ABSTRACT

As the quote above shows, classical programmers recognized that the appearance of a variable's value does not directly reflect its stored value. That is, what you see isn't necessarily what you get. For SAS programmers, quality control with numeric data starts with the recognition that all display, whether on your screen or on a printout, is formatted display. It may be a format you have specified (either in the program or in a stored catalog), or it may be a system default. But what you see is an image.

In this paper I'm going to define quality control in a fairly simple, straightforward, way: The goal of any SAS program is to output all and only the intended output. Quality Control [QC] is the process of making sure that the output of your program is, in fact, all and only the intended output. There's a concept used in scientific theory construction that is useful here. A scientific theory should be falsifiable. That is, it should be so constructed that evidence can be used to determine if it is correct or not. The same is true for programs. We want to construct them so that, if they generate incorrect output, we know about it.

I also hope to make it clear that QC isn't just about checking for errors or omissions. It's also about making informed programming decisions. I will illustrate this later in the paper when the advantages and disadvantages of the LENGTH statement and the COMPRESS=BINARY option are contrasted.

ROADMAP

Understanding how SAS stores and displays numeric data is essential for both accurate computations and effective, useful, reports and tables. I will show the importance of understanding, in some detail, the specific properties of SAS numeric data. But I will begin this paper with a discussion of some general points concerning programming for effective quality control. After this general discussion, I will turn to specific properties of SAS numeric data. For example, what does it mean within the SAS system if a DATA step contains the statement in (1)?

(1) LENGTH NVAR1 3 ;

Effective quality control with SAS numeric data requires an understanding of the basics of how SAS stores number and represents them when performing computations in the DATA step. I will discuss (i) how SAS stores numbers and look at some decisions that SAS programmers must make regarding program and storage efficiency. We will look at two common ways for saving disk space with SAS numeric data: the LENGTH statement and the COMPRESS=BINARY option.

What I am not going to do in this paper is attempt to exhaustively treat all the issues and questions that may occur when programming with numeric data. Rather, the strategy will be to highlight a few important issues that illustrate the need to understand what is happening 'behind the scenes' with respect to storage, computation, and how they relate to good QC.

QUALITY CONTROL [QC]

I'm going to distinguish between 3 types of quality control [QC].

(2) a. INPUT QC
   b. PROGRAM QC
   c. OUTPUT QC

Input QC is the process of learning about the input data to your program. It's important to distinguish between properties that the data is supposed to have, and the properties it actually has. It is also important for the SAS programmer to have a good, general, sense of what the data means and how it will be used. This knowledge allows the programmer to integrate more-intelligent QC measures into the programs than would be possible by blindly following specs.

Program QC refers to writing programs so that quality-control information is available both from intermediate stages of the program as well as in the final output. For example, merges are crucial areas of a program requiring QC.

Output QC is what most programmers are thinking about when QC is mentioned. That is, Output QC is looking at the output data sets to make sure that they have the intended properties. There are various aspects of this. There's the basic level of making sure that $2+2=4$. There are also relationships between variables that are determined by external factors, e.g. IF MONTH = 9 THEN (1 <= DAY <= 30). There are variable values and relationships to check that are specific to the particular data set being input or generated, e.g. number of children in the family was recorded only if they were living at home.

I will discuss a number of specific SAS programming examples, but the goal of this paper is to emphasize as much as possible the general principles which lead to more-effective quality control with SAS numeric data. Throughout I will focus on a common situation for a SAS programmer: you have an input SAS data set (or, sets) and you have specs to modify this data set in some way and output a new data set. Of course there are numerous other situations (e.g. reading in flat files, outputting to Excel spreadsheets, etc.) that require QC. I hope that I am able to communicate my basic approach in such a way that the programmer can readily apply it to situations not covered here.

One thread that will run through this discussion is what I call content-based programming. By that I mean that it is well worth the time for a SAS programmer to learn as much as...
possible about the project the programs are a part of, the type of people who will be working with the data, and the general goals which motivate the particular specs. A lot of time and money can be saved if the programmer is in a position to catch content-based problems before they are delivered to the client. For example, I work a lot with health-care survey data and it is fairly easy to figure out that males should not have valid responses to pregnancy questions. It is less obvious that some questions (e.g. "Have you fully recovered?") are only asked of people with certain types of medical conditions.

An added benefit to content-based programming is that it makes the work more interesting. The more interested I am, the more alert and engaged I am with the data and the programs. So it's efficient in the long run for the programmer to take some time before ever writing a line of code to find out as much as possible about the data, and to plan ahead with QC in mind.

**INPUT QC**

A lot of old clichés are exactly right when it comes to Input QC ("A stitch in time saves nine", "Penny wise but pound foolish", etc.). I'll add two more:

1. a. Don’t overlook the obvious.
   b. Verify what you're told.

The word verify is going to come up quite often in this paper because a lot of basic QC comes down to verifying that what you think is true, or have been told is true, actually is true. One of the great enemies of good QC is time pressure, and this can start right at the beginning by not taking the time to take a good look at the input data.

SAS provides a wonderful tool for initially examining input data sets: the CONTENTS procedure. It is absolutely essential to run a PROC CONTENTS to see, at least, the following:

1. a. Number of Variables
   b. Number of Observations
   c. Sorted?
   d. Engine (V8?)
   e. Variable Name
   f. Variable Type (Numeric/Character?)
   g. Variable Length
   h. Variable Labels
   i. Variable Formats

Initially it's important to know if this general information agrees with the documentation you have about the file. If it doesn't then you need to straighten this out before going any further (perhaps you have the wrong file, one that's similar to the right file).

