INTRODUCTION

A few years ago, systems software for IBM mainframes virtually demanded assembly language programming. Other alternatives were few, and laden with handicaps. PL/I and Pascal have been tested replacements. The situation has brightened considerably as C compilers have become available. There is a chance we programmers will no longer be shackled to the machine's level of direct comprehension. This paper is an attempt to describe the author's progress along this journey.

I have numerous discoveries to share. Due to space limitations, I will focus on 3 principal themes. Each fundamental to productivity gains:

- **Object-oriented programming**
  - Development and debugging aids for portability.

- **Object-oriented programming**
  - The mainframe environment enables C programming on a very large scale. Designing at this level tries the imagination, while posing many organization challenges. Object-oriented programming helps focus design. It also leads to improved product quality. Furthermore, libraries of object-oriented modules are the well from which a diversity of products can spring.

- **Tools and debugging facilities**
  - Higher-level language programming is expected to yield improved productivity. However, this expectation rests heavily on support tools and debugging facilities. Main-frame C languages are newly arrived from UNIX and IBM-PCs. Over the years a thorough set of programming tools were crafted for UNIX. Some crossed the breach to the PC environment. Many remain unavailable for IBM mainframe programmers.

Portability

Programs written in C are believed easily convertible to other environments. This process is not always trivial. Some programs require extensive rework before they finally work as planned. As we develop any C program, there is always the nagging worry: "Will it be portable?" The art of programming portable software is black indeed. Language, library, and environment compatibility are key to all conversion efforts.

OBJECT-ORIENTED PROGRAMMING

Object-oriented programming raised its head a decade ago. Yet, this concept has not been extensively utilized by C programmers. Perhaps with C it is too easy to hack out a solution, and thus the noble software engineering force is deflected. Though this type of programming requires much time and discipline in preparation, the eventual rewards are well worth the effort. Over time an extensive, reliable library of base software can be defined upon which many products, tools, and aids can sprout.

Concepts

Object-oriented programming is based on hiding how data structures are physically implemented. All data structure accesses are performed indirectly via object manipulation functions. Thus, a data object hides within a semantic "black box". Any and all accesses are performed by trusted manipulation functions exclusively. Data content is guaranteed consistent and reliable, at all times.

Objects, instances, and handles

An object class is composed of a data area definition (a C header), and a set of access
functions (C subroutines or macros). Two fundamental access functions create and destroy "instances" of objects within a class. When an instance is created a "handle" serves as a name for subsequent accesses. A handle can be the instance's memory address. Better reliability ensues when the handle locates the instance indirectly. In this case there is no way for the external program to directly locate and alter fields. As an example, in MS-DOS when a file is opened an integer handle is returned. It indexes an array of file locators. There is a design trade-off between object insulation and overall efficiency. When direct addresses are returned, many object accesses can be implemented as in-line macro expansions, with noticeable performance improvement.

Constraints
Often data object fields have constraining value limits. For example, numeric values which are bounded within pre-defined ranges, or printable strings which consist of alphanumeric values exclusively. These conditions can be enforced by the object manipulation functions. During each access, prevailing constraints can be certified. This may seem like extraneous overhead. However, with the C preprocessor facility, constraint testing can be enabled during development debugging, and disabled when the production version is built. For example:

```c
#define DEBUG

if( !range( lrec1, 0, 32768 ) )
  punt();
#endif
```

In the above, constraint checking is enabled only when the "DEBUG" symbol is defined.

Inheritance

Often similar manipulations are performed upon different data objects. Though linked lists, queues, and stacks can be implemented with the same underlying structure, they can be accessed by separate logical actions. In this case we would like to define a single low level data object, while deriving higher objects. The derived objects thus "inherit" the lower level access functions.

Within MVS, partitioned files are an abstraction over sequential files; with subtle differences. Using "inheritance" as a guiding light, sequential and partitioned file objects can be crafted such that many sequential accesses can be reused as partitioned accesses. A single set of read, write, and positioning services suffice. These are extended by specialized directory access services germane to partitioned files.

Message passing
Within Smalltalk(TM), object-oriented programming was enhanced with message passing. This is an excellent way to achieve inter-process/inter-system communication, or remote procedure call (RPC). In essence, an object handle is a network address, routing code, and low-level locator. Inter-process accesses are effected by sending messages to network routing facilities which in turn invoke object manipulation functions.

Advantages
The primary benefits of object-oriented programming are improved design structuring and reliability. With this approach, the structure of the design naturally unfolds. After
necessary data objects and access functions have been defined, upper level programming components can be easily implemented as sequences of object accesses.

