

Understanding macros

I like macros. But sometimes they can be hard to understand.

Part of the problem, I think, is that you have to think in two languages at once. You're using the macro language to generate some code in another language – SAS code, or SCL, or maybe display manager commands. If something goes wrong, you don't immediately know whether the problem is with the code itself, or with the macro that was generating it. It doesn't help when there are similar constructs – if/then/else, “do” loops and so on - in both the languages involved. Long strings of ampersands can also be – let's be honest – a bit intimidating.

The idea of this paper is to eliminate these feelings of confusion and fright by forgetting (as far as possible) about SAS and SCL code, and looking only at the macro language. The downside of this is that all the macros here will be guaranteed utterly useless.

1 A morale-boosting example

Let's start with a simple example, demonstrating some basic macro features; demonstrating also how useless a macro can be. This one writes the complete text of “The Twelve Days of Christmas” to the log.

```
%let cardinal=a two three four five six seven eight nine ten eleven twelve;
%let ordinal=first second third fourth fifth sixth seventh eighth ninth tenth eleventh twelfth;

%let g1=partridge in a pear tree; %let g2=turtle doves;      %let g3=French hens;
%let g4=collybirds;          %let g5=gold rings;      %let g6=geese a-laying;
%let g7=swans a-swimming;    %let g8=pipers piping;    %let g9=drummers drumming;
%let g10=ladies dancing;     %let g11=lords a-leaping; %let g12=turkeys stuffing;

%macro giftline(num,day);
  %if &num=1 and &day ne 1 %then %do; and %end;
%scan(&cardinal,&num) &&g&num
  %if &num=1 %then %do;. %end;
%mend giftline;

%macro capital(line);
  %upcase(%substr(&line,1,1))%substr(&line,2)
%mend capital;

%macro verse(num);
  %local i;
  %put On the %scan(&ordinal,&num) day of Christmas my true love sent to me;
  %do i=&num %to 1 %by -1;
    %put %capital(%giftline(&i,&num));
  %end;
  %put;
%mend verse;

%macro daysxmas;
  %do i=1 %to 12;
    %verse(&i)
  %end;
%mend daysxmas;

%daysxmas;
```

Nothing too difficult or controversial here. Notice the complete absence of quotes, either single or double. Macro instructions always generate text strings, so we don't need to tell the macro processor that's what they are.

Notice also the spacing in the %giftline macro. I would have liked to indent the "%scan" line, but if I had done so, the spaces I inserted would also have appeared in the text the macro generated - in the case where num=1, anyway. Similarly with the next line, there is no space before the full stop. If there were one, it would be faithfully reproduced in the output from the program. For the same reason, there are no embedded spaces in our definition of the %capital macro.

Another point that's brought out by the %capital macro: the text it generates doesn't end in a semicolon. It's not a complete line of code; it's just a string of characters. So beware of adding "missing" semicolons to someone else's macro definitions - they probably don't need them. (If, on the other hand, you find double semicolons, leave those alone too.)

Carriage control is not a concept SAS acknowledges very often - try searching for it in the on-line help! There is no easy way to include a line break in a character string. Hence the null "%put" at the end of "%verse", to leave a blank line between verses.

It's important that %daysxmas and %verse (which it calls) use different variables to count their loops. If they both used the same one, %verse would keep resetting it to 0, so %daysxmas would go into an infinite loop, with me constantly reliving the day my true love first sent me a partridge in a pear tree. Both variables are called "i", but the one in %verse is designated "%local", so that it doesn't interfere with the one in the higher-level macro %daysxmas.

Another standard feature here is the two-stage substitution used in %giftline. The expression

&&g&num

is resolved in two passes of the macro processor. On the first pass, the double ampersand is resolved to a single ampersand, and "&num" is resolved - to, let us say, 5. So we now have

&g5

which, on the second pass, resolves to "gold rings".

2 Across the divide: from SCL into SAS

Bear that in mind as we move away from Christmas and on to a slightly trickier area of macro processing: the interface between SCL and SAS. An interesting situation arises when an SCL variable and a macro variable have the same name, and you want to use one or other of them from within a “submit” block. It works like this – this is SCL code:

```
n=1;
submit continue;
  %let n=2;
  %put Value of SCL variable is &n;
  %put Value of macro variable is &&n;
endsubmit;
```

The rule is, if you want to be sure of picking up the macro variable rather than the SCL variable, use a double ampersand. (If there is no SCL variable of the same name, a single ampersand is enough to pick up the macro variable.)