If the general information (e.g. number of variables and observations) appears to be as it should be, you can look at specifics such as variable names and data types. Whether a variable is numeric or character is not always predictable from its name. An ID variable that consists only of numbers might be either numeric or character. A flag variable whose values are only 0 or 1 might be numeric or character. It's obviously important when you start writing code to know what type of variables you have.

When working with numeric data it's important to know the variables' length. The default is 8 bytes, but this is often reduced to save disk space. If you are going to be modifying the values of numeric data, it's especially important to know variable length. Here's a simple program for taking a first look.

```
(5) PROC CONTENTS DATA= DSN_IN OUT= CHECK1
   (KEEP= NAME TYPE LENGTH LABEL);
   RUN;
   PROC FREQ DATA= CHECK1;
   TABLES LENGTH;
   WHERE TYPE = 1;
   RUN;
```

The output data set CHECK1 will have 4 variables: NAME (= the names of the variables on the DSN_IN data set); TYPE (1 if numeric, 2 if character); LENGTH and LABEL.

The PROC FREQ outputs a table showing the frequency of numeric variables (WHERE TYPE = 1) by LENGTH. There are two reasons to care about numeric length: (i) if it is less than 8 bytes and you will be increasing any values, you need to make sure that the length specification is sufficient for the updated values; (ii) if you are concerned about saving disk space, you may want to consider reducing numeric length for some variables. For more-detailed discussion of this topic, see my paper on SAS numeric data in these Proceedings.

It's important to become familiar with variable labels because they are often important sources of information about the intended content and use of the variables. Sometimes labels offer abbreviated explanations of particular values ("1 IF NUMERIC, 2 IF CHARACTER"). If your editing adds to, or changes, these values, you will need to modify the label as well.

Even the best specs aren't perfect and the programmer who has taken (or, been given) the time to understand the context of the data and the specs, can often spot additional modifications that may be needed. Often these take the form, "If you change X then you should also change Y." This is another potential time saver—and one that is often appreciated by the client writing the specs.

Formats are also important for a number of reasons. If numeric variables are associated with formats, this may affect the operation of particular PROCS. It's good to verify that associated formats should be kept, and to communicate any potential implications.

If formats indicate value ranges, it's good to verify that these ranges are meaningful for the data you have. For example, if the format values are [-1 = '-1'], [0 = '0'] and [ 1-100 = '1-100'] but the actual range of positive values on the input or output data set is 40-60, you should check whether or not this format range should be changed. For this type of case, if you only ran a formatted FREQ, you would never see any problem because the range of actual values is within the format range. The output would only show a certain number of records with each of the format values.
Another important piece of information to know about an input data set is which variable(s) uniquely identify observations, i.e. the key variable (often, but not always, this is the SORTED BY variable). If you are told that PERSON_ID uniquely identifies rows, then you should run the following check:

(6) PROC SORT DATA= DSN_IN OUT= CHECK2
    NODUPKEY;
    BY PERSON_ID;
RUN;

If PERSON_ID really is the key variable, then the LOG will show that 0 observations have been deleted.

Don't overlook the obvious takes many forms. For example if you have MONTH and DAY variables, you could include conditionals such as (7) as part of a DATA step.

(7) IF (MONTH = 'SEPT') AND
    NOT (1 <= DAY <= 30)
THEN PUT ID= MONTH= DAY= ;

You can also use format ranges to test for valid and invalid values for particular variables. Part of the importance of knowing what properties the data SHOULD have concerns how various types of non-response values are coded. For example, what is the value for a MONTH variable if this information is unknown. For many surveys, responses such as Don't Know or refusals are coded as specific minus values (-8, -7, etc.) or out-of-range values (e.g. 99). Knowing what these are is essential before any code is written that will edit or re-code the input data. See Ron Cody's book on data cleaning for many useful approaches to checking the values of input variables.

Here's an example of the verify what you are told theme: suppose the documentation for a flag variable DENTFLAG on a large person-level file says that its value = 1 if the person has dental expenditures greater than zero. Otherwise the value is 0. You could print the first 50 OBS and see if this pattern holds. Or you could use a DATA step conditional to test all the records in the file and put a LOG message if any problems appear. The simple code in (8) does the trick.

(8) IF (DENTFLAG=1 AND DENTEXP <= 0)
    OR
    (DENTFLAG=0 AND DENTEXP > 0)
THEN DO;
    PUT '*** DENTFLAG/EXP ERROR ***' ;
    PUT ID= DENTFLAG= DENTEXP= ;
    END;

This will print a message to the LOG for each record where the expected pattern doesn't hold. If you suspect you have a lot of bad data and don't want to fill up your LOG, you can initially set a flag variable (e.g. DENT_ERR) that is assigned a value of 1 if the conditional evaluates as true. Then you can run a FREQ on DENT_ERR or use it to create an output data set.

This type of approach is much better than dumping 50 or so observations to 'check' on variable values or the relationships between variables. I will discuss this further in the section on Output QC, but it is relevant here as well. Printing a small number of records is an amazingly common practice among SAS programmers. It is often requested by clients. It is useful for getting a kind of 'at-a-glance' initial sense of the data. It may even make you aware of a problem that would never have occurred to you to check on. But it is never a sufficient QC check for the simple reason that, if you print 50 records, there may be a problem with records 51 through 100,000.

**PROGRAM QC**

The point of Program QC is to write DATA steps (e.g. merges) and include PROCs in such a way that you are outputting information that is necessary for evaluating the correctness of the program and its output.

Here's an example where Input and Program QC will help you write the correct code without trial and error. Suppose you have the spec in (9).

(9) Create a variable AGEcat. If a person is under 18 then AGEcat = CHILD. If the person is 18 or older, then AGEcat = ADULT.

