Paper 3285-2015

Unravelling the Knot of Ampersands

Joe Matise, NORC at the University of Chicago

ABSTRACT

We've all heard it before: "If two ampersands don't work, add a third." But how many of us really know how ampersands work behind the scenes? We show the function of multiple ampersands by going through examples of the common two- and three-ampersand scenarios, and expand to show four, five, six, and even seven ampersands, and explain when they might be (rarely) useful.

INTRODUCTION

The SAS language uses the ampersand to define the beginning of a macro variable, a fact commonly known by SAS programmers beyond the beginning level. This allows for substitution and passing data through the macro facility, which is very powerful in itself.

SAS also provides the ability to recursively resolve macro variable references; this ability extends the power of the macro variable significantly by allowing a macro variable name to be defined by other macro variables. This ability gives rise to several powerful techniques, most commonly the macro array technique, whereby a set of macro variables with a common prefix and numeric suffixes are able to be referenced during a %do loop. It also allows macro variable names to be stored entirely inside of other macro variables, building a macro variable name by parts, and several other techniques.

Exactly how this recursive process works, however, is often a mystery to SAS programmers. Most use cases call for two ampersands, and if two doesn't accomplish the desired result, a third is added in the hope that it solves the problem. A detailed understanding of the process SAS uses to parse multiple ampersands not only can help the programmer decide between two and three ampersands, but can also lead to more complex uses of macro variable recursion.

AMPERSAND RESOLUTION: WHEN TWO BECOME ONE

The SAS Macro processor parses macro variable references according to two simple rules. It begins parsing the first character, and once it hits an ampersand, follows the following process:

1. If it encounters two consecutive ampersands, resolve them to one ampersand, then move to the next character and continue parsing.
2. If it encounters a single ampersand not followed by another, capture until a character is encountered that is not a legal part of a macro variable name (A-Z, 0-9, underscore), then attempt to resolve that token in the symbol tables. If it resolves, replace it with its resolution; if it does not resolve, leave it. If this is the final ampersand in this part of the reference, leave a warning that a reference was not resolved.

Once it has passed through the text to resolve once, it will return to the beginning if there are still any ampersands left outstanding (by rule #1), and repeat the process, following the same two rules, as many times as needed for all ampersands to resolve (or be turned into text).

AMPERSAND RESOLUTION WALK-THROUGH

Let's take an extreme example, and watch what happens behind the scenes, one pass at a time. This example consists of a macro variable reference with seven ampersands, a name, three ampersands, and another name.

```
%let var=team;
%let team=WhiteSox;
%let iter=Series;
%let Series=3;
%let WhiteSox3=2005;
```

PASS	TEXT TO PARSE	GROUPED TEXT	RESOLUTION
1	&&&&&&&var&&&iter	[&&][&&][&&][&var][&&][&iter]	&&&team&Series
2	&&&team&Series	[&&][&team][&Series]	&WhiteSox3
3	&WhiteSox3	[&WhiteSox3]	2005

In the first pass, the following actions are taken:

1. Two ampersands are found, grouped, resolved to one.
2. Two ampersands are found, grouped, resolved to one.
3. Two ampersands are found, grouped, resolved to one.
4. One ampersand is found, followed by the text `var`. &var is resolved in the global symbol table to the text `team`.
5. Two ampersands are found, grouped, resolved to one.
6. One ampersand is found, followed by the text `iter`. &iter is resolved in the global symbol table to the text `Series`.
7. First pass is complete, no more characters to scan. An ampersand was found in the result of the first pass and another pass will occur.

Then the second pass, which now has &&&team&Series to parse, begins.

1. Two ampersands are found, grouped, resolved to one.
2. One ampersand is found, followed by the text `team`. &one is resolved in the global symbol table to the text `WhiteSox`.
3. One ampersand is found, followed by the text `Series`. &Series is resolved in the global symbol table to the text `3`.
4. Second pass is complete, no more characters to scan. An ampersand was found in the result of the second pass and another pass will occur.

Now, the third pass has the text &WhiteSox3 to parse.

1. One ampersand is found, followed by the text `WhiteSox3`. &WhiteSox3 is resolved in the global symbol table to the text `2005`.
2. Third pass is complete. No further unprocessed ampersands exist and no further pass will occur.

