Function and procedure libraries have an important purpose in the latest versions of computer languages such as C. The core language remains relatively compact and simple; the libraries grow continuously to extend and amplify the programming system.\(^1\) The SAS System offers a set of core character and list functions. This set has changed surprisingly little during the past decade. The relevant sections in my old 1982 manual read much the same a decade later in Version 6 manuals.\(^3\) Nonetheless, I use the same old functions differently today than I did before.

**Evolving Functions**

At first I tried slightly different sequences of function calls in every program. For example,

* Get last character in string x;
  y=\text{length}(x);
  a=\text{substr}(x,y);

Next I put the sequence of function calls in a simple SAS Macro:

\[
\%macro LastCh;
  y=\text{length}(x);
  a=\text{substr}(x,y);
\%mend LastCh;
\]

I defined the Macro just before the data step in which I first called it as \%LastCh. It actually took more effort to create the Macro and call it, but it made the data step easier to read.

Soon after I began using the same Macro in other data step programs. Starting with a tested macro helped me remember some of the nuances of the SAS function calls. Sometimes I would even remember to assign the correct value to x and assign the value of a to the correct variable.

After using this style of Macro for a while, I began running into conflicts between the variable names used in the Macro and the variable names used in the rest of the data step program. The Macro merely hid the variable names. To get around that snag I collapsed the body of the Macro into a single expression containing expressions in place of data step variable names:

\[
\text{substr}(x,\text{length}(x))
\]

This left only one data step variable name to worry about. I could even give it a really odd name to minimize the risk of conflict, and I could drop the temporary variable name from output data sets.

But these concerns disappeared when I learned to replace the specific variable name references with more general parameters:

\[
\%macro LastCh(VarStr);
  \text{substr}(\&VarStr,\text{length}(\&VarStr))
\%mend LastCh;
\]

Now the function would behave as a proper function: it would act as any other expression and resolve to a value, as in

\[
\text{EndStr}=\%\text{LastCh(ChrStr)};
\]

Having reached this higher plateau, I had gained enough confidence to look for new horizons. When I needed to select the last three
characters from a variable of unknown length, I used LastCh to create Last3Ch:

    %macro Last3Ch(VarStr);
      substr(&VarStr,length(&VarStr)-2)
    %mend Last3Ch;

For a time I envisioned a whole library of LastnCh functions, ranging from LastCh to Lst199Ch! Fortunately, sanity prevailed. I figured out how to use another parameter to combine all of the LastnCh functions.

The latest to evolve in the descent of the LastnCh function family tree now resides in my program library:

    %macro LastnCh(VarStr, __ n);
      reverse(substr(reverse(left(trim(&VarStr)),1, &n))
    %mend LastnCh;

It works as you would expect for case of length (&VarStr)=__n. Run on Westat's networked PC SAS 6.04 installation it produces a Note: on the SAS log when __n>length(&VarStr), and it sets the SAS system variable _error_ to 1.

Setting Up a Basic SAS Function Library

I use a simple set-up that includes general functions and some of the rare flashes of lucidity that appear in recent SAS programs. If your SAS installation does not have a formal function library, you can follow these steps to install a basic SAS function library:

- put the Macro definitions in a text file;
- define a file reference to the text file;
- %include the file reference in a SAS program;
- call the Macros where appropriate.

A good Macro function library contains many of the program segments that you or other programmers have spent long hours developing and refining. A quick glance at a function often helps you remember the little traps that caused other programs to fail. Anyone who spends hours correcting small errors in a program, only to discover that correct use of Macro functions could have prevented many of them, becomes a staunch advocate of function libraries

Developing Better Macro Functions

Remembering these basic rules will help anyone write better functions:

- Make them correct;
- Make them obvious;
- Make them general;
- Make them efficient.

The order of the rules has meaning. First and foremost, the function has to produce correct results. A correct function always produces the same results for the same set of arguments. We look for more. We also insist that the yield (result) of the function have the correct value or values. The other desirable features of a function become important when we compare two or more provably correct functions. Then we consider its ease of use, its flexibility and the costs its use entails.

Rule 1: Make Them Correct

How do we know a function will yield correct values? A routine method of verification helps increase our confidence in a Macro function (or any program for that matter).

First, define carefully a set of constraints on its use. A function designed to concatenate character
strings will not work properly when applied to real numbers. Some functions require unique character values or single character values or some other special data type. To a reasonable extent the function library specifies the restrictions on using a function. Beyond that, the developer of the function may also want to take reasonable precautions to handle invalid function calls. These restrictions on the use of the function establish the context of the verification procedure.