Without having looked at the data, it might be tempting to code this as:

(10) IF AGE < 18
THEN AGEcat = 'CHILD';
ELSE AGEcat = 'ADULT';

But as part of Input QC, you noticed that there were both missing values and zero values. The conditional in (13) will code both of these values as 'CHILD.' This may be want is intended for 0, but certainly not for missing values. Another way to catch this type of problem is to always include a QC crosstab after re-coding. This crosstab should have the basic form BEFORE*AFTER. Of course the trick to such a crosstab is how to format the BEFORE variable. For something like AGE (with perhaps 100 different values) it wouldn't be too annoying to leave it unformatted. But that isn't often a realistic option.

Here you could run a PROC to get the MIN and MAX values (perhaps using a WHERE AGE GT 0 clause) and then format AGE as MISSING, 0, 1-17 and 18-95 (assuming that 95 was the maximum age on the data set). Then you could modify (10) as in (11). Of course, once you know to ask, it may be a simple check to find out if 0 values should be coded as 'CHILD.' If so then (11) can be simplified to include 0 in the CHILD range.

(11) IF (1 <= AGE <= 17)
    THEN AGEcat = 'CHILD';
ELSE IF (18 <= AGE <= 95)
    THEN AGEcat = 'ADULT';
ELSE IF AGE = 0
    THEN AGEcat = 'ZERO VALUE';
ELSE IF AGE = .
    THEN AGEcat = 'MISSING';
ELSE AGEcat = '??????';
The final clause here points to another part of good programming: conditionals should always exhaust the logical possibilities. The final ELSE gives you control over the unexpected. You will have a clear indicator (“???”) that there are AGE values you haven't accounted for.

Here's an example that illustrates how Program QC with numeric data requires not only a knowledge of the specific properties of SAS numeric data, but also the details of how SAS works. We'll discuss some of the details of SAS numeric data later in the paper. For now assume that you have one data set (DSN2001) with a numeric variable for annual 2001 out-of-pocket dental expenditures (DENTOOP). To save disk space, you'd like to use the minimum LENGTH specification you can. You're running SAS on a PC and your documentation shows that 8,192 is the largest integer that can be precisely represented with 3 bytes. All values for DENTOOP are integers and, luckily enough, the largest value is 8,192. So you store DENTOOP with LENGTH 3.

Now let's also assume that you have a second data set with 2000 data (DSN2002) that's identical to the first, except that the largest value for DENTOOP is 10,125. So here you decide that the LENGTH specification for DENTOOP should be 4. Everything is fine, no problems occur with either data set until you are asked to concatenate the files into one. You use the DATA step in (12).

(12) DATA DENT0102;
    SET DSN2001 DSN2002;
    RUN;

What is the LENGTH of DENTOOP on the new data set DENT0102? I'm sure you know that it's 3 because the LENGTH of DENTOOP on DSN2001 (the first encountered data set) determines the LENGTH on the output data set. But this is going to cause problems with values such as 8,193 that were on DSN2002. They can no longer be accurately represented because more than 3 bytes are needed.

Of course if you had done everything right (such as used an explicit LENGTH statement before the SET statement) you could have avoided this problem. But what if it was another programmer a year later doing the concatenation? Even then the problem could have been avoided if that programmer used good Input QC on a PC and your documentation shows that 8,192 is the largest integer that can be precisely represented with 3 bytes. All values for DENTOOP are integers and, luckily enough, the largest value is 8,192. So you store DENTOOP with LENGTH 3.

Unfortunately this wouldn't work because, as we'll see later in the paper, the effect of the LENGTH statement is only on the stored value of DENTOOP, so we'd only be able to actually test the conditional if we, in a subsequent DATA step, read in DENT0102.

We'll return to some other aspects of this type of situation when we turn to examining SAS numeric data more closely. But I hope that these examples illustrate the need, not only for applying general principles of QC, but also knowing specific properties of how SAS works and how SAS stores numeric data. One other thing that this last example shows is the importance for preparing for non-ideal situations. What if that programmer working on your data a year from now isn't perfect? How can you decrease the chance of a problem down the road? We'll consider this and some more examples later on.

The general structure, somewhat simplified, of many SAS programs is to input a data set, create a series of temporary data sets, and output a permanent data set or report. We can illustrate this in (13).

(13) I → T₁ ... → Tₙ → O

It's a good rule of thumb to generate QC output (either as LOG messages or PROC output) for each temporary data set the program generates. That way, if you have to work your way back from some problematic output to its source, you have a series of snapshots to guide the search.

OUTPUT QC

Similar to Program QC, the basic principle of Output QC is to have specific expectations and test for them. Looked at in this way the output of your program (whether it's in the form of data sets or reports) is simply the last in a series of program outputs.

In addition to including QC LOG messages and PROC output for the temporary data sets, it is important to repeat any relevant QC measures with the final output. Not all will be possible because the output file might be different in character (e.g. a person-level rather than a family-level file) from some of the temporary data sets, but what is possible should be done. After all, it's the final output that's important.

One all-too-common way to check the final output is to print 50 OBS or so of a large file to check that everything looks the way it should. A lot of programmers do this, and a lot of clients request it, because it gives a concrete sense of how the variable values look. But it is not a sufficient QC measure. If there are patterns that should be true for all (or a specific subset of) records on a data set, then use a PROC to test for these patterns. You can also use PROC COMPARE to test to make sure that the only changes between an input and output data set are the intended changes. For example, suppose that the output data set (OUTDSN) is supposed to be identical to the input data set except for the addition of NEWVAR1. The following code will test for this.