Most SAS programmers won't need to store information in this way, so let us see how this works in a more common use case scenario. To start with, we have the two most common uses. We will try both

examples with one, two, and three ampersands, to see which is right for which situation, and what happens when you get it wrong, either with too few or too many.

EXAMPLE ONE: RESOLVING A SUFFIX

In the first example, we have two macro variable references, `&team1` and `&team2`, and we will resolve a reference where we provide the numeric portion through an iteration macro variable reference, `&iter`.

```
%let team1 = White Sox;
%let team2 = Cubs;

%let iter = 1;

%put &team&iter.;   *(1);
```

- WARNING: Apparent symbolic reference TEAM not resolved.
- &team1

```
%put &&team&iter.;  *(2);
```

- White Sox

```
%put &&&team&iter.; *(3);
```

- White Sox

Here in the first line, it attempts to parse `&team` in the same pass as it parses `&iter`. Since `&team` is not a macro variable reference itself, it fails, issues the warning, and prints `&team` to the page along with the resolved `&iter` value. There is no second pass, as a single ampersand in one pass does not cause further parsing in a future pass.

PASS	TEXT TO PARSE	GROUPED TEXT	RESOLUTION
1	&team&iter.	[&team][&iter]	&team1

In the second line, the pair of ampersands resolve to a single ampersand and `team` is left alone for the first pass, while `&iter` resolves as normal. Then in the second pass, `&team1` resolves cleanly to `White Sox`.

PASS	TEXT TO PARSE	GROUPED TEXT	RESOLUTION
1	&&team&iter.	[&&][team][&iter]	&team1
2	&team1	[&team1]	White Sox

In the third line, the first pair of ampersands resolve to a single ampersand, and the third ampersand attempts to resolve with `&team`, but can't, so it waits as well. `&iter` resolves as usual to `1`. Then on the second pass, there are now two ampersands; they resolve to one ampersand, and `team1` waits for another pass. Then in the third pass, there is now `&team1`, which resolves to `White Sox`. Here you see the same effect as in the second example, but it takes a longer path to get there.

PASS	TEXT TO PARSE	GROUPED TEXT	RESOLUTION
1	&&&team&iter.	[&&][&team][&iter]	&&team1
2	&&team1	[&&][team1]	&team1
3	&team1	[&team1]	White Sox

This method is commonly used for simple macro variable arrays, such as in Ron Fehd's paper Array: Construction and Usage of Arrays of Macro Variables.

EXAMPLE TWO: RESOLVING A PREFIX AND A SUFFIX

In the second example, we will now use a new macro variable reference, `&prefix`, which stands for the prefix of the ultimate macro variable reference (`team`).

```
%let prefix = team;
%put &prefix&iter.;   *(1);
```
- team1

```
%put &&prefix&iter.;  *(2);
```
- WARNING: Apparent symbolic reference PREFIX1 not resolved.
- &prefix1

```
%put &&prefix.&iter.;   *(3);
```
- team1

```
%put &&&prefix.&iter.; *(4);
```
- White Sox

In the first example, the processor follows this path:

PASS	TEXT TO PARSE	GROUPED TEXT	RESOLUTION
1	&prefix&iter.	[&prefix][&iter]	team1

The second example uses two ampersands. This example points out a common mistake in macro variable reference specification. The first pass resolves the double ampersand to a single, and then also resolves `&iter` to `1`. The second pass then sees `&prefix1` and, since that is in its entirety a valid macro variable, attempts to parse it – which fails, leading to the warning and the undesired result.

PASS	TEXT TO PARSE	GROUPED TEXT	RESOLUTION
1	&&prefix&iter.	[&&][prefix][&iter]	&prefix1
2	&prefix1	[&prefix1]	~~&~~prefix1

The third line is the same as the second, except we add a period after prefix, to make clear where the prefix macro variable should end. The pair of ampersands resolve to one, `&iter` resolves normally, and in the second pass the now-single ampersand resolves with `prefix.` to `team` like in the first line, leaving us with `team1`. Of course, this is not the desired resolution either.

