WHY FORTRAN?

By Viktor K. Decyk, Charles D. Norton, and Henry J. Gardner

AMONG MANY COMPUTER SCIENTISTS, FORTRAN IS AN F-WORD. YET, IT'S STILL THE MOST WIDELY USED LANGUAGE IN SCIENTIFIC COMPUTING, ESPECIALLY WHEN HIGH PERFORMANCE IS REQUIRED. WHY IS THIS SO?

One explanation often given is the huge amount of scientific legacy code in the world—after all, differential equations remain the same over time and so do their solvers, so there's no reason to rewrite such code. But a great deal of new code is written in Fortran95 as well. One of us recently served on a review panel for granting computer time to high-impact scientific computing applications that effectively use thousands of processors, and every single one of the applications he reviewed was written in Fortran. At last year's conference on computational physics in South Korea (CCP2006), most of the plenary speakers who talked about codes used Fortran. Perhaps scientists prefer Fortran because they're productive when using it.

The Rise of Fortran95

One reason developers write new scientific applications in Fortran95 is that it's a natural language for expressing science and engineering ideas. Computer scientists who disparage Fortran are either thinking of Fortran66 or Fortran77, or they're just repeating what someone else has told them. Very few of them are well versed in the details of Fortran95, which is a modern, object-based language that's backward-compatible with Fortran77.

Fortran95 supports most C++ or Java features with the notable exceptions of inheritance and dynamic polymorphism. In fact, Fortran95 arrays act like objects that contain information about size, dimension, and stride; array syntax permits powerful and expressive operations on these "objects." Pointers support complex structures such as linked lists, but alternatives such as allocatable and automatic arrays are available for simple access to dynamic memory. The developer can create classes with the `module` construct, which strictly enforces type-checking and encapsulation. All types are then resolved at compile time—runtime typing errors can't occur, so the debugging process speeds up. In short, Fortran95 isn't your grandfather's FORTRAN!

A second reason for the scientific community's preference for Fortran is performance. Fortran compilers are very mature. Many compiler optimization techniques rely on safely reordering instructions to take advantage of processor features. A rather simple, primitive language by today's standards, Fortran77 is relatively easy for compilers to analyze. All array addressing in Fortran77 occurs via array indices; in languages such as C, direct pointer arithmetic and the possibility that different arguments can refer to the same location (pointer aliasing) greatly complicate code analysis for compilers. As a result, safe code reordering is much easier to determine in Fortran77. (This isn't to say that optimized code isn't possible in other languages, but you must be much more of an expert to write it.)

Although Fortran95 introduces some features that might degrade performance, it's designed to retain much of Fortran77's high-performance advantage: you can still write performance-critical, low-level subroutines in Fortran77 (to help the compiler optimize) and then call them from Fortran95. This is extremely easy because the two languages are compatible.

Beneficial Features

Fortran's object-based features are most useful when writing large, complex applications involving multiple authors. As computers have grown more powerful, scientific software has also become more ambitious and complex. In earlier years, most scientific codes were written by individuals for their own research; collaborative scientific software is a relatively recent development. Although Fortran95 doesn't support inheritance or dynamic polymorphism, developers can emulate these features in software,[1] if needed. *Inheritance* relies on a family of data structures in which the children contain their parents' data, much like Russian matryoshka dolls. Such structures aren't as widely needed in scientific application as in other

 Copublished by the IEEE CS and the AIP 1521-9615/07/$25.00 © 2007 IEEE

John Backus and the Business of Scientific Programming

By David Alan Grier

As a child, John Backus was unconcerned with either time or money. Born into a wealthy Delaware family, he was a restless youth with no clear focus or direction. Though he was able to gain admission to the University of Virginia, he did so badly in his studies that he was forced to leave the school in 1943 and spent the remainder of the war as a draftee in the army. He gained discipline only after he returned from the war and enrolled in a mathematics program at Columbia University. Mathematics led him to IBM. IBM introduced him to computer programming, and computer programming led him to think about the business of science.

By 1950, when Backus joined IBM, programming was becoming a serious economic problem. "The expense of operating a computing installation," he observed, "is almost equally divided between machine costs and personnel cost."[1] Programmers had few tools to assist them in their work, and most wrote their programs either in the basic codes that controlled the machine or with an assembly language that substituted simple names for those codes. IBM had one of the few more advanced programming tools, a system called Speedcode.