(14)

    PROC COMPARE
        BASE= OUTDSN (DROP=NEWVAR1)
        COMPARE= INDSN;
    RUN;

In many ways OUTPUT QC mirrors INPUT QC in that you want to have a very detailed picture of both input and output data sets. In the INPUT QC section I stressed the essential role of PROC CONTENTS. This is equally important with
OUTPUT QC. The list of data set properties in (3), repeated here as (15) should form a minimal output checklist.

(15) a. Number of Variables
    b. Number of Observations
    c. Sorted?
    d. Engine (V8?)
    e. Variable Name
    f. Variable Type (Numeric/Character?)
    g. Variable Length
    h. Variable Labels
    i. Variable Formats

Are the number of variables and observations correct? Should the data set be sorted? Do all the variables have labels and/or formats? I would also add to this list a question about compression. SAS V8 offers good options for compression that are worth considering (COMPRESS= CHAR | BINARY). It is also useful to use the POSITION or VARNUM option to clearly see the position of the variables on the data set. This is often a good way to pick up any stray variables (e.g. I or X from a DO loop) that may have been accidentally left on the data set. These variables will tend to be positioned near the end.

The basic principle of QC is to know your data. Good programming requires that you know your input data, that you know the relevant properties of intermediate data sets, and that you know the details of your permanent output. An important element of this is to distinguish between the values that variables actually have, and the values that they should have. As much as possible this should be done for all records on all the files you are working with.

SAS programmers are fortunate to have a variety of data-analysis tools at their disposal. The minimal added cost of including QC PROCs or conditional LOG messages in SAS programs is usually more than offset by the benefit of preventing the generation of unintended output.

In the next section we turn to specific properties of SAS numeric data as a way of illustrating the use of quality control principles in particular situations.

SAS NUMERIC DATA

In this section we will look at SAS numeric data in some detail, with the goal being to highlight the interplay between general principles of quality control and specific properties of how SAS stores and computes with numeric data.

Different systems use different methods to store numbers. SAS uses floating point representation and, by default, stores numeric values using eight bytes. This paper will begin with a brief, practical, sketch of floating point representation and some of its underlying concepts. The goal of this sketch is not to exhaustively explain how to translate base 10 numbers into floating point representation, but rather to give you a more-concrete sense of how the SAS system interprets a statement such as:

(16) LENGTH NVAR1 3 ;

One reason for not delving into the minutiae of floating point representation is that it is rarely of practical value in the day-to-day work of SAS programming and making decisions concerning the LENGTH of numeric data. It is better to have a small set of guiding principles for making decisions in a timely manner. Of course it is always good to know where to go to get detailed information if you need it, and the REFERENCES section at the end of this paper contains two useful references from SAS Institute (they include additional references).

HOW SAS STORES NUMBERS

There's quite a bit of detail in this section. In general, there would seem to be little reason to pay attention to this level of detail in making day-to-day programming decisions. So why go to the trouble of devoting time to it? For two reasons: (i) so that you are in a position to evaluate the general claims made here in terms of your particular situation; (ii) you may, at some point, encounter a programming problem that requires a detailed understanding of how SAS stores numbers—and how this differs from how SAS displays numbers.

FLOATING POINT REPRESENTATION

Most papers on floating-point representation and numeric precision only tell part of the story. The part of the story I'm going to tell focuses on giving you enough information to understand, or, at least give you a jumpstart in understanding, more-detailed discussions of these topics. I also hope it allows you to evaluate the guidelines that I give at the end of this section. As I said at the beginning of the paper, QC is about decisions—and knowing a few important points about how SAS stores numbers will allow you to make informed decisions.

The basic unit of storage is the bit (binary digit). As the term binary suggests, there are two possible values: 0 and 1. A sequence of 8 bits is called a byte.

SAS stores numeric data using 64 bits (eight bytes). The eight bytes are divided among 3 different types of information: the sign, the exponent, and the mantissa. An important fourth type of information is the base, i.e. the number raised to a power. These are terms for the parts of a number in scientific notation. Floating point representation is just one form of scientific notation. In this system, each number is represented as a value (the mantissa) between 0 and 1 raised to a power of 2 (in a binary, base 2, system). The term decimal in 'decimal point' assumes a base 10 system, so the term 'radix point' is often used when the base is not 10. The radix point for a number is moved (i.e. it floats in the picturesque speech of computer science) to the left until the number is a fraction between 0 and 1.

If we use the more-familiar base 10 system, the number 234 would be represented as \(234 \times 10^1\). Placing the decimal point to the left of the number is called 'normalizing the value.' This normalization yields a value between 0 and 1. In addition to the mantissa (.234) and the exponent (3), a full representation of 234 would include the sign (negative or positive).

Here's the byte layout for a 64-bit number in the IEEE system used by Windows (where S = sign; E = exponent; and M = mantissa):
That is, there's 1 bit for the sign, 11 bits for the exponent, and 52 bits for the mantissa. The number of exponent bits determines the magnitude of the numbers that can be represented. The number of mantissa bits determine the precision. For example, an IBM mainframe system will use 56 bits for the mantissa, allowing for greater precision than you get on a PC. In the following discussion I will focus on the mantissa since that is the part of the representation affected by the LENGTH statement.

For 52-bit systems, the functional equivalent of 53 bits is achieved by assuming an 'implied' bit. That is, since the only possible non-zero digit is 1, we can just assume an initial mantissa bit with a value of 1. We'll see that the existence of this implied bit solves an apparent puzzle when we look at numbers in binary where all the mantissa bits have values of 0.

When you use a LENGTH statement such as (16), you are telling the SAS system to only use the first 3 bytes to store the number, i.e. bytes 1-3 in (17). You save 5 bytes per file record this way, but you are potentially losing precision because you are sacrificing 40 mantissa bits. Note that you cannot specify length in terms of bits, or specify which bytes to use. If LENGTH is 3 then you only have the first 3 bytes in (17).

