ELEMENTS OF GOOD PROGRAMMING STYLE

Amy Caron
Syntex Research

ABSTRACT

The importance of good programming style cannot be underrated. Unfortunately, programmers have a strong tendency to do so. We all like to believe that once a piece of code is put together, however haphazardly, the code will work properly from the point it is put into production until forever after. Besides, why waste cleaning up the code if it is only going to be used once? Writing code cleanly the first time not only increases the chance of getting it right but also eases the task of fixing it in case it is not right. Furthermore, chances are likely that the program will be used more than once and it will probably be used by someone other than the author of the code. So in the interest of those writing code and those having to use code that has already been written, some elements of good programming style will be discussed.

INTRODUCTION

How can one program be considered good and another bad when they both solve the same problem? The answer is a matter of style. A well-written program not only outwits the computer but is also comprehensible to someone reading it years later. A basic precept of good programming involves implementing the literal set of instructions necessary for a computer without losing the reader, who deals in concepts. A clean, clear program involves several elements of good programming style. Those elements to be discussed below are expression, control structure, input/output, and documentation.

EXPRESSION

A computer program consists of a sequence of statements. How these statements are expressed determines the understandability of the whole program. These statements also determine what the program actually does. Commenting, formatting or supplementary documentation cannot replace well expressed statements. One rule of good programming practice is:

WRITE CLEARLY. DO NOT BE TOO CLEVER.

Practically every programming language allows the programmer to form arithmetic expressions, and each language has an operator precedence to eliminate ambiguity for expressions without parentheses. Unfortunately, some programmers take great liberties with this capability. For example, the following equation calculates body surface area:

$$BODYSA = (0.007184)\ WEIGHT^{0.425}\ HEIGHT^{0.725}\ .$$

Putting this equation into SAS® code yields:

```
BODYSA = 0.007184*WEIGHT**0.425*HEIGHT**0.725; .
```

Notice that the statement is syntactically correct; however, it is not at all clear what operations are being performed or in what order. One solution is to insert parentheses in meaningful places to clarify the sequence of arithmetic operations being performed. Thus the body surface area statement might look like this

```
BODYSA = 0.007184 * (WEIGHT**0.425) * (HEIGHT**0.725); .
```

Going a step further, breaking the statement into a number of simpler statements is even clearer.

```
WTCOMP = WEIGHT ** 0.425;
HTCOMP = HEIGHT ** 0.725;
BODYSA = 0.007184 * WTCOMP * HTCOMP;
```

All of these SAS statements will calculate body surface area correctly; however, in the last two versions the programmer has made his computations more legible to the reader.

Another important principle of clear expression is:

USE MEANINGFUL VARIABLE NAMES.

A name can carry an enormous amount of information (and misinformation). Therefore, names should be as meaningful as possible. For instance, the following statements compute sales tax and sales price of an item and calculate the amount of change given, if any.

```
S = T * P;
C = P + S;
CH = A - C;
```

Without any type of identifying information these statements are meaningless. The variable names have no significance and so the intent of the code is not clear. In general, mixtures of similar characters, such as the letter O and the digit 0 or the letter I and the digit 1, are unsafe. Long identifiers that differ only at the end, such as HEIGHT1 and HEIGHT2, are also poor choices since misspellings and typographical errors are more easily made. Well-chosen names clarify the purpose of the variable. The SAS system allows 8 character variable names, and the programmer is advised to use as many of those characters as possible. Using more meaningful names, the above example can be rewritten as

```
SALESTAX = TAXRATE * PRICETAG;
COST = PRICETAG + SALESTAX;
CHANGE = AMOUNT - COST; .
```

This time the variable *names* indicate the function of the variable and clarify the intent of the statements. These statements are also self-documenting.

CONTROL STRUCTURE

A computer program is shaped by its data representation and the statements determining its flow of control. Together these define the structures of a program. These control structures in turn provide the framework of the program. How these elements are used determines the overall intelligibility of the program. A rule in this category is:

AVOID NEGATIVE BOOLEAN LOGIC.

Negative Boolean logic seems to invite misunderstanding. For example, the following SAS statements

```
IF NOT MARRIED THEN TAX = SALARY;
ELSE TAX = SALARY/2;
```

can be rewritten into the following equivalent positive statements that are easier to understand:

```
IF MARRIED THEN TAX = SALARY/2;
ELSE TAX = SALARY; .
```

Notice that the two sets of statements perform the same actions but the second set of statements does so in a less confusing manner. Again, the programmer is making the statements more legible.

Another rule of good programming style in the area of control structure is:

IF A VARIABLE HAS N CONDITIONS,
TEST FOR THE (N+1) CONDITION.

