Paper 229-2012

What Do You Mean, Not Everyone Is Like Me: Writing Programs For Others To Run

Jack Hamilton, Division of Research, Kaiser Permanente, Oakland, California, USA

ABSTRACT

Writing programs that other people will run can be difficult, and programs that might be run on multiple operating systems and versions of SAS® can be even more difficult. And if you're like me, a program you wrote a year ago might as well have been written by someone else.

This presentation discusses some of the challenges of writing programs to be run by others, and some possible guidelines to get around those challenges - some based on beyond-the basics programming, some based on programming style.

Some of the topics that will be discussed are: DOW loops, hash objects, regular expressions, choosing system options, and minimizing log messages.

INTRODUCTION

I've been programming for a very long time, perhaps longer than some readers have been alive, alas. For most of that time, I've worked in small groups, one or two or three, maybe as many as a dozen, but nothing larger.

In 2011, I changed jobs. My new employer, the Division of Research at Kaiser Permanente in northern California, participates in a number of research projects. Some of these projects are strictly internal - it's easy to meet everyone involved, and even get them in the same room. If you have a question, you can just drop by their office or drop them an e-mail.

But other projects, including some I'm working on, are much more geographically diverse. Several have participants across as many as five time zones. Because these participants work for several different companies, there is no standard operating system, set of SAS products, or even SAS version.

In one of these projects, I'm the lead programmer, so I am responsible for developing programs that will run across all of these diverse systems. It has been an interesting challenge, with unexpected problems.

Because these programs need to run on multiple systems, they have to be very flexible. SAS advertises itself as being a multiplatform system, and in general code is both forward and backward compatible across different versions. But there are some odd exceptions. Also, the amount of data at different sites may vary widely in size. One site might have 500,000 records in a table I'm interested in. Another might have 5 billion. A technique that works for a small table might not work for a big table.

Another challenge is that some participating sites don't allow me to see their job logs; I can get back only summarized data. So the logs have to be reviewed programmatically, and they must also be easy to review manually.

Different sites may have chosen different default system options, and this also can cause some surprising problems.

This paper discusses some of the problems, solutions, and alternatives I found while working on these projects. Although the initial goal was to help myself and others manage distributed programs, I have found that all of these tools and techniques are helpful for all of my programs, even knows I'm only going to run wants. As if I can be absolutely certain that a program will only be run once, ha ha.

This version of the paper is by no means complete; please check on SAS community.org for an updated version. The URL is < http://www.sascommunity.org/wiki/Writing_programs_for_others_to_run >. The version on SASCommunity.org should be considered the best version, because it can be revised and commented upon. I strongly urge you to look at that copy rather than this one. I needed about 4 more hours to work on this paper, adding another section and fleshing out the examples, but life got in the way. More will be added later and covered in the presentation.

MAKE THE PROGRAM FLOW EASY TO FOLLOW

After projects get past a certain size, typically a few hundred lines of code, it's a good idea to break them up into smaller programs. This is especially important if multiple programmers will be working on the same project, but it's useful even if it's only you - smaller programs make editing easier.

I try to break programs into semantically useful pieces; for example, I might put initialization code in one file, code to

read in input data in another, and the reporting programs in the third. In addition, I put macros and formats in their own file.

Personally, I find it useful to number my programs so it's obvious which is run first, which is run second. And so forth. In addition I subdivide the numbers into meaningful parts. Here's an example:

```
000-initialization.sas
001-macros.sas
002-formats.sas
010-run_programs.sas
020-read_text_data.sas
021-read_oracle_data.sas
030-generate_reports.sas
```

Even if you know nothing about what these programs are supposed to do, you can figure out what the programs do by looking at their names, and you can get a good idea of the sequence in which they are run by looking at their numeric prefixes.

Now the thing here that would be confusing is which program to run first. You might guess that it's the 000 program, or the 001 program. Well, in this case it's the 010 program that is run first. Sometimes there's no substitute for some good simple documentation.

The 010 program itself would be simple, something like this:

```
%include '000-initialization.sas'   / source2;
%include '001-macros.sas'           / source2;
%include '002-formats.sas'          / source2;
%include '020-read_text_data.sas'   / source2;
%include '021-read_oracle_data.sas' / source2;
%include '030-generate_reports.sas' / source2;
```

Notice the use of the source2 option, which causes the included code to be printed in the SAS log.

This code assumes that the programs are in SAS's current directory. In some environments, that's not a good assumption. I will discuss an alternate method later.

