

We have read Making the Fortran-to-C Transition: How Painful Is It Really? by Theurich *et al.* (*Computing in Science & Engineering*, Vol. 3, No. 1, p. 21) with interest. We believe the discussion presented there is incomplete and we are prompted to comment. Today's ubiquitous uses of computers, by commercial enterprises and private persons using commercially developed software, are the majority of all uses. The operations of these applications require the means to access databases, windows and networks. More traditional uses of computers, including numerical simulations by scientists, engineers and economists, are now a minority of all uses. The operations of numerical simulations require many floating point operations on ordered values. These calculations are well expressed by Fortran. For other uses of computers, or for employment as a professional programmer making commercial applications, other languages may be more advantageous. That there are very diverse uses of computers is, perhaps, obscured by the use of the same commodity hardware to run very diverse software.

We were surprised to read that it may be considered "unethical" to teach Fortran to science (or engineering or economics) undergraduates. If one considers the purpose of the technical undergraduate education to be a step in preparing the next generation of researchers, we argue that it is irresponsible **not** to teach Fortran. If one considers the undergraduate technical education to be simply a good Liberal Arts education, then one is free to teach whatever programming language(s) one chooses. In the latter case, we argue that HTML or SQL are at least as important, and probably more important, than C, C++ or Fortran. If a research or engineering program has a large investment in Fortran code, then the students should learn Fortran. If a student doesn't want to learn Fortran, then perhaps the student isn't interested in numeric computation, but would rather learn to use computers for other, perhaps commercial purposes or would rather not study programming at all.

Theurich *et al.* reported that their students have difficulty with Fortran's argument passing mechanism. Since Fortran's argument passing mechanism always appears the same to the programmer, where C's argument passing mechanism varies by the kind of argument passed (and C++'s mechanism is more complex still), we wonder why they chose to regard Fortran's simpler mechanism as the cause of the difficulty. Use of Fortran's argument intents catches many, if not all, of the bugs Theurich *et al.* found hard to locate, provides greater clarity of expression, and encourages greater optimization by the compiler. We note that the next Fortran standard will support the interoperability of C and Fortran procedures, thus differences in argument passing may be clearly expressed in the code in a portable way and handled by the compiler (as is appropriate for such low-level detail). We wonder how efficiently the C argument passing mechanism will operate in the ever more complex memory hierarchies of modern SMP and DMP computers. Fortran's mechanism allows greater flexibility to the compiler for implementation in the variety of architectures on which modern numerical simulation programs are, and will be, expected to perform well.

We agree with Theurich *et al.* that the public domain f2c tool is inappropriate for making a language conversion. We agree that line-by-line conversions by tools which need the support of proprietary libraries are inappropriate for software where the source is to be freely shared. Still, the course of action followed by Theurich *et al.* puzzles us. If the automatic tools are not useful, why did Theurich *et al.* follow a strategy of making a conversion by hand? Further, in their discussion of the manual conversion of their program's memory management, we don't understand the finding of "fundamental differences"

between the memory management of C (malloc/free) (or C++ (new/delete)), and Fortran (allocate/deallocate). We think Fortran's memory management scheme (with distinct allocatable, pointer and target attributes) is safer, simpler, provides greater clarity of expression and encourages greater optimization by the compiler.

Theurich *et al.* stated that a "plethora" of code maintenance tools is available for C, many of them free. We agree. Similar tools are also widely and freely available for Fortran. For example, formatting and other maintenance chores may be handled by the free Moware suite (<http://www.ifremer.fr/ditigo/molagnon/fortran90/contenu.html>). Fortran 95 compilers have no need for a code checker, such as lint, since they are required to perform many, and in practice perform all, of these checks themselves with each compile. Fortran 95 compilers support, in practice, a strict superset of previous standards, so a modern Fortran compiler itself would have found all the issues mentioned by Theurich *et al.*, even if their Fortran code is written to a previous standard. Conversion of fixed format sources to free format is certainly desirable, these conversions may be automated by free tools such as `convert` (<ftp://ftp.numerical.rl.ac.uk/pub/MandR/convert.f90>) and `to_f90` (http://www.ozemail.com.au/~milleraj/misc/to_f90.f90). All the tools described here are freely available as Fortran source code so they can be available on every platform. We also note many tools for C and Fortran code maintenance are available for sale.

