Teaching C & Fortran Simultaneously to Beginning Engineering Students

Harley R. Myler

Department of Electrical & Computer Engineering

University of Central Florida

Orlando, Florida 32816-2450

407.823.5098

hrm@engr.ucf.edu

Abstract

This paper addresses a common concern amongst engineering faculty: Which computer programming language should be taught to engineering students? The University of Central Florida (UCF) has initiated a novel course offering where both C (the modern language of engineering) and Fortran (the foundation language) are taught simultaneously using a problem solving methodology. An interesting contrast is developed between engineering approaches to programming and that of computer science, as well as the importance of proper background in computer languages to the graduating engineer.

Introduction

The teaching of programming skills to all engineering majors has become widespread and interdisciplinary. Increasingly, the question of which language to teach has become an issue and is sometimes hotly debated. Fortran has been giving ground to C, although many senior faculty see this as a problem since Fortran is still the language of choice for many physicists and mathematicians and its use is far more widespread than many realize.

At the University of Central Florida, a novel approach to teaching both Fortran and C simultaneously was initiated. Here C is taught as an implementation language, while a reading knowledge of Fortran is developed at the same time. Few, if any, engineering colleges have the luxury of teaching computer programming to all engineering majors for more than one semester or three credit hours. The suggestion that engineers take a semester of C programming and a semester of Fortran would most likely be met with serious dissent, if not ridicule. As engineers become more specialized, colleges and departments are faced with a shrinking credit hour budget that will not allow waste or inefficiency in the total number of credits required. The program developed takes advantages of similarities between the two languages to generate an economy of presentation. Additionally, the overall course goal of "read Fortran, write C" introduces further efficiency into the program. Finally, the approach to programming is one of *rapid prototyping*, where the engineer is interested in getting *accurate* results quickly and with a minimum of overhead in the way of algorithm elegance or program structural attractiveness. Those aspects of programming are best developed over time and with practice, rather than during critical early stages where the abstraction of the concept of programming a machine must be overcome.

Motivation

The development of the course began five years ago when the author was faced with that occasional predicament of finding a course to teach over the summer session. The only possible option at the time was to teach a college core offering entitled “Engineering Analysis and Computation”, an articulation course targeted at entry-level engineering students who did not take a programming course in High School. The language taught at that time was Fortran, and the book being used was Etter’s excellent text (see references). For many years, Fortran was the first language that engineers were taught and was the first computer language that this author learned, later using it off and on during years in industry and also in consulting. For these reasons the teaching of it was both challenging and refreshing. Nevertheless, the author found himself making refer-

ences to C during the instruction. After a few weeks into the course, some students would ask, "How is that done in C?". This prompted an experiment the following semester where C instruction sequences were introduced formally as an alternative to the Fortran constructs. By the end of the semester, a set of notes had been compiled that illustrated the similarities and contrasts between the languages.

At the time, the faculty of the college had been divided between Fortran and C, with more junior professors desiring a complete switch to C and senior faculty demanding that Fortran remain the in the curricula as the primary language of instruction. In the third semester offering, it was elected to drop the Etter Fortran text in favor of her newer C version, but retain the examples of Fortran constructs and syntax. Ultimately, it was determined that it was not possible to do justice to both languages using the materials at hand. This prompted the development of a set of course notes that were bound and sold to the students as a "Course Pac" by the campus bookstore and local copy shops.

As the course evolved, the bound notes grew in size and transitioned from copies of overhead transparencies to transparencies with text descriptions, then to chapter sections, and ultimately the notes took the form of a textbook manuscript. When the manuscript was submitted to Prentice-Hall, the response of the editorial staff was that the book looked great and could be published if all of the Fortran material was removed. This response was taken as an affront as the newly developed concept of dual-language learning was being seriously questioned. Additionally, the request was somewhat absurd considering the overwhelming dearth of C programming books that are currently on the market. The novelty of this text hinged on the Fortran component, yet whether or that novelty was useful was still unproven.