PASS	TEXT TO PARSE	GROUPED TEXT	RESOLUTION
1	&&prefix.&iter.	[&&][prefix.][&iter]	&prefix.1
2	&prefix.1	[&prefix.][1]	team1

Finally, the fourth line is the desired result. Here, the first two ampersands resolve to a single ampersand; the third ampersand resolves along with `prefix.` to `team`, and then `&iter` resolves as normal; then, in the second pass, the remaining ampersand resolves along with team1 to the value of &team1, in this case White Sox. Here the period is optional, as the ampersand before `&iter` will terminate the reference for `&prefix`, as it resolves during the first pass and not the second.

PASS	TEXT TO PARSE	GROUPED TEXT	RESOLUTION
1	&&&prefix.&iter.	[&&][&prefix.][&iter]	&team1
2	&team1	[&team1]	White Sox

Note here, as compared to the first example's three ampersand case, you actually only have two passes: that's because in the first example you had a pass that didn't do anything. Here the third ampersand is consumed by turning `&prefix` into `team`, and so no extra pass occurs.

THE AMPERSANDS COME WALKING TWO BY TWO

Of course, more than three ampersands can be used in some situations. However, not all combinations of ampersands are equal.

Even numbers of ampersands have one specific use: delaying resolution one or more passes. This is because even numbers only do one thing: they delay resolution a round (or more), while something else is resolving. Each pass, an even number of ampersands is parsed to one-half that number in the next pass, and eventually must parse down to an odd number in order to do anything (including 1).

This can be very helpful if the second half of your macro name needs to be resolved in several steps. In that case, you need the two to the power of (the number of steps) ampersands. Two ampersands delays one round, four ampersands delays two rounds, eight delay three rounds, etc.

```
%let one=two;
%let two=three;
%let three=four;
%let startfour=done;
%put &&&&&&&&start&&&&&&&&one;
```

That can get to be a lot of ampersands quickly!

Eight ampersands delayed resolution of start for three rounds, but seven ampersands caused three rounds of resolution for the right half of the macro variable. If we wanted to show four rounds of

resolution, we would need 16 ampersands before `start`, and 15 ampersands before `one` (and some new macro variables).

Other quantities of ampersands in between the powers of two (2^n or 2^n-1) serve the purpose of being delayed versions of the 2^n-1 quantities. Six ampersands for example is the same as three ampersands plus one delayed round of resolution before the first round; five ampersands is the same as three ampersands plus one delayed round of resolution prior to the second round. This delay can be helpful if the second parameter needs to resolve at a different time than the first. $2*(2^n-1)$ (six, for example) can be very helpful in particular because it enables an iterator suffix to be used.

Here is a table of different numbers of ampersands and how they are useful.

Number of Ampersands	Common Example	Use
Two	&&var&i	Allow simple resolution with a suffix.
Three	&&&var	Allow double resolution (resolves a macro variable, then resolves the result as a macro variable)
Three	&&&var.&i	Resolves a prefix and a suffix into a single variable, then resolves that.
Four	&&&&var&&&i	Like the two-ampersand solution, but with a double resolving suffix.
Six	&&&&&&var&i	Combines the first two examples into one: resolves a suffix, then double resolves the result after the suffix is added.
Seven	&&&&&&&var	Triple resolves a macro variable.
Seven	&&&&&&&var.&i	Resolves a prefix and a suffix, then resolves that twice.

For more information on the patterns in macro variable resolution with different numbers of ampersands, read *The Role of Consecutive Ampersands in Macro Variable Resolution and the Mathematical Patterns that Follow* (Michael Molter, SUGI 29), which goes into more detail about the numeric patterns seen in macro variable resolution.

SOLUTIONS USING MANY AMPERSANDS

Most of us will never use more than three ampersands in our daily work. However, for those who are interested in practical applications of larger numbers of ampersands, here are a few ideas.

STORING PARAMETERS AND NAMES OF PARAMETERS

In The Six Ampersand Solution, John Gerlach uses a six ampersand prefix to store parameters used in a study. He creates multiple variables in an array, where `&&var&i` stores the name of the parameter and `&&&&&&var&i` then resolves to the value of the parameter. The six ampersands are needed in order to give `&i` a chance to resolve before trying to resolve the entire variable, allowing then `&&&var1` to fully resolve twice.

