INTRODUCTION

Jerry, a fictional programmer, was a recognized genius. No one knew just exactly how he did it, but all his work was done wonderfully. The trouble started after Jerry quit, to become a mystic in Tibet, and you inherit his assignments.

Have you ever found yourself assigned to cope with any of the following conditions:

- Inheriting undocumented, or tersely documented programs?
- Code elegantly designed to be densely unreadable?
- Code that rambled incoherently; where entire pages were commented out but not deleted?
- Data dictionaries out of date?
- Multiple copies of nearly exactly the same programs?
- Permanent format libraries having multiple members doing not quite, but almost the same value conversions?
- Undocumented data where no clear idea of the subset parameters that created them are given?

Jerry is more than fictional, he is my alter ego. Any project done too quickly, under short deadlines, has potential for disaster. Long ago I gave up the illusion that time is ever adequate to go back, clean up and fully document. Another project looms and takes main stage.

What can you do? How can you prevent perpetuating these conditions? Programming should be fun, not a constant audit trail - but without some standardizing methods, you will find that much of your time is spent on figuring out how, what, and where those year old report figures came from. The ones you may be required to duplicate.

Before we get down to defensive programming strategies, it is good to start with a clean sweep of your program library. Weed out and delete earlier versions, multiple copies, temporary attempts.

Keeping a tight control of your program library, format library and permanent data sets is work. Space limits force you to do this sooner or later, so it is best to keep on top of it as you go along.

The next pre-defensive issue is your data:

GIGO (Garbage in, garbage out). An old truism. Much crazy programming logic initiates from coping with anomalous data. Know your input data. Obtain, or create an accurate data dictionary.

- Run PROC CONTENTS DATA=ALL; to get a file layout: the variable names, labels, formats, creation date, (version 6.06 gives last modified date) and number of observations.
- Change non-arithmetic numerics like Social Security number to character. Why? you may want leading zeros; or avoid a PROC MEANS reporting total or average zip code, which would look foolish. Defining variables by there proper type is simply good programming practice.
- Knowing the extreme values of your data will help you spot bad data. For example, an AGE value of 600 (which should have been 60) may not be caught in the calculation of average age in a big data set. Use PROC MEANS N NMISS MIN MAX SUM;

PRE-PROCESS DATA WITH AN EXCEPTIONS REPORT:

The goal is to collect and print the things that have been identified as problems from prior experience with the data. For example, an in-house database rotates two years of active data on a monthly update schedule. A month can be dropped from the on-line database onto an archive tape right in the middle of a project. One standard exceptions report function is to compare minimum and maximum date range in the data to the requested date range. If these do not match up a message prints.

A clean program library and accurately described data source put you on terra firma instead of quicksand. Now the fun begins.

Organize the design of your code. One structure is to group code into distinct units:

1) DOCUMENTATION
2) FORMAT TABLES
3) MACRO FACILITY
4) SUBROUTINES
5) DOWNLOADING
This paper will progressively develop a budget allocation model, in a program named BUD92, to demonstrate these five units. Several overall defense strategies are mentioned at the end.

1) DOCUMENTATION
An introductory comment should frame the history of your program. For example:

```
/*BUD92 IS THE 1992 MODEL *
* FOR BUDGET AND REVENUE *
*
(DATA SOURCES:
* 1 STATE TAPE *
* 2 IN-HOUSE DATABASE *
* 3 REMITTANCE TAPES *
*CONVERTED BY (GRP92) PROGRAM*
* TO 1992 REIMBURSEMENT *
* METHODOLOGY *
*
*PRODUCES: DOWNLOAD DATA TO *
* PC MODEL *
*PROGRAMMER: E. QUINT *
*
*WRITTEN: APRIL 1991 *
*UPDATED: JUNE 1991 *
*THIS PROGRAM USES *
* %INC MACLIB(REV1) *
*MERGE TO DB.RANKS DATA SET *
*/
```

This header comment states three possible input sources; it tells me that the data has been pre-processed by a program called GRP92 (to standardize the data to a common rate year). It gives the goal, to DOWNLOAD the results. It identifies the programmer, dates written, and the sub-program (REV1) and data set (DB.RANK) that are also needed to run the program. If a copy of this code is requested by another user, the need to also copy the subroutine program and data set is seen immediately.

