Paper CC166

Your Own SAS Macros Are as Powerful as You Are Ingenious

Yinghua Shi, Department Of Treasury, Washington, DC

ABSTRACT

This article proposes, for user-written SAS macros, separate definitions for function-style macros and for routine-style macros, respectively. A function-style macro returns a value, while a routine-style macro does not. Implementation of function-style macros follows a set of rules that are not identical with the rules for routine-style macros. In client code, the method for invoking those two types of macros is not the same, either. Just like SAS language has a distinction between functions and call routines, it is natural for macros to also have that distinction.

With that distinction in place, this article describes the proper approach and rules to follow for writing function-style macros, and for writing routine-style macros. Usage of each kind of macro is also provided to describe the proper way of invoking the macros in client code.

This article also points out some common problems in writing and invoking macros that appeared in published articles. Some of those problems can cause errors in SAS programs, and therefore should be guarded against when designing and coding macros.

For each key point discussed in this article, working sample SAS code is provided to further illustrate what to do, what not to do, what to avoid, and what are good, or not-so-good coding practices in writing user-written macros.

INTRODUCTION

SAS® Macro Language has a set of system-defined macro functions. Examples of those macro functions are `%index`, `%scan`, `%substr`, `%syseval`, and `%upcase`, just to name a few.

SAS® Macro Facility, on the other hand, lays out the foundation for SAS programmers to write their own macros. User-written macros have the potential of tremendously enhancing SAS programs, and they are the focus of this article.

It is out of the scope of this article to discuss SAS macros in general. The first reference listed at the end of this article is SAS online document that describes SAS® Macro Language.

Published articles abound on the topic of writing SAS macros. Most of them are good, but often for beginners. Beyond beginner level, a distinctly good article on user-written macros can be found on the web. The second reference listed at the end of this article has the URL to that article.

This article does not cover the basics of writing SAS macros. Instead, focus is on how to write macros of both function-style and routine-style.

FUNCTIONS VERSUS ROUTINES

A common feature of SAS system-defined macro functions is that there is a return value, and that all macro parameters are input parameters. In C language syntax, SAS system-defined macro function prototype may be declared as follows.

```
char function_1(char parm_1, char parm_2, ...);
```

To simplify the discussion, in the above prototype we assume that each of the parameters is of character type, and that the return value is also of character type. Note that the means of data exchange from macro function to the client is through the macro function return value.

SAS® Macro Facility allows programmers to expand the means of data exchange by passing back values to client through macro function parameters. Macro function prototype with that capability may be declared, in C syntax, like this.

```
char function_2(char parm_1, char * parm_2, ...);
```

Parameter `parm_1` is still an input parameter, while `parm_2` is an output (or input/output) parameter which passes back a value to the client.

A common feature of `function_1` and `function_2` is that there is a return value. That feature defines macro functions, or function-style macros for user-written macros, a term to be used in this article. Whether a macro has output parameter or not is irrelevant to the definition of function-style macros.

Because function returns a value, the client code invokes a macro function in this manner.

```
return_value = some_function(...);
```

Routines, such as SAS call routines, on the other hand, do not return a value. In C language syntax, a routine prototype may be declared like this.

```
void routine_1(char parm_1, char parm_2, ...);
```

All parameters are input parameters, as the case for SAS call routines. The means of data exchange is one-way, from client to routine, through macro parameters.

SAS® Macro Facility allows programmers to expand the means of data exchange to make it two-way, by passing back values to client through macro parameters. Macro routine prototype with that capability may be declared, in C syntax, like this.

```
void routine_2(char parm_1, char * parm_2, ...);
```

Parameter `parm_1` is still an input parameter, while `parm_2` is an output (or input/output) parameter which passes back a value to client code.

A common feature of `routine_1` and `routine_2` is that there is no return value. That feature defines routines, or routine-style macros, a term to be used in this article. Whether a macro has output parameter or not is irrelevant to the definition of routine-style macros.

Because routines do not return a value, the client code invokes a routine-style macro in this manner.

```
some_routine(...);
```

While macro call of the following kind makes no sense.

```
return_value = some_routine(...);
```

The distinction between macros of function-style and routine-style has practical reasons, which will be made clearer in the following sections.

A good coding practice for either functions or routines is that they exchange data with client through the return value and/or through explicitly declared parameters of functions or routines. In other words, the means of data exchange of a function (or a routine) should be explicitly specified in the application programming interface (API) of the function (or the routine). That practice should be followed when designing user-written macros.

An example of explicitly defined means of data exchange is parameter `char * parm_2`, as is specified above in the prototype of `function_2` and that of `routine_2`.

Data exchange between macro and client through some implicit, global variable is not a good coding practice, and should be avoided.