MAKE THE CODE EASY TO READ

Many people place a lot of emphasis on detailed comments in their code. You know, stuff like this:

```
/* Sort the data set by medical record number */
proc sort data=mystuff;
    By mrn;
run;
```

Personally, I think that's useless. I expect people who read my programs to have some knowledge of the subject matter, and at least an intermediate level knowledge of SAS. For non-technical readers, there should be a separate prose description of what the program is intended to do.

If there's a complicated piece of code in one of my programs, I will just point to an external source (both past user group papers and SAScommunity.org are good places to go).

What I mean by "make the code easy to read" is "make the physical structure of the code reflect the logical structure". Primarily, that means "indent and space your code properly".

I used to care a lot about the details of code formatting. Three spaces, not two, and not four. "then" goes on the same line as "if". That kind of thing.

But over the years, I realized that the details aren't important. What matters is not two spaces versus four spaces, or where the line breaks occur, but that there's any kind of consistent structure at all! Three spaces in one part of the code and five in another? Doesn't matter.

What's important is that you don't have code that looks like this:

```
If x = 1 and y = 12 then do; call missing(a, b); output; end; else delete;
```

It should look like this:

```
If x = 1 and y = 12 then
    do;
    call missing(a, b); output;
    end;
else
    delete;
```

Or this:

```
If x = 1 and y = 12
then
    do;
    call missing(a, b);
    output;
    end;
else
    delete;
```

The worst thing you can do is to write code that looks like it has some kind of indentation, but doesn't:

```
If x = 1 and y = 12 then
    do;
    call missing(a, b);
output; end; else
delete;
```

When this style is used for long sections of code, or for nested do loops or if then elses, it becomes deeply misleading, and makes the code almost it possible to follow.

But, of course, inventing his work! You have to press the tab key, or the space key or the return key, instead of just typing merrily along until you hit the right margin (and yes, I've seen code that was apparently written that way).

Happily, SAS Institute has a tool to help lazy typers create relatively well indented code – SAS Enterprise Guide. Take your poorly formatted code, paste it into an EG code window, and press Control-I. Voila! Indented code:

```
If x = 1 and y = 12 then
    do;
        call missing(a, b);
        output;
    end;
else delete;
```

There are several things I would change about the way this indentation was done, but the indenter is pretty set in its ways. There's not much you can do in the way of persuading it to do things differently. But even so, it's better than the original, and even more importantly, all of your code can be formatted *quickly* with little work.

An interesting feature is that the code inside macro definitions is further indented:

```
%macro test;

    If x = 1 and y = 12 then
        do;
            call missing(a, b);
            output;
        end;
    else delete;
%mend test;
```

You might find that this is very helpful when quickly scanning through code.

MAKE THE LOGS EASY TO REVIEW BY ELIMINATING EXTRANEOUS MESSAGES

It's a hassle to review logs, especially if the output is dozens of pages longs and contains a zillion extraneous messages. It can be a lot of work, but with a bit of extra code here and there you can eliminate almost all of the messages that don't matter.

Many of the extraneous messages are a result of what you might think of as sloppy coding. These can be fixed fairly easily.

Implicit Conversion Messages

```
6    dm 'clear log;' log;
7    data _null_;
8        a = '3';
9        b = a + 4;
10   run;

NOTE: Character values have been converted to numeric values at the places given
by:
      (Line):(Column).
      9:9
NOTE: DATA statement used (Total process time):
      real time           0.01 seconds
      cpu time            0.00 seconds
```

If A has to be a character variable (perhaps in real life it came from an, external source), put it inside an INPUT function. Then you'll get a message only when the conversion can't be done:

```
data _null_;
    a = '3';
    b = input(a, best.) + 4;
run;
```

If A doesn't contain a string that can be converted to a number, you'll still get a message in the log, but this time it's one you might want to pay attention to.

If you need to go in the other direction, use the PUT function.

Expected Invalid Data

Sometimes you're reading from an external source that contains data you expect will sometimes be invalid. In that case, you might not want to see the message – either because it's not a real problem, or because you'll handle it in a better way than by looking at the log.

If you're using an INPUT statement or function, you can use the ?? format modifier, and the message about invalid data will not appear. Your variable will simply be assigned a missing value

But you'll still get a message when you use that missing value later; it will say you've performed operations on missing values. Happily, there's a little-known function, DIVIDE, that suppresses even those messages:

```
6    dm 'clear log;' log;
7    data _null_;
8        a = 12 / .;
9        b = divide(12, .);
10   run;

NOTE: Missing values were generated as a result of performing an operation on
missing values.
      Each place is given by: (Number of times) at (Line):(Column).
      8:12
NOTE: DATA statement used (Total process time):
      real time           0.04 seconds
      cpu time            0.04 seconds
```

You can see that the message appears only for the first division in line 8 and not for the second division in line 9.

