

Paper 134-29

Writing Code That Lasts

Sandra Minjoe, Genentech, Inc., South San Francisco, CA

ABSTRACT

With SAS, there are often many different ways to write code that provide the same answer. So how do you decide which method to use? The answer depends greatly on your environment. This collection of tips can help you decide which programming method makes the most sense for your situation.

These tips are useful for anyone interested in writing code that you can walk away from (and not have to worry about). This could be in your role as a contractor, or just because you want to be promoted or moved into another job area you find more interesting. New programmers can use these tips to start them off on the “right track”.

This paper contains both suggestions and warnings to assist you in writing code that is easy to read and understand, flexible, portable, and that requires little future modifications. In other words: code that lasts.

INTRODUCTION

As a programmer, I’m often put into situations that are not drastically new. When someone before has handled a similar issue, it seems that the fastest solution for me should be to use that code, modifying whatever is needed to fit the current task. But too often this is not possible, because the code isn’t something that anyone but the original author can easily understand.

Sometimes I can contact the original author to make the modifications that are needed for the new situation, or have them help decipher the code. More often, it turns out that the original author is no longer available. Thus I’m left to either delve into the program in hopes that I’ll learn it well enough to make the necessary modifications, or scrap it and start over from scratch. Both are time-consuming options, and both are a waste of money.

If we take a step back, though, we can see that situations like the one described above could easily be avoided if the program had been originally developed to be more understandable. This paper focuses on how to initially write code that others can easily use in the future. No, it won’t solve the current problems with existing confusing code, but could help prevent it from happening again.

WHAT IS CODE THAT LASTS?

Code that lasts is easy to read and understand, flexible, portable, and requires little future modifications. It doesn’t happen by accident. It happens because we are aware of our programming environment, our coworkers, and our customers. It happens because, while writing code, we think about those that will come later.

We’ve all discovered that there are often many different ways to write code that produces the same results. Good programming is more than just writing code that works, it’s writing the “best” code that works. However, the way “best” is defined differs between industries, companies, and sometimes even departments within the same company. For example, those with slow processors are more concerned about speed than those dealing with space limitations; different companies buy and use different modules of SAS®; and some love PROC SQL while others hate it.

Because of these and other differences, we can’t have a set of rules for the entire SAS® user community. On the other hand, there are a number of things we can do to make our code long lasting. The following few simple ideas can help you in any industry and company, and in any language (not just SAS®):

1. Clarity before cleverness
2. Document, document, document!
3. Choose names wisely
4. Write for your final audience
5. Avoid “Go To” code
6. Take advantage of external sources
7. Use in-house standards and tools
8. Use standard indenting and program layout
9. Check for missing and unexpected data
10. Practice code reviews with your coworkers

Here are some details, examples, and suggestions about how to apply each of these ideas to your code, so that we can all create code that lasts.

1. CLARITY BEFORE CLEVERNESS

Sometimes programmers strive to write the fastest running code possible. But is shaving 0.5 seconds off the run time by using a complicated `set` statement really helpful if people can't understand how it works? If that chunk of code is in a loop that needs to be executed a few thousand times in a program that is run daily, then maybe it is. But in most cases, the amount of time spent running the program is minor compared with the time spent by others trying to understand and re-use it.

There are programmers who feel that they can demonstrate how smart they are by writing tricky code, and by using features that their coworkers are not aware of. They think of this as a way of achieving "job security". These programmers believe that because they are the only ones who can easily understand and thus update their code, they will never be fired or always be brought back at a high hourly rate. This attempt at job security can backfire, though. I've seen cases where people are stuck in a job and never promoted, or even have a difficult time scheduling a vacation, just because no one else can understand their code and thus take over for them. And I've seen contractors black-listed and the company spend money re-doing their work because they don't want to deal with that kind of coding again.

I'd like to suggest an alternative: making complicated code appear simple is a talent. In a technical writing course, I learned that we need to present difficult material in a way that is as comfortable as possible for the reader. For example, a technical document being developed for high school graduates might be written at an 8th grade reading level. In this way, the readers don't need to navigate both difficult technical concepts plus a fancy writing style. I think the same idea can be applied to programming: the more complex the idea we're trying to get across, the more important it is to use simple and straightforward code.

One final warning on "clever" code: we often come away from a training class or a conference (like this one) with all sorts of new techniques to use. It's easy to get excited and try to apply these new ideas in situations that don't really call for them. To help make your code more useful in the future, use only the minimum required complexity for the task at hand.

2. DOCUMENT, DOCUMENT, DOCUMENT!

Documentation can simply be thought of as a laymen's explanation of what the code is doing. The amount of documentation necessary in code will vary, but generally the more complicated the program, the more documentation that is needed.