Why didn't Theurich *et al.* modernize their Fortran code? They could have used Fortran modules, with public and private entities, to implement data hiding. They could have used Fortran derived types, associated generic procedures and operator overloading, to implement information abstraction. They could have used argument intent (mentioned above) to improve compile time error checking and run time efficiency. These features may be added to the existing Fortran code incrementally, as the programmer learns the new features of modern Fortran, with systematic testing as an integral part of the effort. This strategy maintains the high confidence that a large code may have earned over years of use in a variety of circumstances. New ideas may be implemented quickly as clearly expressed modifications localized in the appropriate module. The Fortran module and use mechanism (including the `only` and `rename` features) is, in our view, clearly superior for these purposes to the C header file and `include` mechanism.

Unlike Theurich *et al.*, we do not regard the lack of a single-precision math library or the lack of an intrinsic complex type as shortcomings of C. For the purposes for which C is well suited, these numeric amenities are unnecessary. Simply put, the demands of C's customary applications have not given C compiler vendors sufficient reason to supply these features. We note that full language support for a complex type corresponding to each available precision of real type would imply a significant enhancement to C.

We were glad to read that, after several months of hand tuning their C code, Theurich *et al.* were able to beat, slightly, the performance of their original Fortran application. We wonder how great a performance increase they would have achieved if they had spent that time tuning their Fortran. We wonder how much of this performance will be preserved, with how much effort, as they move to C++. Theurich *et al.* commented that they eliminated several unneeded procedure arguments, variables and array copy

operations. Clearly, these observations merely indicate that the Fortran code needed some maintenance, which may be true of any sizable code which has grown in scope as it is used and modified. The elimination of any significant amount of unneeded array copying, on the cache based computers used, explains to us the performance gained by months of hand tuning, as Theurich *et al.* conceded.

We note that Theurich *et al.* used commercial "high performance" C compilers to obtain the performance eventually achieved by their application. We wonder which Fortran compilers were used to make the comparison. We note that the time spent hand tuning the application for the C compilers is of the same approximate duration as the time between major releases of commercial compilers. We wonder how much of this hand tuning is either unnecessary with a newer release of the C compiler, or must be redone to meet the demands of a different C compiler (perhaps on hardware where use is unanticipated when the initial hand tuning is done).

Pursuing portability issues further, the high-level Fortran kind mechanism for selecting the precision of variables is more expressive and, in our view, clearly superior to the usual low-level C (and C++) methods of conditional compilation of typedefs and/or macros. Indeed, we wonder how much the C code of Theurich *et al.* now relies on conditional compilation for portability, including compiler-specific performance tuning, where their Fortran code did not. We wonder, will their application now port to computers yet unknown with the least effort, the greatest confidence and, without further hand tuning, the highest performance? Can new modifications be made to work quickly and reliably through all paths of the conditional compilation scheme? Theurich *et al.* did not report their experiences with these issues. They did not discuss source code portability at all.

The manual use of the "restrict" keyword is tedious and error prone, the use of a compiler option to automatically place a restrict keyword on all pointers, which Theurich *et al.* found necessary to gain high performance, is extremely risky. It is all too easy to find situations where the restrict keyword is valid for most input datasets but not for unusual edge cases, or valid on some hardware but not on others (for example, when switching between DMP and SMP, pointers which were not aliased may become aliased when separate arrays in distinct address spaces become parts of one larger array in a shared address space. But then, Theurich *et al.* did not discuss parallelization at all.) Of course, one can always claim that the programmer will be aware of, and react appropriately to, all these instances and be able to do so without affecting performance. But as memory hierarchies become more complex this claim becomes more dubious as the burden of low-level analysis by the programmer becomes greater. Experience with compiler directives on flat-memory vector computers has taught us that such general assertions are hard to make reliably without great study of the whole code, and that mistakes can lead to subtle, hard to find bugs particularly if a code is developed over time by several researchers, each familiar with only a portion of the whole code.

We tend to agree with Theurich *et al.* that much of the performance debate misses the point. Performance of the resulting code is only one factor among several which must be considered when deciding upon a strategy of computation. Portability and clarity of expression are other important factors. We find Fortran to be more portable and more clearly expressive of numerical problems than C or C++. We also find Fortran to be a safer programming language, especially when several researchers are contributing to the

overall code, and the code is evolving as a research program advances.

