Paper 024-2010

Functioning at an Advanced Level: PROC FCMP and PROC PROTO

Peter Eberhardt, Fernwood Consulting Group Inc.,
Toronto ON Canada

ABSTRACT

In version 9.2, SAS® added the ability to create user-written functions and call routines using the SAS DATA step language, and make these functions and call routines available to the DATA step programmer. In an earlier paper, I demonstrated the basics of creating and using your own functions. This paper goes beyond those basic steps and shows some more advanced uses of the PROC FCMP and its cousin, PROC PROTO.

In this paper we will review creating DATA step language functions with PROC FCMP. After showing the creation and use of these functions, the paper will shift to PROC PROTO and show how to incorporate C language functions into SAS. This paper is for advanced SAS programmers. Basic knowledge of the C programming language will be useful to help understand some of the examples.

INTRODUCTION

SAS® programmers inherently understand the concept of breaking a complex problem into manageable pieces; a quick look at any SAS programme with the rich variety of DATA and PROC steps will demonstrate this. However, within the main programming environment, the DATA step, the ability to break down problem complexity into manageable pieces has been limited to two basic methods: ***Link/Return*** blocks and ***Macros***.

Link/Return blocks allow the SAS programmer to isolate (encapsulate) business logic using familiar DATA step code; although useful, there are some drawbacks to this method. First, the SAS code is embedded into the DATA step. This means the ***Link/Return*** block needs to be copied to each DATA step where it is to be used. As with all cut-and-paste operations this is both error prone and maintenance heavy. Second, since the ***Link/Return*** block is part of the DATA step, it uses DATA step variables not parameters. This can lead to the need to a series of variable assignments before and after the Link statement so the DATA step variables that need processed are appropriately renamed to and from the ***Link/Return*** block variable names. Once again, this can lead to errors.

SAS ***Macros*** help to eliminate some of the re-use and parameter issues at the expense of introducing non-DATA step syntax. In addition, the overuse of macros can lead to code that is more difficult to read and maintain.

To overcome the limitations of these two alternatives, SAS 9.2 has made PROC FCMP available to the DATA step programmer; prior to SAS 9.2 PROC FCMP functions could be used in several SAS/STAT®, SAS/ETS® and SAS/OR® procedures. PROC FCMP allows the SAS programmer to create and reuse parameterized functions using familiar DATA step code. This paper will step you through the creation, testing, and deployment of your own functions. Although PROC FCMP uses DATA step like syntax, there are a few differences between its syntax and the DATA step syntax; these differences will be highlighted.

PROC PROTO provides a facility that allows you to use functions written in C or C++ . Once these functions are registered using PROC PROTO they can be called from any SAS function or call routine defined in PROC FCMP. Some simple examples will be demonstrated in the paper.

Before we look at how to create a function with PROC FCMP, let’s examine the definition of a function.

WHAT IS A FUNCTION

Every year the local ballet company holds a black-tie function to raise money; in this case, a function can be defined as a social gathering. This is not the type of function to be created by PROC FCMP. The function of a soda bottle is to hold the soda; in this case a function can be defined as the purpose for which the object exists. This is not the type of function to be created by PROC FCMP. What then is the thing we SAS programmers call a function?

A general definition of a function is:

> A function is a rule for transforming zero or more values called ***arguments***; the result of the transformation is called the value or result of the function. The transformation can also have side effect; that is permanent changes in the values of arguments.

The SAS 9.2 online help has a similar definition:

> A **SAS function** performs a computation or system manipulation on arguments, and returns a value that can be used in an assignment statement or elsewhere in expressions.

What is not explicit in either the general description above or in the definition supplied in the SAS help is the fact that functions have names. Not only does the function have a name but also that name should be meaningful. So although the statement:

```
rc = foo(1500, 3);
```

may be a valid way to call a function, a function called ***foo*** does not convey very much meaning to most people reading the code. On the other hand the statement:

```
interestPaid = monthlyInterest(2500, 3);
```

would make us guess that function is probably calculating a monthly interest on $2,500 at 3% interest. This would be reinforced if the function were called with variables instead of constants:

```
interestPaid = monthlyInterest(balance, iRate);
```

The SAS 9.2 online help has a definition for a second type of function, called a CALL Routine:

> A **CALL routine** alters variable values or performs other system functions. CALL routines are similar to functions, but differ from functions in that you cannot use them in assignment statements or expressions.
>
> All SAS CALL routines are invoked with CALL statements. That is, the name of the routine must appear after the keyword CALL on the CALL statement

In our general definition of a function we introduced side effects, that is, changes in the values of the arguments. In the case of a SAS CALL Routine these side effects are the method for returning values. If our function ***monthlyInterest*** were written as a CALL Routine, it would look something like:

```
call monthlyInterest(balance, iRate, interestPaid);
```

In this case the variable interestPaid is passed in as an argument and its value is changed in the CALL Routine.

In this paper I will be using the term function to include both functions and CALL Routines as defined in the SAS help; in cases where there could be ambiguity I will explicitly refer to SAS functions or SAS CALL Routines.