Many times a programmer tests a variable that is assumed to have one of two values, such as true/false, positive/negative or yes/no. If the variable is not equal to the first possible value, then it must equal the second possible value and the appropriate conditional statements are executed. However, unless the programmer is so close to the data that he knows all the expected values, an assumption like the one above is a bad one. Suppose ideal weight needs to be calculated for both sexes. SAS statements to do this computation might be:

```
IF SEX = 'MALE' THEN WEIGHT = (HEIGHT-60) * 5 + 100;

ELSE WEIGHT = (HEIGHT-60) * 3 + 100; .
```

However, can the programmer guarantee that the only values for sex are 'male' and 'female'? In many cases, he probably cannot. Therefore, a better way to handle the calculations is to incorporate all possible input values for sex.

```
IF SEX = 'MALE' THEN WEIGHT = (HEIGHT-60) * 5 + 100;
ELSE IF SEX = 'FEMALE' THEN WEIGHT = (HEIGHT-60) * 3 + 100;
ELSE PUT '** INVALID VALUE FOR SEX ** ' SEX=;
```

Program statements like these not only test for the N known values for a variable, but also include an (N+1) error-checking test for the none-of-the-above situation.

INPUT/OUTPUT

Input and output is the interface between a program and its environment. Input data generated by other programs or people should be viewed with suspicion. In order to handle the inevitable, a program should be as foolproof as is reasonably possible, meaning that it is easy to use and it behaves intelligently even when it is used incorrectly. A general rule concerning the area of input/output is:

USE MNEMONIC INPUT AND OUTPUT.

The use of numeric codes is bad practice in a program that people use directly, as opposed to a program that is only accessed through another program. For instance, the following SAS code matches a season with a sport.

```
IF SEASON = 1 THEN SPORT = 3;
ELSE IF SEASON = 2 THEN SPORT = 1;
ELSE IF SEASON = 3 THEN SPORT = 4;
ELSE IF SEASON = 4 THEN SPORT = 2;
ELSE PUT '** INVALID SEASON ** ' SEASON=;
```

Without any documentation, these statements are rather cryptic. If the programmer knew the numeric codes at the time of creating these statements, would he still know them two months or even two weeks later? A better way to write the statements would be to include meaningful mnemonic names instead of numeric codes. The example above can be rewritten as

```
IF SEASON = 'WINTER' THEN SPORT = 'SKIING';
ELSE IF SEASON = 'SPRING' THEN SPORT = 'BASEBALL';
ELSE IF SEASON = 'SUMMER' THEN SPORT = 'SWIMMING';
ELSE IF SEASON = 'FALL' THEN SPORT = 'FOOTBALL';
ELSE PUT '** INVALID SEASON ** ' SEASON=; .
```

These statements are self-explanatory and are their own documentation.

Another good rule concerning input and output is:

TEST INPUT DATA FOR VALIDITY.

Never trust any data. Input prepared by people or programs will inevitably contain errors. A good program will test the input data for validity. Suppose a man's weight and waist measurements are needed in order to compute sizes for tailoring purposes. The statements to do this might look like this:

```
INPUT MAN WEIGHT WAIST;
** COMPUTE SIZES **;
NECKSIZE = 3.0 * (WEIGHT/WAIST);
HATSIZE = NECKSIZE/2.125;
SHOESIZE = 50.0 * (WAIST/WEIGHT); .
```

What happens if either weight or waist contains the value zero and the input data is not properly checked? The program will attempt to perform the computations and when it encounters the zeroes, a message like "division by zero" is printed to the log. However, if the statements include input checking, bad or implausible data will be identified and handled in such a way as to avoid a fatal program error. Statements to check input might look like the following:

```
INPUT MAN WEIGHT WAIST;
** CHECK FOR BAD INPUT **;
IF WEIGHT=0 THEN DO;
   PUT '** QUESTIONABLE WEIGHT ** ' MAN= WEIGHT=;
   WEIGHT=.;
   END;
ELSE IF (WEIGHT LE 100) OR (WEIGHT GE 400) THEN
   PUT '** QUESTIONABLE WEIGHT ** ' MAN= WEIGHT=;
IF WAIST=0 THEN DO;
   PUT '** QUESTIONABLE WAISTLINE ** ' MAN= WAIST=;
   WAIST=.;
   END;
ELSE IF (WAIST LE 20) OR (WAIST GE 55) THEN
   PUT '** QUESTIONABLE WAISTLINE ** ' MAN= WAIST=;
** COMPUTE SIZES **;
NECKSIZE = 3.0 * (WEIGHT/WAIST);
HATSIZE = NECKSIZE/2.125;
SHOESIZE = 50.0 * (WAIST/WEIGHT); .
```

By "laundering" the data with validity testing statements such as these, fatal errors can be avoided. Using this approach, the program behaves correctly when used properly and also acts intelligently when used incorrectly. The generally suspicious approach is therefore advisable.