Speedcode consisted of an elementary language and a special set of routines to handle scientific (floating-point) calculations, but it created bloated and inefficient programs. Backus defended the system by conceding the problems with the final code and then arguing that "Speedcoding reduces [overall] coding and testing time considerably," and hence, "it will often be the more economical way of solving the problem."[1]

In 1953, Backus proposed that the company create an alternative that could translate mathematical formulae into computer code, a project that quickly acquired the name Fortran. He felt the system would have to have a more expressive language than that of Speedcode and be more efficient. If the "object program [was] only half as fast as its hand coded counterpart, then acceptance of our system would be in serious danger," he conceded.[1]

The problem of producing efficient code wasn't easily solved. Backus's Fortran compiler was a complicated system that divided the source program into its fundamental units, analyzed the connections between these units, and, from this analysis, tried to create code that would make good use of the machine. "We were often astonished at the surprising transformation in the indexing operations and in the arrangement of the computation which the compiler made," Backus recalled. "We would not have thought to make [such changes] as programmers ourselves."[1]

continued on p. 70

areas of computing, an observation that computer scientists don't always appreciate. *Dynamic polymorphism* describes the ability to use a single name to refer to a family of types and procedures—such polymorphic types and procedures are examples of *abstractions*. (It's unfortunate that C++ bundles the concepts of inheritance and polymorphism together. Java does not.) In scientific applications, we've found that polymorphic procedures are more useful than inheritance; in Fortran, they're emulated with a case statement in a controlling procedure.

Computer scientists have gone beyond object-oriented programming to consider how classes can be organized into more abstract units. Specifically, these language-independent abstractions, called *design patterns*,[2] organize classes to solve recurring programming problems. A guiding principle of design patterns is to avoid inheritance in favor of object composition, which is fortunate for languages that naturally support the latter rather than the former. Although most design patterns were first formed in nonscientific contexts, some are useful for science, and all three of us—the authors of this article—have effectively implemented them in our own scientific codes.[3] The basic idea behind design patterns is to encapsulate, in one place, the variation and control of some important code capability. Developers can then use the resulting polymorphic procedures without making other procedures aware of those choices.

In general, Fortran95 adopts new ideas conservatively, after they've proved themselves in use by trying to maintain the high performance that's so important to scientific calculations. Fortran's competition comes from high-level, sometimes interactive, languages such as Matlab, Python, or Mathematica, rather than from C++ or Java. This is natural: as computers become more powerful, some of the problems that originally required high performance no longer do. (The Macintosh on your desktop is more powerful than the Cray C90 you used to use.) When this happens, the human interface becomes much more important than performance. However, a great many problems of interest in science and engineering won't even be solved with exaflop computers.

The Next Step

Fortran continues to evolve. The current standard, Fortran2003,[4] introduces full object-oriented support, standardized interoperability with C, and procedure pointers, among other features, but compilers have been slow to appear. Fortran programmers are at the frontier of writing massively parallel applications, so they're attempting to enhance Fortran itself with features to support parallel com-

continued from p. 69

When Fortran was released in April 1957, it found quick acceptance among users of the IBM 704, the company's big scientific processor. At every site, a few individuals claimed that they could write better code in assembly language, but the majority found Fortran to be simpler and adequately efficient. By 1960, other vendors had started to write their own Fortran compilers; by 1964, the American Standards Association had created a common definition of the language. Fifty years after its creation, Fortran is still employed for scientific computation and is one of the oldest computer artifacts still in use.

When Backus reviewed his creation, he concluded that the language's initial goals remained unfulfilled. The "plain fact is that few languages make programming sufficiently cheaper or more reliable," he wrote in 1977. "There is a desperate need for a powerful methodology to help us think about programs and no conventional language even begins to meet that need."[2] These claims were a bit of an overstatement. Certainly, few people believed that the software of 1977 could have been produced with only the tools of machine code and assembly languages. Fewer still would accept such an opinion in 2007. Nonetheless, Backus's statements point to a fundamental motivation for the development of computer languages: the desire to make computing less expensive.

References

1. J. Backus, "The IBM 701 Speedcoding System," *J. ACM*, vol. 1, no. 1, 1953, pp. 4–6.
2. J. Backus, "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs," *Comm. ACM*, vol. 21, no. 8, 1978, pp. 613–641.