WHAT IS A RULE

In our general definition of a function we called it "*a rule for transforming zero or more values*". The rule can be a simple formula such as:

- Square(x) == x * x
- Product(x, y) == x * y

The rule may be more complex such as:

1. Discount(quantity) == if quantity > 100 then discount = 0.15 else discount = 0.0

In addition, some of the rules we will implement will end up being more complex than originally conceived. Consider a simple rule of division:

2. Divide(dividend, divisor) == dividend / divisor

This is fine, as long as divisor is a number and not equal to zero. Our simple rule can be expanded to something more complex, but also more robust:

3. Divide(dividend, divisor) == if divisor is missing then missing else if divisor is 0 then 0 else dividend / divisor

Such a divide function in a data step where we expect a number of missing or 0 values in the divisor field would simplify our code and clean up the ***Division by Zero*** and ***Missing Value*** NOTEs in our log.

With this introduction to functions, let's turn to creating our own functions.

CREATING FUNCTIONS

Starting in SAS 9.2 DATA step programmers can start to take advantage of PROC FCMP to create and use DATA step language functions. PROC FCMP was available before SAS 9.2, but the functions could only be used in selected SAS/STAT and SAS/ETS procs. We will start by looking at how we could create 2 simple functions, one to return an integer constant and the other to return a character constant. In order to create these functions we will have to learn some of the PROC FCMP syntax.

PROC FCMP SYNTAX

The following is the syntax for PROC FCMP:

PROC FCMP option(s);

ARRAY	array-name[dimensions] </NOSYMBOLS | variable(s) | constant(s) | (initial-value(s))>;
ATTRIB	variable(s) <FORMAT=format-name LABEL='label' LENGTH=length>;
FORMAT	variable(s) <format> <DEFAULT=default-format>;
FUNCTION	function-name(argument-1, argument-2, ..., argument-n) <$> <length>;
LABEL	variable='label';
LENGTH	variable(s)<$> length <DEFAULT=n>;
STRUCT	structure-name variable;

SUBROUTINE	subroutine-name (argument-1, argument-2, ..., argument-n);
	OUTARGS out-argument-1, out- argument-2, ..., out-argument-n;

As you can see, there are some familiar elements in this syntax. We will be concentrating on two of these elements, ***function*** and ***subroutine***, as well as some of the proc ***options.***

```
   proc fcmp
1      outlib=work.funcs.Test;  /* where will the functions be saved */

2  function whatAmI();      /* declare a function returning a number */
3  return(42);       /* return the number */
4   endsub;

5  function whereAmI() $;      /* declare a function retuning a string */
6  return('In Test');   /* return the string */
7  endsub;

   quit;
```

Let's examine the code above.

1. ***outlib=work.funcs.test***:

   this option tells SAS where to store the function. If no ***outlib=*** is entered then none of the functions are saved.

2. ***function whatAmI()***

   function is the keyword to start the definition. Functions return a value, either numeric or character; since there is not character option in the definition, this function will return a number. NOTE: the function name, like all SAS names, is not case sensitive.

   whatAmI is the name of the function.

   () indicates there are no arguments to the function.

3. ***return(42);***

   the return statement is used to return the value of the function to the DATA step. In this case we are returning a constant (42); however, it could be, and usually will be, a variable calculated in the function.

4. ***endsub***

   this ends the definition of the function.

5. ***function whereAmI()*** $

   the ***$*** modifier tells SAS that this function will return a character string. Since there is no length specified the function can return a string of any valid SAS string length. NOTE: the function name, like all SAS names, is not case sensitive.

6. ***return('In Test')***

   the return statement returns the constant string ***In Test***; however, it could be, and usually will be, a variable calculated in the function.

7. ***endsub***

   this ends the definition of the function.

After we submit this code, we check the log. The log for this code is:

```
1    proc fcmp
2        outlib=work.funcs.test;  /* where will the functions be saved */
3    function whatAmI(); /* declare a function returning a number */
4      return(42);       /* return the number */
5    endsub;
6
7    function whereAmI() $; /* declare a function retuning a string */
8      return('In Test');   /* return the string */
9    endsub;
10
11   quit;

NOTE: Function whereAmI saved to work.funcs.test.
NOTE: Function whatAmI saved to work.funcs.test.
NOTE: PROCEDURE FCMP used (Total process time):
      real time           1.48 seconds
      cpu time            0.14 seconds
```

We see the function appears to have compiled correctly, so let's try using them in a simple DATA _NULL_ step:

```
data _null_;
     rci = whatAmI();
     put rci=; /* should be 42 */
     rcc = whereAmI();
     put rcc=; /* should be In Test */
run;
```

If we look at the log, there appears to be a problem, our new functions could not be found:

```
12
13   data _null_;
14        rci = whatAmI();
                -------
                68
ERROR 68-185: The function WHATAMI is unknown, or cannot be accessed.

15        put rci=;
16        rcc = whereAmI();
                --------
                68
ERROR 68-185: The function WHEREAMI is unknown, or cannot be accessed.

17        put rcc=;
18   run;
```

In order for SAS to recognize our functions we have to use the SAS option ***CMPLIB=***; this option can be set in the configuration file, in the autoexec.sas file, or in your program; in these examples the ***CMPLIB=*** option will be set in the program.