A Label is useful in tracing a data set to its source. A Label can be up to 40 characters, it prints whenever a PROC CONTENTS is run. In this example a data set label option states the program name, date range, and input source. In this way the output "remembers" its creation.

```
DATA ONE (LABEL=BUD92 CY91 TAPE);
```

Every permanently saved data set should have an identifying label.

Also use a label statement to describe variables.
```
LABEL variable1 = "label 1"
  variable2 = "label 2";
```

This will take the mystery out of cryptic naming conventions.

There are usually identifying values you would want to come directly from the data.

Getting the DSN information straight from the JCL is a powerful automatic audit tool.

```
sample JCL:
//DB DD DSN=DISK.PSD.DO7001DB,
  DISP=SHR
```

```
sample program module:
DATA_NULL;
LENGTH DSNAMES $40;
INFILE DB JFCB=DSNAME OBS=1;
%GLOBAL DSNAMES;
CALL SYMPUT(DSNAMES,DSNAME);
RUN;
FOOTNOTE1 "INPUT IS DSNAMES";
```

A Subsequent PRINT statement will now footnote automatically as:
```
INPUT IS DISK.PSD.DO7001DB
```

The JFCB=variable-name is an option on the FILE and INFILE statement. Under version 6.05 the option is:
```
INFILE DB FILENAME=DSNAME;
```

Note that in this example a SAS data set is read with an INFILE statement. Even though the JCL specifies DISP=SHR, if you accidentally use a FILE statement in the above example, you will replace a perfectly good file with junk. The FILE statement overrides the JCL. When you are using FILE statements you are in systems territory, and do not get the same protection you would using a SET statement or PROC Step from the JCL DISP=SHR parameter.

Using the JFCB option, DSNAMES was assigned the file name, some blanks, and then a string of strange stray characters. To refine the value use:
```
HOLD=INDEX(DSNAME," ");
DSNAME=SUBSTR(DSNAME,1,HOLD);
DSNAME=COMPRESS(DSNAME);
```

This seems to remove the problem, which may be site specific.
It would be good to note here that a GLOBAL macro variable cannot be used in the same data set that creates it. It can be used in any code outside and after the data step that creates it. Macros will not resolve within single quote’s, you must use double quotes.

To get descriptive values from the data set itself, use CALL SYMPUT. The CALL SYMPUT statement assigns a variable’s value to a macro variable.

In this example, PROTO is a variable that contains a protocol number which identifies the study. This is useful in titles.

```
DATA_NULL;
SET DB.DEMOG(OBS=1 KEEP=PROTO);
%GLOBAL PROTO;
CALL SYMPUT("PROTO",PROTO);
RUN;
TITLE1
"CLINICAL STUDY - &PROTO"
```

This prints as:
CLINICAL STUDY - DAZ-8807

These two methods use macros to document rather then process data, so it is best to keep them separate from the main logic of your program. Keep a template of this code, copy it in and modify as needed.

2) FORMATS
Grouping formats together, even though they may not be used until much later in your program, lets you block out a table look up area that is in one place. Formats common to many different data sets, or large format tables are better put into a permanent format library. Creating data with non-standard user defined formats saves time, but beware, a variable can be difficult to read if it cannot link to its format. If you somehow lose the input format table use the statement:

```
OPTIONS NOFMTERR;
```

in a mainframe environment. In PC SAS this options does not exist, but you can null out the format with

```
FORMAT variable-name;
```
in your data step.