STORING PARAMETERS AND NAMES FOR MULTIPLE STUDIES

Seven ampersands can be used in a similar fashion, if you stored the parameters for several different studies; so instead of `&&&&&&var&i` resolving to `&&&var1` resolving to `&DXCODE` to `4019`, you then had `&&&&&&&study&i` which resolved to one of several studies (say, studyA, studyB, studyC) which each had something like `studyA1` => `DXCODE` => `4019` and `StudyB1` => `DXCODE` => `4017`, for example (and `StudyC1` => `DZCODE` => `12345`).

Both of these examples can be used to efficiently store data in macro variables in an efficient way. While this is normally something to be avoided (as the data step does this better), in some instances this can be

helpful (such as when the data needs to be used to reference variables in a datastep) or for programmers who are more comfortable in the macro language.

MACRO VARIABLE LINKED LISTS

A linked list of macro variables is possible, using 2^n-1 ampersands to search for the name n steps down the list, and 2^n ampersands to resolve the value associated with it. This wouldn't be terribly practical to use directly , but a macro that searched down a linked list and created the appropriate reference would be easy to use and write. This could be used much in the same fashion as linked lists are used in object oriented programming applications.

```
%let a=b; %let val_a = 123;
%let b=c; %let val_b = 234;
%let c=d; %let val_c = 345;
%let d=.; %let val_d = 456;


%put &&&&&&&a &&&&&&&&val_&&&&&&&a;
```

➢ d 456

You could then define macros that give you the next value or item in the list after the current item:

```
%macro next_val(current_item=);
  &&&&&&val_&&&current_item.
%mend next_val;

%macro next_item(current_item=);
  &&&current_item.
%mend next_item;

%next_val(current_item=&cur);
```

And even define macros to find an arbitrary value.

```
%macro find_val(current_item=,item_number=);
  %let twon = %eval(2**&item_number.);
  %let amps = %nrstr(%sysfunc(repeat(&,%eval(&twon.-2))));
  %let val_name = %unquote(val_&amps.&current_item.);
  &&&val_name.;
%mend find_val;
```

Here we use some macro quoting/unquoting, because normally ampersands resolve down to one ampersand instantly, so we have to delay that process with `%nrstr` and then allow it with `%unquote`. We also have to do it in two steps to make it resolve correctly.

CONCLUSION

Ampersand resolution can be confusing, but following the macro parser's passes step by step can help a SAS programmer to better understand exactly what is happening when their macro variables resolve. Going beyond three, to six, seven, or more can also open up several new options for SAS programming once macro variable parsing is understood.

REFERENCES

Gerlach, John. "The Six Ampersand Solution." Presented at NESUG 1997. Available at http://www.lexjansen.com/nesug/nesug97/posters/gerlach.pdf .

Fehd, Ronald. "Array: Construction and Usage of Arrays in Macro Variables". Presented at SUGI 29 (May 9-12, 2004). Available at http://www2.sas.com/proceedings/sugi29/070-29.pdf .

Molter, Michael. "The Role of Consecutive Ampersands in Macro Variable Resolution and the Mathematical Patterns that Follow". Presented at SUGI 29 (May 9-12, 2004). Available at http://www2.sas.com/proceedings/sugi29/063-29.pdf .

ACKNOWLEDGMENTS

Thanks to Jeff Vose and the Data Delivery team at NORC at the University of Chicago for supporting me during the writing process. Special thanks to Megan Moldenhauer (of NORC) and Marcus Maher (of Ipsos) for proofreading and design suggestions for the paper and associated e-Poster. Also, thanks to Nina Werner whose paper, along with Marcus' questions, at MWSUG 2014 ("Understanding Double Ampersand SAS Macro Variables") inspired this topic.

RECOMMENDED READING

- *"The Role of Consecutive Ampersands in Macro Variable Resolution and the Mathematical Patterns that Follow", Michael Molter, SUGI 29.*
- *Referencing Macro Variables Indirectly. SAS Macro Language Reference. http://support.sas.com/documentation/cdl/en/mcrolref/61885/HTML/default/viewer.htm#a001071915.htm*

CONTACT INFORMATION <HEADING 1>

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

Joe Matise
NORC at the University of Chicago
matise.joe@gmail.com
Or online on SAS-L Listserv or in a StackOverflow question.