That’s simple enough too. So let’s put that together with the two-stage substitution technique we were looking at before, and see what happens. Suppose we have a set-up like this:

```
num1='uno';
num2='due';
i=1;
submit continue;
  %let num1=one;
  %let num2=two;
  %let i=2;
  %put Macro variable num1 is &&num&i;
  %put Macro variable num1 is also &num&i;
  %put Macro variable num2 is &&&&num&&i;
endsubmit;
```

We can get at the macro variables easy enough, as shown. If we use our standard “double substitution” syntax,

```
&&num&i
```

then this resolves to “one”. However, the way this happens is not the same as when we were in a purely SAS environment. When SCL encounters this expression, it leaves the “&&” unaltered, and resolves only “&i”. So what SCL passes to SAS is

```
&&num1
```

SAS now takes one pass to resolve “&&” to “&”, and a second pass to resolve “&num1” to “one”.

So in fact we had one more ampersand than we needed, as the second %put statement demonstrates. Here we use the syntax

```
&num&i
```

SCL looks first at “&num”, and sees that there is no SCL variable of that name, so it leaves that alone. “&i”, though, it can resolve. So what it passes to SAS is

```
&num1
```

which SAS can do in a single pass.

There was no problem when we used two ampersands at the beginning – and in fact we could have used 4, 8, 16 or any power of two, if we wanted the SAS macro processor to do a bit more work. Any other number of ampersands would have given an error.

We need a few more ampersands if we want to use the macro variable I, rather than the SCL variable. The simplest expression which works is the one used in our third %put:

```
&&&&num&i
```

Here, SCL can resolve nothing, so passes the whole thing to SAS unchanged. SAS, on its first pass, resolves each instance of “&&” to “&”, leaving

```
&&num&i
```

which it takes two more passes to resolve in the way we know so well.

[But... this expression, with 4 leading ampersands, is not the only one that works. 6, 7, 8, 10 or 11 are also OK. In fact, most numbers of ampersands are OK, but there are a few that generate errors – for example 5, 9, 12, 17 and 20. We have not the slightest hesitation in leaving the explanation of this as an exercise for the reader.]

What is harder to do is to get at our SCL character strings – the ones in “Sicilian” – from within our “submit” block. The kind of expression we have been looking at so far is not going to do it for us, however many ampersands we use. SCL will resolve only things that it can resolve immediately i.e. on its first pass. It’s not going to do a second pass, so there’s no way we can make it do a two-stage substitution before it passes text to SAS.

One way of achieving an equivalent result is to have a single variable containing all the words we are interested in, and use %scan. For example:

```
numlist='uno due';  
i=1;  
  
submit continue;  
  options mprint mlogic symbolgen;  
  %let numlist=one two;  
  %let i=2;  
  %put The word for 1 in Sicilian is "%scan(&numlist,&i)";  
  %put The word for 2 in Sicilian is "%scan(&numlist,&&i)";  
  %put The word for 1 in English is "%scan(&&numlist,&i)";  
  %put The word for 2 in English is "%scan(&&numlist,&&i)";  
endsubmit;
```

We use double quotes here, of course. Single quotes would have the program telling us unhelpful things like:

```
The word for 1 in Sicilian is '%scan(uno due,1)'.
```

rather than

```
The word for 1 in Sicilian is "uno".
```

3 Macro quoting

You need to quote text in macro code and macro definitions if it includes things that might otherwise be misinterpreted – things like semicolons, ampersands, percent signs, keywords such as “NOT”, and so on. The trouble is, there are a few too many macro quoting functions for comfort.

Here’s an example that illustrates some of the most common issues:

```
%macro explain(details,n=1,alt=);
  %put Input parameter is "&details".;
  %if &n=1 %then %let type=character; %else %let type=string;
  %put The &type %bquote(%substr(&details,1,&n)) is known as "%substr(&details,%eval(&n+1))".;
  %if &alt ne
  %then
    %put %qsubstr(&alt,1,&n) is sometimes also known as "%substr(&alt,%eval(&n+1))".;
%mend explain;

%explain(%str(;semicolon));
%explain(%str( blank));
%explain(%str(,comma));
%explain(%str(*splat));
%explain(%str(@whirlpool));
%explain(%str(^sharkfin));
%explain(%str(%"rabbit ears (unmatched)));
%explain(%str(%(wax (unmatched)));
%explain(%nrstr(&deckchair),alt=%nrstr(&ampussyand));
%explain(%nrstr(%per cent));
%explain(%str(NOTlogical not),n=3);
```

Here we are deliberately making life difficult for ourselves by passing a single parameter called “details” which contains a special character followed *immediately* by its name.

