

MACROS AND FUNCTIONS IN C AND SAS® SOFTWARE

Lillian L. Randolph, Ph.D.
American Medical Association

Programs written in C or SAS can be compiled and executed on everything from small personal computers to powerful workstations (linked or not in networks) to large mainframes. Both languages are "portable", meaning that programs written in these languages can be compiled and executed in any of several systems as long as the required compiler type (i.e., for C or SAS) is available on the system. What counts is the availability of the appropriate compiler, not the size of the system.

In C and SAS, macros and functions have a similar definition and a similar usefulness. Macros result in macro expansion, replacing the macro call with its replacement text and the macro parameters with its arguments. Thus, their usefulness is to save the programmer from writing the code in the macro expansion. Functions return a value from one or more arguments. We might create functions by naming the function and specifying its parameters (and, in C, the data type that the function returns). Their usefulness is to define a procedure which is always done in the function call, the protocols of execution, and the result of executing the procedure.

Macros and functions are examples of modularization, the division of programs into small structures, each performing a single task. Modularization is crucial when you want to create reusable, sharable program units to make new development more efficient or improve the maintainability of existing applications. It is also invaluable to move applications off the mainframe and into a client/server environment. Slicing off program modules can split out functions that will run in the new environment from those that will not without modification.

Macros and functions are similar in another respect: their value is independent of the variables (e.g., the DATA step variable) being processed. In fact, the macro variable contains one value that remains constant until explicitly changed. The function value depends not on variables but on the argument(s) defined in the function.

*Macro and function are viewed here as discrete entities, even though SAS USER'S GUIDE describes "macro functions" as referring to functions within macros, such as %QUOTE or %SCAN. These "macro functions" are indeed functions, because their invocation can return a value from an argument. However, most SAS macros are true macros as defined herein.

However, the advantages and disadvantages of macros and functions are different. Macros have drawbacks not extant in functions. One drawback is that the macro code appears everywhere the macro is called. A function's code appears only once, irrespective of how many times the function is called. Thus, long and frequently used macros should be written as functions, if possible.

Macros have advantages not available in functions. Unlike functions, macros can be passed a type as an argument. For example:

```
#define INRANGE(x,y,v) ((v)>=(x) && (v) <=(y))
```

on a macro call of `INRANGE(1,100,i)`, expands into:

```
((i)>=(1) && (i) <=(100)).
```

Thus, `INRANGE` can find the minimum of any pairs of values with the same data type (ints, doubles, etc.). If we code `INRANGE` as a function, we must specify a single type for its parameters, with the result that we must write different versions of `INRANGE` for each data type.

Another advantage of macros is that macro calls are done in preprocessing instead of at run time, eliminating the extra computer tasks in a function call (argument passing, variable allocation, calling and returning from the function).

In a table look-up using a pointer to an existing array to fill in with a dynamically allocated array, the C programmer must code a macro. This situation is an example of a procedure that cannot be written as a function, because the computer needs to know what kind of elements are in the arrays.

Even when we could use functions, we often use macros to make our program more readable without sacrificing efficiency. `FSTALLOC` is a variant of `ALLOC` in the example following that automatically computes a string's length and allocates enough space for the string.

```
#define FSTALLOC(s) (malloc(strlen(s) + 1))
```

By using `FSTALLOC` we save ourselves additional typing and lessen the chance that we will forget to allocate space for the string's null character. We could write `FSTALLOC` as a function, but it is much more efficient as a macro.

In C, programmers oftentimes write their own macros and functions. This is the case even though most C preprocessors contain predefined macro names along with processing code, and the

frequently used functions, such as `#include` and `#define`,* are drawn from header files maintained in the C compiler. In SAS, however, while programmers might write their own macros, there is disinclination to do so, arising from the nature of SAS programs. For example, SAS programs must define the location of variables on input files, and the names of output files, both of which in C can be defined by pointers and in header files. As a result, there is more coding and greater possibility of errors. And functions in SAS are prescribed in the language, and accord with predefined names. While these are many and varied, there is no opportunity to define particular functions.

The comparative freedom in C to write macros and functions is a mixed blessing. There is greater possibility of faulty code. Many a C programmer experiences incorrect orders of evaluation for seemingly unknown reasons. For example, the function

```
#define SQUARE(x) (x*x)
```

causes SQUARE to expand into $i + j * i + j$, which does not return $(i + j)$ squared. The correct result comes from carefully placed parentheses:

```
#define SQUARE(x) ((x*x)).
```

This feature in C can be advantageous, as when the programmer uses the preprocessor to select lines of the source file actually compiled. The expression:

```
#if constant expression
    first-group-of-lines
#else
    second-group-of-lines
```

evaluates constant expression and compares its value with zero to determine which group of lines to process. If the expression is not zero, the preprocessor will process first-group-of-lines; otherwise, second-group-of-lines. Thus, `#if` can be used to "comment out" sections of code with comments, which is useful because comments don't nest. The code:

```
#if 0                                /* begin the ignored part */
    first-group-of-lines
```

*For functions, C uses `#` in column 1 followed by the name of predefined functions, or the programmer's defined function without the `#`. For macros, C uses two special operators that can appear in macro definitions: `#`, which precedes a parameter name, and `##`, which turns two tokens into a single token. Thus, the appearance of macros and functions may be identical.

will comment out (ignore) this first group of code.

The directive **#undef NAME** lets us select between a macro and a function to accomplish a particular task. Without the directive, all subsequent macro calls are replaced with its replacement text. Sometimes, however, we want to limit a name definition's scope, either to highlight that the name is used only in a small section of the program or to allow the name to be redefined. But with the directive, we undefine a defined name, which stops any further macro substitution for it. We can then use **#define** to redefine the name. Otherwise, the compiler will give an error message if we try to redefine a name without first undefining it.

An example of a programmer-written function which makes coding complex programs much easier is the use of generic functions, such as **bsearch** and **qsort**. To use **bsearch**, for example, we not only provide a target value, an array, and the number of elements in the array, but also extra information describing the size of each element and the function to use to compare elements.

```
ptr = bsearch(target, tab, n, sizeof(char *), cmpstr);
```

Then, we can perform a binary search with a simple call:

```
ptr = SEARCH_STRINGS(target, tab, n);
```

We can accomplish this easily by using header files. For each type of string we want to search, we create a header file having an interface macro (a good use of macro code) as well as prototypes for the comparison functions we want. (In the above notations, **cmpstr** defines the string comparison functions). We put the comparison functions in a separate source file and compile it separately. All we have to do now is include the header file, use the macros, and link in the comparison functions. This approach makes searching tables of strings a breeze!

From the above statements, both C and SAS programmers are advised to consider the usefulness of macros and functions, and the results from coding them in C or SAS. Macros give us speed, and functions save space. The advantages and disadvantages lend a rationale to the choices available.