ABSTRACT
This paper will demonstrate several techniques using Base SAS and macros that will take your programming skills to a new level by giving you the power to write programs that write programs. These simple techniques, which are powerful used alone, are awesome when combined. Using these techniques when your application is appropriate, you can write one program that will act as though you wrote dozens of individual programs. These programming techniques will give you the ability to write more robust, highly flexible, intelligent programs that can extract data from little known sources, make decisions, write and execute program code depending less on user input. Programs using these techniques are more difficult to debug, but once validated, can be used over and over again by many projects. Although the techniques are simple, this paper is aimed at experienced programmers who know when it would be appropriate and efficient to use them.

INTRODUCTION
"There are a hundred different ways to do everything in SAS."

The code shown in this paper is only one of many ways to write the programs. Many other tools and techniques are available in SAS. This paper presents these techniques as a means of sparking interest. Feel free to explore different ways of writing the code, combining the techniques, and maybe develop other ways to write programs that write programs.

Host specific examples given are be based on the Microsoft Windows environment using SAS system for Personal Computers. Most of the host specific examples work very similarly in the UNIX environment.

MACROS

Technique 1
SAS Macros are easily the most common method used to write programs that write programs. Macros, simply put, are text substitution. Wherever you call a macro, text is substituted. For example:

```sas
%macro print(dsn);
proc print data=&dsn;
title "&dsn";
run;
%mend print;

%print (alpha);
%print (beta);
```

generates the program below:

```sas
proc print data=alpha;
title "alpha";
run;

proc print data=beta;
title "beta";
run;
```

Technique 2
Macros can generate code conditionally. Here is an example that will print the data sets only when the value of the macro variable SHOW equals YES:

```sas
%macro print(dsn);
%if &show=YES %then %do;
proc print data=&dsn;
title "&dsn";
run;
%end;
%mend print;
%print (alpha);
%print (beta);
```

This is the same example written differently and more efficiently since the macro is never actually called unless &SHOW = YES:

```sas
%macro print(dsn);
proc print data=&dsn;
title "&dsn";
run;
%mend print;
%if &show=YES %then %do;
%print (alpha);
%print (beta);
%end;
```

Technique 3
There will be circumstances when you need slightly different statements. Macros can write programs with varying statements. In this example, when printing the data set ALPHA the variables FIRST, SECOND, and THIRD are dropped.

```sas
%macro print(dsn);
proc print data=&dsn
%if &dsn='alpha' %then %do;
(drop=first second third)
%end;;
title "&dsn";
run;
%mend print;
%print (alpha);
%print (beta);
```

The macro above generates the code below:

```sas
proc print data=alpha(drop=first second third);
title "alpha";
run;

proc print data=beta;
title "beta";
run;
```

Notice what appears to be an extra semicolon on the %END statement in the source code. The first semicolon terminates the
%END statement, the second terminates the PROC PRINT statement.

BASE SAS
Using Base SAS to write programs that write programs is a bit more complex. Instead of writing the code within a %MACRO and %MEND statements, you need to actually get your SAS program to write the program then %INCLUDE the program to run it.

You will be given a SAS data set that contains the names of other SAS data sets and asked to provide a PROC PRINT for each of those data sets. You know you need statements that look like:

```
proc print data=alpha;
  title "alpha";
  run;
```

The number of observations that will be in the data set you will be given has not been specified. You may only have a few data sets to print, or maybe hundreds. It may not be feasible to write all the PROC PRINT statements by hand. Instead, write a program to write the program for you.

Technique 4
This technique creates a text file or “flat file” containing SAS program code. The PGM extension on the flat file tells you the program was computer generated, distinguishing it from “normal” SAS code.

```
data _null_; /* contains DSNs to print */
  set final; /* contains DSNs to print */
  file 'c:/temp/step01.pgm';
  put 'proc print data =' dsn ' ';
  put 'title "' dsn '"';
  put 'run';
run;

%include 'c:/temp/step01.pgm';
```

The %INCLUDE above includes the generated code and executes it as if you had written it yourself. Using eight lines of code, you can generate hundreds or thousands of lines of code. The generated code will contain all the PROC PRINT statements you need to handle however many observations were provided in the data set FINAL.