INTEGERS

Below is a table showing (what is often referred to as) the largest integer that can be represented accurately for a given LENGTH specification. The numbers in the table will be different for IBM and VAX computers (consult the SAS Companion for your operating system).

<table>
<thead>
<tr>
<th>LENGTH IN BYTES</th>
<th>LARGEST INTEGER (PC/UNIX)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>NOT ALLOWED</td>
</tr>
<tr>
<td>3</td>
<td>8,192</td>
</tr>
<tr>
<td>4</td>
<td>2,097,152</td>
</tr>
<tr>
<td>5</td>
<td>536,870,912</td>
</tr>
<tr>
<td>6</td>
<td>137,438,953,472</td>
</tr>
<tr>
<td>7</td>
<td>35,184,372,088,832</td>
</tr>
<tr>
<td>8</td>
<td>9,007,199,254,740,992</td>
</tr>
</tbody>
</table>

Now let's look at the 64-bit representation of 8,192 and 8,193 to see why 8,192 is the 'largest integer' that can be accurately represented. You can create 64-bit output by using the BINARY64. format (The spaces between bytes were added later).

(18) The 64-bit representation of 8,192:

\[
01000000\ 11000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000
\]

The first thing to notice is that the difference between 8,192 and 8,193 is in byte 4. If you are limited to the first 3 bytes then this difference is eliminated and, as far as the computer is concerned, the numbers are identical (because bytes 1-3 are identical).

We can show this with the SAS program in (20).

(20) DATA ONE;
    LENGTH VAR1 VAR2 3 ;
    VAR1 = 8192 ;
    VAR2 = 8193 ;
    RUN;

DATA TWO;
    SET ONE;
    PUT VAR1=22.16 ;
    PUT VAR1=BINARY64. ;
    PUT VAR2=22.16 ;
    PUT VAR2=BINARY64. ;
    RUN;

Here's the LOG output you'd get:

(21) VAR1= 8192.0000000000000000
VAR1= 01000000 11000000
VAR2= 8192.0000000000000000
VAR2= 01000000 11000000

There are a couple of things going on here we need to unpack. First, VAR2 was initially assigned a value of 8,193, but in data set TWO it's 8,192. The reason is that the LENGTH specification of 3 has removed the distinguishing information in byte 4. When SAS stored this variable it only used three bytes.

Second, and this is a point we'll come back to later, in the DATA step (i.e. the program data vector [PDV]) SAS uses the full 8-byte representation of numbers (Note that, for character variables, the LENGTH statement applies both to the PDV and the output data set). If a variable was stored as LENGTH 3, then bytes 4 through 8 are 'filled in' with zeros. This is what has happened with VAR1 and VAR2. When data set ONE is stored, VAR1 and VAR2 are stored with 3 bytes. When they are read for the DATA step that creates TWO, they are expanded to eight bytes, with the last 5 bytes all zeros.

For VAR1 (8,192) this does not result in any loss of precision because the last 5 bytes are all zeros in the original 8-byte representation. Note that there are no mantissa bits with a value of 1. This is because 8,192 is a power of 2 (\(2^{13}\)) and so only the implied bit needs a value of 1.

For VAR2 (8,193) there is a loss of precision when stored as LENGTH 3 because the 4th bit contains information. This
information is lost when the data set is stored. In this case the filling in of bytes 4-8 with zeros results in a value equivalent to 8,192—so the original value of VAR1 (8,193) has been changed to 8,192.

Let's expand on the DATA step in (20) to illustrate a couple of points.

(22) DATA ONE;
    LENGTH VAR1 VAR2 VAR3 3;
    VAR1 = 8192;
    VAR2 = 8193;
    VAR3 = 16384;
    PUT VAR1=22.16;
    PUT VAR1=BINARY64.;
    PUT VAR2=22.16;
    PUT VAR2=BINARY64.;
    PUT VAR3=22.16;
    PUT VAR3=BINARY64.;
RUN;

VAR1= 8192.0000000000000000
VAR1= 01000000 11000000 00000000 00000000 00000000 00000000 00000000 00000000
VAR2= 8193.0000000000000000
VAR2= 01000000 11000000 00000000 10000000 00000000 00000000 00000000 00000000
VAR3= 16384.0000000000000000
VAR3= 01000000 11010000 00000000 00000000 00000000 00000000 00000000 00000000

DATA TWO;
    SET ONE;
    PUT VAR1=22.16;
    PUT VAR1=BINARY64.;
    PUT VAR2=22.16;
    PUT VAR2=BINARY64.;
    PUT VAR3=22.16;
    PUT VAR3=BINARY64.;
RUN;

VAR1= 8192.0000000000000000
VAR1= 01000000 11000000 00000000 00000000 00000000 00000000 00000000 00000000
VAR2= 8192.0000000000000000
VAR2= 01000000 11000000 00000000 00000000 00000000 00000000 00000000 00000000
VAR3= 16384.0000000000000000
VAR3= 01000000 11010000 00000000 00000000 00000000 00000000 00000000 00000000

The first thing to notice is that the LENGTH statement in the first DATA step has no effect on the length of the variables created in the DATA step. The LENGTH statement only takes effect when the variables are stored.

The second thing to notice is that (unlike with VAR2: 8,193), there is no loss of precision when VAR3 (16,384) is stored. This is because in the full, 8-byte, representation of 16,384, bytes 4-8 contain only zeros. In fact, because 16,384 is a power of 2 (2^{14}), only the implied mantissa bit has information.