When we invoked PROC FCMP we used the ***outlib=*** option to tell SAS where to store the function; in our case we used ***work.funcs.test***. We will look at the library structure later, for now we will take on faith that the ***outlib=*** option is a three level reference (1 – work, 2 – funcs, 3 – test) but the ***CMPLIB=*** options takes only a two lever reference ***work.funcs*** (1 – work, 2 – funcs). The following is the SAS log when we add the options statement and re-execute our code:

```
19   options cmplib=work.funcs;
20   data _null_;
21        rci = whatAmI();
22        put rci=; /* should be 42 */
23        rcc = whereAmI();
24        put rcc=; /* should be In Test */
25   run;

rci=42
rcc=In Test
NOTE: DATA statement used (Total process time):
      real time           0.14 seconds
      cpu time            0.07 seconds
```

Now we see the results we expected.

To recap our progress, here are the highlights:

1. we used the PROC FCMP ***outlib=*** option to tell SAS where to store the compiled functions.
2. we create two functions, one returning a number and one returning a string, using the ***function*** statement.
3. we used the ***cmplib=*** system option to tell SAS where to find our functions.
4. we invoked the two new functions in a DATA step and confirmed the results.

The next example will demonstrate the use of function arguments and building in some decision rules to our functions. In this example the function will take one numeric argument and return a value based on the argument. If you had a series of variables (e.g. survey questions) that needed to be recoded the same way, you could create a similar function. The following is the SAS log showing the call to PROC FCMP:

```
   27   proc fcmp
   28       outlib=work.funcs.test;   /* where will the functions be saved */
1  29   function whatAmI(startValue); /* declare a function returning a number */
2  WARNING: Function whatAmI is already defined in packet test. Function whatAmI
   as defined in the
            current program will be used as default when packet is not specified.
3  30     if missing(startValue)   then rc=.S;
   31     else if startValue <  0  then rc=.Z;
   32     else if startValue < 20  then rc=0;
   33     else if startValue < 50  then rc=20;
   34     else if startValue < 100 then rc=50;
   35     else                          rc=100;
4  36     return(rc);
   37   endsub;
   38
   39   quit;

   NOTE: Function whatAmI saved to work.funcs.test.
   NOTE: PROCEDURE FCMP used (Total process time):
         real time           0.12 seconds
         cpu time            0.06 seconds
```

Let's examine the code above.

1. ***Function whatAml(startValue)***

   we are creating a function that takes one numeric argument (***startValue***) and will return a numeric

2. ***WARNING: Function whatAml…***

   a SAS warning telling us we are redefining the function ***whatAml***.

3. ***If … then … else… rc=***

   a series of if/then/else statements where we set a value of the variable ***rc***. For more complex logic we could have ***do… end*** blocks.

4. ***return(rc)***

   return the variable ***rc*** to the DATA step.

A check of the log of a DATA _NULL_ step shows the new function working.

```
40
41   options cmplib=work.funcs;
42   data _null_;
43        iAm = .;
44        rci = whatAmI(iAM);
45        put iAM= rci=; /* should be .S */
46        iAm = -1;
47        rci = whatAmI(iAM);
```

```
48         put iAM= rci=; /* should be .Z */
49         iAm = 10;
50         rci = whatAmI(iAM);
51         put iAM= rci=; /* should be 0 */
52         iAm = 30;
53         rci = whatAmI(iAM);
54         put iAM= rci=; /* should be 20 */
55         iAm = 80;
56         rci = whatAmI(iAM);
57         put iAM= rci=; /* should be 50 */
58         iAm = 20000;
59         rci = whatAmI(iAM);
60         put iAM= rci=; /* should be 100 */
61    run;

iAm=. rci=S
iAm=-1 rci=Z
iAm=10 rci=0
iAm=30 rci=20
iAm=80 rci=50
iAm=20000 rci=100
NOTE: DATA statement used (Total process time):
      real time           0.10 seconds
      cpu time            0.07 seconds
```

The second type of function that can be defined is a subroutine, or in SAS terms, a CALL Routine. We can redefine our function as a subroutine as follows:

```
    proc fcmp
        outlib=work.funcs.test;         /* where will the functions be saved */

1   subroutine whatAmI(startValue, rc); /* declare a subroutine             */

      if missing(startValue)   then rc=.S;
      else if startValue <  0  then rc=.Z;
      else if startValue < 20  then rc=0;
      else if startValue < 50  then rc=20;
      else if startValue < 100 then rc=50;
      else                          rc=100;
2     return;
    endsub;
    quit;
```

The important change here is

1. ***subroutine whatAmI(startValue, rc)***

   we are using ***subroutine*** instead if ***function***. In addition, we have two arguments instead of one. We will use the ***rc*** argument to return the value to the DATA step.

2. ***return***

   the return statement takes no arguments. In this example we could have omitted the ***return*** statement since it is implicit with the ***endsub*** statement.

When we examine a portion of the SAS log for this code we see some warning messages:

```
   62
   63   proc fcmp
   64       outlib=work.funcs.test;         /* where will the functions be saved
   */
   65
   66   subroutine whatAmI(startValue, rc); /* declare a subroutine             */
1  WARNING: Function whatAmI is already defined in packet test. Function whatAmI
   as defined in the
            current program will be used as default when packet is not specified.
   67
   68     if missing(startValue)   then rc=.S;
2  WARNING: The variable rc should not be the result of the '=' operation because
   it is a read-only argument to subroutine whatAmI. Any changes to this argument
   will not be returned from the subroutine whatAmI. Use the OUTARGS statement to
   allow return values from a subroutine.
```

1. ***WARNING: Function whatAmI***

   the WARNING telling us we are redefining the definition.