The word “ampussyand”, by the way, comes from Chambers English Dictionary. Names for some of the other characters are taken from *Tonsil A* of the *Intercal Reference Manual*, which you can find at

[<http://www.muppetlabs.com/~breadbox/intercal-man/home.html>](http://www.muppetlabs.com/~breadbox/intercal-man/home.html)

(Intercal is a programming language of considerable charm, which features instructions such as COME FROM, IGNORE and GIVE UP. It is recommended for people who are truly paranoid about job security, and people who find time hanging heavy on their hands.)

The general rules for macro quoting are

- For quoting at compilation time, use a “str” function – either %STR or %NRSTR.
- For quoting at execution time, use a “quote” function – either %BQUOTE or %NRBQUOTE

%NRSTR and %NRBQUOTE mask out practically everything; they are stronger than %STR and %BQUOTE, which do not affect “%” and “&”.

There's a special rule for characters that normally come in matched pairs – i.e. single quotes, double quotes and parentheses. When unmatched and quoted using %STR or %NRSTR, these characters must be preceded by a “%”. The “%” is stripped off by the macro processor. So in our example above, when we invoke %explain with the parameter “%”rabbit ears (unmatched)”, what the %explain macro actually sees is “”rabbit ears (unmatched)”. The parentheses don't need ‘%’s of their own, because they are matched.

Once a string has been quoted, you don't normally need to quote it again. When a string is quoted, what actually happens behind the scenes is that the string is “tagged” with some special characters fore and aft. These “delta characters” remain there until either the macro processor has finished with the string, or something happens to get rid of them – specifically, they are operated on using %CMPRES, %SCAN, %SUBSTR, %UPCASE, %LOWCASE, %LEFT, %TRIM or %SYSFUNC. If you don't want this to happen, there is a variant of each of these macro functions available - %QCMPRES, %QSCAN, %QSUBSTR, %QUPCASE, %QLOWCASE, %QLEFT, %QTRIM and %QSYSFUNC.

In our example, the first %put in the %explain macro can use &details without a macro quoting function, because this string was quoted (using %STR or %NRSTR) when the macro was invoked.

The second %put has more trouble, because it uses %SUBSTR to extract the first character from the quoted string. This strips off the delta characters, leaving that character as naked and troublesome as it ever was. We therefore need to quote it again. Since this is happening at execution time, we use %BQUOTE. Why is %BQUOTE (rather than %NRBQUOTE) good enough, even when the character could be a “&” or “%”? Because we are dealing with a single character, and “&” and “%” are only really dangerous when followed by something that could be the name of a macro or a macro variable.

The third %put neatly avoids the need for %BQUOTE by using %QSUBSTR rather than %SUBSTR.

There are some other macro quoting functions. %SUPERQ is really useful if you want to quote great swathes of macro code without resolving anything. You won't often need %QUOTE, %NRQUOTE or %UNQUOTE.

4 Statement style macros

You can get rid of the % signs when invoking your macros, so that it looks as though you've added new statements to the SAS language. Here's an example:

```
options implmac nosource;
%macro he / parmbuff stmt;
  %global boy;
  %let boy=%str(&syspbuff);
%mend he;
%macro she / parmbuff stmt;
  %global girl;
  %let girl=%str(&syspbuff);
%mend she;
%macro place / parmbuff stmt;
  %put &boy met &girl in &syspbuff.;
%mend place;
%macro gambit / parmbuff stmt;
  %put He said to her, "&syspbuff".;
%mend gambit;
%macro riposte / parmbuff stmt;
  %put She said to him, "&syspbuff".;
%mend riposte;
%macro result / parmbuff stmt;
  %put The consequences were &syspbuff.;
%mend result;
He      Dennis the Menace;
She     the Lady of Shallott;
Place   a cafe in Rickmansworth;
Gambit  Do you come here often?;
Riposte Only as a last resort;
Result  nothing to frighten the horses;
```

The output looks like this:

```
Dennis the Menace met the Lady of Shallott in a cafe in Rickmansworth.
He said to her, "Do you come here often?".
She said to him, "Only as a last resort".
The consequences were nothing to frighten the horses.
```

Here we have used the STMT option in each macro definition, and turned on the system option IMPLMAC to enable statement-style macros. The only trouble with this is that SAS now has to suspect *every* statement as a possible implicit macro call, so that it has to work a great deal harder.

