Writing Portable C Code: The SAS Institute Experience

Richard D. Langston
SAS Institute Inc.

This paper illustrates how SAS Institute Inc. went about converting a large software system from PL/I and IBM® 370 assembly language into a new version written almost entirely in C. The paper briefly discusses the history of software development at the Institute, then explains the rationale behind the company's decision to convert to a new language. From there, the Institute's experience in writing portable software is discussed. In addition, the paper touches on the development process the Institute used, including the adaptation of software coding standards.

Let us begin by tracing the history of programming language usage at the Institute. SAS® software was originally implemented for OS-type operating systems (MVT, MFT, MVS, VS1). The SAS System was implemented using a combination of PL/I and assembly language. The assembly language portion was known as the "supervisor" and was implemented as such for efficiency, speed, code size, and any other characteristic for which assembler is best suited. The PL/I portion of the system was that of the procedures (programs written to access and create data using the supervisor through a common set of routines). PL/I was chosen here because the procedures were performing a higher-level set of functions, such as statistical analysis and report writing. Procedure writers could spend more time concentrating on the implementation of complex algorithms instead of concerning themselvses with how many base registers to have available.

This hybrid of language served the Institute well until it was decided to transport the SAS System to other operating systems. Recall that at the time the system was first implemented, the IBM OS family of operating systems was much more widespread than the VM family, so the emphasis was placed on performance within OS systems instead of portability. Of course, the user community quickly recognized the usefulness of the VM family and began an allegiance to those operating systems that precipitated the Institute's need to convert the SAS System to run under VM.

This effort was accomplished while maintaining the aforementioned hybrid of languages. Very little PL/I code was added to support VM; most of the system modifications were to the supervisor. Therefore, most modifications were made in assembly language. The conversion goal was to change as small amount of procedure code as possible and allow the supervisor to recognize the environment and deal with its eccentricities, relieving the procedure writers to once again concentrate on their specialties.

However, certain aspects of the VM operating system certainly made their way to the procedure writer's level. For example, the MVS operating system fills dynamically allocated memory with zeros before the application receives the pointer to the memory. The PL/I programmers were sometimes unknowingly taking advantage of this fact by not initializing variables (whose initial values would be zero nonetheless). Only when their procedures were tested under VM did this fact arise. This was the first of many occurrences of recognizing the difference an operating system can make on the performance of an application that is supposed to be operating system-independent.

The conversion for VM was accomplished primarily by OS simulation; that is, simulating OS SVCs and control blocks when system calls were made. However, when the Institute converted to the DOS/VSE system, the system developers recognized the need for isolating the system-dependent portions of the code so that these could be replaced when moving to a new environment. Once again, very little of the procedure code had to be changed to function under DOS/VSE.

The SAS System remained an IBM mainstay through the 82.4 release of the product. The system did not function on any other computers other than those within the 370 family (using the operating systems VS/1, MVS, VM/CMS, DOS/VSE). However, just as the Institute recognized a growing force in the user community for the VM family, there was acceptance of the fact that even more operating systems seemed likely candidates for conversion. The user community had been asking for minicomputer support, primarily on Digital Equipment Corp.'s VMS® Data General's AOS/VS, and Prime's PRIMOS® operating systems.

But converting to those operating systems would be much different than converting from MVS to VM/CMS. After all, the IBM operating systems all use a common assembly language and PL/I compiler (not a completely true statement but relatively speaking, close enough). The three minicomputer operating systems all had their own assembly languages and different PL/I compilers (all different implementations of the ANSI standard), and they were architecturally far removed from the IBM 370 family. A totally new implementation of the SAS System would be needed. And, to simplify this implementation, the largest possible subset of the system needed to be implemented in a portable language so that the same code would function on all three minicomputers.

The final outcome of the design was that the supervisor for the system was written in PL/I. This language was chosen for several reasons. First, almost all of the Institute programming staff was familiar with it, having written procedures in this language. Second, the intention was to once again have the procedures be as portable as possible across operating systems (IBM mainframes and the minicomputers). At the time, PL/I was the language with the most power and flexibility for which compilers were available for all the systems. Third, conversion would not be as painful since the procedures were already in PL/I, albeit using the PL/I Optimizing Compiler as a base language.