A string of IF THEN’s that re-code data can often be changed to a format look up table. The original program stated:

```
DATA HOSP OOPS;
SET INDATA;
RANK=99;
IF TYPE="INLIER" AND _NAME_="CMICASE" THEN RANK=1;
ELSE IF TYPE="OUTLIER" AND _NAME_="CMICASE" THEN RANK=2;
ELSE IF TYPE="PSY" AND _NAME_="CASES" THEN RANK=3;
* {...33 more IFs THENs...};
ELSE IF TYPE="INLIER" AND TYPE="SIW" THEN RANK=37;
IF RANK=99 THEN OUTPUT OOPS;
ELSE OUTPUT HOSP;
PROC PRINT DATA=OOPS;
TITLE2 "REVIEW - NO RANK FOUND";
```

A string of thirty seven IF-THEN statements is hard to read. A better approach is:

```
PROC FORMAT;
VALUE RANK
  "INLIER-CMICASE" = "1"
  "OUTLIER-CMICASE" = "2"
  "PSY-CASES" = "3"
  ...{etc...}
  "INLIER-SIW" = "37"
  OTHER = "99"
```

```
DATA HOSP OOPS;
SET INDATA;
LENGTH RTYPE $20;
RTYPE=COMPRESS(TYPE | "-" | _NAME_);
RANK=PUT(RTYPE,SRANK);
IF RANK="99" THEN OUTPUT OOPS;
ELSE OUTPUT HOSP;
```

```
RTYPE is the concatenation of TYPE and _NAME_.
with padding blanks removed by using the COMPRESS function. The PUT statement table look up is easy to maintain and debug.
```
One reflection on using the catch-all "other", for example:

PROC FORMAT;
VALUE $PAYOR
"A"="MEDICARE"
"M"="MEDICAID"
"C"="BLUECROS"
OTHER="OTHER";

It is often better to let data print as is, unformatted, then unaccounted for. OTHER="OTHER" as a category is often a lazy way to cause yourself trouble when a new category appears in your data that perhaps should be assigned elsewhere.

One way of trapping a logic error is to initialize a variable to an impossible value. In the above example, RANK is given an initial value of "99". If any 99 value exists after RANK has passed through this logic, a data set OOPS is created. OOPS should always have 0 observations unless a data value exists that has not been accounted for. Coding for unknown data values makes it easier to spot both new data values and logic errors.

3) MACRO FACILITY
The macro facility is one of the most powerful features of the SAS language. It lets you divide an application into core similarities which need to be standardized; conversely, to address specific conditional requirements and reporting differences; and it effectively can be used as a self documenting audit tool.

If your are making minor changes to a tried and true program, then saving it as a new program, you will end up with many maverick programs. None of which you are sure does what.

The %LET statement is a nice way to declare at the top of your program:
- data set names
- conditional parameters
- critical constants

Constants that are hard coded act like "magic bullets", they will kill you when they are buried in hundreds of lines of code because you forget they are there.

Notice how titles and footnotes have been used to begin a self documenting audit trail.

The title and footnotes decode to print:
DB.MEDICAID-'01JUL91'D TO '30JUN92'D
CREATED BY PROGRAM BUD92
USES 1.0305 INFLATION RATE

The %LET statement can go almost anywhere in a SAS program, as long as it is above the &macro-name that invokes it. The structure is:

%LET macro-variable = <value>;

It is preferable to group %LET statements together. They represent the most often changed parameters of a program. They are global. Also, if a subsequent %LET is given the same macro variable name, it will overwrite the original parameter value.

Macro modules can also keep a "memory" of all these slight amendments to the original program.

The BUD92 model provides information to two departments: Revenue and Budget. Each has its own favorite data source and unique requirements. One input data source does not even remotely resemble the other.
At first we had two programs, one for Revenue, one for Budget. The question asked most by these departments when they compared results was:

How much of the difference was caused by:
- the different data source?
- an update in data?
- a change in time frame, or some other conditional parameter?
- a change in program logic formulae?

Since the results were summarized to represent very different group specifications, much time was wasted cross checking these programs.

The solution was to keep everything in one program. Set up macro modules to process components of the application discretely, and in logical progression. With planning, large segments of code can then be turned on or turned off by only a few changes in the macro parameters. A driver macro "shell", which will be described in detail below, surrounds this code. One summarized data set is created. It can be used for either a REVENUE report format, or a BUDGET report format.