The manuscript was then sent to Cambridge University Press, a publisher that had expressed an interest in a new engineering programming text. After the manuscript had been reviewed, the following was the response from the editor: "there isn't a ringing endorsement from this particular set of referees". This sort of comment is always disconcerting, however, when the reviews were examined, it became clear that two of the reviewers were most likely computer scientists, while only one was an engineer. This was obvious, in spite of the anonimity, from the comments of two of the reviewers belaboring the lack of coverage of *recursion*. The editor suggested that two more reviews be sought, and confessed that the backgrounds of the reviewers in question were as surmised. At the request of the author, engineers were asked for the extra reviews. The response, to the surprise of the editor, was a firm endorsement of the text and the approach, with comments such as:

> "It seems to me that the main advantage of the present book by Myler is the combination of both Fortran and C. In my opinion this is a significant plus, since C is currently favored by most programmers but Fortran remains widely used in many engineering communities. Thus students who are proficient in both languages are clearly better equipped than those who learn only one language."

The text, *Fundamentals of Engineering Programming with C & Fortran*, was published in the fall of 1998. Although the book serves as the foundation of the course, the effectiveness and efficiency of teaching two languages simultaneously will now be examined independent of the book and course material.

Engineering Programming

Engineering programming has developed as an entity unto itself over the years, beginning with Fortran and moving through various languages and systems that include BASIC, FORTH, MATLAB and others. The expert programmer might find difficulty with this, in point of fact, the author's brother, a chemical engineer, wrote a program during his graduate student days to compute fluid-flow dynamics that is still in use worldwide. When the program was demonstrated, the author was forced to admit that the complexity of the program was impressive. This impression changed dramatically from admiration to outright disdain when it was revealed that the program had been written in Visual BASIC™. In spite of the author's confessed professional snobbery, this program is not the exception, but the rule. This conclusion is predicated on the simple fact that engineers want results from computers and they typically want the results quickly with a minimum of programming overhead accompanied by a good deal of confidence in the computed result.

Although the forgoing is anecdotal, the attitude of many engineers towards programming is not unlike their attitude towards mathematics. Engineers are not mathematicians. Notwithstanding this statement, engineers use mathematics to achieve goals

and often see the elegance of math not in a proof but in an application. A similar phenomenon occurs with computer programming. A good deal of time is spent in computer science classrooms teaching the student the structure and details of computer languages, much more so than the actual use and applications of these languages beyond operating systems and language support programs (compilers, assemblers, etc.). In computer science, the data structure is king and woe be the student who fails to develop elegance and structure in their programs. Contrast this with engineering. One might pose the question: "Which is more important? Elegance or functionality?" Which of these will guarantee that the bridge does not collapse, or the circuit remains within operating parameters? Finally, which approach makes the algorithm the simplest to follow and understand?

Before the author is run out of town on a rail for implying that computer scientists are more concerned with the appearance of a program rather than its functionality, let it be stated that the approaches to programming are different due to the different goals of the disciplines and those approaches are what will ultimately dictate pedagogy. The engineering student has scant little time within an ever-demanding curriculum to spend excessive time learning a computer language fluently to the point that elegance and structure are co-mingled with engineering functionality. Does this mean that engineering should abandon the teaching of programming and leave computers to the computer scientist or to the computer engineer? Many would strongly disagree. Historically, engineers have been the implementers of theory and the implementation phase involves a great deal of trial-and-error, testing and experimenting. One of the characteristics of a good engineer is perseverance. Edison stated that "invention is 1% inspiration and 99% perspiration" in describing his own work—and who better typifies one of the greatest engineering minds of the post-industrial age. Likewise, with programming, the computer becomes the tool of evaluation and test, a vehicle whereby quick evaluation and rapid prototyping can take place.

An argument against the use of a high-level language in the foregoing justification for "rapid prototyping" and "quick evaluation" might be the replacement of high-level languages by mathematics tools such as MATLAB™, MathCAD™ or Mathematica™. At UCF, we use MATLAB heavily in our controls and systems courses, and some faculty have argued that it should be used in our numerical methods course. It is important, however, to avoid the bandwagon effect that an individual or group can have on a curriculum decision that might lead to overspecialization and compartmentalization. It is possible to teach a high-level language course in such a way as to promote good programming skills and an understanding of machine computation that is readily transferable to other systems. The old adage "Only the first computer language that you learn is difficult, the rest are easy" is most apropos in this instance.