Improved reliability is a consequence of error localization within object access functions. In classical structured programming, or structured analysis and design, data accesses are dispersed throughout program components. When a data layout is revised, all components must be studied for required changes. Bugs will appear if these are not all consistently updated.

Data objects and access functions are often reusable as "building blocks" in a diversity of programming endeavors. Over time as object access functions increase in sophistication and reliability, new components and projects can often be quickly implemented.

Object-oriented programming example
I have developed a detailed object-oriented C example, which implements a "topological sort". Space constraints do not permit the example to be included here. However, the reader can contact the author to obtain a royalty-free copy of example materials.

DEVELOPMENT AIDS
The initiate to mainframe C development will quickly realize that some important productivity aids are absent. This topic deserves a lot of attention, but space constraints prohibit this. Thus, I will focus on utilities used often, or those greatly needed. Many were derived from previous specifications.

Before delving into specific aids, I would like to first describe various conceptual notions. These will help the reader understand how programming large C efforts differs from tinier standalone C projects.

Version incompatibilities and data area changes
Within several days of launching a large C software development project, one must come to grips with three major difficulties:

1. how to avoid using down-level data area definitions (headers), and subroutines,
2. how to localize the impact of data area changes on referencing program logic,
3. how to assure proper arguments are supplied to subroutines.

"make"
UNIX_RTM and PC environments have a special tool called "make" which helps address the first problem. Even though this tool is unavailable in the mainframe environment, I believe that it would be inadequate for large scale projects. "Make" is able to ascertain via time stamps in source, object, and module files which programs and subroutines require re-compilation and link-edit. In a large scale project, adding a field to a data area description can spawn 100's of re-compilations. Whether or not these run in foreground or are submitted as batch jobs, the developer must wait a long time for this process to complete.

"lint"
UNIX_RTM and PC developers use a tool called "lint" to address the third problem. This is able to assure that subroutine invocations:

1. pass the proper number of arguments
2. have argument types which match those in the subroutine definition
3. have a consistent subroutine return type

Lacking "make" and "lint" I approached the above issues in three ways. First, an object-oriented approach localizes data area access to private subroutines. Second, I compile all subroutines into one very large module. Third, I access the large subroutine module from small main modules using an inter-module vectoring technique. Object-oriented programming was described extensively above. Now, the latter techniques will be described.

Version avoidance by mass compilation

There are four fundamental ways to resolve subroutine references.
1. Mass compilation
2. Separate compilation into link-time libraries
3. Separate compilation into run-time libraries, for dynamic link resolution
4. Subroutine sharing by inter-module vectoring

Some may prefer to maintain a tangled web of subroutine dependency linkages and generate 100's of compilations. It is far easier and more reassuring to submit a single job which recompiles them all in one fell swoop. No routines are down-level if all are re-compiled. No time is wasted debugging problems which arise when subroutines are not re-compiled after a dependent data area definition is changed, but the corresponding "make file" changes are overlooked!

Subroutine sharing by inter-module vectoring

An inter-module vectoring technique is used to access the subroutines in the massive module from small separately compiled and linked main routines. The subroutines are defined in the main routines using standard C function prototypes definitions in the prolog section of the program. All references are resolved to entries in a vector table, located within a single assembler program called VECTOR. This program locates a global data area to find a second companion vector table in the massive subroutine load module. VECTOR then transparently transfers control through the vector table to the corresponding C subroutine. When the C subroutine completes, control automatically reverts to the appropriate point in the calling program. To work in a multitasking environment, the massive subroutine module must be reentrant.

I feel that I have adequately dealt with the first two problems described above. The third problem is partially addressed by the object-oriented style of programming; whereby subroutines are quite formally defined. Yet, I still debug subroutine linkage errors, which could have been avoided by a "lint" utility.

Available tools

REXX
REXX is a flexible procedural language which can do double-duty as a C preprocessing language. For us non-VHers, REXX/PC can be used to generate programs in a PC environment, for uploading to MVS.

Available ISPF tools

Most of us, and even some VM addicts, are familiar with the range of tools available in IBM's ISPF product, which is a good base for SAS/C-VM software development. Version 2.3 offers several additional features which are worth extra attention.
The ISPF editor is described as a robust editor complemented by a powerful macro facility, the initial spark which ignited ISPF's success.

Library Management Facility (LMF)
For large projects, when multiple developers revise the same source files, life is too hard unless LMF is used.

File comparison (Version 2.3)
Performs excellent file comparison without the presence of sequence numbers. A boon for comparing new vs. old versions of C programs.