To turn on/off input selection we used something like the example shown on the right:

The first %IF &INDATA=TAPE fails, DISK is input. Structure that optionally state input / output sources, I call a "SHELL" macro. A SHELL macro can drive input, output, comments, titles, data set label, just about anything as more parameters are added. The data set label

```
LABEL=(&dslbl.&indata);
```

decodes to BUD92-CY89-DISK. The dot delimiter allows the INDATA macro to be read and attached to the DSLBL.

On subsequent runs, by commenting out the first *

```ruby
%_0 AND putting the next macro choice right under it, a record of what combinations had been run, the data sets they created, and the parameters that created them, begins to accumulate on the program. This is a valuable tool in recording the history of a project.
```
This macro convention recognizes the comma as a delimiter, not a blank or semi-colon, hence &DATES is a complete SAS statement. It can be several statements if the mood takes you to design it that way. REVPARM=. is null, it is needed for TAPE only. And SFORMAT allows the flexibility to use different formats as they are needed. The inner macros %_1, %_2 are turned off (i.e. commented out) until final tables are needed, only %_3 download is requested.

With a little bit of ingenuity, you can get a dizzying confusion of macro loops and inner loops. If this begins to happen rethink your design. Dense, impossible to debug code, should be broken into several much easier to read modules. KEEP THE LOGIC SIMPLE. Remember the goal is to write a program that is flexible.

After setting the SHELL parameters, all data is filtered through the same processing routines. In this example a subroutine is used.

4) SUBROUTINES
To assured that complex calculations are being done identically across many different programs use subroutine programs. It also allows the appearance of a simpler line of coding logic. However, using too many sub-programs, for obscure one shot routines, causes its own problem. Do not abuse this tool.

5) DOWNLOADING: The Black Box of audit trails.
When you download data to an ASCII flat file you are relying on human intervention to parse, and re-label your variables. Missing numeric dots create havoc with parsing. A dynamic data set storage structure will make formulas in a cell bound PC spread-sheet useless. A little planning ahead on the mainframe side will help. Design a subroutine that removes numeric missing dots, puts the data in the exact position it needs to be in for the spreadsheet, automatically writes its own variable names on the first line, and carries an audit link from mainframe to PC. The ideas for this are given below:

For cell location merge your OUTDATA data set with a previously created RANKing data set. This data set, in our example, DB.RANKS, reflects the row order of a PC spreadsheet. After the merge, a line number for RANK exists for all combinations of _NAME_ and _TYPE_. Dynamic data became static.

Your ASCII download does not care about numeric or alpha. Use a data step to re-create all variables as character. This changes the numeric missing 'dot' to a character missing blank space.

For example:

```sas
DATA TEXT(KEEP=VAR1 VAR2 VAR3 DLBL);
SET HOSP(KEEP=RANK RTYPE PAYOR);
LENGTH VAR1 VAR2 VAR3 $8.
DLBL $30;
VAR1=RANK;
VAR2=RTYPE;
VAR3=PAYOR;
DLBL='LABEL';
RUN;
```

Next create a data set with all the variable names as values:

```sas
DATA VARNAMES;
LENGTH VAR1 VAR2 VAR3 $8.;
VAR1='RANK';
VAR2='RTYPE';
VAR3='PAYOR';
DLBL='LABEL';
RUN;
```

Put these two data sets together:

```sas
DATA _NULL_;
FILE DOWNLOAD;
SET VARNAMES TEXT;
PUT @1 VAR1 @10 VAR3 @20 VAR3
@30 DLBL;
RUN;
```

The resulting data set will look like this:

```
RANK PAYOR1 CMI LABEL
1 MEDICARE 1.145 BUD92
2 BLUECROS .894 BUD92
3 MEDICAID 1.203 BUD92
```

That top line is the first line of your data! After downloading, parse twice. The first parse of only the first line will get the variable name headings. The second parse, from the second line down, is the spreadsheet data.
DSLBL was created solely to identify the mainframe program, it could be extended to include data source, date ranges, or any conditional parameters. A literal character can be up to 200 spaces. It is the link to the mainframe source.