Your calculation isn't a division? Just use the entire expression you want to evaluate as the first argument to the divide function, and use 1 as the second argument. This lets you use the divide function to suppress messages without changing the returned value.

AUTOMATE LOG CHECKING TO THE EXTENT POSSIBLE

Even if you've written your program to eliminate as many extraneous messages as you possibly can, you may have overlooked something. And maybe, just maybe there was a message that you really need to look at – other sites may have data that cause messages that don't appear at your home site, so a message appears that never occurred to you. Reviewers at other sites might have different ideas from you about what constitutes a meaningful warning or error message.

For all these reasons, it's a good idea to have a program that automatically checks your logs for significant notes, warnings, and errors.

Happily, there are some features in SAS that make this straightforward.

The first is that on directory-based systems like Windows and UNIX, you can specify a wildcard in the filename statement. For example, if all of your log log files are in the current directory, you can read them all in like this:

```
18   filename logfiles './*.log';
19   data _null_;
20       infile logfiles;
21       input;
22   run;

NOTE: The infile LOGFILES is:
      Filename=Y:\SGF2012\pgm1.log,
      File List=Y:\SGF2012\*.log,RECFM=V,LRECL=256

NOTE: The infile LOGFILES is:
      Filename=Y:\SGF2012\pgm2.log,
      File List=Y:\SGF2012\*.log,RECFM=V,LRECL=256

NOTE: The infile LOGFILES is:
      Filename=Y:\SGF2012\pgm3.log,
      File List=Y:\SGF2012\*.log,RECFM=V,LRECL=256
```

You can combine that feature with the FILENAME= option on the INFILE statement to get the filename associated with each line of input:

```
      infile logs filename=filename;

      input;

      if filename ne lag(filename)
      then
          do;
          line_num = 0;
          logfilename = filename;
          shortlogfilename = prxchange('s/.*([\\\/])(.*)([\\\/])(.*)/$2$3$4/', 1,
  filename);
          retain logfilename shortlogfilename;
          end;
```

The PRXCHANGE function examines the complete file name and path returned in the FILENAME variable and returns the filename part. It's too complicated to explain here, but look in sascommunity.org for further examples, and look in user group papers for more explanations and examples. In particular, look for recent papers by Toby Dunn. You can drop me an e-mail if you want specific references, but I think you'll learn more if you go look for yourself, and you might make some serendipitous discoveries along the way.

You can then look at each line to see if it contains anything interesting. What's interesting to you might not be interesting to me, but here is the code I use in one of my programs:

```
    line_num + 1;

    linetext = _infile_;

    if prxmatch('/(will be expiring soon)|(Please run PROC SETINIT)/i', _infile_)
    then
        delete;  /* License expiration warnings  */

    if prxmatch('/(ERROR( |:))|(WARNING( |:))|(stopped processing this step)|(DATA
STEP stopped)|(symbolic reference)|(Errors printed on page)|(repeats of by
values)/', _infile_)
    then
        output warnings_errors;

    else if prxmatch('/(was too small)|(have been converted)|(Invalid \(or
missing\) arguments)|' ||
                '(Division by zero detected)|(multiple lengths)|' ||
                '(Missing values were generated)|(operations could not be
performed)/i', _infile_)
    then
        output notes;

    else
        ;  /* Nothing we care about.  */
```

You'll notice that I make extensive use of the prxmatch function. The code here is more straightforward. The regular expression "/(will be expiring soon)|(Please run PROC SETINIT)/i" tells SAS to look in the searched string for either "will be expiring soon" or "Please run PROC SETINIT"; the parentheses create groups to search for, and the "|" symbol means "or". The "i" at the end tells SAS to ignore case when doing the comparison. The expressions can get long, but the alternative (a series of INDEX functions) would be even longer, and I think it would be more difficult to read.