The programming staff quickly discovered that each vendor's implementation of the PL/I language was different. As the staff gained experience with the compilers, a set of coding standards was adapted to ensure that the code would compile on any of the machines. Some of the simpler decisions made were that such characters as '#' and '@' could not be used in the names of variables because the ANSI Standard did not include them. The use of # and @ in variables was a common practice among the developers using the more amiable Optimizing Compiler. Another common practice was incrementing pointers by overlaying an integer on the pointer and incrementing the integer. This became verboten in the portable implementation due to pointer sizes on some machines. These are but two of many examples of differences that were dealt with. Striving for portability across operating systems became a common goal among all developers, and this virtue carried on throughout the minicomputer implementation and is still apparent in the latest design effort using the C language.

The minicomputer and mainframe implementations merged with the advent of the Version 5 SAS System. The procedure code implemented for the minicomputers was also functional on the mainframe system. Only the supervisor was different between the two systems. The supervisor on all three minicomputer systems was the same, with the exception of small amounts of host-dependent code, necessary to perform the low-level tasks demanded of a complex system.

As the Version 5 SAS System made its way into the user community, the IBM PC was quickly evolving into a viable environment for a system as complex and expansive as the SAS System. An experimental project began on this microcomputer, using C as the implementation language. C was chosen because there was not a fully acceptable PL/I compiler available for the machine. C has many characteristics of PL/I, so its choice was not an unreasonable one. And its functionality on a microcomputer was well recognized.

Once again, the Institute was faced with needs of the user community that would require massive changes to system implementation. Was it better to develop a system written in C solely for the PC environment, similar to the Version 5 PL/I system written for the minicomputers? It was decided that the best approach was to rewrite the entire system in C in such a way that it would be portable across all operating systems, so that minimal reimplementation would be necessary when still more computer systems became viable as environments for the SAS System.

The biggest problem with this decision was compiler availability. All of the minicomputers on which the SAS System ran had good C compilers, and there were several available on the IBM PC. However, such was not the case for the IBM mainframe. The compiler from Whitesmiths Ltd. was tested, but for various reasons, it did not meet the needs for MVS development. Neither did the Waterloo compiler, which at the time was strictly a VM compiler. The solution was to transfer a PC DOS C compiler to the IBM mainframe so that sufficient control could be ensured.

The outgrowth of this solution was the agreement with Lattice, Inc. to implement their C compiler on the mainframe. After these efforts ensued, Lattice, Inc. agreed to be acquired by the Institute. After the acquisition, further enhancements were made to the PC DOS version of the compiler, concurrent with the mainframe development. Examples of these enhancements include the introduction of built-in functions to produce in-line code instead of subroutine calls, better diagnostic messages to ensure that a function declaration or prototype is always present, and improved floating-point processing to improve performance.

So, at this point, the Institute had access to a PC DOS compiler, complementary to a mainframe version, along with the C compilers of other operating systems (VMS, AOS/VS, PRIMOS, UNISYS's OS/1100, and Control Data Corporation's NOS/VE. The Institute also realized that many other operating systems had C compilers, and if a new implementation would function successfully on all of the systems currently available to the Institute, new operating systems could be added expediently and with much less "pain" than with VM/CMS, DOS/VSE, or the minicomputers using PL/I.

Most developers accepted the C language quite readily. As stated earlier, there are similarities between PL/I and C. The most obvious are the use of semicolons as statement delimiters and /* */ as comment delimiters. My own experience with learning C was that it was initially a "step down" from PL/I; the string-handling capabilities of C are minimal, and it has the feel of a low-level language, unlike PL/I. However, as more developers began to convert to the new language, more subroutines and macros were created and C began to have the look and feel of a high-level language like PL/I, but with the portability that C enjoys.

