequate. It will be informative to look at the details of a specific sample output. For the purpose of clarity, meaningless information has been omitted:

```
METRICS FOR C:\PROJECT X FILES\SAMPLE-CODE.SAS

The MEANS Procedure

<table>
<thead>
<tr>
<th>Variable</th>
<th>Label</th>
<th>Sum</th>
<th>Mean</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>lif</td>
<td>lines in file</td>
<td>1385</td>
<td>6.04</td>
<td></td>
</tr>
<tr>
<td>indent</td>
<td>spaces indented</td>
<td>6.04</td>
<td>8.52</td>
<td>57.36</td>
</tr>
<tr>
<td>white</td>
<td>embedded non-leading blanks</td>
<td>8.52</td>
<td>6.04</td>
<td>172</td>
</tr>
<tr>
<td>line_len</td>
<td>length of line</td>
<td>57.36</td>
<td>172</td>
<td></td>
</tr>
<tr>
<td>comments</td>
<td>non-macro comments</td>
<td>28</td>
<td></td>
<td></td>
</tr>
<tr>
<td>com_macs</td>
<td>macro comments</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ncss</td>
<td>non-comment source statements</td>
<td>1297</td>
<td></td>
<td></td>
</tr>
<tr>
<td>n_if</td>
<td>if statements</td>
<td>39</td>
<td></td>
<td></td>
</tr>
<tr>
<td>n_select</td>
<td>select statements</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>n_do</td>
<td>do statements</td>
<td>47</td>
<td></td>
<td></td>
</tr>
<tr>
<td>n_data</td>
<td>data steps</td>
<td>29</td>
<td></td>
<td></td>
</tr>
<tr>
<td>n_proc</td>
<td>proc steps</td>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>n_lower</td>
<td>lower case alpha characters</td>
<td>55125</td>
<td></td>
<td></td>
</tr>
<tr>
<td>n_upper</td>
<td>upper case alpha characters</td>
<td>8379</td>
<td></td>
<td></td>
</tr>
<tr>
<td>v_align</td>
<td>consecutive line similarity</td>
<td>0.14</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Of course, the `PROC MEANS` output could be directed to a dataset, and that could be used to create a more user-friendly output. For example, the results above might be used to generate:

```
C:\PROJECT X FILES\SAMPLE-CODE.SAS has
- 1385 lines (4.1% blank)
- 1328 statements (2.3% comments)
- 36 steps (29 data, 7 proc)
- 86 if/select/do statements (39/0/47 respectively)
- 0.13 fraction of alpha that is upper case
- 6.04 average spaces indented
- 8.52 average embedded spaces per line
- 57.36 average line length
- 172 characters in longest line
- 0.14 fraction of characters same as previous line
```
With this specific example, there is a wealth of information about the programming style, some of which leads to questions. Not necessarily in order of importance, we note that:

- Indentation is being used. Does the amount of indentation properly reflect the context of the program?
- There is embedded white space. Again, is the number appropriate within the context of the program?
- There is roughly one blank line per 25 lines of code. That may not be enough to ensure easy reading and proper interpretation of the code.
- There is approximately one comment per 40 lines of code. That may seem sparse – is it sufficient?
- A combination of upper and lower case is used. This may reflect an organization or individual preference, the inclusion of quoted character strings, or a confusing style. In general, lower case code is easier to read and is preferred.
- This program contains a large number of DATA steps. Is the program overly complex?
- The longest line contains 172 characters. Will that be printed / read properly?
- Approximately 14% of the characters on each line of code were the same, and in the same position, as in the preceding line. That is about 8 characters, slightly more than the average indentation. This may indicate weak alignment, depending on the context.

Some of the numbers stand alone. For example, we would generally expect a program to contain comments, and seeing that a program contained very few comments would raise concern. Similarly, a long average line length is uncommon and would lead to a deeper look at the program.

But in general, the numbers cannot be used alone. They are meaningful only within the context of the program.

Special attention should be given to the use of mixed case. As longer variable names have become popular, those names are often formed by concatenating multiple words and using upper case for the first letter in each word, e.g. "ShipToAddress" or "Ship_To_Address". Since this is inconsistent with common English usage, it tends to create confusion or, at the very least, to increase the time needed to read and understand the code. A similar argument can be made for using all upper case names or keywords. SAS programs that use mixed case or predominately upper case and which are robust and easy to understand seem to be relatively uncommon. It is far better to consistently use lower case, simply because that is what we are accustomed to reading.