Concatenated file search (Version 2.3)
Searches concatenated PDS libraries for source strings. The utility is not sensitive to character case, and complex searches involving multiple strings can be conducted.

Unavailable UNIX_RTM tools
C programming was born in the UNIX_RTM environment. Over years of progress a number of excellent tools were devised. All of these are not essential in the mainframe environment (we have others), but some remain sorely needed. Due to space constraints, only the names of missing tools are listed. To find out more about these identities refer to a UNIX System V publication. The more important missing tools are: awk, lex, lint, make, sccs, and yacc.

Other tools and techniques
So much for UNIX_RTM, now I will describe some utilities I prepared on an as-needed basis, and use regularly.

Port
I prepared the port program to move entire source libraries downward from the host to the PC, and library members upwards from the PC. In the process, various source translations are performed. System using some preprocessing of course.

Helper/heek
My programs generally run in an independent environment which does not use run-time libraries. I encountered difficulties locating the hexadecimal offsets of fields in large data areas. I circumvented this by developing an independent data area generation and diagnosis facility. Driven by symbolic dictionary input, it generates an output dictionary for run-time diagnosis, data area layouts similar to those which appear in RMS debugging handbooks, and the C header files as well.

Cross-reference facilities
Large environments can have intricate relationships between programs and data areas, which are difficult for new project members to comprehend. Cross-reference facilities can help considerably.

Inter-language data format generation/conversion (C-Asm)
Sometimes the same data area is accessed from both assembler and C programs. Rather than maintain consistency of the definitions by hand, C header and assembler DSECT definitions are prepared from a simpler source form. There are a number of variations on this theme, especially when starting with preexisting DSECT definitions delivered by another vendor.

Relational development assistance
Many facts regarding programs and data are best stored and queried in relational databases. For example, often I need to know what is a function's 3rd argument and data type. Or, what data areas are used by each
On a smaller scale, Turbo-Prolog or an equivalent can be useful for many relational purposes.

Trial with PC compilers
Sometimes I encounter C programming intricacies with border-line correctness. The recent availability of quick PC compilers (ie. Turbo-C or Quick C) allows speedy concept validation, before casting volumes of code improperly.

DEBUGGING AIDS AND TECHNIQUES
It is one thing to write C programs, it is yet another to debug them. As a friend of mine observed: unlike accounting, when programs are 99.99% correct they are wrong. C programming is not an exception to this rule. The best advice I can give about debugging C programs follows:
- write programs as often as possible
- debug them immediately after writing them
- keep a log of unusual C quirks
- keep a separate log of C portability quirks
- never give up

The C learning curve is a long, upwards slope. The more C programs you write and debug, the better you will become at writing and debugging future programs.

Debugging approaches

Printf's
In the old days, the only way to debug C programs was to use many printf's. This method remains occasionally suitable, especially in smaller programs. But, now many additional tools and techniques have become available.

Constraint checking
Assuring data values are proper before and after subroutine calls is an excellent way to debug. In the object-oriented section above I described how constraint checks are easily added, without introducing overhead in the production version of a product.

Symbolic debugger
We should all praise the availability of the SAS/C_TM symbolic debugger. This is a giant leap forward in C program debugging. Like the symbolic debuggers available in PC environments, we can now observe instruction flow at the C source level, and reference data values (and structures and arrays) using C source syntax.

TSO test
TSO test is an old standby greatly in need of upgrading. I use it extensively to debug "independent" modules, which are off-limits for the symbolic debugger. The basic technique is to trap subroutine calls and study arguments and returned results. C object code between subroutine calls is virtually indiscernible. SAS provides an object module disassembler (ODD), which helps. But, ODD listings of large modules require excessive compilation time, and consume beaucoup of spool or disk space.

Diagnostic trail with data area snapshots
One of the most useful techniques that I use produces voluminous diagnostics during the course of operation. These may not be useful in every session. However, when problems occur, the trail often has key indicators of where things went astray. My diagnostics include data area formatting similar to system dumps.

Abend recovery
I use a single program for abend recovery in all tasks. When an ESTAE exit receives control, it checks minimal conditions, copies the SDWA to a
pre-allocated area, and then immediately attempts to enter the retry routine. The retry routine formats:

- the copied SDVA save area back-chains at the time of the abend and diagnostic information

If the abend occurred in a program which received control via the LINK system service, it may no longer be present when the retry routine receives control. Critical modules are pre-loaded to increase their "use count". This assures that they can be referenced by the diagnostic routines. Furthermore, before any data area is formatted, its presence is assured by using SPIE or ESPIE, in non-XA and XA environments respectively. Once formatting is complete, the retry routine gracefully returns to the upper-level caller. There is no attempt to proceed onward.