To ensure that this portability would truly exist and that the SAS System would indeed be transferable to many different operating systems, a more rigid design was constructed for Version 6 of the SAS System. This design, known as Multi-Vendor Architecture (MVA), is a three-tiered approach steeped with the experience of previous reimplementations. The lowest tier was the host level. A set of formally specified and documentated routines would be implemented by a host development team on each host machine. The choice of implementation would be up to that team, although the desire was to use C whenever possible. Examples of functions at the host level were memory acquisition and freeing, opening and closing files, task management, and conversion of generic global representations to native forms. The end result of the host level was a host supervisor. The second tier of the design was the core supervisor. The core supervisor would be written entirely in C, and it would be portable. It would not have to be reimplemented on other hosts and would function the same on all hosts. The core supervisor would be responsible for managing user requests to the hosts. These requests may be of the form of screen management as part of the SAS Display Manager System (the interactive editor and output manager used by the SAS System) or in the form of parsing program input, to name but two. Another characteristic of the core supervisor would be to accept calls from the procedures for services, such as managing output, reading/writing observations to SAS data sets, or managing pools of memory.

The third tier of the design was the application. The application would also be written in C. Most applications are procedures that are either part of the base product or one of the add-on products licensed by the Institute. In any case, the application would consist of a set of calls made to the core supervisor, along with the real work of the application, such as factor analysis, tabular reporting, full-screen editing of SAS data sets, and so on. The point is, as has always been the case in the SAS System design, the application writer can concentrate on implementing the task using his specialized skills; the work of interfacing to the system is handled by the core supervisor.

With this three-tiered system in place, the SAS System operates and has a similar look and feel across all operating systems. SAS code (that is, SAS program statements issued by the SAS user) can function the same regardless of the operating environment. There is ease of use for both the user of the system and the implementer of the system at SAS Institute.

The development staff, being aware of the three-tiered approach, set out on the monumental task of implementing a new system and converting existing software yet again, this time from PL/I to C. Naturally, a large number of programmers, especially those who were not conversant in C, needed a stringent set of guidelines to follow to properly code in a style of C that would be acceptable to the widest range of compilers. Developers throughout the Institute experimented with the peculiarities of the various compilers, studied the Kernighan and Ritchie standards, and came up with a set of house standards. The house standards were intended to minimize compiler complaints on all systems. The Institute had much experience with the concept of implementation standards, having found similar difficulty in dealing with several vendors' PL/I compilers in the project mentioned earlier in this paper.

The following are some examples of the house standards: 1) All external names must be seven characters or less. This is because some systems have a restriction on the size of external names. The C language itself and the PC version of the compiler do not have this restriction, but some systems do. (The PL/I Optimizing Compiler actually has this restriction, in that it chooses the first four and last three characters of the name and then fashions external names with this seven-character root). 2) Included H files must have names that are eight characters or less. (Once again, due to some systems' limitations). 3) Character range checking should only be performed by C library functions or by Institute-supplied functions. For example, if x is a numeric digit, the condition

```
(x >= '0')
```

will be true on an EBCDIC-based system only for the digits 0-9 (and the unlikely characters 0xFA through 0xFF), but on an ASCII system this condition will be true for almost all characters. The proper way to test for numeric digits is

```
(isdigit(x))
```

4) Always use the sizeof operator to get the size of any object. This is because different compilers pad structures differently, and some systems even have unconventional lengths for simple elements. For example, PRIMOS has a 6-byte char pointer in addition to its 4-byte pointer. 5) Do not use the & operator on non-lvalues in function calls. In the example below

```
char x[50];
...
abc(&x);
```

the argument to abc should not be passed with '&'. Some compilers get confused here and do not put the address of x on the stack. Omitting the & will remove the confusion.

The house standards document was very useful in assisting neophyte C programmers to adjust to the new language. However, there were still limitations due to the inherent nature of the C language. One such limitation was in the use of external functions.

In the C language, an external function being used in a program is declared as being external. However, there is no mention of the attributes of the arguments to that function. For example, a function abc that expects an int and a pointer as arguments and a function def that expects two longs will both be declared as

```
external int abc(),def();
```

If the caller does not use the correct arguments when calling the routines (for example, an int instead of a long), there is no validity checking that can be done by the compiler. Furthermore, type differences such as int vs. long may be acceptable on one machine (when an int and a long are the same thing) and may cause serious problems on another where they are different.