An Approach to Teaching Engineering Programming

Our introductory programming course, EGN3210 Engineering Analysis and Computation, is listed in our catalog with the following description:

> Engineering analysis and computation with structured constructs. Subscripted variables, subprograms, input/output. Batch processing and timesharing. Engineering applications will be emphasized.

This blurb reveals how dated a description can become, a function that is directly proportional to the amount of administrative paperwork and committee activity that is needed to update it! Nevertheless, in the classroom, subscripted variables are easily explained as arrays, subprograms as functions and I/O discussions are dependent on the language in use. As for "Batch processing and timesharing", it is always an amusement for an instructor to explain how in the "good old days" programming jobs were submitted as card decks and *teletypes* were used to access a time-share computer—descriptions that are often lost on students who have only known the programming pleasures of the *Integrated Development Environment*. This course description is fairly generic and offers the possibility of fairly wide interpretation of how the course should be taught. Since the course is currently structured after the text, which was an evolutionary result of many semesters of offering the course, any discussion of approach must follow the outline of the book. The rest of this section proceeds from that premise.

The first topic covered is the understanding of basic computer architecture. The Von Neumann model is explained in terms of a register-ALU-memory transfer system. The typical engineer, regardless of discipline, tends to seek a physical reality or baseline of understanding for a complex process. The introduction, at this level, of registers and ma-

chine computation relieves a large portion of the mystery of computer behavior once programming begins. This discussion of register transfer is then integrated into an explanation of Tannenbaum's virtual machine hierarchy to illustrate the multiple levels of translation and interpretation that exist in modern computers. Understanding this hierarchy is critical to the engineering programmer, because it illustrates and emphasizes the fact that, although computation only takes place at the digital logic (hardware) level, a substantial mount of translation takes place from the coding of the algorithm in the high-level language until it is seen by the computing hardware. The opportunity for error is great and the more understanding of the process that the engineer develops, the less likely error will occur or go unnoticed.

Errors in this translation sequence are analogous to the party game of whispering a phrase into someone's ear, then that person whispering the phrase to their neighbor, and so on until the last person to hear the whispered phrase speaks it aloud. Invariably, particularly with a large group, a phrase such as "Never give up the ship!" becomes "Whether to go up or slip," or some other such nonsense. Unlike science, engineering does not operate on conjecture or theory, an inaccurate result from mathematical calculations can lead to failure with serious consequence. To complete this segment, the relationship of C and Fortran programs to this hierarchy is then explained through diagrams and illustrations, so that the student develops a strong mental picture of computer function through language.

A methodology to problem solving is then explored where the primary issue to be resolved is the determination of what the problem actually is. This is illustrated by the following story:

> *A student and his professor are backpacking in Alaska when a grizzly bear starts to chase them from a distance. They both start running, but its clear that eventually the bear will catch up with them. The student takes off his backpack, gets his running shoes out, and starts putting them on. His professor says, "You can't outrun the bear, even in running shoes!" The student replies, "I don't need to outrun the bear; I only have to outrun you!"*
>
> *from Strategies for Creative Problem Solving by Fogler and LeBlanc, Prentice Hall, 1995.*

Six problem solving steps are introduced that lead the student from problem statement to a working computer solution. The steps followed are:

1. State the problem clearly.
2. Describe resources, data needed (input), expected results (output) and the variables required for the problem.
3. Work a sample data set by hand.
4. Develop an algorithm to solve the problem.
5. Code the algorithm.
6. Test the code using the edit-compile-run cycle on a variety of data sets with known results.

The edit-compile-run cycle is then described where a program is first typed into a computer using an editor, is compiled and then run if compilation was successful. It is explained that one rarely completes the sequence without re-editing and recompiling to fix bugs, hence the term *cycle*. This is followed by a discussion of the techniques used for off-line code development, flow charts and pseudocode. At this point it is possible to introduce the first program exercise, the ubiquitous "Hello World!" program. Many students are intimidated by the computer and the thought of actually programming one, so the simplicity of this program helps them gain confidence for the substantially more complex tasks ahead.