COMMON PROBLEMS

The code segment listed below illustrates some common problems in writing macros. The code, as was alleged, is the implementation of a macro function that tests for existence of a given SAS dataset. The macro returns YES if the dataset exits, or NO otherwise.

```
%macro exist(dsn);
%global exist;
%if &dsn ne %then %do;
data _null_;
stop;
if 0 then set &dsn;
run;
%end;
%if &syserr=0 %then %let exist=YES;
%else %let exist=NO;
%mend exist;
```

A sample usage of that macro in client code (as provided in the same article where macro `%exist` was introduced) is listed below.

```
%exist(sasuser.bigdat)
if "&exist"="YES" then do;
... more SAS statements ...
```

The above macro and the client code would work. But what are the problems, then?

The first problem is declaring macro variable `exist` as global. Being global, the value of variable `exist` may be changed inadvertently by any other macros or SAS code that are running in the same SAS session. That will especially be a problem should macro ***%exist*** be used as a utility in a complex (or not very complex) SAS program.

The second problem is that the means of data exchange is not explicitly specified in the API of the macro. Instead, the macro code takes an implicit approach of declaring a global variable `exist`, which is used to pass back data to client code. The appropriate approach, for this particular macro, is to return value YES or NO, or to pass back YES or NO to client code through a parameter.

Lastly, macro ***%exist*** is a regular macro, not a macro function, as is alleged.

WRITING FUNCTION-STYLE MACROS

Start with a simple macro that takes no parameter and returns a value of character type.

For SAS code to be platform-independent, it is often necessary to know at runtime whether to use forward slash or back slash, as is used on Unix or Windows, respectively. This sample macro returns the proper slash based on the operating system of SAS runtime environment. In C language syntax, the macro prototype may be declared like this.

```
char slash();
```

Here is the implementation of the macro.

```
%macro slash(     /* returns '\' for Windows platform, '/' otherwise. */
```

```
);
  %if &sysscp. = WIN %then \;
  %else                    /;
%mend slash;
```

At runtime upon finishing executing the macro, the macro resolves to either '\' or '/', which is then placed, by the macro processor, on stack for the rest of the program to consume.

It is crucial to understand that function-style macro body must not contain base SAS statements. That rule ensures that when macro processor finishes executing the macro, there is no base SAS code left on the stack. Otherwise there could be unexpected results in client code, often leading to compile errors or causing the whole program to error out. Refer to either one of the references to further understand the reason.

Next is a sample usage of the macro `%slash`.

```
%macro slash_usage(
);
  %local slash;
  %let slash=%slash();
  %put slash is &slash..;
  %local file_pathname;
  %let file_pathname =
    %str(C:&slash.SAS&slash.MacroFacility&slash.src&slash.macros.sas);
  %put file_pathname is &file_pathname..;
%mend slash_usage;

%slash_usage();
```

Here is what is displayed in SAS log.

```
slash is \.
file_pathname is C:\SAS\MacroFacility\src\macros.sas.
```

Note that macro variable `slash` is declared as local. That is necessary to keep its value from being inadvertently changed by other macros or SAS code running the same SAS session.

In macro %slash_usage, it is totally fine to use this statement.

```
%let file_pathname =
  %str(C:%slash()SAS%slash()MacroFacility%slash()src%slash()macros.sas);
```

If the above statement is used in client code, there would be no need to even use macro variable `slash`.

The next example is a function-style macro that tests for the existence of a given folder. The macro returns 1 if the folder exists, or 0 otherwise. It takes one input parameter. In C syntax, the macro prototype may be declared like this.

```
int folder_exist(string folder);
```

Here is the implementation of the function-style macro.

```
%macro folder_exist(  /* returns 1 if folder exists, 0 otherwise. */
  folder =
);
  %if %length(&folder.)=0 %then %do;
```

```
    0
    %return;
  %end;
  %local fileref rc did;
  %let rc = %sysfunc(filename(fileref, &folder.));
  %let did = %sysfunc(dopen(&fileref.));
  %if &did=0 %then 0;
  %else            1;
  %let rc = %sysfunc(dclose(&did.));
  %let rc = %sysfunc(filename(fileref));
%mend folder_exist;
```

Note again that the whole macro body does not contain base SAS statement, a rule that function-style macro must conform to. All macro variables used in macro body are declared as local, another rule for writing function-style macros.