With luck, both the WARNINGS_ERRORS and NOTES data sets will be empty, so there won't be anything to report on. But you don't want your colleagues to have to look through the log to find that out. It's better to create an "empty" report saying there's nothing to report. There are lots of ways to do that – for example, see my old paper "How Many Observations Are In My Dataset?" – but this is a very simple case, so you can do it simply:

```
data warnings_errors;
    if 0 then modify warnings_errors nobs=nobs;
    if nobs = 0
    then
        do;
        linetext = 'No warnings or errors were found.';
        output;
        end;
    stop;
run;
```

If there are any observations in the data set, nothing is changed. If there are no observations, a single observation will be written out. You can then print the data set; I like to use PROC REPORT:

```
options nobyline;
title "WARNINGS and ERRORS in Log File:  #BYVAL(shortlogfilename)";
proc report data=warnings_errors missing nowindows nocenter;
    by shortlogfilename;
    column line_num linetext;
    define line_num  / order   width=10  flow 'Line'     style={ cellwidth=.7in  };
    define linetext  / display width=150 flow 'Message'  style={ cellwidth=9.5in };
run;
```

This will print one page for each log file that contains a warning or error, with the name of the log file in the title.

CONSIDER CREATING SEPARATE LOG AND LIST FILES FOR EACH OF YOUR SUBPROGRAMS

Just as cooperative programming can be made easier by splitting source code into several files, there are benefits to splitting the output into multiple files. I usually associate one input file with one log file, but there's no reason you couldn't put the output from two programs into a single log file, split the output into two log files, or partition the output across several log files. The same holds true of list files.

The easiest way to do this is with PROC PRINTTO. I know, PRINTTO is very old-style and not very flexible, but it works, and has the big advantage over the ALTLOG system option of not requiring changes to the SAS startup options – something it turns out is not possible everywhere.

Here's how it works:

```
Proc printto log='./020-read_text_data.log' new; run;
```

In practice, if you are going to run programs on both Unix and Windows systems, you will probably want to set the line-end character(s) to be the same everywhere. To do that, you will need to use a FILENAME statement, and then use the fileref in PROC PRINTTO:

```
filename runlog "020-read_text_Data.log" termstr=crlf;
proc printto log=runlog new; run;
```

"MACROTIZE" WHEN YOU CAN, BUT DON'T GO OVERBOARD

One of the biggest mistakes rookies make when learning the macro language is to put everything into a macro, not realizing that many things can be done without using macros. Also, some rookies think that you can't use macro variables and functions outside of a macro, which is not true.

Macro variables can be used anywhere in SAS; you just have to make sure that the variable is defined before you need to use it

You can even simulate a macro %IF using the %sysfunc macro function. For example, if you want to send output to an ODS destination if and only if a macro variable has been set to a particular value, you can do it using the IFC function:

```
29   dm 'clear log;' log;
30   %let testvar = 0;
31   %put The statement that would be executed is %sysfunc(ifc(&TESTVAR=0, * ,ods
pdf
31 ! file='abc.pdf'));
The statement that would be executed is *
32
33   %let testvar = 1;
34   %put The statement that would be executed is %sysfunc(ifc(&TESTVAR=0, * ,ods
pdf
34 ! file='abc.pdf'));
The statement that would be executed is ods pdf file='abc.pdf'
```

STUDY YOUR SYSTEM OPTIONS

One of the most interesting discoveries I've made is the wide variety of system options set at different sites. These are the system options I ending up needing to set explicitly:

```
options ls=240 nocenter pagesize=50 date number stimefmt=h formdlim='-' dtreset;
options dldmgaction=fail;
options msglevel=i mprint;
options sortdup=logical;
options dsoptions=nonote2err errorcheck=strict serror merror;
options dkricond=error dkrocond=error;
options papersize=letter formchar='|----|+|---+=|-/\<>*' sysprint='';
```

The most interesting of these options are dkricond and dkrocond, which I bet most users have never heard of. They control what happens when a Drop, Keep, or Rename data set option on an Input or Output data set contains an invalid variable name (shouldn't ever happen, but oops happen). The most puzzling was the sysprint system option. My programs never try to print anything – they only write to files – but at a few sites an error message about the sysprint option would randomly appear. After discovering that no one at SAS Tech Support could tell me for sure what was happening, I skirted the issue by setting SYSPRINT='', which seemed to have no effect except for making the messages go away.

DON'T MAKE YOUR COLLEAGUES – OR YOURSELF - DO UNNECESSARY WORK

This is really the major theme of the paper. If a manual task can be automated, it should be automated. That applies to creating standard macros and naming standards to save yourself work just as much as it applies to creating log checking utilities to save your colleagues work.

One of the requirements for most of our projects is to create a ZIP file containing log files, list files, and selected SAS data sets that are created by the programs. But the way to do that varies on every system – a site might be on Unix or Windows, and might have WinZIP or PKZIP or InfoZIP or nothing installed. In any case it would be a partly or completely manual process to create the ZIP file. Getting exactly the right files into the right ZIP subfolders requires more attention to detail than I like to demand of myself, so I didn't want to demand it of others.