DOCUMENTATION

The best documentation for a computer program is a clean structure. However, writing a program is like writing an instruction booklet. Just including all the facts is not enough. They must be presented so that even a casual reader can follow. This is where comments come into play. Although comments do not effect the program, they are still part of it and therefore provide much of the program documentation. A good suggestion for documentation is:

FORMAT A PROGRAM SO THE READER CAN UNDERSTAND IT.

The physical layout of the program should help the user understand the logical structure. In the SAS system, a semicolon signifies the end of a statement. Several statements can be on one line and are interpretable as separate statements when they are executed. For example:

```
PROC SORT; BY PATIENT;

PROC PRINT; ID PATIENT; VAR AGE HEIGHT
WEIGHT; TITLE 'DEMOGRAPHIC VARIABLES'; .
```

These statements sort data and print them. Once again, these statements are syntactically correct but are not very readable. They are clustered together and are even split across a line. A better way to present the statements is:

```
PROC SORT DATA=DEMOG;
     BY PATIENT;

PROC PRINT DATA=DEMOG;
     ID PATIENT;
     VAR AGE HEIGHT WEIGHT;
     TITLE 'DEMOGRAPHIC VARIABLES'; .
```

Notice that each statement is on a separate line and the statements applying to a particular procedure are indented. Statements presented in this manner allow no confusion.

Another important rule for documentation is:

MAKE SURE COMMENTS AND CODE AGREE.

Many times software has been written to compute something but somewhere along the line, the definition of that computation has changed. Too often the program is modified but the comments describing the computation does not. For example, a White Russian drink consists of one ounce of Kahlua, one ounce of vodka and two ounces of cream. The comments and statements to reflect this definition might be:

```
**********************************;
** A WHITE RUSSIAN CONSISTS OF ONE **;
** OUNCE OF KAHLUA, ONE OUNCE OF   **;
** VODKA AND TWO OUNCES OF CREAM.  **;
**********************************;


IF KAHLUA = 1 AND VODKA = 1 AND CREAM = 2
     THEN WRUSSIAN = 'YES';
ELSE WRUSSIAN = 'NO'; .
```

Suppose vodka is not a necessary ingredient for a White Russian. The program statements are modified but in a panic the comments are not modified. After all, comments are not executable statements. Thus, the SAS statements now look like

```
************************************;
** A WHITE RUSSIAN CONSISTS OF ONE **;
** OUNCE OF KAHLUA, ONE OUNCE OF   **;
** VODKA AND TWO OUNCES OF CREAM.  **;
************************************;

IF KAHLUA = 1 AND CREAM = 2
     THEN WRUSSIAN = 'YES';
ELSE WRUSSIAN = 'NO'; .
```

The problem with comments that do not accurately reflect the code is that the reader, unaware of the discrepancy, might not examine the program statements themselves carefully enough. For a new user who may not know the definition of a computation or even the program's function, comments are the programmer's way of relaying information about the program statements. If the comments and program statements do not agree, then questions arise regarding the validity of both the comments and the program statements. The lesson here is to make the same modifications to both the comments and the code. The modified example above should look like:

```
********************************;
** A WHITE RUSSIAN CONSISTS OF **;
** ONE OUNCE OF KAHLUA, AND    **;
** TWO OUNCES OF CREAM.        **;
********************************;

IF KAHLUA = 1 AND CREAM = 2
   THEN WRUSSIAN = 'YES';
ELSE WRUSSIAN = 'NO'; .
```

CONCLUSION

Several areas of programming where style can be implemented have been touched upon. Within each area, several rules have been mentioned with supporting examples. These are but a few of the rules concerning good programming style. Many programs can be improved upon by the application of a few of these principles of good practice and a little common sense. So when sitting down to write that next program, remember the golden rule of style which is:

> A PROGRAM SHOULD BE AS EASY FOR A HUMAN BEING TO READ AND UNDERSTAND AS IT IS FOR A COMPUTER TO EXECUTE.

REFERENCES

Kernighan, B.W. and P.J. Plauger, The Elements of Programming Style, McGraw-Hill, 1978.

SAS Institute Inc., SAS User's Guide: Basics, Version 5 Edition, Cary, NC: SAS Institute Inc., 1985.

Yourdon, E., Techniques of Program Structure and Design, Prentice-Hall, 1975.