As a simple example, documentation is probably not needed for code such as:

```
if      sexcode = 'M' then sex = 'MALE';
else if sexcode = 'F' then sex = 'FEMALE';
```

but we might want to include a comment if a format is used instead:

```
sex = put(sexcode, sexfmt.);
```

In this second case, it would be useful to comment on where the format is derived and what it does. For example, a comment directly preceding the code, such as

```
* Use format sexfmt., defined in formats.sas, to convert codes M and F;
*   to MALE and FEMALE;
```

would go a long way to helping the reader understand what is happening with the format.

In SAS® there are different ways to comment. In addition to the `*` comment ; style, we can, for example, use a `/* comment */`

The second option has the advantage of not requiring a semicolon, so it can span several SAS® statements or be put within a statement. The disadvantage is that some programmers like to use the `/*` and `*/` to temporarily block large sections of code from running; this method of commenting can cause trouble when a `*/` is found before it was expected, due to nested `/*` and `*/` comments.

Keep in mind that the commenting method we use should be consistent and based on our company's preference, if there is one.

Some companies have standards of what should be included in comments at the top of a program. This might include information such as program name, author name, purpose, input datasets used, external code and macros used, version of SAS® used, and output generated. Additionally, some companies require a comment at the end of the code to signal its terminus, such as:

```
*****;
* End of program ABC.SAS *;
*****;
```

If documentation standards exist, we should use them. This will help others to read our code, because it will look like code they are already familiar with.

In terms of timing, some programmers find it easier to write comments at the same time as they code. Others prefer to code first and comment later. I tend to do a little of both, commenting complicated work as I go along to help me figure out what I'm trying to do, and then going over the program later to add more details. In this way if I get pulled away to another task and can't finish the program right away, at least when I eventually get back to it I'll have a general idea of what I was doing at the time.

If you include comments so that a year from now you could pick up the code and understand what it does, then others probably will understand it, too. Good documentation can make even the most complicated code usable. And useable code is code that lasts.

3. CHOOSE NAMES WISELY

In programming classes, I was taught with variable names that were short but so generic they did not describe the data. For example:

```
x = a - b;
```

doesn't tell me anything about what variables x, a, and b are. It is much more clear to use meaningful names, such as

```
diff = month4 - baseline;
```

or

```
diff_wt = mo4_wt - bl_wt;
```

Of course, with SAS® v7 and later, we can use variable names longer than 8 characters, and this allows for even more descriptiveness. Keep in mind, though, that if version 5 transport files are needed to send out data (such as is required in an FDA submission), we're stuck with the 8-character limit and need to be especially creative.

One distinct advantage to using meaningful names is that we can often get away with less documentation. In an earlier example, I said that the code

```
if sexcode = 'M' then sex = 'MALE';
else if sexcode = 'F' then sex = 'FEMALE';
```

probably did not need documentation. If that same code had meaningless names, such as

```
If a = 'M' then b = 'MALE';
else if a = 'F' then b = 'FEMALE';
```

then documentation might be necessary.

Meaningful names are also useful for naming datasets as well as variables. Frequently I see programs with work datasets, used during the running of a program, called by names such as a, b and c. The justification for using these meaningless names is usually that it saves time typing. But again, how much time do others lose when trying to decipher the code?

A few extra minutes spent typing during program development can create much more understandable and long-lasting code.

4. WRITE FOR YOUR FINAL AUDIENCE

Our final audience is often the other programmers and managers within our own company. Remember that we are writing code for their use as well as our own, and need to consider their experience level. If most of our coworkers are at a more junior level, it won't be easy for them to decipher complex code.

That doesn't mean we should never make use of any advanced techniques. If we want to use more complicated code, we just need to be sure to document what it does and how it works. We might also want to hold a little "coders' corner" or training session about that nifty piece of code, so that our coworkers and managers can learn from us.

Sometimes our coworkers aren't the only ones with access to our programs. If our customers are even remotely knowledgeable about coding (and these days many are), it can be helpful to include comments in the code written especially for them. For example, many years ago I worked with a county budget department to develop some budget prep software. The budget analysts helped me understand the topics I was writing code for, and

also helped me write comments in their own “budget” terminology. After I left the county, I heard from another programmer that the budget analysts now make requests like “the code in program XYZ on lines 275-280 needs to change to instead do blah-blah-blah.” Because the users understood the comments surrounding the code and they could direct the programmers, this code ended up very easy to modify by others well after I was gone.