2. ***WARNING: The variable rc should not…***

   We wanted to use ***rc*** to return a value to the DATA step, however SAS is telling us that any changes to the variable will not be returned to the DATA step. It also reminds us we missed the ***outargs*** statement.

The subroutine did compile and was saved. We could try accessing it in a DATA step; this SAS log shows what the result will be:

```
   79   data _null_;
   80        length rci 8.;
1  81        rci = -1;
   82        iAm = .;
2  83        call whatAmI(iAM, rci);
   84        put iAM= rci=; /* should be .S */
   85        iAm = -1;
   86        call whatAmI(iAM, rci);
   87        put iAM= rci=; /* should be .Z */
   88        iAm = 10;
   89        call whatAmI(iAM, rci);
   90        put iAM= rci=; /* should be 0 */
```

```
  91        iAm = 30;
  92        call whatAmI(iAM, rci);
  93        put iAM= rci=; /* should be 20 */
  94        iAm = 80;
  95        call whatAmI(iAM, rci);
  96        put iAM= rci=; /* should be 50 */
  97        iAm = 20000;
  98        call whatAmI(iAM, rci);
  99        put iAM= rci=; /* should be 100 */
  100  run;

3 iAm=. rci=-1
  iAm=-1 rci=-1
  iAm=10 rci=-1
  iAm=30 rci=-1
  iAm=80 rci=-1
  iAm=20000 rci=-1
  NOTE: DATA statement used (Total process time):
        real time           0.10 seconds
        cpu time            0.07 seconds
```

From the log we see:

1. parameters ***iAm*** and ***rci*** are initialized

2. the subroutine is called with the two arguments

Note there are no WARNINGs or ERRORs, so the subroutine was called.

3. iAm=. rci=-1***…***

Note that the value of rci has not been changed in any of the subroutine calls

The fix for this is simple; add the ***outargs*** statement as the following SAS log shows

```
  103  proc fcmp
  104      outlib=work.funcs.test;         /* where will the functions be saved
  */
  105
  106  subroutine whatAmI(startValue, rc); /* declare a subroutine             */
  WARNING: Function whatAmI is already defined in packet test. Function whatAmI
  as defined in the current program will be used as default when packet is not
  specified.
1 107            outargs rc;             /* argument rc will return a value */
  108
  109    if missing(startValue)   then rc=.S;
  110    else if startValue <  0  then rc=.Z;
  111    else if startValue < 20  then rc=0;
  112    else if startValue < 50  then rc=20;
  113    else if startValue < 100 then rc=50;
```

```
114    else                             rc=100;
115  endsub;
116
117  quit;

NOTE: Function whatAmI saved to work.funcs.test.
NOTE: PROCEDURE FCMP used (Total process time):
      real time           0.14 seconds
      cpu time            0.06 seconds
118
119  options cmplib=work.funcs;
120  data _null_;
121       length rci 8.;
122       rci = -1;
123       iAm = .;
124       call whatAmI(iAM, rci);
125       put iAM= rci=; /* should be .S */
126       iAm = -1;
127       call whatAmI(iAM, rci);
128       put iAM= rci=; /* should be .Z */
129       iAm = 10;
130       call whatAmI(iAM, rci);
131       put iAM= rci=; /* should be 0 */
132       iAm = 30;
133       call whatAmI(iAM, rci);
134       put iAM= rci=; /* should be 20 */
135       iAm = 80;
136       call whatAmI(iAM, rci);
137       put iAM= rci=; /* should be 50 */
138       iAm = 20000;
139       call whatAmI(iAM, rci);
140       put iAM= rci=; /* should be 100 */
141  run;
2 iAm=. rci=S
iAm=-1 rci=Z
iAm=10 rci=0
iAm=30 rci=20
iAm=80 rci=50
iAm=20000 rci=100
NOTE: DATA statement used (Total process time):
      real time           0.12 seconds
      cpu time            0.07 seconds
```

Let’s examine the code above.

1. ***outargs rc;***

   this tell SAS the value of ***rc*** can be altered and the altered value returned

2. ***iAm=. rci=S…***

   our CALL Routine now returns the correct values.

After looking at these examples of a ***function*** and a ***subroutine***, you might ask yourself “Why would I use a subroutine when a function calling a function is clearer?”.. One main reason is that a subroutine can return multiple values whereas a function can only return one value. The following scenario demonstrates the use of a subroutine to return multiple values.

One data set has the following columns related to payments:

Column	Description
ID	Unique entity identifier
GROUP	Group in which ID is a member
PAYRULE	Payment rule
PAYDATE	Payment Date. There is only one payment date per month for everyone
PAYAMOUNT	Amount of the Payment

There are several hundred thousand of these records covering three years.

A second dataset has the following columns relating to payments for specific services.

Column	Description
ID	Unique entity identifier.
GROUP	Group in which ID is a member
SERVICECODE	A code identifying the service
SERVICEDATE	Service Date. These are the actual date of service
SERVICEAMOUNT	Amount of the Payment

There are about 140 million of these records per year covering the same three year period. The goal is to combine the payments by ID, GROUP and MONTH. In order to get a common ‘month’ value it was decided that the PAYDATE column would be the appropriate value to use. In order to do that, the SERVICEDATE had to be converted to the equivalent PAYDATE. In addition, since this process would be repeated on a monthly basis, a repeatable and automated process was required. Essentially a simple lookup that would convert any date to the appropriate PAYDATE was needed. To minimize the number of times the larger dataset had to be accessed a format lookup to convert the SERVICEDATE to the PAYDATE was chosen. The problem then is how to automate this process.