We have also used the PARMBUFF macro option, which enables us to pass all of the text following the macro invocation as a single parameter. A more useful macro might analyse this using %SCAN and process the individual words with a degree of intelligence.

Note the double full stops in the %place and %result macros. The first one signals the end of the name of a macro variable, but is then thrown away by the macro processor. Since we actually want a full stop, we have to supply a second one.

5 Working with numbers in macros

Since all macro variables are character strings, we need to use special functions if we want to treat them as numbers and do calculations with them. For integer operations, the function is %EVAL. Here's a macro to print out a multiplication table:

```
%macro table(n);  
  %do i = 1 %to 12;  
    %put &i times &n is %eval(&i*&n);  
  %end;  
%mend table;  
  
%table(7);
```

%EVAL is also used implicitly by a number of other macro functions, such as %IF, %DO and %SCAN. A common error message is:

```
ERROR: A character operand was found in the %EVAL function or %IF condition where a  
numeric operand is required
```

It can be disconcerting to get this message when your code doesn't appear to include a %EVAL or %IF anywhere. It may mean what it says – that a value you intended to be numeric has contrived not to be – or it may mean that the value of a macro variable contains an unmatched quote or something similar.

You can do floating point arithmetic as well, using %SYSEVALF. Here's a macro to help fight the tide of metrication:

```
%macro demeter(metres);  
  %let yd_to_m = 0.9144;  
  %let yd_in_ch = 22;  
  %let ft_in_yd = 3;  
  %let in_in_ft = 12;  
  %let chains = %sysevalf(&metres/(&yd_to_m * &yd_in_ch));  
  %let w_chains = %sysfunc(floor(&chains));  
  %let yards = %sysevalf(&yd_in_ch*(&chains-&w_chains));  
  %let w_yards = %substr(&yards,1,%eval(%index(&yards,%str(.))-1));  
  %let feet = %sysevalf(&ft_in_yd*(&yards-&w_yards));  
  %let w_feet = %sysfunc(floor(%sysevalf(&ft_in_yd*(&yards-&w_yards))));  
  %let inches = %sysfunc(floor(%sysevalf(&in_in_ft*(&feet-&w_feet))));  
  %put &metres metres equals &w_chains chains &w_yards yards &w_feet feet &inches inches;  
%mend demeter;  
  
%demeter(100);  
  
100 metres equals 4 chains 21 yards 1 feet 1 inches
```

This one also uses the standard technique of calling SAS functions – in this case FLOOR – from macro code, using %SYSFUNC. For the sake of variety, the “whole yards” figure W_YARDS is arrived at by looking for the decimal point in YARDS and taking everything before it.

6 What's new in V8?

Not a lot, on the macro front. There are a few new system macro variables, and you can now use %put to print coloured messages to the log e.g.

```
%put ERROR: You have broken my poor program.
```

will come out in red.

Under V8.2, you can now delete a macro variable:

```
%let a=apple;  
%symdel a;
```

(It would be nice if there were also a %symtest, to check whether a macro variable existed.)

The one that you may need to know about is %SYSLPUT, known and loved by users of SAS/CONNECT. %SYSLPUT is the macro you execute on the local machine to set the value of a macro variable on the remote machine. It's been around for some time. Under V6.12 it was documented in the Macro Reference Manual, but not shipped, though it was readily available from SAS Institute. Under V8 it is officially regarded as a "new" macro, which means that the change to the syntax does not count as a compatibility issue.

The V6 syntax was:

```
%syslput(remvar, locstr, remote=HOST);
```

The V8 syntax is:

```
%syslput remvar=locstr;
```

This is in some ways a nicer, more logical syntax, but it does mean that your V6 invocations of %SYSLPUT will need to be recoded.

Note that you can't now specify the name of the remote host. If there is more than one possibility, you may need to do an RSUBMIT to the remote host you want, immediately before the %SYSLPUT call.

Note also that the V8 version of %SYSLPUT assumes that the remote host is also running V8 – otherwise it won't work.

Conclusion

Macros can be fun, if you keep a clear head and refuse to be intimidated. The "pros" in this field all say "amper" rather than "ampersand". Cultivate this. It sends out the message that you think macros are a picnic. In time, you'll believe it too.