Diagnostic services
I use standard sequential services in my diagnostic routines. But, there are some restrictions which must be observed in a multitasking environment. First, multiple tasks can attempt to record diagnostics at the same time. These must be serialized; I use ENQ and DEQ. Furthermore, the task which initiates an I/O request must be the task which "checks" I/O completion status. This requires avoidance of exchange buffering techniques.

PC debugging of code fragments
Often it is quicker to debug mainframe C programs with PC debugging facilities, if these will fit within PC constraints. Alternatively, fragments of mainframe programs can be independently debugged, if a suitable driver can be quickly concocted.

Multitask debugging
Programming for a multitasking environment is not very different than a stand-alone environment. But, debugging the original set of services is definitely a challenge, especially when using TSO Test. TSO Test has undocumented deficiencies in a multitasking environment. Specifically, if a breakpoint is reached in one task, subsequent breakpoints may not be acknowledged. The task requiring service of the subsequent breakpoint is not allowed to continue. If this is the master task, it is removed - possibly with your TSO session to boot. I was able to circumvent the problem by developing special abend diagnostic software, and assuring that only a single breakpoint would arise at any given time.

The XA version of TSO Test is superior to the non-XA version. Its functionality is more durable. It can also execute client's test commands at any point in the debugging session; useful for debugging unexpected hangs.

My company has also acquired a license for Yale's Debugging Controller (DBC). Others in the company perceive it as far superior to TSO Test for debugging multitask and authorized program environments. Anyone suffering through the throes of "real" program debugging should give DBC a hard look.

Large data area field location
Analyzing large C data structures within storage dumps, TSO test, or DBC displays is counter-productive. I hit my head against this rock too long. Finally, I came up with a smarter approach of developing a data area layout formatter; the
PORTABILITY

The "GOAL"

Many have talked of portability in the past. I have a simple way of defining the "goal". It should be possible to develop and test a C component with any C compiler, ship it across a gateway, and recompile and execute it cleanly: without a source change! The entire range of facilities outlined in the ANSI library definitions should be available, and perform as advertised. The generated modules should be able to execute "standalone" without run-time library dependencies. Any C developer should be able to nonchalantly complete standard C training and begin developing "any" project, without having to learn a new "library" of services.

Portability barriers

There are three fundamental "barriers": language divergence, library interface differences, and environment.

C language standardization

The core of C has remained quite resilient over the years. Nearly all compilers are equal in transforming a pure K&R source program into a working executable module. ANSI has drafted a standard set of extensions to the K&R core. These are also accepted by the majority of compilers as proper programs. Each vendor tries to voyage beyond these bounds. The reader is advised to stick with pure K&R, and refrain from all extensions as much as possible. Though new ANSI function prototypes and enumerations are enticing, these may not be properly accepted by some compilers. Pragmatically, "pragmas" should be sidestepped as well.

Suspicious language extensions

Those living near Seattle have recently brought alternative execution memory models and four(4) pointer types, where before there was one. We must now learn to respect the following definition of a faraway function which locates a distant integer:

```
int far %pascal far func(void);
```  

Before "int *func();" sufficed. The above can be portably coded using conditional defines of "far" and "pascal" to nothingness.

There is also the ability to concoct a fastest pointer with 20-bits of addressability from two 16-bit segment and offset values, if that is what you really want to do.

Library differences

Over the past 5 years, considerable divergence of C libraries has been witnessed. We hope that IBM's OS/2 and SAA digressions do not further exacerbate this trend. Otherwise, C developers may be pigeon-holed into incompatible dialects, much akin to the original Tower of Babel.

SAS has striven hard and diligently to buck this trend. They have bent hard to the task of cooperating with Lattice to provide common libraries for both the PC and mainframe environments. They deserve much credit for their efforts.

I have prepared comparisons of subroutine libraries offered by: SAS/C_TN, SAS/C_TN/INDEP, UNII_RTN System V, ANSI, SII (IBM/C), Borland, and Microsoft. The divergence is significant enough to alert all readers. There seem to be three fundamental arteries of
dispersion: ANST, UNIX RTM, and Microsoft. IBM, SAS, and Lattice remain close to ANSI library standards. Everyone is moving fast beyond what ANSI has defined in communications, database, and presentation management (graphics). The comparisons are voluminous and so could not be included in this paper. These can be obtained by contacting the author.