We are curious as to why Theurich *et al.* concluded that commercial competition will allow further optimizations for C++ compilers, but not for Fortran compilers. Indeed, we conclude that just the opposite is true. Most vendors who offer both Fortran and C++ compilers (including The Portland Group and MIPSPro, whose compilers Theurich *et al.* reported using) use a common code generator technology. Thus, any optimizations present in code generation are always available to both languages. The question becomes which language can take best advantage of the optimizations made available by the common code generator. The answer will always be Fortran. In fact, it is difficult for us to think of an optimization in the code generator for which C++ would benefit but Fortran would not. Indeed, Fortran can take advantage of many optimizations more aggressively than C++. With Fortran, optimizers benefit greatly, for example, from the knowledge that procedure arguments must represent distinct entities (unless they are targets or pointers), this allows powerful dependency analysis, dead code elimination, common subexpression elimination, and keeping variables and subscript expressions in registers. These are optimizations associated with large loops containing many array references, as is typical of programming numerical simulations. Perhaps there is little competitive need in the C++ marketplace for these rather specialized optimizations. Fortran compilers, traditionally, do compete on the basis of high performance of the resulting executables.

The most competitive market for compilers today is probably the largest market for applications, the Microsoft Windows on Intel market. We checked the display advertising in the latest edition (15.6b) of the Programmer's Paradise catalog. We note three C++ compilers (Microsoft, IBM and Borland) represented by display advertisements. Not one of them advertised that its code generation technology produced superior results either in speed of execution or in size of the resultant object. What they did advertise was reduced time to market for new applications due to the number of predefined classes supplied (allowing the use of various protocols to access networks, windows, and databases), ease of debugging including remote debugging, and ease of collaboration in a team programming environment. It seems clear to us that the separate markets for numerical and commercial computing are such that the reasonable expectation of numeric programmers (that compilers will compete on the basis of the optimization of the generated code) just doesn't hold true in the commercial market. To find the word "optimization" in a display advertisement for a compiler in the Programmer's Paradise catalog, one must check a Fortran advertisement (Compaq).

We are aware of several C++ compiler vendors for the Microsoft Windows on Intel platform and slightly more Fortran compiler vendors. Many of these Fortran vendors also supply the Linux on Intel market, which the C++ vendors, generally, do not. In the Free Software marketplace, the g95 project is alive and well (<http://g95.sourceforge.net>), although it is admittedly about two years away from being released for public use.

We find ourselves in complete agreement with Theurich *et al.* with regards to their sympathy for the Free Software Foundation and their enthusiasm for the tools which have emerged therefrom. We further agree that researchers will mutually benefit by the free sharing of codes. But we don't see why Theurich *et al.*

seemed to think the Gnu GPL (or LGPL) must be associated with something written in, or is intended for use solely with, or even that it exists solely because of, C or C++. Indeed, we have seen many works, from Ada and Fortran compilers and tools, to as far afield from computing as scripts for dealing with telemarketers (<http://www.junkbusters.com>) published under the GPL. One of us (TM) maintains the g77 compiler http://gcc.gnu.org/onlinedocs/g77_news.html, while the other (DN) maintains a web site (<http://users.erols.com/dnagle>) where software is available for free download under the terms of the GPL and LGPL (most of it written in Fortran). Thus, we both fully utilize the GPL and utilize it with regards to Fortran.

In two years time, the g95 project will be approaching the time of its public release, and the Fortran 2000 standard will be approaching the time of its publication. Compiler vendors are already starting to implement the new features of Fortran 2000 (the allocatable components T.R. and the IEEE arithmetic T.R.). A scientist, engineer or economist can now incrementally learn the new features of Fortran 95 while keeping the advantages of their current code, and be ready to adopt the full object oriented features (class inheritance, polymorphic pointers, type bound procedures) of Fortran 2000.

In summary, we are not persuaded by Theurich *et al.* either that there was a need to make a Fortran to C to C++ transition, or that their not yet completed transition is not, in fact, rather painful.

Dan Nagle

<mailto:dnagle@erols.com> – phoneto: 703 471 7968

12142 Purple Sage Ct., Reston, VA 20194-5621 USA

<http://users.erols.com/dnagle>

Toon Moene

<mailto:toon@moene.indiv.nluug.nl> - phoneto: +31 346 214290

Saturnushof 14, 3738 XG Maartensdijk, The Netherlands

Maintainer, GNU Fortran 77: http://gcc.gnu.org/onlinedocs/g77_news.html

Join GNU Fortran 95: <http://g95.sourceforge.net/> (under construction)

We thank Richard Hendrickson for his constructive comments of a draft of this paper.