The first step was to create a list of distinct PAYDATES; remember, each month has only one PAYDATE. With only one date per month, we would need just two other values to create the format – the first day of the month and the last day of the month. The following subroutine shows how to do this:

```
1 proc fcmp outlib=work.Funcs.Dates;
2 subroutine datesInMonth(inDate, startDate, endDate, days);
3     outargs startDate, endDate, days;

4     nextMonth = intnx('month', inDate, 1);
      startDate = intnx('month', nextMonth, -1);
```

```
     endDate = nextMonth - 1;
     days = nextMonth - startDate;
     return;
endsub;
```

Let’s review this code:

1. ***outlib=work.myFuncs.Dates***

   the compiled functions will be going into a different 3rd level location.

2. ***subroutine datesInMonth(inDate, startDate, endDate, days)***

   an attempt to give both a meaning routine name, and meaningful argument names.

3. ***outargs startDate, endDate, days***

   three of the arguments will return values. In this way we need only one call to get three new values. ***startDate*** will be the first day of the month for ***inDate***, ***endDate*** will be the last day of the month for ***inDate***, and ***days*** will be the number of days in the month. For the problem as described above this is not needed, but for the real problem it was also required.

4. ***nextMonth = intnx(….***

   using the built-in SAS function ***intnx()*** and some simple date arithmetic we can calculate the values we want to return.

With this subroutine in place, the following code shows how to create and use the format:

```
data cntlinPaydates;
     retain fmtName 'payDates';
     set payDates;
     call missing(start, end, days);
1    call datesInMonth(payDate, start, end, days);
     put start= end= days= paydate=;
     label = paydate;
     format payDate start end yymmdd10.;
run;

2 proc format cntlin=cntlinPaydates;
run;

data test;
     set allVisits;
3    paydate = input(put(servicedate, paydates.), 7.);
     if _N_ = 1000 then stop;
     format paydate yymmdd10.;
run;
```

The main parts of this code are:

1. create the ***cntlin*** dataset using the first day in the month as the start value, the last day of the month as the end value and the ***PAYDATE*** as the label.
2. create the format using the ***cntlin*** dataset just created.

3. Apply the format to ***serviceDate*** to lookup the appropriate ***paydate***. Our two datasets now have a common date value we can use in a join.

In this section we looked at how you create functions and subroutines. The example functions were very simple since the purpose here is to show the basic mechanics of creating a function. Needless to say, complex rules can be encapsulated in functions; to see examples of a complex functions look at Secosky (2007).

Now that we know how to create functions, let's look at how to test them before compiling them into a permanent library.

TESTING/DEBUGGING FUNCTIONS

In the previous section we looked at creating and storing functions, then using them in a DATA _NULL_ step. For simple functions this works well, however for more complex functions we need to look at alternatives. We do not have access to the DATA step debugger so we will have to make judicious use of the ***PUT*** statement within the function to help test and debug new functions. The results of the ***PUT*** statement can go to the PRINT (default) and LOG destinations.

Let's revisit our ***daysInMonth*** routine and add some ***PUT*** statements to see the results as we execute:

```
  154  proc fcmp outlib=work.Funcs.Dates;;
  155  subroutine datesInMonth(inDate, startDate, endDate, days);
  WARNING: Function datesInMonth is already defined in packet Dates. Function
  datesInMonth as defined in the current program will be used as default when
  packet is not specified.
  156       outargs startDate, endDate, days;
1 157       FILE log;
2 158       put inDate yymmdd10. ;
  159       nextMonth = intnx('month', inDate, 1);
  160       put nextMonth yymmdd10.;
  161       startDate = intnx('month', nextMonth, -1);
  162       put startDate yymmdd10.;
  163       endDate = nextMonth - 1;
  164       put endDate yymmdd10.;
  165       days = nextMonth - startDate;
  166       put days;
  167       return;
  168  endsub;
  169  QUIT;

  NOTE: Function datesInMonth saved to work.Funcs.Dates.
  NOTE: PROCEDURE FCMP used (Total process time):
        real time           0.10 seconds
        cpu time            0.04 seconds



  170  data _null_;
  171       format firstDay lastDay yymmdd10.
  172              numdays 2.;
  173       call missing(firstDay, lastDay, numdays);
3 174       PUT 'calling routine:';
```

```
   175       call datesInMonth('15jan2009'd, firstDay, lastDay, numdays);
   176       PUT 'end of routine:';
   177       put _all_;
   178
   179       PUT 'calling routine:';
   180       call datesInMonth('15FEB2009'd, firstDay, lastDay, numdays);
   181       PUT 'end of routine:';
   182       put _all_;
   183  run;

4  calling routine:
   2009-01-15
5  2009-02-01
   2009-01-01
   2009-01-31
   31
   end of routine:
   firstDay=2009-01-01 lastDay=2009-01-31 numdays=31 _ERROR_=0 _N_=1
   calling routine:
   2009-02-15
   2009-03-01
   2009-02-01
   2009-02-28
   28
   end of routine:
   firstDay=2009-02-01 lastDay=2009-02-28 numdays=28 _ERROR_=0 _N_=1
   NOTE: DATA statement used (Total process time):
         real time           0.10 seconds
         cpu time            0.06 seconds
```

The main points here are

1. ***FILE log***

   direct the results of the PUT statements to the log.