Most C developers over the years have used the LINT utility to assist them in finding these argument mismatches. LINT is given the source for all interrelated programs, and it will determine if there are mismatches. However, the Institute did not find LINT to be acceptable for a variety of reasons. For one, we had to rely on a different implementation of LINT for every machine, and the inconsistency in performance was apparent. Some versions would abend quite readily. Also, we felt it was much more desirable to have the compiler tell us if we were calling the routines incorrectly. We as developers appreciated the functionality of PL/I on this issue: any deviation from the function declaration was stated in compilation warnings.

Our solution was to use function prototyping, which had been specified and approved by the ANSI C Standards Committee. The Lattice PC DOS compiler (preacquisition) had function prototyping as a feature. However, certain compilers within our target operating system range did not have this feature implemented. In order to incorporate prototyping where possible, we introduced macros to be expanded on each host as appropriate. So, for example,

```
int abc U_PARMS (( long ));
```

could be expanded into a function prototype (if the compiler supports it) or simply a function declaration (if the compiler does not).

Simultaneously, we introduced the concept of structured macros. With structured macros, if a program were accessing routines from a given subsystem, the appropriate #include statement would be given for that subsystem. The "h" file included would have the necessary structure declarations within, along with the function prototypes for all the routines of the subsystem. A drawback, of sorts, was that there was no requirement that the prototypes be given for a function that was called. The compiler would accept external function calls without any declaration. We did not want to depart fully from a standard in forcing complete compliance.

Another decision made in trying to develop this large portable system was a decreased reliance on the C library functions. We found that many of them did not completely meet our needs. For example, the strcpy function requires a null terminator at the end of the source string, and that terminator is copied to the target string. We wanted to have our own copy function that did not rely on null terminators since that concept was somewhat foreign to our coding methods. The strlen function was found to sometimes give incorrect results with null strings. Granted, some problems may have been due to a given implementation of the compiler, but it was necessary for us to work within those constraints. The less we would have to contend with peculiarities of a given vendor's compiler, the better.

In the automated process to allow for portable code was old-fashioned human intervention: the code review process. Each subsystem of the core supervisor was scrutinized to ensure that coding standards were followed. Other benefits of the review were to check for more efficient ways of coding algorithms, how to cut down on memory usage (a crucial factor in PC DOS), clarity of code, adequate in-line documentation, and the recognition of esoteric features that could be used by other members of the development team.

More human intervention was apparent in host committee meetings. Each of the subsystems within the host supervisor had a corresponding committee, composed of host developers from all hosts for which implementation was proceeding. The developers would discuss proposed changes to their routines, such as calling sequence, documentation enhancements, or elaboration of functionality. These committees submitted their subsystems' changed specifications periodically, resulting in continually updated host supervisor implementation guides. This allowed for a robust development environment where feedback was quickly accepted and acted upon.

In summary, we at SAS Institute have had a great deal of experience with dealing with a changing user community, and have redesigned and reimplemented our systems accordingly. One important underlying characteristic that has been maintained throughout the years is that as much code as possible should be usable on as many operating systems as possible with little or no change. PL/I served this design well in the past; the C language is serving well for Version 6 and the foreseeable future.

Furthermore, the peculiarities of each vendor's compilers and operating systems must be taken into consideration when developing a portable system, regardless of whether the language or operating system is said to be portable. To accomplish this, coding standards are absolutely essential. And, the system design must encourage the majority of the development to be totally system-independent, with the system-dependent portion layered for easy re-implementation on new hosts.

We at SAS Institute have accomplished this with the three-tiered design, implemented following our house standards. We recognized the knowledge gained from previous re-implementation projects and from the experience brought by expertise of many different operating systems. We have found that this approach has been most productive in having our software functioning properly on as many different operating systems as possible.

SAS is a registered trademark of SAS Institute Inc., Cary, NC, USA.

IBM is a registered trademark of International Business Machines Corporation, Armonk, NY, USA

PRIMOS is a registered trademark of Prime Computer, Inc.

VMS is a trademark of Digital Equipment Corp.