Technique 5
Conditional programming can be handled as well. In this example, when printing the data set ALPHA the variables FIRST, SECOND, and THIRD are dropped.

```
data _null_; /* contains DSNs to print */
  set final; /* contains DSNs to print */
  file 'c:/temp/step01.pgm';
  put 'proc print data =' dsn '@';
  if dsn = 'alpha'
    then put '(drop=first second third)'; @;
  put 'title "' dsn '"';
  put 'run';
run;

%include 'c:/temp/step01.pgm';
```

If the data set FINAL contains two observations, ALPHA and BETA, the program above will generate the following code:

```
proc print data=alpha(drop=first second third);
  title "alpha";
  run;

proc print data=beta;
  title "beta";
  run;
```

The %INCLUDE statement will bring in the generated program code and execute it.

Technique 6
Whole blocks of base SAS code can be turned into a macro and executed conditionally. The code below might be a group of steps you have in your program to assist you with validation.

```
data _null_; /* contains DSNs to print */
  set final; /* contains DSNs to print */
  file 'c:/temp/step01.pgm';
  put 'proc print data = merged (obs = 100);'
  put 'run;'
run;

%include 'c:/temp/step01.pgm';
```

As your program development progresses, you may want to comment this section out. But if you need it to function again later you have to go back and uncomment the code.

Another solution might be to “turn on” or “turn off” the code as you execute your program. Enclose the section of code in a macro, add a condition under which to execute the macro code and make that section of code switchable based on your condition.

```
%macro validate;
  if &validate = Y %then %do;
    proc contents data = merged;
    proc freq data = merged;
    tables _all_;
    proc print data = merged (obs = 100);
    run;
  %end;
%mend validate;
%validate;
```

The execution of the procedures depends on the value of the macro variable VALIDATE. The macro variable VALIDATE is set earlier in the program by the user and/or by program conditions. The conditional execution of these steps can be turned on during testing or troubleshooting, and turned off in production, reducing generated output.

DETERMINING IF A DATA SET CONTAINS DATA
Technique 7
Use the SAS dictionary table VTABLE to see if a table contains data.

```
%let nobs = 0;
proc sql;
  select nobs into :nobs
  from sashelp.vtable
```
The macro variable NOBS is initialized to zero because if the data set DEMOG does not exist the PROC SQL statement will not create the NOBS macro variable. A minor change of logic in the code above could make the program check if the data set DEMOG exists. It is possible for a data set to exist and contain no data.

The macro variable NOBS can now be used in conditional programming to use, or avoid using, the data set DEMOG depending on its existence.

CREATING A DATA SET WITH VARIABLES BUT NO OBSERVATIONS
Technique 8

```
data bad_obs;
  attrib dsn length=$8
    label='data set name'
  num length=8.
    label='number of bad obs';
  if _n_ = 0;
run;
```

The code above will produce a data set containing two variables and zero observations as shown in the log below.

1. data bad_obs;
2. attrib dsn length=$8
3. label='data set name'
4. num length=8.
5. label='number of bad obs';
6. if _n_ = 0;
7. run;

The log above notes the variables are uninitialized. These notes are correct and expected.

CREATING A DATA SET WITH NO VARIABLES AND NO OBSERVATIONS
Technique 9

The following code will produce a data set containing zero variables and zero observations.

```
data bad_vars;
  if _n_ = 0;
run;
```

The program log:

1. data bad_vars;
2. if _n_ = 0;
3. run;

NOTE: The data set WORK.BAD_VARS has 0 observations and 2 variables.
NOTE: The DATA statement used 0.02 seconds.

The log above notes the variables are uninitialized. These notes are correct and expected.

ISSUING OPERATING SYSTEM COMMANDS
Technique 10

There are a number of ways to issue operating system commands from SAS. The FILENAME PIPE option method is not used frequently, however is easy and does not require XWAIT or XSYNC option involvement. Any operating system command can be issued using this method.