David Alan Grier is the editor in chief of the *IEEE Annals of the History of Computing* and writes the "In Our Time" column for *Computer* magazine. He's also the associate dean of academic affairs in the Elliott School of International Affairs at George Washington University. Contact him at grier@gwu.edu.

puting. The proposed high-performance Fortran (HPF) standard combined additions to Fortran90 with directives for data distribution, but HPF's performance was generally worse than that obtained from message passing in Fortran90; thus, support for it withered in the US. Nevertheless, work on HPF-J lives on in Japan: the Japanese Earth Simulator has experienced performance with it that's nearly as good as that achieved with message passing.[5] In the meantime, the desire to make parallel computing easier in Fortran hasn't abated in the US, and the Fortran standards committee is currently evaluating co-Array Fortran—a parallel programming paradigm from Cray—for inclusion in Fortran2008.[6]

One concern we often hear is that Fortran is rarely taught to students anymore—any programming course they take generally covers whatever language is in vogue at the moment. Currently, this means Java, but the concerns of computer science and scientific computing rarely overlap, so computational science students who need to learn Fortran must do so on their own. They seem to pick it up quite well, and it helps for computational science students to be "multilingual." Unfortunately, many computational science students don't know how to design complex programs in any language. We think that teaching design patterns in scientific contexts would be a good approach to solving this problem.

In the early days of computing, scientific programming dominated, and FORTRAN was king, but the world has moved on: computers are now mostly used for tasks that have very little to do with science, and other languages now dominate. Nevertheless, scientific computing is still important, especially on high-performance computers, and Fortran still rules this niche for very good reasons.

For perspective, we close with some insightful quotations about Fortran from Bjarne Stroustrup, the designer of C++:[7]

> It would be nice if every kind of numeric software could be written in C++ without loss of efficiency, but unless something can be found that achieves this without compromising the C++ type system, it may be preferable to rely on Fortran... .
>
> …
>
> Fortran is harder to compete with. It has a dedicated following who [...] care little for programming languages or the finer points of computer science. They simply want to get their work done.
>
> …
>
> I see C++ as a language for scientific computation and would like to support such work better than what is currently provided. The real question is not "if?" but "how."
>
> …
>
> C++ was designed to be a systems programming language and a language for applications that had a large systems-like component.
>
> …

I am not among those who think that a single language should be all things to all people.

Acknowledgments

Viktor Decyk and Charles Norton's work was performed, in part, at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with NASA. Decyk was also partly supported by the US Department of Energy, under the SCIDAC program. We acknowledge useful discussions with Michael Metcalf.

References

1. V.K. Decyk, C.D. Norton, and B.K. Szymanski, "How to Express C++ Concepts in Fortran90," *Scientific Programming*, vol. 6, no. 4, 1997, pp. 363–390.
2. E. Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.
3. C.D. Norton et al., "The Transition and Adoption to Modern Programming Concepts for Scientific Computing in Fortran," to be published in *Scientific Programming*, vol. 15, no. 1, 2007.
4. M. Metcalf, J. Reid, and M. Cohen, *Fortran 95/2003 Explained*, Oxford Univ. Press, 2004.
5. H. Sakagami et al., "14.9 TFLOPS Three-Dimensional Fluid Simulation for Fusion Science with HPR on the Earth Simulator," *Proc. IEEE/ACM SC2002 Conf.*, IEEE Press, 2002; http://ieeexplore.ieee.org/ie15/10618/33527/01592887.pdf?tp=&arnumber=1592887&isnumber=33527.
6. R.W. Numrich and J. Reid, "Co-Arrays in the Next Fortran Standard," *ACM Fortran Forum*, vol. 24, no. 2, 2005, p. 4.
7. B. Stroustrup, *The Design and Evolution of C++*, Addison-Wesley, 1994.

Viktor K. Decyk is a research physicist and adjunct professor in the Department of Physics and Astronomy at the University of California, Los Angeles. He's also a senior member of the technical staff at the Jet Propulsion Laboratory at the California Institute of Technology. Decyk has a PhD in physics from the University of California, Los Angeles. Contact him at decyk@physics.ucla.edu.

Charles D. Norton is a principal member of technical staff at the Jet Propulsion Laboratory, California Institute of Technology. His technical interests include high-performance computing applications, model-based validation of precision structures, and instrument payload modeling. Norton has a PhD in computer science from Rensselaer Polytechnic Institute. Contact him at Charles.D.Norton@jpl.nasa.gov.

Henry J. Gardner is an associate professor at the Australian National University. His technical interests include scientific software, e-science, and virtual reality. Gardner has a PhD in theoretical plasma physics from the Australian National University. Contact him at Henry.Gardner@anu.edu.au.