Environment differences
Writing C programs for different environments is as difficult as programming in different languages. Today a C program may be ported between MV5, MV5/IA, V5, V5/SP, V5/IA, UNIX RTM, Tenex, PC-BOSS, OS2, Macintosh, and the list goes on. Considerable standardization remains to improve "data", "address", and "display" portability. Exchanging data between systems is almost straightforward, provided all information is delivered in printable ASCII string format. Character transfer within internal data areas is another matter. In the IBM mainframe world, bytes are counted left to right, not so in the Intel (IBM-PC/PS2) world. "Even" and "odd" characters occupy high- and low-order bytes of 16-bit words respectively. For example, can you read:

(AC NOT UERDAT IN SRCVO EMA DONCYE?E).
At least you might be able to "swab" the above, should this C service is available.

Surprisingly, exchanging integers can also be a challenge. First there are numerous kinds of integers: short, int, and long with signed and unsigned distinctions. At least all hardware vendors access the bits within these consistently right-to-left (they may be counted differently), but negative numbers may be represented in either one's- or two's-complement. And, the lengths of these are environment-specific. The only integer which is easily exchanged is "unsigned long"; with the exception of those machines with 16-bit words. As an extra test, how can "unknown" integer values be passed when all values are meaningful?

What about exchanging floating point numbers? This is clearly an exercise for the reader and not the author to solve.

Independent compilation vs. run-time libraries
Run-time libraries are a nuisance which should NEVER have been conceived. Not only must EACH USER revise the "path"/"bin"/STEPLIB/minidisk-s to access these, but also the level of these libraries must be INTERFACE-STAMP-COUPLING with those used by the original developer. The SAS/C_TM compiler at least provides the ability to compile independent modules, at the sake of over 50% of XTR/ANSI library services. I have personally built 80% of the missing services I need the most.

Portability frontiers
Some frontiers remain unstandardized, for example:

Display control
There is a plethora of alternative toolkits for composing meaning, windowing, and graphics front-ends. Furthermore, consider the variations in devices to control:

Glass teletypes, dumb terminals, monochrome, multiple gray scale, CGA (color char/color raster), EGA (color raster), VGA (color analog). Some day expect monochrome/gray/color vector.

Data management
Standards are lacking for:

- "keyed" and some "direct" file architectures
- sharing files in a multi-user environment
- controlling user/program access privileges
- SQL/SEQUEL/QBC interfacings

Operating system interfacing

There have been few efforts to define standard interfaces to operating system services. Indeed, it is not easy to standardize across the many systems described previously (MVS, MVS/ESA, VM, etc.). AT&T has published UNIX RTM System V standards. SAS has published a document defining a "Generalized Operating System (GOS)" interface. IBM made no steps in this direction with its SAA architecture.

Inter-system data exchange

How does one connect two C programs running on different host processors? How about with heterogenous architectures? Is the communication synchronous? What if the clock ceases on the other system? Is anybody out there?

IBM is trying to reign in these frontiers with their strategic SAA architecture. Other vendors are not so eager to comply. Certainly, UNIX RTM standards should be clearly monitored as well.

Looking SAA arrival

IBM recently woke up to new economic challenges. If Digital has it, IBM will. And, they can back this claim with orderable documentation; and lots of it. IBM document SC26-3353 is the SAA Common Programming Interface C Reference publication. The language described and subroutine libraries are very closely tied with the Proposed ANSI Draft Standard; with differences so slight as to be insignificant. IBM claims they will have compatible compilers in MVS, VM, S/3X, and OS2 environments. Yet, the SAA document recognizes C/2 as the only licensed program IBM supports. Their MVS and VM program offerings (5773-AAC and 5773-AAB) are flagged as available "to gain early experience with the C interface" (sic).

Other SAA documents define many more C interfacing considerations. These go far beyond ANSI C standards. For example, the database reference describes how embedded SQL statements are transformed in C programs prior to compilation. Clearly these specifications significantly affect C portability considerations.

SUMMARY

This paper has addressed object-oriented programming in C, development and debugging aids, and portability issues. These should be helpful for all those getting started with the SAS/C TM compiler.

I believe C stands at a cross-roads faced by many languages in the past. Many of my associates are now eager to start programming in C. There is much entrepreneurial pressure encouraging advancement beyond solid ground. Evidence of this can clearly be seen in the incompatibilities between various C subroutine library packages. Should this fraying continue, even though we share the language stem (C), the net result will be as many separate languages as those which arose from Latin. I hope we have the collective patience to allow necessary standards to develop, so that we can proceed coherently onwards.