The following is sample syntax for the Windows environment. The UNIX syntax is the same with the exception of the command issued inside the quotes of the filename statement.

```
filename delete pipe 'del c:test.pgm';
data _null_;
  infile delete;
run;

filename delete clear;
```

The code above will produce this log:

1. filename delete pipe 'del c:test.pgm';
2. data _null_;
3. infile delete;
4. run;

NOTE: The infile DELETE is:
FILENAME=del c:test.pgm,
RECFM=V,LRECL=256
NOTE: 0 records were read from the infile DELETE.
NOTE: The DATA statement used 0.21 seconds.

Here is another example of what can be done using the PIPE option.

```
filename delete pipe
  'cd d:
    cd test
    mkdir results
    move *.lst results
    cd results
    rename *.lst *.sav';
data _null_;
  infile delete;
run;

filename delete clear;
```

Further information regarding the FILENAME PIPE option can be located in host specific documentation.

GETTING DATA INTO YOUR PROGRAM
Technique 11
Parameters can be sent into a SAS program on invocation using the SYSPARM SAS system option. The syntax for both the UNIX and Windows environments is:

\[ \text{sas program-name} -\text{sysparm} \ '\text{string}' \]

The value of 'string' can be one or more parameters separated by any delimiter. The string must be enclosed in quotes if spaces or special characters are used. The parameters can be accessed within the SAS program using the SYSPARM() function or the automatic macro variable &SYSPARM. For example:

```sas
data _null_;  
if sysparm() = 'one' then put '11111';  
if sysparm() = 'two' then put '22222';  
put '=====';  
runc;
```

To run the program above, stored as test.sas, from the command prompt in the same directory where the program is stored, you would type:

```sas
sas test.sas -sysparm one
```

and press enter. The log generated would be:

```
data _null_;  
if sysparm() = 'one' then put '11111';  
if sysparm() = 'two' then put '22222';  
put '=====';  
runc;
```

```n
11111
=====  
NOTE: The DATA statement used 0.07 seconds.
```

The value returned by the SYSPARM() function can be parsed like any character string. Therefore, multiple parameters can be sent into a program using any delimiter. The parameters in the following call

```sas
sas test.sas -sysparm 'one|two|three four'
```

can be parsed with this, or similar code:

```sas
data _null_;  
call symput('parm1',scan(sysparm(),1,'|'));  
call symput('parm2',scan(sysparm(),2,'|'));  
call symput('parm3',scan(sysparm(),3,'|'));  
runc;
```

The macro variables PARM1, PARM2, and PARM3 will have the values 'one', 'two', and 'three four' respectively.

Using this and the technique above, you could write a SAS program that calls other SAS programs when necessary based on your data conditions.

---

**COMBINING THE TECHNIQUES**

**Problem**
Given a SAS data set containing an unknown number of names of other SAS data sets, drop blank observations from the named data sets while ignoring certain specified “housekeeping” variables. For this example, the housekeeping variables will be INVEST, SITE, PTNO.

```sas
%macro blanks(dsn);
  /* find the names of all */  
  /* the variables in the DS */
  proc sql;  
    create table columns as  
    select name, type  
    from sashelp.vcolumn  
    where memname = upcase(&dsn)  
    and name not in ('INVEST','SITE','PTNO');  
  run;

  /* check to see if variables */  
  /* are blank */
  data _null_;  
    if sysparm() = 'one' then put '11111';  
    if sysparm() = 'two' then put '22222';  
    put '=====';  
  run;

  proc print data=del;  
    title 'Blank records in &dsn';  
  run;

  %include 'c:\temp\columns.pgm';
%mend blanks;
```

```sas
/* generate calls to macro */
%include 'c:\temp\step01.pgm';
```

**NOTES:**
1. Use the SAS dictionary tables to get information about the data set.
2. Data set name must be in upper case because the value in the variable MEMNAME is stored in upper case.
3. Variable names must be in upper case because the value in the variable NAME is stored in upper case.
4. The program generated here will change with each call to the macro. The macro itself remains unchanged but the results change with each different data set.
5. Quoting here does not matter. Either type of quote can be used in the generated program to check for a null value.
6. Different types of quotes are necessary to achieve desired affects. Here the code can be generated exactly as shown, so single or double quotes can be used.
7. In order to get the value in the macro variable in the generated code, double quotes must be used here.
8. Quotes on the outside must be double here to cause the macro variable to evaluate and become part of the inner single quotes.
9. Single quotes are necessary here, otherwise the SAS System will try to execute %BLANKS at compilation and all sorts of errors will occur.