Currently I work for a biotech company, and understand that the FDA is a likely “customer” of my code. FDA reviewers are statisticians and MDs, and they have no programming support. Most of the statisticians know some SAS® and other statistical packages, but aren’t professional SAS® programmers. Thus I need to write code and comments that will be understandable by them.

A little background about our customers can be very helpful in creating code that lasts.

5. AVOID “GO TO” CODE

Some of the most confusing code to understand is that written with go to statements. It’s hard to get a feel for program flow when forced to jump around with go to’s, so it can be difficult for the reader to understand the point.

Because go to’s can easily lead to infinite looping, most schools of programming discourage using them. However, some programmers find the freedom of movement allowed with go to’s too tantalizing to resist. If tempted to use a go to, remember that there is probably a more clear way to get there.

6. TAKE ADVANTAGE OF EXTERNAL SOURCES

In this electronic age we live in, detailed specifications are often given to us in a Microsoft ® Word table or Excel spreadsheet. When writing a program from these specs, we’ll usually turn around and re-type that information into SAS® code. This can introduce typos, especially if we’re working with long text strings. When the specs have quite a bit of text to them, we might save a little time by copying sections of the specs and pasting them into the code. If the specs change later (and who hasn’t seen that happen!), the code must then be manually revised.

Instead of manually typing code, or even copying and pasting from specs we received electronically, we can programmatically bring that data into SAS® as a dataset to use directly. For example, if we receive a spec in Excel titled “Group 1 Terms.xls” that is laid out as:

Term
Aaaaa
Bbbbb
Ccccc
Ddddd
...

we can easily import this into a SAS® dataset called GROUP1 with the variable called TERM.

We can then write code to convert this long list of terms into a character string that can be searched. For example, consider the following code:

```
* Create macro variable to hold all Group 1 terms, based on specs from Dr. X';
* titled 'Group 1 Terms.xls' and converted to SAS dataset group1 in the input;
* directory.    ;
data _null_;
    set input.group1;

    * Create long text to temporarily hold all terms, until defined to a macro var ;
    format longtext $5000.;
    retain longtext;

    * Add each term to the end of long text;
    longtext = trim(left(longtext)) || ', ' || group;

    * Create macro variable from long text;
    call symput ('grp1list', longtext);
run;
```

This code creates a macro variable called grp1list that contains all these text terms in a list. Later in the code, when we want to keep only those terms from our data that are in Group 1, we can simply say:

```
if term in (&grp1list) then output;
```

This same idea can be expanded to Excel spreadsheets with multiple columns by creating SAS datasets with multiple variables. And with only one extra step, we can even tackle specs given to us in a Word table, by first converting to Excel and then to a SAS® dataset. In fact, there is probably a way to convert any electronic spec, from any software, into a SAS® dataset.

A huge advantage to automating this task is that the code doesn't need to be updated when specs change. Instead we simply need to re-do the steps that create the SAS® dataset and macro variables, and then we can use those updated variables. Keep in mind that because this process of converting from electronic specs to code involves multiple steps using multiple software packages, we should be sure to document it in such a way that all the files can be found and all the steps run.

Whenever we receive detailed electronic specs that need to be brought into our code, we should consider, instead of manually re-typing or copying/pasting them, bringing those specs into a SAS dataset and using them to create macro variables. Code such as this lasts a long time because even when specs are updated, it doesn't need to change.

7. USE IN-HOUSE STANDARDS AND TOOLS

We all like to be creative. It keeps our jobs from getting dull. And sometimes we programmers don't like using standards or other tools because they impinge upon our freedom to write code our own way. Instead try to look at the many advantages of using standards:

- Using a standard or tool instead of writing our own code will probably take less time.
- Standards and tools can't possibly cover everything, so less time spent on the boring everyday stuff that they cover means we'll have more time for the unique and interesting projects.
- Standard code and tools are designed to be usable by other programmers, so we can go on vacation, get promoted, etc. and not worry about someone else getting confused by the code.

In other words, we need to let standards work to our advantage. Besides the fact that standard code will last simply because it is standard, using it also gives us time to pursue more exciting challenges.

8. USE STANDARD INDENTING AND PROGRAM LAYOUT

Company standards regarding indenting and other program layout make it much more comfortable for others to read our code. Whether the standard is indenting 2 or 3 spaces, leaving a blank line between data steps, or lining up our end and run lines of code, this is an easy way to make our code readable.