Second, devise a simple model of the way the function operates on the types of data expected as parameter values and other data objects. Starting with an initial state of the data, step through the actions of the function. Each action changes the state of the data. If the function always concludes by returning the correct value or values to the program that called it, consider it provably correct.

A simple model of the function tCount provides an illustration. tCount has a very simple design:

```
%macro tCount(___l,
    ___n=n,
    ___dl='quote(' ')');

    *Counts tokens in character variable;
    *Use as a data step statement;
    *Supply character expression as ___l
    *Returns digit string in data step variable (default name ___n);

    &___n=0;
    do while(scan(___l,___n + 1,___dl) ne &"");
        &___n=&___n +1;
    end;
%mend tCount;
```

Once called, the function only refers to three expressions that change. The sequence of these changes follows two simple patterns:

```
&___n&___n +1 scan(&VarStr,___n +1)
0 1 token[1]="
0 1 token[1]
1 . .
n n+1 token[n+1]="
```

Assuming the scan function and the addition operator work as documented (and if you can't trust the SAS Institute, whom can you trust?), we have every reason to believe that the scan function will return an empty string when it tries to locate the nth+1 token in a character variable that contains n tokens. (We define tokens as character strings separated by blanks or other delimiters.) If the character variable contains only blanks or other delimiter values, the function tCount tries to locate the first token and returns an empty string. The do while condition fails and the function ends with the value assigned to variable name at "address" &___n still equal to its initial value of 0. If the character variable contains one or more tokens, the function increments &___n each time the scan function returns a token. When the function has located the n tokens in the character string, it stops with &___n equal to n. This type of proof increases my confidence in the validity of the function, and it takes less time to complete it than to correct subtle errors in the most dangerous type of program: the one that works almost all of the time.

Finally, test for contradictions of our proof. I prove the function correct beyond a shadow of a doubt, then I humiliate myself by testing the function and finding all of the holes in my proof. (A mathematical genius can skip the testing part. I have to have test results to back up my
claims.) Testing also strengthens our faith in our understanding of the way the SAS System processes observations in datasets and applies functions to expressions. A combination of analytical proof and testing helps us avoid most types of errors. 6

Rule 2: Make Them Obvious

If the form of a function call makes its use and result obvious, it has a greater value than a more complicated one subject to misinterpretation or error. In its simplest form the function tCount requires only one parameter value. When used as a data step statement, it assigns a value to an ordinary data step variable name. Applied correctly, the variable resolves to a character value representing an integer. Both its name and its form help the user understand that it counts the number of tokens in a character variable.

Obvious Macro functions work in a way that users understand intuitively. The method for calling a function fits its purpose. A function yielding a value in an expression has the data source as its main argument. A function yielding a variable (or array or some even more complex data object) has both the data source and the name of the output data object as arguments.

Documentation within the text of the Macro should provide succinct instructions on the method used to call it, the proper context for calling it and limitations on its use. The user needs to know the data types of the arguments, and whether to call the Macro as an expression, a data step statement, a data step, or anywhere in a SAS program. The more obvious the function, the less documentation required.

Obvious Macro functions take less time to learn and to use. The Macro ChkDigit (see Sample Program Library in Appendix 1) has an obvious function. The user does not have to know much about the code to call it. A quick look at the program segment,

```
a=compress(put(IDNum,8.));
* Returns check digit for character value;
%ChkDigit(a);
ID=trim(put(IDNumZ8.))||"-"
||put(_ChkDigit,1.);
```
gives the user a good idea of the function and use of %ChkDigit.

Rule 3: Make them General

Two kinds of generality enhance a Macro function. First, a function that accepts a larger set of argument values saves time and effort during training and documentation. The function has fewer restrictions to describe. For example, Macro function %ChkDigit detects the type of each character in a string and acts accordingly. The function handles ID’s composed from any set of characters as well as ID’s composed solely of digits. Second, a function that does more than a limited counterpart does means that we need fewer functions to do the same number of tasks.

But generality does have a price. It usually requires more exact and detailed programming, and it may require more parameters. If these changes make proving the function more difficult, or make the use of the function less obvious, then their benefits may not offset their costs.