Two programs will be generated.

The first generated program, COLUMNS.PGM, is a block of code that is being created based on the variables in a given data set, then immediately %INCLUDEd and executed as part of a macro in which it is contained. This section of the %BLANKS macro is being rewritten for each individual data set. A few lines of code are responsible for the creation of hundreds or thousands of lines of code. The code in COLUMNS.PGM will look something like:

```sas
data alpha del;
set alpha;
if CRFNO = . AND SUBIN = '**' AND DOB = . AND SEX = '' AND RACE = . AND STATUS = '' then output del;
else output alpha;
proc print data=alpha;
title "blank records in alpha";
run;
```

The variable names in the code above will be upper case because they are retrieved from the SAS dictionary table VCOLUMN which stores them in upper case.

The second generated program, STEP01.PGM, uses the list of data set names from FINAL to create the calls to the macro. Thus, the program STEP01.PGM will look something like:

```sas
%blanks(alpha);
%blanks(beta);
.
.
```

When the program STEP01.PGM is executed, the macro %BLANKS will be called once for each observation in the data set FINAL. The program COLUMNS.PGM, will be generated and executed for each call to %BLANKS.

**DEBUGGING AND VALIDATION**

There are a number of debugging and validation issues related to using these techniques.

- Quoting becomes a special challenge. Writing a program that writes a program that needs to use macro variables and double quotes requires additional attention, as was demonstrated in the problem above.
- Semicolons are often missed when writing SAS programs to a "flat file" because the statement you are having the computer generate must end in a semicolon, and the statement you are using to tell the computer what to write must also end in a semicolon.
- Particular care needs to be given if the generated programs need to be readable by humans. The computer does not care about spacing, indentation, blank space or how many program statements are on one line of code, but humans do.
- Taking the time and effort to make the generated programs more readable will make debugging considerably easier, both for you and the people who will inherit your programs.
- Storing computer generated code could use lots of space. Permanent storage may be required by your Quality Control department.
- Reproducibility will be an issue if you have ever changing data. Obviously, with final data, any program should produce the same results over and over.
- The concept of programs that write programs is sometimes difficult to grasp, therefore some people will be uncomfortable understanding the techniques employed. As a result they may not accept such code as fixed and unchanging. The program code is unchanging, and when the data is fixed, the results will be fixed.
- Validation of programs using electronic sources of data, especially sources such as the SAS dictionary tables, should be relatively quick since the SAS product is validated, therefore the source of the data is validated.
- One of the biggest advantages to using these methods is that changing a program in one place will effectively change all the programs in your project.
- One of the biggest disadvantages to using these methods is that changing a program in one place will effectively change all the programs in your project.

**CONCLUSION**

These techniques work best with systems requiring repetitive operations, such as exercises across all variables in a data set or all data sets in a library. Being able to see commonalities of need across programs or data, knowing how to exploit the SAS dictionary tables, and being able to “see the forest through the trees” are all skills which will assist you in efficiently using these techniques.

Using these programming methods requires a crisp programming skill, style, and superior commenting. Programs written using sloppy style or that are uncommented or poorly commented will make understanding the code virtually impossible even a few days after you have written it.

Caution MUST be exercised when using these methods. One small change can have a very large “trickle down” effect. This can be both an advantage and a disadvantage.

The techniques demonstrated in this paper are basically very simple, yet very powerful. The methods have been used successfully to produce programs and output in hours or days that would have taken many weeks or months to write and validate individually. The tables for an Integrated Safety Summary (ISS) were written using these techniques. A dozen programs and a few simple small macros were responsible for producing over 800 reports for the ISS.

Once you start using these techniques and get comfortable with their application, function and interaction your programs will be more robust, efficient, intelligent and flexible, and you will become hooked.

**REFERENCES**