**Additional Style Attributes**

There are numerous style attributes that have not been identified above. Some of these can be readily measured, others cannot.

One of these attributes is the length of variable names, e.g. "acct_num" vs "account_number". Conciseness leads to clarity, but so does a complete name. Longer names may substantially increase a program’s volume, and that will also tend to impact readability. Some industries or organizations may require shorter names to ensure compatibility with older systems or other applications.

SAS allows different programming constructs that are equivalent, and the programmer must decide what to use. For example, in a logical comparison **EQ** is the same as the equal symbol "=". This author has observed that SAS programmers who have a background or education in the mathematical sciences tend to use symbols, while programmers who have a different background or education tend to use the letter abbreviations.
Organizations frequently mandate a specific header block format that should be placed at the top of each program. These typically include the program name, function, author, revision history, and similar data. The presence of this block, the number of lines, and the inclusion of specific lines could be measured.

It is common, but not universal, practice to clearly identify the end of a logical block of statements. This means including the name of the macro on a %MEND statement, adding comments after END/%END statements that refer to the associated DO/%DO statements, and similar coding.

The generic programming approach described above can be readily adapted to measure these other attributes and report the results.

**Future Direction**

Many different style attributes have been identified, along with a suggested approach for objectively measuring some of them. It would be interesting for an organization to take a set of SAS programs, measure style attributes, and subjectively evaluate each program for quality, say on a 1-to-10 scale. A logistic regression model could be constructed that would indicate which style attributes contributed most significantly to programming quality, and by how much. An organization could then apply that information to develop or solidify its own programming style guidelines, and a simple tool such as the one noted in this paper could be used to help ensure compliance. The up-front attention to style could reasonably be expected to lead directly to improved quality, which yields many benefits.

An individual programmer can also use this approach to help ensure that his / her programming products are consistent. Consistency is universally recognized as an important factor of quality.

A very preliminary analysis with limited data suggests that the author of a program can be identified based upon the style attributes of the program. In other words, a program's style is a form of signature. This sounds very abstract, but actually offers a very practical benefit. If an organization is working with older programs that may not be well-documented, it may be possible to use this technique to identify the author, who may still be available for consultation.

Robust objective measures of indentation and vertical alignment have been elusive, and this may be a fertile area for research, experimentation, and discussion. But there are rough measures of these that can be useful if we also consider the context of the program.

**Conclusion**

We would not patronize a grocer that displayed merchandise in an unorderly fashion, or a bank that gave us garbled and hard-to-read statements, or a physician who did not maintain a tidy office. We would naturally question the quality of the products or services. High quality, or the lack of it, is reflected in the styles in which products and services are presented to us. Our customers and coworkers deserve the same attention to quality in our programming products.

Programming style is not an absolute. There is minimal scientific evidence that supports the use of one particular style over another, and common sense must prevail. It must be understood that a good program will enhance communication with people, e.g. future maintenance programmers. An organization may decide to mandate a specific style, and that may be dependent on the context of the programming, the perceived need to share source code with others, program complexity, the anticipated lifetime of the code, and other factors.
Programming style cannot be ignored. Style is an important component of overall programming quality. Rather than rely solely upon subjective and casual observation, a simple SAS program can be used to objectively measure selected style attributes. The results can be used by individual programmers to help develop and maintain mature, consistent, and high quality styles, and by management to help ensure that the organization's programming products are high quality and meet expectations.

Standards and conventions exist for good reasons. Without them, communications suffers. They should not necessarily be regarded as inviolable, but rather as sound guidance. Arbitrary deviations should be minimized.

**Acknowledgments**

The author thanks colleague Marianne O'Connor for fruitful discussion and insight. Constructive comments made by reviewers of a draft of this paper are also appreciated.

**References**


SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Allan Glaser is a member of the Clinical Programming management team at Octagon Research Solutions, Inc., and can be contacted at a glacier@octagonresearch.com. Comments regarding this work would be appreciated.