The automatic Macro variable SYSSCP returns the operating system code. This can be used to automatically turn on/off directory parameters that are the great divide between mainframe and PC SAS programming code.

```
%MACRO DIR;
  %IF &SYSSCP=OS %THEN %DO;
    "your-mainframe-library" DISP=SHR;
  %END;
  %ELSE %IF &SYSSCP=PC DOS %THEN
    "D:\PCSAS\";
  %MEND DIR;

  LIBNAME DB %DIR;
  RUN;
  DATA HOSP;
  SET DB.HOSP;
```

SOME OVERALL DEFENSE TIPS:

DO NOT RELY ON SYSTEM DEFAULTS. Explicitly using the DATA= option in each PROC to name the data set allows you to move blocks of code around without the worry that you've disrupted an implicit order of design logic.

CODE FOR MISSING VALUES. SAS Procedures handle missing values for the most part by ignoring them. Some will note this, others won't. I for one can never remember the rules.

MAKE YOUR CODE READABLE. A separate line for each statement, generous spacing and indenting DO-END loops will give the eyes a rest. Simplify long or complex expressions with parentheses, or break them into several easier to understand lines of code.

USE RUN STATEMENTS. It is important, especially when writing macro code to CLEARLY SEPARATE YOUR PROGRAMMING STEPS. RUN defines this boundary. Not only does this make your code more readable, it causes SAS log messages to print with the step, not ambiguously somewhere below it.

READ YOUR SAS LOG. In a long program that you have run many times over, who reads the log every time? At least check for the key words: ERROR will stop you cold, but its smart to scan for: WARNING, UNINITIALIZED (you may have spelt a variable name wrong), STOPPED, CONVERTED and LOOPING. Also scan the number of OBSERVATIONS being created at each data step.

FREEZING A PROGRAM

Freezing a program means that it is essentially dead, and will never be changed, but cannot be eliminated. An important report may have gone out on this version, you do not want it altered. Create a separate library for these dead versions. If moving programs to a separate library is not an option, rename these programs. Use a special first character, "Z", "@", "#" are good. They will sort together when you list your library members so it is less likely they will be accidently updated during the next head long rush to complete a similar project. Also, be sure to copy any subroutines into the frozen program, replacing the %INCLUDE statement. This will remove the danger of subsequent updates occurring to a subroutine that would invalidate any attempt to trace back the logic. Freeze programs sparingly or you will have the multiple program problem you started with.

COVER SHEET DOCUMENTATION

If the final results must print letter perfect, identifying footnotes are not allowed. Create a segment of code that prints a for-your-eyes-only cover sheet. It should print all the data set names, program and sub-program names, data range values and macro parameters which were created to automatically drive, select and track your data.

IN SUMMARY. When a project is completed to a milestone, that is the time to list the programs involved; print summarized results; draw a quick informal flow chart of inputs / outputs. Combine this with the folder that has your PROC CONTENTS, PROC MEANS, exception reports, and cover sheet. Tape the disk sheath holding a copy of your downloaded data set inside. THIS COPY IS YOUR AUDIT TRAIL, NEVER TO LEAVE YOUR OFFICE, really.

Eva Quint
Pfizer Inc.
219 East 42nd Street
New York, New York 10017
(212) 573-1477

SAS is a registered trademark of SAS Institute Inc., Cary, NC, USA
This example turns the header comment into a dataset and prints a cover sheet.

Data creation: HC dataset

PROC PRINT DATA=HC;
PROC PRINT DATA=HC1;
MERGE INDEX.HC HC1;
BY PROGRAM;
PROC PRINT DATA=INDEX.HC;

Program documentation:

DATA HOLD (KEEP=PROGRAM);
SET HC;
LENGTH PROGRAM 515.;
IF TYPE='PROGRAM:';
PROGRAM=COMPRESS(DESCRIPT);

DATA HC1; IF _N_=1 THEN SET HOLD;
SET HC;
IF TYPE='STUDY:';
PROC SORT NODUPE DATA=INDEX.HC; BY PROGRAM;
PROC SORT DATA=HC1; BY PROGRAM;
DATA INDEX.HC (KEEP=PROGRAM DESCRIPT);
MERGE INDEX.HC HC1;
BY PROGRAM;
PROC PRINT DATA=INDEX.HC;

This dataset lists your program member names with a description update if the program automatically added.