So it's not actually true that 8,192 is the largest integer that can be accurately represented with LENGTH 3 (on a PC or UNIX system). What is true is that 8,192 is the largest integer of a continuous range of integers that can be accurately represented with 3 bytes. The same is true for the other integers in the right-hand column in the table. These 'largest integer' tables are still handy references for quick decisions, but it's important to know that exceptions exist.

FRACTIONS

It's also important to remember that these tables apply only to integers. The situation is somewhat different when we look at fractions. Let's compare 1/10 and 1/2.

(23) DATA ONE;
    VAR1 = 1;
    VAR2 = 0;
    DO X = 1 TO 10;
        VAR2 + 0.1;
    END;
    VAR3 = 0;
    DO X = 1 TO 2;
        VAR3 + 0.5;
    END;
    IF VAR1 = VAR2
        THEN PUT VAR1 'EQ' VAR2 ;
    ELSE PUT VAR1 'NE' VAR2 ;
    IF VAR1 = VAR3
        THEN PUT VAR1 'EQ' VAR3 ;
    ELSE PUT VAR1 'NE' VAR3 ;
RUN;

If you run this DATA step, the LOG output will be:

(24)
1 NE 1 [for 0.1]
1 EQ 1 [for 0.5]

The first LOG message looks like a clear contradiction. How can the result be that 1 does not equal 1? The answer, of course, is that the stored value of VAR2 differs from its display form. Here's the 8-byte representation of VAR1 and VAR2:

(25)
VAR1= 00111111 11110000 00000000 00000000 00000000 00000000 00000000 00000000
VAR2= 00111111 11101111 11111111 11111111 11111111 11111111 11111111 11111111

Once we see the 8-byte representation, it's clear that VAR1 and VAR2 have different stored values. The output of the comparison also shows that SAS is comparing stored values and not display values. The reason that VAR2 does not equal 1 is
because 0.1 cannot be represented precisely in a binary system (though it's straightforward in a decimal system) and this imprecision iterates with each addition in the DO loop.

Now let's compare VAR1 with VAR3:

(26)

VAR1= 00111111 11110000 00000000 00000000 00000000 00000000
VAR3= 00111111 11110000 00000000 00000000 00000000 00000000

Here we see that adding 0.5 to 0.5 results in a stored value that is equal to 1. Why here and not with 0.1? Notice that, in contrast to VAR2, bytes 3-8 are all zeros. In fact, in binary representation, 0.5 looks a lot like 8,192 and 16,384. This is because these numbers are all powers of 2. 0.5 = 2⁻¹.

DECISIONS ABOUT STORAGE

It's clear that the magnitude of a number is an imperfect predictor of whether or not it can be accurately represented with a reduced number of bytes. The numbers 1/2 and 16,384 can be accurately represented with 3 bytes, but 8,193 cannot.

Let's consider a numeric variable HOSPITAL_DAYS that indicates how many days in a year a person was in the hospital. The largest possible value is 365 in non-leap years. But let's assume that half-days (but no other fractions) are possible, so values such as 189.5 are allowed. This is a variable that, despite the non-integer values, is a candidate for LENGTH 3.

Recognizing candidates for reduced LENGTH is an important part of the decision process, and knowing a bit (no pun intended) about how SAS stores numbers is essential for this. The larger question concerns other factors that are involved in making such a decision.

Here's an example, similar to (12), that illustrates how you often need to consider a larger context when making, what appear to be, narrow, technical decisions. Assume that you have a SAS data set with a variable for annual out-of-pocket dental expenditures. The values are all integers and the largest value for this variable on the file is 8,192. Should you set LENGTH 3?

Let's say you do. Fast forward a year or so and the task is to input this file, increase dental expenditures by 13% (to account for inflation), and round the result to the nearest integer. Here's a DATA step that shows the potential problem.

(27) DATA ONE;
    LENGTH DENTOOP1 3 ;
    DENTOOP1 = 8192;
    DENTOOP2 = 8192;
    RUN;
    DATA TWO;
    SET ONE;
    DENTOOP1 = ROUND((DENTOOP1*1.13));
    DENTOOP2 = ROUND((DENTOOP2*1.13));
    PUT DENTOOP1= ;
    DENTOOP1=9256;
    DENTOOP2=9257;

In data set ONE the variable DENTOOP1 has a LENGTH of 3 whereas DENTOOP2 has a LENGTH of 8 (the default). After the 13% increase and rounding in the second DATA step, we see that the modified values depend on the LENGTH specification. What you don't want is a phone call from the client asking why, when a simple check with a hand calculator is used, they get a value of 9257 (i.e. 9256.96 rounded) but your program output is 9256. Finding out why could ruin your whole afternoon.

Of course the problem could have been avoided if the programmer updating dental expenditures had simply (i) run a PROC CONTENTS and noticed that the LENGTH is 3 bytes; (ii) known the integer values for which accuracy is preserved with this LENGTH; (iii) run a PROC and noticed that there are values that, if increased by 13%, would exceed 8,192; (iv) realized that variable attributes like LENGTH are inherited by one data set from another; (v) known how to stop the default attribute inheritance for this variable; and (vi) placed a new LENGTH statement in the DATA step that updated the variable's values.

If the steps listed in the last paragraph are second nature to you, everyone you work with, and everyone you will ever hire, then this would favor a decision to reduce numeric length based simply on the values in the data set.

I would argue, however, for a more conservative approach. This approach would first distinguish between actual and permissible values. By actual values I mean the set of values that happen to be true for a variable on a particular data set. Given reasonable QC, the set of actual values should be a subset of the set of permissible values.