Consider these two code examples:

```
*Example 1: Code with no layout;
data labstot;
set labs; length high $3;
if day > 0 then do;
if gr < 3 then high='NO';
else high='YES';
output; treat = 'Total'; output;
end;
proc sort data=labstot;
by treat patient day high;
run;
```

```
*Example 2: Code with some layout;
data labstot;
set labs;
length high $3;
if day > 0 then do;
if gr < 3 then high = 'NO';
else high = 'YES';
output;
treat = 'Total';
output;
end;
run;

proc sort data=labstot;
by treat patient day high;
run;
```

Even with no additional comments, example 2 is easier to follow. In example 1, it is hard to tell that only those records with day > 0 are output, let alone twice. But in example 2, indenting the code between the if and the end and splitting up the three tasks of

```
output; treat = 'Total'; output;
```

onto three lines makes this much more clear.

Determining what the company standards are and following them will make our code easier for others to read and thus re-use.

9. CHECK FOR MISSING AND UNEXPECTED DATA

The data that we work with is nothing like the stuff they gave us to use in programming classes. Let's face it: we have to deal with dirty data. Often we have no control over how it is entered and we're often surprised by what we find. Sometimes we end up with unexpected results in a variable, and sometimes a variable that should have a value is missing. If we write code to cover those conditions that we don't expect but could happen, it can save us time later.

Consider the case where we write a program that will be run on data received, say, once a week. Each time a new set of data passes through the code, any number of entry errors could be included. We don't want to have to rewrite code or do a lot of extra data checking every week. If the code is written to cover the possible data errors, then our output will contain only what it should.

For example, we might expect the variable sex to contain either "M" or "F", and could write

```
if sex = 'M' then output males;
else          output females;
```

But what if on some records sex is missing or contains something other than "M" or "F"? With the code above, those records would end up in the females output dataset, which would probably cause us some trouble later on in the program.

What we can do instead is to specifically look for each value and handle the remaining cases separately. If we change the code to

```
if      sex = 'M' then output males;
else if sex = 'F' then output females;
else          output baddata;
```

then the females dataset contains only the data we expect. Any surprises go into the baddata data set, which needs, of course, to then be checked.

Another option is to print messages to the log for checking. Instead of the last "else" code from above, outputting problem records to a separate dataset, we could do the following

```
else put 'Entry error: ' patient = sex =;
```

to print our concerns to the program log. In this case, the log and not a separate dataset would need to be checked.

Coding for unexpected data is not only useful for the situation mentioned above, where we receive new versions of the dataset on a regular basis, but it also allows our code to easily be used in other places. The next time we need to deal with these same types of variables in other datasets, we can copy and re-use that chunk of code and trust that it will work even if something completely different is wrong with that data.

By writing code that checks for the conditions we expect, plus including an escape clause for anything else, we go a long way toward making code usable on other data, long after it is written.

10. PRACTICE CODE REVIEWS WITH YOUR COWORKERS

It sounds intimidating at first, but having someone else read and critique our code can be a valuable learning experience for both the writer and the reviewer. By reading the programs of our coworkers, we can each learn how the others code, compare different coding styles, and even spark discussions about other coding challenges. I've personally picked up some wonderful techniques in code reviews, and have hopefully been able to contribute some as well. And whenever another programmer has questions with our code during a review, this provides a signal of where we could include more documentation.

Additionally, because code reviews get us familiar with each other's code, it is then easier for us to give and get help when needed. It will take less time to get up to speed on a new project when we already understand some of the current programs and our fellow programmers style of coding.

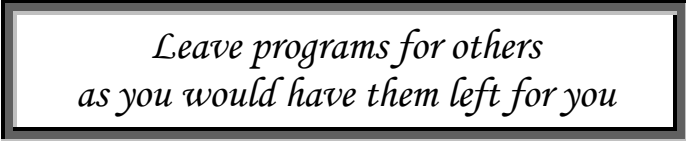
Code reviews allow us to improve our code as we're writing it, and help others to understand it; both of which help make that code last.

CONCLUSION

You are probably already practicing at least a few of the ideas and suggestions offered here. Give some consideration to others when you sit down to write your next program.

But be aware that the benefits of long lasting code are often not seen immediately. Some of the suggestions offered here may take you a few minutes longer at programming time, but they could save hours of work later on. Those who come after you will certainly appreciate your thoughtfulness.

Think of these tips as a helpful tool in leading you toward what I consider the “golden rule” of programming:



*Leave programs for others
as you would have them left for you*

ACKNOWLEDGMENTS

I'd like to thank the people I've worked with or who've left their code behind for me to use. They provided me with a wealth of examples, both good and bad, to draw from when writing this paper.

CONTACT INFORMATION

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

Sandra Minjoe
Genentech, Inc.
1 DNA Way
South San Francisco, CA 94080
(650) 225-4733
fax: (650) 225-4611
email: sminjoe@gene.com

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. Other brand and product names are trademarks of their respective ☐ companies. ☐