Rule 4: Make Them Efficient

Often we consider efficiency a matter of minimizing runtime costs. In fact, the hidden costs of detecting and correcting errors and of learning to develop and maintain correct programs usually exceed by far any likely reductions in runtime costs. Methods that reduce runtime costs at the expense of higher development costs only provide savings when
- an application requires that a program segment execute a very large number of times, or
- a program segment includes many conditions, assignments or iterations, or
- better technique makes the use of the function more obvious and general as well as equally correct and more efficient.

Only in the last case do we want to begin rewriting functions in a Macro function library. In the first two cases it makes better sense to write a specialized program for these special situations.

Not that efficiency does not matter. Discovering better techniques and using them to improve a function in a library has great importance. The benefits of such efficiency multiply with the number of users of the library, and improved technique in one function leads to improved technique in others. Since new functions use more primary ones as building blocks, a small gain in the efficiency of a primary function increases with the number of functions using it.

Techniques for Developing Better Macro Functions

Several techniques lead to more correct, obvious, general and efficient functions:

- intelligent use of positional and named parameters makes the use of the function more obvious and general. If correct execution of a Macro function always requires a small set of parameter values, defining them as positional parameters and placing them first in the parameter list stresses their role in the function. If the user has the option of including a parameter or not, defining it as a named parameter gives the user the option of specifying a value for it or omitting it entirely. If the Macro has a long parameter list, defining most of them as named parameters means that the user does not have to remember an order for listing them. Named parameters, in contrast to positional parameters, can appear in any order.

The %ReCode Macro in the Sample Program Library has a parameter list containing both positional and named parameters. The Macro requires the first two parameters, which we have defined as positional parameters, but it does not always require the named parameters.

- defining appropriate default values for parameters gives the user the choice of defaulting to a general and obvious default value or, for special situations, specifying another value.

%InSet (Sample Program Library) has a default value of NO for the _verbose parameter. A user may choose to override the default by specifying something other than NO.

- defining the names of data step variables as Macro variables allows the user to avoid conflicts between the default name for the variable and other data step variables. SAS makes no distinction between data step variables defined in Macro functions and variables defined elsewhere in the data step. The SAS system does not in general support local variables within a data step. Using Macro variables as data step variable names serves as a
workaround. The user specifies unique names in the function call.

Note the default value for the variable name _InSet in the Macro InSet. The function will assign a value to the data step variable _InSet unless the user specifies another name. The use of the same name for the macrovariable parameter and the data step variable name does not cause a conflict. The macrovariable name resolves to a pointer to an memory address; the data step variable name appears on the catalog of the dataset created by the data step.

- providing options triggered by a parameter value can make a Macro function more general. Different values in the function call for a parameter may, for example, cause the program to use that value in an expression. The Macro variable defined by a parameter may also serve as a condition that the Macro function tests and, depending on the outcome of the test, execute one sequence of statements or another.

- customizing the Macro function to the SAS environment helps solve a number of system-specific problems and offers major advantages. The user gains the advantages of advanced programming techniques without having to learn them. The function includes system-specific code. The user only has to choose the function library that matches the SAS environment used. A good function library eases transitions from one SAS environment to another. I use code developed under three different host systems and across many versions of SAS. Using a function library to isolate the system-specific parts of programs can minimize the effort required to develop programs that will run correctly and efficiently on multiple platforms.

Differences between ASCII and EBCDIC collating sequences of alphanumeric characters, for example, may affect the results of a program. Using a function library that takes this subtlety into account will make it possible to run the same main program on ASCII and EBCDIC host systems. Properly designed and implemented, program libraries can minimize the differences among SAS installations.

Summary

A library of character and list functions programmed as SAS Macros helps extend the SAS System in several ways. Use of the functions suppresses some of the details of programming and makes programs shorter and easier to understand. Access to the library encourages programmers to preserve and refine proven program segments. The library resource leads to savings of training and coding time, and it helps enhance the capabilities of programmers at all levels. Finally, a library designed for a particular SAS installation helps hide some of the peculiarities of that installation and gives all SAS programmers easier access to it.
Dalia Kahane, Robin McEntire, Mike Rhoads and Ian Whitlock made suggestions that led to significant improvements in the article; the author retains full responsibility for all remaining defects.

1 The UNIX system, for example, has a predefined directory for C subroutines. See Kernighan, Brian W. and Rob Pike, The UNIX Programming Environment (1984) Englewood Cliffe, NJ, p.63. UNIX is a trademark of Bell Laboratories

2 SAS is a registered trademark of SAS Institute, Inc., Cary, N.C., USA


6 "We should run test cases not to look for bugs, but to increase our confidence in a program we are quite sure is correct; finding an error should be the exception rather than the rule."