A dilemma that appears early in the teaching of C is how to introduce file and console I/O methods without a discussion of functions. This is particularly messy when one considers the fact that *scanf* requires the use of the address operator, a concept that is not fully explained until much later in the course. Nevertheless, the material presented is enough to allow the student to get early response from a program, and to allow simple numerical data entry. The details of "whistles and bells", elements that add style and user interest to a program but do not add to the engineering functionality, are discussed, and the student is cautioned to add these items after the program is algorithmically functional and correct.

The introduction to editing, compiling and running a program is followed by discussions of types, operators and expressions in both the C and Fortran languages. Recall that our goal is to teach the student to read Fortran but write C. Exercises in writing new code in C as well as translating Fortran code to C are

used to develop skill and test understanding of these important concepts.

The approach to translation and comparison that is used to illustrate the similarities and differences of the langauges is line-by-line translation. programs with sufficient simplicity are used so that a relatively straightforward translation is possible.

The use of fundamental language constructs for control flows are then introduced, with the pitfalls associated with using conditionals and looping constructs illustrated. These topics are presented with engineering problem examples for each. The limitations of Fortran become apparent in that DO loops are little competition for the looping capabilities in C. However, the contrast between the two languages enhance the understanding of the languages. Advanced concepts that include programming with complex numbers, structures and pointers complete the book.

The fundamental approach of the book and the course is one of *rapid prototyping*. Emphasis is placed on understanding the fundamental processes that a computer employs to derive calculations and to "get the right answer," as opposed to producing a slick application that is decorated with "whistles and bells". This is done in the context of both Fortran and C, which underscores the need for flexibility and multiplatform awareness. The book is self-contained and useful as a self-study tutorial, although students are advised to seek other texts to enhance the contrast of tutelage techniques and improve learning.

Conclusion

This paper has discussed an approach to the teaching of engineering programming that involves the use of two languages with an emphasis on functionality and correctness. A recent query to a group of Fortran programmers on the *World Wide Web* was posed. The question asked was "Is Fortran dead?" and the collective and quite emphatic response was, "NO!". Fortran is in heavy use by physicists and mathematicians, predominantly in the area of high-performance and parallel computing for very large-scale algorithms (finite element analysis, fluid dynamics, plasma physics, etc.). One responder lamented the fact that Fortran savvy programmers were difficult to locate, and that job opportunities abound. The simple fact is that a great deal of Fortran code is still used, and new Fortran code continues to be written. The engineer with an exposure and some small experience with the language cannot go wrong.

Nevertheless, the ubiquitous use of C throughout the computer engineering field, as well as its increasing use by the signal processing and communications communities indicates that a lack of knowledge by the engineer entering these fields would be tantamount to closing a large and potentially lucrative doorway. As new platforms and operating systems are developed, the first compiler available is often the C compiler—so an exposure and experience in C is increasingly valuable to the non-computer specific disciplines. Our approach to the teaching of these two languages meets those needs.

References

Etter, Dolores M. (1992) *Structured Fortran 77 for Engineers and Scientists*, Addison-Wesley, New York.

Etter, Dolores M. (1996) *Introduction to ANSI C for Engineers and Scientists*, Prentice Hall, Englewood Cliffs, New Jersey.

Myler, Harley R. (1998) *Fundamentals of Engineering programming with C & Fortran*, Cambridge University Press, New York.

Kerrigan, James F. (1991) *From Fortran to C*, Windcrest, Blue Ridge Summit, Pennsylvania.

Fogler and LeBlanc (1995) *Strategies for Creative Problem Solving*, Prentice Hall, Englewood Cliffs, New Jersey.

Harley R. Myler

Harley R. Myler is a Professor in the Department of Electrical and Computer Engineering at the University of Central Florida in Orlando. Dr. Myler did his graduate work at the Electronic Vision Analysis Laboratory at the New Mexico State University earning the MSEE in 1981 and the doctorate in 1985. He is currently the Director of the Machine Intelligence and Imaging Laboratory at UCF and has published over 30 articles and four books in the areas of imaging science and engineering, computer programming and architecture and engineering education. Dr. Myler is a senior member of the IEEE, a member of the SPIE, a Tau Beta Pi eminent engineer and a member of Eta Kappa Nu.