Let's take the dental expenditure variable as an example. On the data set I discussed earlier, the maximum value was 8,192. But this just happened to be true. It could have been different. We can contrast this with a numeric variable such as BIRTH_MONTH, which might have a permissible range of 1-12 and be restricted to integers. For various reasons it is possible that, on a particular data set, not all of these are actual values. But the point is that, if LENGTH 3 is set for this variable, it's hard to imagine a situation where values for this variable would ever create imprecision issues. Of course the variable might have a value of 99 or whatever to indicate "UNKNOWN", but, unlike the dental expenditure variable discussed earlier, the permissible range is well defined and clearly within the reduced LENGTH specification.

For the HOSPITAL_DAYS variable, given the permissible range, and the fact that the only permissible non-integer is 0.5, I would decide to store this variable as LENGTH 3.

One reason to be tempted by a less-conservative approach is the potential saving in storage space. In the next section I will discuss an alternative way of reducing the storage space required for numeric variables. It is important to remember that the only reason to reduce LENGTH with numeric variables is to
save disk space. There is no increase in computational efficiency. All numeric variables are expanded to 8 bytes for computations performed in DATA and PROC steps.

**COMPRESS= BINARY**

In SAS Version 8 one of the options for compressing SAS data sets is COMPRESS= BINARY. This is a compression method that is recommended for numeric data. It is efficient in two ways: (i) it can dramatically reduce storage requirements; (ii) it can decrease the time it takes to read in the data set. You can use COMPRESS= BINARY as a system or program option, or as a DATA step option, as in (28).

(28) DATA TWO (COMPRESS= BINARY);
    SET ONE;
    ....
    RUN;

Here's one example of the type of reduction you will find in storage requirements. The 1997 Medical Expenditure Panel Survey [MEPS] public use file HC-020 ("The 1997 Full-Year Consolidated Data File") has 1,215 variables and 34,551 observations. Observation length is 6,544. Almost 99% of the variables are numeric (1,201). Of these, 621 are LENGTH 3 and 580 are LENGTH 8.

Uncompressed the file uses 275 MB of disk space. Compressed with the COMPRESS= BINARY option it uses 43 MB. Also, the number of data set pages is reduced from 17,285 to 2,718. This latter reduction, in part, allows for faster I/O times because the SAS system will usually transfer a complete page in a single I/O operation. So, the fewer pages per data set, the fewer I/O operations required.

For many data sets this creates a win-win situation: you save both storage space and the time it takes to read in a file. I have run numerous tests on a variety of data sets and the following generalization holds:

(29) If the COMPRESS= BINARY option reduces file size, then SAS takes less time to read in the file.

SAS will give you an informative LOG message, e.g.

(30)

NOTE: There were 34551 observations read from the data set HC020.P20V4X.
NOTE: The data set DTEST.PUF20 has 34551 observations and 1215 variables.
NOTE: Compressing data set DTEST.PUF20 decreased size by 84.28 percent. Compressed is 2718 pages; uncompressed would require 17285 pages.

Obviously COMPRESS= BINARY is an option with lots of advantages. More CPU resource will be required, but this cost is usually more than offset by the benefits (see below).

In some SAS documentation you will see statements such as the following.

(31) In Release 8.2, when a request is made to compress a file, SAS determines if the compressed file will be larger than an uncompressed file. If so, SAS creates an uncompressed file and issues a warning message that compression is not enabled.

In a subsequent SAS Note (SN-005687), this general statement is corrected because there are cases where compression will be enabled despite it increasing data set size. As a test of this you can compress a data set with one variable, a character variable of LENGTH 13, and 100,000 observations. The LOG note will let you know that compressing the data set increased file size.

The general rule of thumb is that the effectiveness of compression increases with observation length. The overhead associated with a SAS-compressed data set is 12 bytes per observation for 32-bit hosts and 24 bytes per observation for 64-bit hosts, so this must be taken into account when making decisions about compression.

One final advantage of using SAS-internal compression: there is no need for an explicit utility or command to uncompress the data set. SAS will automatically uncompress the file.

Two important caveats must be mentioned regarding the use of SAS compression. The first is that CPU time is likely to increase. In my own work situation this is not a factor because (i) the cost of my program runs is not affected by CPU usage, and (ii) the savings in I/O time always more than compensates for any increase in CPU time (see Karp and Shamlin's SUGI 28 paper for a discussion of this and other issues with SAS compression). The second is that utilities such as Stat/Transfer® can only take uncompressed SAS data sets as input.

The general guidelines for storing SAS numeric variables are given in (32).

(32) a. Distinguish between permissible and actual values for a variable in a data set.
    b. Use the default LENGTH specification of 8 bytes unless all the permissible values for a variable can be accurately represented in fewer bytes.
    c. Unless there is a specific contraindication, use the COMPRESS= BINARY option whenever it reduces data set size.

These are guidelines, not rules, and (as always) you need to take into account any specific properties of your data set or project that would argue for a different approach.
COMPARISON OPERATORS

In the section on fractions, we have already seen that comparing two numbers can lead to counter-intuitive results if we do not consider the effects of numeric imprecision. In (23), because 1/10 cannot be accurately represented in a base 2 system, we appeared to have LOG output stating that 1 does not equal 1. Obviously this apparent contradiction only existed due to the numeric appearance of the variables. The actual comparison was between the stored representations.