Next is a sample usage of the macro `%folder_exist`.

```
%macro folder_exist_usage(
);
  %local folder_exist;
  %let folder_exist
     = %folder_exist(folder=C:%slash()SAS%slash()MacroFacility%slash()src);
  %put folder_exist is &folder_exist..;
  %if %folder_exist(folder=C:%slash()SAS%slash()MacroFacility%slash()src) %then
    %put Yes.;
%mend folder_exist_usage;
%folder_exist_usage();
```

Here is what is displayed in SAS log.

```
folder_exist is 1.
Yes.
```

In client code, the return value from function-style macro must form seamlessly as part of a macro statement or base SAS statement. That is exactly the case for the `%let` statement and `%if` statement in the code body of macro `%folder_exist_usage`. That is a rule that client code must follow.

The next example uses the two function-style macros (`%slash` and `%folder_exist`) discussed above. This example is a macro (routine-style) that creates a sub-folder under a given parent folder.

```
%macro create_folder(
  root_folder = ,         /* name of root or parent folder */
  sub_folder  =           /* name of sub-folder to be created */
);
  %if %folder_exist(folder=&root_folder.)=0 %then %do;
    %put ERROR: Folder &root_folder. does not exist;
    %return;
  %end;
```

```
  %local folder;
  %let folder=%str(&root_folder.%slash()&sub_folder.);
  %if %folder_exist(folder=&folder.)=1 %then %do;
    %put Note: Folder &folder. already exists.;
    %return;
  %end;
  %let folder=%sysfunc(dcreate(&sub_folder., &root_folder.%slash()));
  %if %folder_exist(folder=&folder.)=1 %then
    %put Folder &folder. created.;
  %else
    %put ERROR: Folder &folder. creation failed.;
%mend create_folder;
```

Since the entire macro body does not contain base SAS code, it is not difficult to re-write the macro into a function-style macro. Those who are interested may take that as an exercise.

Next is a sample usage of macro `%create_folder`.

```
%macro create_folder_usage();
  %create_folder(
    root_folder = %str(C:\SAS\MacroFacility\logs),
    sub_folder  = test
  );
%mend create_folder_usage;

%create_folder_usage();
```

Here is what is displayed in SAS log.

```
Folder C:\SAS\MacroFacility\logs\test created.
```

As the last example of this section, consider a function-style macro that takes one parameter, which is for both input and output. The macro function reverses a given input string, and passes back the revered string via the same macro parameter. The macro function returns 1 on success, or 0 otherwise.

In C syntax, the macro prototype may be declared like this.

```
int reverse(string * value);
```

Here is the macro implementation.

```
%macro reverse(
  value =
);
  %if %length(&value.)>0  and %length(&&&value.)>0 %then %do;
    %let &value. = %sysfunc(reverse(&&&value.));
     1
  %end;
  %else
    0;
%mend reverse;
```

The first statement checks for validity of input value. The macro function performs reverse only when there is something to reverse.

In the macro code body, there is the use of multiple &. A short explanation is this. One & requests SAS to reference one time the macro variable following &, that is to get the value that the macro variable points to. Triple & requests SAS to reference the macro variable two times, that is to get the value pointed by the value that the macro variable points to. Using multiple references is the mechanism of implementing passing value by reference in SAS language.

For the use of multiple &, refer to the first reference listed at the end this article.

Next is a sample usage of macro `%reverse`.

```
%macro reverse_usage(
);
  %local my_value return;
  %let my_value = qwerty;
  %let return= %reverse(
    value = my_value
  );
  %put my_value is &my_value..;
  %put return is &return..;
%mend reverse_usage;
%reverse_usage();
```

Here is what is displayed in SAS log.

```
my_value is ytrewq.
return is 1.
```

Note again that in client code macro variables are declared as local.

As summary of this section, listed below are key points for writing function-style macros and for writing client code that invokes function-style macros.

- All macro variables in function-style macro body must be declared as local.
- Function-style macro body must contain macro statements only.
- The mechanism for data exchange must be explicitly specified in the API (return value and/or parameters) of the macro.
- The client code must declare its macro variables as local which are used for data exchange with the invoked macros.
- In the client code, the return value from function-style macro must seamlessly integrate into a macro statement or a SAS statement.

WRITING ROUTINE-STYLE MACROS

Routine-style macros are simpler to write than function-style macros, because of two reasons. First, routine-style macro does not return a value, and that simplifies code logic. The second is that an arbitrary mix of macro statements and base SAS statements may be used in writing routine-style macros.

Start with a simple macro that appends a token to a string. In C syntax, the macro prototype may be declared like this.

void `append_token` (string * str, string token);

The value of parameter `token` will be appended to the value of parameter `str`. The result will be passed back through parameter `str`. Parameter `str` is used for both input and output.

Here is the macro implementation.

```
%macro append_token(
  str   = ,
  token =
);
  %local temp;
  %let temp = %str(&&&str.);
  %let temp = %str(&temp.&token.);
  %let &str.= &temp.;
%mend append_token;
```

Refer to the previous example for the use of multiple & in macro code body.

