

Command Line Argument Processing: No Interface Required!

Paul Dalberth
Trilogy Consulting Corporation

Abstract

User specifications often suggest that source code should be user-modifiable but should also shield the user from the specifics of SASTM programming. Code in this class needs to be dynamic but may not justify the complexity of a full-blown user interface such as those provided by SAS/AFTM or macro windows. Programmers can satisfy these almost contradictory specifications by adding command line argument capability to source code. This paper will address the usefulness of command line arguments and some – though not all – OS generic methods for processing them. Finally, the paper will include a SAS macro implementation of command line argument processing that loosely adheres to C language conventions.

The Problem

Help desk personnel notice a common thread in requests for assistance from the end-user community. Alternatively, a programmer realizes that she would like a "quick and dirty" application to provide a specific piece of information about a process or a database. Situations like these call for utility programming; the idea is to make simple pieces of information available with a minimum of effort.

A natural example is a meta-data utility, which could be pointed at a SAS data library and would describe the data sets that reside there by reporting, say, the last modification date and the number of observations the data sets contain. Unless the utility is to be distributed widely, and suppose that this one will not be, a "pretty" interface – read GUI – is not necessary. The effort required to develop a SAS/AF application, then, may not be justified. What is required is a utility called from the command language interface – the '\$' prompt under VMS – with syntax something like this:

```
showme <path to data library>
```

(Throughout this discussion, <path to data library> represents a host specific path name and will be abbreviated <ptd>). We will incorporate more flexibility later, but this is a good opportunity to review some command line argument passing mechanisms.

Some Solutions

There are at least two options for building our 'showme' application. We could execute a libname statement directly, or, we can get a macro variable that resolves to the appropriate library name into the code somehow.

"Macroless" Execution

The first method is simple enough. The INITSTMT option could be used to execute the appropriate libname statement,

like this:

```
sas608/initstmt="libname target <ptd>" showme
```

The rest of the source code comprising showme.sas could be a simple SQL select statement that accesses the dictionary tables:

```
proc sql;
  SELECT memname, memlabel, moddate, nobs
  FROM dictionary.tables
  WHERE libname = 'TARGET' ;
quit;
```

Similarly, we could use the AUTOEXEC option to specify a file containing a libname statement to be executed prior to the rest of the source code:

```
sas608/autoexec=<libname code> showme.sas
```

Macro-based Solutions

There are many possibilities for a macro-based solution. In general, though, we just want to execute a statement that looks like this:

```
libname target "&libn_mac"
```

where macro variable libn_mac resolves to the SAS data library of interest.

We only need to determine how to initialize the macro variable. An obvious solution is to use the macro free methods above to execute a %let statement. For instance,

```
sas608/initstmt="%let libn_mac=<ptd>;"
showme.sas
```

This syntax, though, is a bit convoluted.

Alternatives are provided by the SYSPARM macro and the SYSPARM() function. Both methods depend on specification of the SYSPARM option at SAS invocation:

```
sas608/sysparm="a passed argument" showme.sas
```

The SYSPARM() function assigns the value of the SYSPARM option to a character string. The SYSPARM macro resolves to the value of the SYSPARM option. We could invoke our "macronized" version of showme.sas, then, like this:

```
sas608/sysparm="<path to data>" showme.sas
```

The source code would include these lines:

```
libname target "&sysparm"
```

if we use the SYSPARM macro. If the SYSPARM() function is preferred, then these lines would be included:

```
data _null_;
call symput ('libn_mac', sysparm)
run;

libname target "&libn_mac";

proc sql;
etc.
```

Something That Might Even Be Useful

The INITSTMT and AUTOEXEC options don't really insulate the user from the specifics of SAS code to the extent that we would like. The SYSPARM options are good, but what if we wish to extend our showme utility to be more useful and less contrived? Suppose, for example, that we want to be able to limit our report to only some of the data sets in a specified SAS data library. We also want, moreover, to be able to process an indefinite number of SAS data sets. Invocation syntax could look like this:

```
sas608/sysparm="<ptd> data_set1 data_set2
data_set3 ,,,,"
```

Here's a short macro that might help:

```
%macro get_args;

%let i = 1 ;
%let arg = %scan(&sysparm, &i, %str( ));

%do %while (&arg ^= );
  %global arg&i;
  %let arg&i = &arg;
  %let i=%eval(&i+1);
  %let arg=%scan(&sysparm, &i, %str( ));
%end;

%global num_args;
%let num_args = %eval(&i-1);

%mend get_args;
```

After this macro runs, we will have one macro variable called NUM_ARGS which has a value indicating how many arguments were passed into the code. There will also be one macro variable for each passed argument, named ARG1, ARG2, ARG3, etc. which resolve to the arguments that were passed. For example, if the invocation line looked like this:

```
sas608/sysparm="first second third"
somecode.sas
```

Then NUM_ARGS would resolve to 3 and ARG1 would resolve to "first", ARG2 to "second", and ARG3 to "third." There is some similarity to C language conventions here, except that the name of the program is not counted as one of the command line arguments. I never liked that about the C implementation and I wrote the macro so I get to do what I want.

The GET_ARGS macro could be accessed via an autocall library or a %INCLUDE statement.

For our showme utility, the first argument is the path to the SAS data library which will be connected to a libref this way:

```
libname target "&arg1";
```

A supplemental macro which is also useful in this case is:

```
%macro concat;

%do i = 2 %to &num_args;
  %if &i ^= &num_args %then
    "&arg&i" ,;
  %else
    "&arg&i";
%end;

%mend concat;
```

CONCAT concatenates all of the arguments after the first one into a comma-delimited string for use in a SQL IN clause. Our SQL select statement could, finally, could be:

```
proc sql;
SELECT memname, memlabel, modate, nobbs
FROM dictionary.tables
WHERE libname = 'TARGET'
AND memname IN ( %checkarg );
quit;
```

Now we've built a small utility that can accommodate a variable number of arguments. There are, however, some missing elements. There is, for example, no error checking, a user could specify an invalid path or a series of invalid data set names and SHOWME would happily generate a bunch of error messages. It would be useful, in addition, if the default behavior was to summarize all of the data sets in the library if no data set names were specified on the command line. Of course, enhancements like these might take our little utility out of the "quick and dirty" realm.

Conclusion

The GET_ARGS macro can form the foundation of a library for dealing with command line arguments. SAS software includes so many techniques for dealing with this kind of programming that some level of standardization is useful. Most SAS programmers don't venture into the utility programming area without some accompanying smirks from 3GL programmers, but the journey can be fruitful if you go prepared.

Contact

The author may be contacted at

Glaxo Research Institute
Office phone: (919) 990-5332

Internet: PSD17969@USAV01.GLAXO.COM or
PDALBERTH@DELPHL.COM

SAS and SAS/AF are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. TM indicates USA registration.

Other brand and product names are registered trademarks or trademarks of their respective companies.