Let's modify (23) as follows:

(33) DATA ONE;
    VAR1 = 1 ;
    VAR2 = 0 ;
    DO X = 1 TO 10 ;
        VAR2 = VAR2 + 0.1 ;
    END;
    VAR3 = 0 ;
    DO X = 1 TO 2 ;
        VAR3 = VAR3 + 0.5 ;
    END;
    IF VAR1 = ROUND(VAR2) THEN PUT VAR1 'EQ ' VAR2 ;
    ELSE PUT VAR1 'NE ' VAR2 ;
    IF VAR1 = VAR3 THEN PUT VAR1 'EQ ' VAR3 ;
    ELSE PUT VAR1 'NE ' VAR3 ;
RUN;

If you run this DATA step, the LOG output will be:

(34)

1 EQ 1  [for 0.1]
1 EQ 1  [for 0.5]

The round function corrects for the imprecision by rounding to the nearest integer. In (32) the rounding is within the comparison, the value of VAR2 is unchanged, i.e. exactly what you see in (24). But you may want to actually change the stored value of the variable. If so, then the DATA step in (34) is what you want.

(35) DATA ONE;
    VAR1 = 1 ;
    VAR2 = 0 ;
    DO X = 1 TO 10 ;
        VAR2 = ROUND((VAR2+0.1),.1) ;
    END;
    VAR3 = 0 ;
    DO X = 1 TO 2 ;
        VAR3 = VAR3 + 0.5 ;
    END;
    IF VAR1 = VAR2 THEN PUT VAR1 'EQ ' VAR2 ;
    ELSE PUT VAR1 'NE ' VAR2 ;
    IF VAR1 = VAR3 THEN PUT VAR1 'EQ ' VAR3 ;
    ELSE PUT VAR1 'NE ' VAR3 ;
RUN;

The LOG output of this DATA step will be the same as for (32), but for a different reason. Here the ROUND function is part of the assignment statement and the stored values of VAR1 and VAR2 are equal, as shown in (36).

(36)

VAR1 = 00111111 11110000 00000000 00000000 00000000 00000000 00000000 00000000
VAR2 = 00111111 11110000 00000000 00000000 00000000 00000000 00000000 00000000

SAS gives you the choice of rounding for the comparison or rounding the actual value of the variable. With the ROUND function you can specify a round-off-unit as a second argument that allows you to specify the decimal level for rounding. For example, ROUND(N,.1) rounds to the nearest tenth. If you omit the second argument, as in (32), then SAS rounds to the nearest integer, i.e. it assumes a second argument of 1.

A reasonable guideline is to store the more-precise (unrounded) values for a variable and round for specific purposes. That way if you ever need the more-precise values, you have them. Of course, as is often true in programming, the trick is being able to distinguish between what you see and what you have.

There are situations where the ROUND function itself may be affected by numeric imprecision For a detailed discussion of this issue see TS-230 from SAS Institute.

The CEIL and FLOOR functions may also be useful in specific situations. Both functions return integer values, i.e. unlike ROUND, you cannot specify a second argument indicating the decimal level.

The CEIL function rounds up to the nearest integer, i.e. it returns the smallest integer that is greater than or equal to the argument. The FLOOR function rounds down to the nearest integer, i.e. it returns the largest integer that is less than or equal to the argument.

CONCLUSION

It is important to understand that the display properties of numbers in SAS differ from the stored properties. Being aware of these differences and how they affect computations and reports is an essential part of good SAS programming.

Knowing the basics of how SAS stores numbers leads to more informed decisions regarding LENGTH specifications for numeric variables. The default specification in SAS is also the maximum specification, and this is exactly right. The costs and benefits of reduced numeric length should be carefully considered.

One guideline for storing numbers in SAS data sets is to only reduce length if the set of permissible values can be accurately represented in the number of bytes specified. Reducing numeric length does not affect DATA or PROC steps, so the only reason for doing so is to save disk space. For many SAS data sets, considerable saving of disk space can be achieved with the COMPRESS= BINARY option.
Even with a full 8-byte representation, not all numbers can be accurately stored. The ROUND function may be used to correct for this either by rounding the stored values of numeric variables, or by rounding for the specific purpose of a comparison within a DATA step.

SAS provides a variety of formats for displaying numeric data. SAS also allows users the option of creating their own formats. SAS formats are extremely useful and extremely powerful. They have the potential to reveal interesting patterns in the data, but also to obscure information. Again, this points to the importance of distinguishing between the appearance of a number on a screen or piece of paper, and how it is represented in the computer.

In the section on INPUT QC, I said that time pressure is one of the great enemies of good QC. It's clear that in the real world you can't always run every check that you'd like. But there's an awful lot that can be done fairly efficiently. In the same way that there are patterns in data we want to capture with our programs, there are patterns to QC that often recur on projects we are working on. As with input data before beginning a program, it's worthwhile at the beginning of a project to anticipate QC needs and build them into program specs. Often this will allow a number of QC steps to be 'macrotized' for a greater saving of project resources by being applicable to a number of different programs.

In general, the net cost in time for good QC is probably minimal. In many cases there is a net saving in time and money. Many of us know from painful experience the feeling that comes when someone asks, right after the final data delivery, "Did you know that those expenditure values we summed had –1 values for missing data?" Often it only takes a few simple checks on the patterns of the intermediate and final output data to reveal a problem—or to provide evidence that the output is all and only the intended output of the program.

REFERENCES

"Numeric Precision 101," available from:

"Dealing With Numeric Representation Error in SAS Applications," available from:


ACKNOWLEDGMENTS

I would like to thank all my SAS-programming colleagues at Social & Scientific Systems, Inc. SSS has a great community of SAS programmers and I've learned a lot being part of it. I would also like to thank the audience at DCSUG who gave me good feedback on early version of the presentation on numeric data.

CONTACT INFORMATION

Paul Gorrell
Social & Scientific Systems, Inc.
8757 Georgia Avenue, 12th Floor
Silver Spring, MD 20910

Email: pgorrell@s-3.com
Telephone: 301-628-3237 (Office)
301-628-3000 (Main)
FAX: 301-628-3201

-----
Stat/Transfer® is a registered trademark of Circle Systems, Inc.