Next is a sample usage of macro `%append_token`.

```
%macro append_token_usage(
);
  %local my_str;
  %let my_str=qwerty;
  %append_token(
    str   = my_str,
    token = _xyz
  );
  %put my_str is &my_str..;
%mend append_token_usage;
%append_token_usage();
```

Here is what is displayed in SAS log.

```
my_str is qwerty_xyz.
```

The next example illustrates a macro that passes back a list of strings, instead of a single string as in the previous example. The macro code body contains a mix of macro statements and base SAS statements.

Sometimes there is the need to find out files with a specific file extension in a given folder. File extensions can be, for example, txt, xls, sas, or log, etc.

The first two parameters of the macro specify the target folder and the target file extension, respectively. The third parameter is an output parameter that passes back a list of file names delimited by space.

Here is the macro implementation.

```
%macro get_file_names (
  folder    = ,           /* name of the target folder to search */
  extension  = ,          /* name of the target file extension */
  file_names =            /* a list of file name with target file extension */
);
  %if %folder_exist(folder=&folder.)=0 %then %do;
```

```
    %put ERROR: Folder &folder. does not exist;
    %return;
  %end;
  %if &sysscp.=WIN %then filename dir_list pipe "dir &folder.";
  %else                  filename dir_list pipe "ls &folder.";
  ;
  data ds_files (drop=temp);
    infile dir_list pad;
    length temp $256;
    input temp $256.;
    if index(temp, ".&extension.")>0;
    file_name=scan(temp, -1, " ");
  run;
  %local temp;
  proc sql noprint;
    select file_name into : temp separated " "
     from ds_files;
     drop table ds_files;
  quit;
  %let &file_names. = &temp.;
%mend get_file_names;
```

Instead of using, as shown above, system command `ls` or `dir` to find out the files in target folder, base SAS functions may be used just as well. But it is the opinion of the author that using `ls` and `dir` makes the code logic less convoluted and easier to understand.

Next is a sample usage of macro `%get_file_names`.

```
%macro get_file_names_usage (
);
  %local log_file_names;
  %get_file_names (
    folder = C:\SAS\logs,
    extension  = log,
    file_names = log_file_names
  );
  %put log_file_names is &log_file_names..;
%mend get_file_names_usage;

%get_file_names_usage();
```

Here is what is displayed in SAS log.

```
log_file_names is log_1.log log_2.log log_3.log.
```

As summary of this section, listed below are key points for writing routine-style macros and for writing client code that calls routine-style macros.

- All macro variables in routine-style macro code body must be declared as local.
- Routine-style macro is free to use an arbitrary mix of macro statements and base SAS statements.

- The mechanism for data exchange must be explicitly specified in the API (return value and/or parameters) of the macro.
- The client code must declare its macro variables as local which are used for data exchange with the invoked macros.

CONCLUSION

This paper proposes separate definitions for function-style macros and for routine-style macros. Function-style macro returns a value, while routine-style macro does not.

There are practical reasons for having that distinction, because separate rules should be followed for implementing those two types of macros, and for implementing client code that invokes those two types of macros.

Function-style macro body must contain macro statements only. That ensures that at the end of macro execution, the return value is the only item resolved by the macro processor, and placed on stack by the macro processor. Function-style macro must declare its own macro variables as local to preserve data integrity. In client code, return value from function-style macro must seamlessly integrate into a macro statement or a base SAS statement.

Routine-style macro may contain an arbitrary mix of macro statements or base SAS statements. The macro must declare its own macro variables as local to preserve data integrity.

Good coding practice requires that the mechanism for data exchange between macro and client must be explicitly specified in macro API (return value and/or parameters) for macros of both styles. Data exchange through some global variable must be avoided.

User-written SAS macros are a powerful tool for implementing custom logic in SAS programs. When properly designed and implemented, user-written SAS macros are able to contribute to better code modularity, higher code re-use, easier code maintenance, and less efforts in overall program development. They provide you with the potential of enhancing your SAS program in so many ways. The limit is your own ingenuity.

REFERENCES

SAS Institute Inc, Copyright © 2011, *SAS® 9.3 Macro Language Reference*
Available at http://support.sas.com/documentation/cdl/en/mcrolref/62978/PDF/default/mcrolref.pdf

Roland Rashleigh-Berry, *Tips on writing SAS macros*
Available at http://www.datasavantconsulting.com/roland/macrotips.html

CONTACT INFORMATION

Yinghua Shi
Phone: 646-250-3115, 202-957-8308
E-mail: yinghua_shi@yahoo.com