Author Contact
For questions, comments or suggestions for Macro functions, contact the author:
Sig Hermansen
Westat, Inc.
1650 Research Blvd.
Rockville, MD 20850-1520
(301) 251-4268

***Sample Library of SAS Macro Functions***;

*** S.W. Hermansen, Westat 7/20/92 ***;

%macro chkdigit(id, _chkdigit=_chkdigit, _i= _i, __digit=__digit);
* Use as data step statement;
* Requires character expression as argument (<= 40 characters);
* Assigns _chkdigit to numeric variable (default name _chkdigit) with _chkdigit;
* Remove extraneous characters before executing function;
  _chkdigit=0;
  do __i = 1 to length(&id);
  if not indexc(substr(&id, __i,1), '0123456789') then
    __digit=input(substr(&id, __i,1),PIB1.);
  else
    __digit=input(substr(&id, __i,1),1.);
  if not mod(_i,2) then __digit = 2
  * __digit;
    _chkdigit=_chkdigit+__digit;
  end;
  _chkdigit=mod(_chkdigit,10);
%mend chkdigit;

%macro InSet(_varstr, _set=, _InSet= _InSet, _type=, _verbose=NO);
*Use as data step statement. Note: case sensitive."
*Returns InSet (default name _InSet) = 1 (true) = 0 (false).";
* _varstr of character type: use put function to convert numerics.";
* _set = set of character values (<percent>quote() all but numbers).";
* Examples: %InSet(x,0123456789)
%InSet(x, _set='quote(ABCD))
* %InSet(x, _type=Number)
%InSet(x, _type=Alpha)
& _InSet=1;
%if _type=Number
%then %do;

if verify(substr(&__varstr,1,1),"+-0123456789") ne 0 then &__InSet=0;
else if length(&__varstr)>1 then
  if verify(substr(&__varstr,2),"0123456789,.") ne 0
    then &__InSet=0;
  %end;
%else %do;
  if &__type=Alpha
    then %do;
      if verify(trim(&__varstr),'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZLMNOPQRSTUVWXYZ') ne 0
        then &__InSet=0;
    %end;
%else %do;
  if verify(trim(&__varstr),"&__set") ne 0
    then &__InSet=0;
%end;
%mend;

* Macro ReCode error messages and main Macro;

%macro ReCodeM1;
  %put Function ReCode Error 1:;
  %put ChrttoNum=YES requires;
  %put %str( NewName not blank);
  %put %str( Both Fmt and InFmt);
%mend ReCodeM1;
%macro ReCodeM2;
  %put Function ReCode Error 2:;
  %put NumtoChr=YES requires;
  %put %str( NewName not blank);
  %put %str( Fmt not blank);
%mend ReCodeM2;
%macro ReCodeM3;
  put "Function ReCode Error 3:"
    put " non-numeric value found"
    put " numeric value required.";
  _error_+1;
%mend ReCodeM3;
%macro ReCode(VarName, Fmt, InFmt=, NewName=, ChrttoNum=NO, NumtoChr=NO);

  * VarName = SAS data step variable name;
  * Warning: specify NewName when VarName numeric variable:
    recoded variable will have character type;
  %global _error;
  %let _error=0;
  %if &ChrttoNum=YES and
    ((&NewName=) or (&Fmt=) or
    (&InFmt=))
    %then %do; &ReCodeM1;
    %let _error=1;
  %end;
  %if &NumtoChr=YES and
    ((&NewName=) or (&Fmt=))
    %then %do; &ReCodeM2;
    %let _error=1;
  %end;
  %if (not &__error) %then %do;
    %if (&ChrttoNum=YES)
      %then &NewName=input(
        put(&VarName,&Fmt •• ),&InFmt •• );
    %else %do;
      %if (&NumtoChr=YES)
        %then &InSet(&VarName,_type=Number)
          if InSet
            then
              &NewName=put(&VarName,&Fmt •• );
          else do; &ReCodeM3;
        %end;
    %else %do;
      %if (&NewName=)
        %then %let NewName=&VarName;
      &NewName=put(&VarName,&Fmt •• );
    %end;
  %end;
%mend;

* Use as data step statement;
* Precede data step with proc format defining Fmt parameter;
* See error messages for constraints;