The following program recursively reads all the files in and below a given directory, selects the ones to put in the ZIP file, and creates the ZIP file. The ZIP portion requires ODS packages, which are available only in SAS 9.2 and above, but the code that reads the directories will run on any recent version. I have not seen this recursive directory technique used elsewhere; it does not require PROC FCMP.

```
%macro get_file_names_recurse(
    root=..             ,
    dirs=dirs_found     ,
    files=files_found   ,
    filefilter=1        ,   /* Data set name filter  */
    sepchar=/
    );

    /* Data set dirs_found starts out with the names of the root folders    */
    /* you want to analyze.  After the second data step has finished, it    */
    /* will contain the names of all the directories that were found.       */
    /* The first root name must contain a slash or backslash.               */
    /* Make sure all directories exist and are readable.  Use complete      */
    /* path names.                                                          */
    data &DIRS. (compress=no);
        length Root $120.;
        root = "&ROOT.";
        output;
    run;

    data
        &DIRS.                  /* Updated list of directories searched */
        &FILES. (compress=no); /* Names of files found.                 */

        keep Path FileName FileType;

        length fref $8  Filename $120 FileType $16;

        /* Read the name of a directory to search.             */
        modify &DIRS.;

        /* Make a copy of the name, because we might reset root.  */
        Path = root;

        /* For the use and meaning of the FILENAME, DOPEN, DREAD, MOPEN, and     */
        /* DCLOSE functions, see the SAS OnlineDocs.                             */

        rc = filename(fref, path);

        if rc = 0 then
            do;
            did = dopen(fref);
            rc = filename(fref);
            end;
        else
            do;
            length msg $200.;
            msg = sysmsg();
            putlog msg=;
            did = .;
            end;

        if did <= 0
        then
            do;
            putlog 'ERR' 'OR: Unable to open ' Path=;
            return;
            end;

        dnum = dnum(did);

        do i = 1 to dnum;
            filename = dread(did, i);
```

```
            fid = mopen(did, filename);
            /* It's not explicitly documented, but the SAS online  */
            /* examples show that a return value of 0 from mopen   */
            /* means a directory name, and anything else means     */
            /* a file name.                                        */
            if fid > 0
            then
                do;
                /* FileType is everything after the last dot.  If */
                /* no dot, then no extension.                     */
                FileType = prxchange('s/.*\.{1,1}(.*)/$1/', 1, filename);
                if filename = filetype then filetype = ' ';
                if &FILEFILTER.
                then
                    output &FILES.;
                end;
            else
                do;
                /* A directory name was found; calculate the complete  */
                /* path, and add it to the &DIRS. data set,      */
                /* where it will be read in the next iteration of this */
                /* data step.                                          */
                root = catt(path, "&SEPCHAR.", filename);
                output &DIRS.;
                end;
        end;

        rc = dclose(did);

    run;

    proc sort data=&DIRS.;
        by root;
    run;

    proc sort data=&FILES.;
        by path filename;
    run;

%mend get_file_names_recurse;

/* Get list of files in selected project directories. */
/* Keep only certain file types, and exclude a few    */
/* specific files.                                    */
%get_file_names_recurse(
    root=..             ,
    files=dist_files    ,
    dirs=dist_dirs      ,
    filefilter=(prxmatch('Programs)|(Logs)|(Data)/i', path) or path='..')
                and prxmatch('/(sas7b)|(pdf)|(doc)|(txt)/', filename)
                and filename ne: '040-create_zip'
    );


data _null_;

    call execute('ods package(distzip) open nopf;');

    if not EndOfFile then
        do until (EndOfFile);
            set dist_files end=EndOfFile;
            /* Add this file in its subdirectory.  */
```

```
            call execute('ods package(distzip) add file="' || strip(path) || "/" ||
strip(filename) || '" path="' || strip(substr(path, 4)) || '" ;');
        end;

    call execute("ods package(distzip) publish archive
properties(archive_name=""&REQUEST_ID..zip"" archive_path=""../"");");

    call execute('ods package(distzip) close;');

    stop;

run;
```

ACKNOWLEDGMENTS

Jamila Gul, Kaiser Permanente Division of Research

Tom Billings, Union Bank

Nicolas Beaulieu, Harvard Pilgrim Health Care

Roy Pardee, Group Health Cooperative

Ron Nicols, UC Davis Health System

Don Bachman and Gwyn Saylor, Kaiser Permanente Center for Healthcare Fesearch

CONTACT INFORMATION

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

Jack Hamilton
jfh@acm.org
http://www.sascommunity.org/wiki/User:JackHamilton