2. ***PUT inDate yymmdd10.*** (and others)

   Put the contents of inDate to the log using the specified date format. Similar statements are used for the other variables.

3. ***call datesInMonth…***

   call the routine with a constant value in the ***inDate*** parameter. Note that messages are also written before and after the call.

4. The message from the DATA step before the call to the routine.

5. The values of the variable within the routine.

Earlier, when we first saw ***outlib=*** it was mentioned that if not library is given then the function is not saved. At first glance this seems odd. That is, why would you create a function if you were not going to save it? One answer is “So you can test it”. You can call functions within PROC FCMP to test them. For example:

```
184  proc fcmp ;
185  subroutine datesInMonth(inDate, startDate, endDate, days);
WARNING: Function datesInMonth is already defined in packet Dates. Function
datesInMonth as defined
         in the current program will be used as default when packet is not
specified.
186        outargs startDate, endDate, days;
187        FILE log;
188        put inDate yymmdd10. ;
189        nextMonth = intnx('month', inDate, 1);
190        put nextMonth yymmdd10.;
191        startDate = intnx('month', nextMonth, -1);
192        put startDate yymmdd10.;
193        endDate = nextMonth - 1;
194        put endDate yymmdd10.;
195        days = nextMonth - startDate;
196        put days;
197        return;
198  endsub;
199  format testDate1 testDate2 yymmdd10.;
200  format days 3.;
201        call datesInMonth('15jan2009'd, testDate1, testDate2, days);
202        put 'date1: ' testDate1  ' date2: ' testDate2 ' days: ' days;
203
204  QUIT;

2009-01-15
2009-02-01
2009-01-01
2009-01-31
31
date1:  2009-01-01  date2:  2009-01-31  days:   31
NOTE: PROCEDURE FCMP used (Total process time):
      real time           0.12 seconds
      cpu time            0.03 seconds
```

We can see from this SAS log:

1. PROC FCMP was called with no outlib= statement
2. Variables for the call are given formats
3. The routine ***datesInMonth*** is called
4. The variables that were returned will be displayed in the log

Although it is possible to test new functions within PROC FCMP, using a DATA step does provide more flexibility for testing and debugging.

STORING AND SHARING FUNCTIONS

Looking back at the ***outlib=*** option for PROC FCMP we see there are three levels in the name; in our ***datesInMonth*** example the library was ***work.funcs.Dates***. What does this mean? All of the functions defined in this call to PROC FCMP will be saved in a package called ***Dates***; the package is stored in the data set ***work.funcs***. A package is simply the collection of functions and subroutines, each with a unique name. Although it is possible to have the same function name in two different packages on the same data set there is no way within a DATA step to specify which package to use.

Although it is not possible to be able to access the two versions of a function if they are in different packages in the same data set, it is possible to access different versions of the same function if they are in different data sets. For example, the Risk department may have a larger set of parameters for a particular function (or functions) than the rest of the company. The subset of risk related functions could be in the package ***allfuncs.risk.dates*** while the common set (including the general set of functions intended for Risk) could be in the package ***allfunc.company.dates***. Then, to make sure each gets the appropriate versions the ***cmplib=*** options could be set as follows:

1. Risk: ***cmplib=(allfunc.company.dates***. ***allfuncs.risk.dates)***
2. Everyone else: ***cmplib= allfunc.company.dates***.

The ***cmplib=*** option can set a “search path” of libraries. Note that the search path is left to right (***allfuncs.risk.dates*** is search first, then ***allfunc.company.date***); this behaviour is not the same as other SAS “search paths” such as ***fmtsearch=***, so be careful when using it.

In order to share functions they simply need to be compiled into packages that are saved in a public folder. The SAS autoexec and/or configuration files can then appropriate ***cmplib=*** and ***libname*** statements.

RECURSION

A function that calls itself, either directly or indirectly, is called a recursive function. Essentially the function does part of the work, then calls itself to do the rest of the work. Every recursive solution involves two major parts or cases, the second part having three components.

1. base case(s), in which the problem is simple enough to be solved directly, and
2. recursive case(s). A recursive case has three components:
   - divide the problem into one or more simpler or smaller parts of the problem,
   - call the function (recursively) on each part, and
   - combine the solutions of the parts into a solution for the problem.

The base case is the ‘stop’ condition – once hit the function stops calling itself. Needless to say, if there were no ‘stop’ condition the function would theoretically never end. In reality it would end with an ‘out of memory’ error.

Each time a recursive function is called, it allocates a bit of private memory for itself and when the function exits (returns a value), this private memory Is freed. If the function had no stop condition, as each call to itself used a little memory, eventually it would use all of the memory available. In fact, it could be possible for a recursive function to run out of memory if it attempted to solve too ‘large’ a problem.

PROC FCMP supports recursion. For an excellent example of recursion to do a directory walk see Secosky [2007]. In this paper we will look three traditional examples of problems that can be solved with recursion:

1. calculating a Fibonacci number
2. calculating the factorial of a number

FIBONACCI NUMBER

A Fibonacci number is a member of the sequence of which the first two are 0 and 1, and each remaining number is the sum of the previous two. Some sources omit the initial 0, instead beginning the sequence with two 1s. Leonardo of Pisa, discovered the series in 1202 when he was studying how fast rabbits could breed in ideal circumstances. A Fibonacci series would start like

Index	0	1	2	3	4	5	6	7	8	9	10
Fibonacci Number	0	1	1	2	3	5	8	13	21	34	55

```
   function  fibonacci(n);
①    if n = 0 then return(0);
     else if n in (1, 2)
       then return (1);
②      else return (fibonacci (n-1) + fibonacci (n-2));
   endsub;
```

From this we see:

1. the ‘Base Case’ or end condition. In this example we have two end conditions
   a. when the index is 0
   b. when the index if 1 or 2
2. the recursive call. Since a Fibonacci number is the sum of the previous two numbers we actually have two recursive call.

The function is initially called using the index:

```
data _null_;
      x = fibonacci(10);
      put x=;
run;
x=55
```

FACTORIAL

Another function that is a natural recursive function is a factorial. A factorial of a number is the product of all the numbers from 1 up to and including the number

n	0	1	2	3	4	5	6	7	8	9	10
Factorial	0	1	2	6	24	120	720	5040	40320	362880	3628800

```
   function factorial(val);
     if val > 1
②      then return( factorial(val - 1) * val) ;
①      else return(val);
   endsub;
```

From this we see:

1. the ‘Base Case’ or end condition when val equals 0
2. the recursive call.
3.

The function is called the number for which the factorial is needed:

```
288  data _null_;
289       do i = 0 to 10;
290       x = factorial(i);
291       put i= x=;
292       end;
293  run;
i=0 x=0
i=1 x=1
i=2 x=2
i=3 x=6
i=4 x=24
i=5 x=120
i=6 x=720
i=7 x=5040
i=8 x=40320
i=9 x=362880
i=10 x=3628800
```

From these two examples it can be seen that recursive functions are clean and simple, although initially not always clear or easily understood. It should also be noted that these are common functions for teaching recursion, however they are not good candidates for implementing as recursive functions – as the initial calling argument gets larger the amount of memory and overhead needed kills the performance.

Once again, for an excellent and practical example of using recursion see Secosky [2007].

PROC PROTO

So far we have looked at creating functions with the DATA step like language of PROC FCMP. For locations that may have a library of C language functions, PROC PROTO and PROC FCMP can make them available to the DATA step programmer.

According to the SAS Online help:

> *The PROTO procedure enables you to register, in batch mode, external functions that are written in the C or C++ programming languages. You can use these functions in SAS as well as in C-language structures and types. After the C-language functions are registered in PROC PROTO, they can be called from any SAS function or subroutine that is declared in the FCMP procedure, as well as from any SAS function, subroutine, or method block that is declared in the COMPILE procedure.*

From this we see we first have to run PROC PROTO to let SAS know about these C functions. The DATA step cannot call these functions directly, so we have to also create a “wrapper” function in PROC FCMP. To show how this works I will use an example with a C library file available from the SAS website; this will allow you to test the functionality if you do not have a C compiler or access to C libraries. To download the library see:

http://support.sas.com/kb/32/080.html

The following SAS log will be used to explain the process:

```
   1       /* Specify the storage location of the ranmtnrm.dll file */
   2
1  3    %let root = C:\proto;
   4
   5     /* Define the library in which to store the SAS data set that contains */
   6     /* the functions that are declared using the PROTO procedure.         */
   7
   8    libname lib "&root";
   9
   10      /* Specify a seed value for the random-number generator functions. */
   11
   12   %let seed=1;
   13
   14      /* Use PROC PROTO to define the library, data set, and function  */
   15      /* package in which to store the function prototypes.           */
   16
2  17   proc proto package=lib.Proto_ds.ranmtnrm
   18              label="Random Number Generators"
   19              stdcall;
   20
   21         /* Link SAS to the DLL that contains the compiled C functions. */
   22
3  23      link "&root\ranmtnrm.dll";
   24
   25         /*  Declare the function prototypes. */
   26
4  27      double genrand(void) label="Uniform random number generator";
   28      double ranmtnrm(void) label="Normal random number generator";
   29      void sgenrand(unsigned long seed) label="Initialize seed value";
   30
   31   run;
   NOTE: 'C:\proto\ranmtnrm.dll' loaded from specified path.
   NOTE: Prototypes saved to LIB.PROTO_DS.RANMTNRM.
   NOTE: PROCEDURE PROTO used (Total process time):
         real time           0.37 seconds
         cpu time            0.09 seconds
```

1. The location of the C library file (ranmtnrm.dll)
2. Package is the location where PROC PROTO will save the function information.
   a. Label= is optional
   b. stdcall: for Windows PC platforms only, indicates that all functions in the package are called using the "_stdcall" convention
3. LINK – tells PROC PROTO where to find the load module (.DLL).
4. The C prototype of the functions within the module .

Since the DATA step cannot access these functions directly, we have to create a wrapper in PROC FCMP. A wrapper is essentially a function that simply calls another function.

```
32
33      /* Use the FCMP procedure to wrap the C functions calls so that they */
34      /* can be used in the DATA step. Specify which SAS data set to read
*/
35      /* that contains the function package with the function prototypes,  */
36      /* and specify the function package in which to store the wrapper    */
37      /* functions that are being defined in this FCMP step.               */
38
39   proc fcmp inlib=lib.Proto_ds
40             outlib=lib.FCmp_ds.ranmtnrm;
NOTE: 'C:\proto\ranmtnrm.dll' loaded from specified path.
41
42         /* Define a wrapper function for each of the C functions. */
43
44      subroutine sas_sgenrand(seed);
45         call sgenrand(seed);
46      endsub;
47
48      function sas_genrand();
49         x1=genrand();
50         return(x1);
51      endsub;
52
53      function sas_ranmtnrm();
54         x2=ranmtnrm();
55         return(x2);
56      endsub;
57
58   quit;
NOTE: Function sas_ranmtnrm saved to lib.FCmp_ds.ranmtnrm.
NOTE: Function sas_genrand saved to lib.FCmp_ds.ranmtnrm.
NOTE: Function sas_sgenrand saved to lib.FCmp_ds.ranmtnrm.
NOTE: PROCEDURE FCMP used (Total process time):
      real time           0.64 seconds
      cpu time            0.07 seconds
```

1. The INLIB= for PROC FCMP is the two-level part of the package name from PROC PROTO.
2. Each of the C functions are ‘wrapped’ by a PROC FCMP function with a similar name and calling argument. Since sgenrand does not return a value (void sgenrand) it is declared as a call routine. The others are declared as functions since they do return values.

With the functions declared in PROC FCMP we can now access them.

```
59
60      /* Specify the search path for the function packages that were  */
61      /* created with the PROTO and FCMP procedures so that SAS knows */
62      /* where to find them when the functions are used.              */
63
64   options cmplib=(lib.Proto_ds lib.FCmp_ds);
65
66      /* Call the C functions from within the DATA step and print the results
*/
67      /* to the log window.
*/
68
69   data _null_;
70
71      call sas_sgenrand(&seed);
NOTE: 'C:\proto\ranmtnrm.dll' loaded from specified path.
72      x1 = sas_genrand();
73      call sas_sgenrand(&seed);
74      x2 = sas_ranmtnrm();
75
76      put "--------------------";
77      put "PROTO Function Calls";
78      put "--------------------";
79      put "For seed = &seed";
80      put "GENRAND returns: " x1;
81      put "RANMTNRM returns: " x2;
82      put "--------------------";
83
84   run;
--------------------
PROTO Function Calls
--------------------
For seed = 1
GENRAND returns: 0.8838658645
RANMTNRM returns: 1.0120214427
--------------------
NOTE: DATA statement used (Total process time):
      real time           0.18 seconds
      cpu time            0.07 seconds
```

1. We have to tell the dataset where the PROC FCMP functions are stored.
2. Calling the wrapper functions to call the C functions
3. The results

One feature of PROC PROTO that is interesting is the ability to write and compile C code. Although not an ideal way to develop C code, it can be used to create and access functions written in C. For a complete list of rules, see the PROC PROTO documentation. Let's look at the Fibonacci example again, this time in C.

```
   1
①  2    proc proto package=work.user.cfuncs;
   3
②  4          long fibonacci_c( int nNumber);
③  5          externc fibonacci_c;
   6          long  fibonacci_c(int nNumber)
   7                {
   8                 if (nNumber == 0)
   9                   return (0);
   10                if (nNumber == 1)
   11                  return (1);
   12                return (fibonacci_c(nNumber-1) + fibonacci_c(nNumber-2));
   13               }
   14
③  15      externcend;
   16      run;
   NOTE: Prototypes saved to WORK.USER.CFUNCS.
   NOTE: PROCEDURE PROTO used (Total process time):
         real time           0.03 seconds
         cpu time            0.03 seconds
   17
④  18      proc fcmp outlib=work.user.sasfuncs
   19                inlib=work.user;
   20         function  fibonacci( index);
   21            return(fibonacci_c( index));
   22         endsub;
   23      quit;
   NOTE: Function fibonacci saved to work.user.sasfuncs.
   NOTE: PROCEDURE FCMP used (Total process time):
         real time           0.11 seconds
         cpu time            0.07 seconds
   24   options cmplib=work.user;
   25   data _null_;
⑤  26        x = fibonacci(10);
   27        put x=;
   28   run;

   x=55
```

1. PROC PROTO will write to a WORK package
2. The C prototype for the function fibonacci_c
    a. long fibonacci_c(int nNumber);
3. externc/externcend bracket the C code. Within this we have the C code
4. PROC FCMP declares the functions. It specifies the INLIB= that was in the call to PROC PROTO.
5. The function is called from within a DATA step.

CONCLUSION

SAS DATA step code has often been maligned as being 'spaghetti code' and 'wall paper' code because there was no simple way to encapsulate complex logic into the DATA step. The introduction of user defined functions for DATA step programmers in SAS 9.2 has changed that. This paper has provided a glimpse into the workings of user defined functions,

REFERENCES

SAS Institute Inc. 2008. ***SAS® 9.2 Language Reference: Dictionary.*** Cary, NC: SAS Institute Inc.

Secosky, Jason. "User Written DATA Step Functions" ***Proceedings of the SAS® Global Forum 2007 Conference***

CONTACT INFORMATION

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

Name: Peter Eberhardt
Enterprise: Fernwood Consulting Group Inc.
Address: 288 Laird Dr
City, State ZIP: Toronto, ON, Canada M4G 3X5
Work Phone: (416)429-705
Fax:
E-mail: peter@fernwood.ca
Web: www.fernwood.ca