Using C in CS1
Evaluating the Stanford Experience

Eric S. Roberts
Department of Computer Science
Stanford University
September 1992

ABSTRACT

In 1991, the Stanford Department of Computer Science decided to abandon Pascal in its introductory computer science courses and to adopt ANSI C as the language of instruction. We based this decision on several factors: the inadequacy of standard Pascal as a base for teaching modern programming concepts, the need to prepare our students for more advanced courses in which they will be expected to use C for programming projects, and increasing pressure from students and faculty throughout the School of Engineering for instruction in a language that has become the industry standard. We also believe that it is not reasonable to expect students to learn C on their own; students must receive instruction in C in order to become good C programmers. C has several known deficiencies that make it a challenging language to teach. Based on our experience at Stanford, we believe that it is possible to minimize the problems associated with teaching C at the introductory level by applying standard software engineering strategies—procedural abstraction, modular decomposition, and information hiding—to good pedagogical effect. This paper expands on the reasons behind Stanford's decision to adopt C and summarizes the pedagogical approach.

1. INTRODUCTION

Any introductory course in computer science has one overriding mission: to teach students the basic programming concepts and methodologies that make more advanced work possible. To a large extent, these concepts and methodologies are independent of any particular programming language. However, since most of these concepts are best mastered through use, the introductory course must choose some programming language as its medium of instruction.

For many years, most universities in the United States used Pascal to introduce programming concepts. Compared with

many other programming languages, Pascal is conceptually simple and easy to learn. Pascal is also highly constrained, in the sense that it does not include certain powerful features that are easy to abuse. Such features tend to confuse students at the introductory level, even though they may be extremely useful to more experienced programmers.

Over the last five years, however, many schools have begun to move away from Pascal. The reasons behind this shift include:

• Pascal is getting old. There are several modern programming concepts, such as separate compilation, string manipulation, and the use of functions as first-class objects, that are not supported by standard Pascal and that are therefore difficult to teach.

• The restrictiveness of Pascal's design occasionally forces its users to "program around the language." Doing so means that students develop certain bad habits—habits that they must later unlearn when they begin to work with more modern languages.

• Pascal is not widely used beyond the introductory sequence, and students are therefore being taught programming style and the discipline of software engineering in the context of a language that they will never again use. The first course must provide an introduction to good programming practice. Unfortunately, the effectiveness of that instruction is compromised when the linguistic tools used to illustrate the underlying concepts are so quickly abandoned.

Even though a consensus seems to be emerging among educators that Pascal is becoming increasingly obsolete, there is no corresponding consensus on what language should replace it. Various informal surveys of colleges and universities indicate that the first course in computer science is moving in several competing directions. Many schools have shifted their curriculum to a language that adheres to the general philosophy and tradition of Pascal, but includes more modern concepts. In this category, the principal contenders are Ada [Winslow89, Frank90] and Modula-2 [Gabrini86, Koffman88]. Other schools have begun to use a Lisp-like language, the favorite being Scheme as it is presented in the textbook *Structure and*

Interpretation of Computer Programs [Abelson85]. A growing number of institutions, however, are choosing to adopt ANSI C, in light of its status as the language most often used for systems programming applications. Each of these approaches has merit, and it is not clear that there is a single answer that fits all academic institutions. At Stanford, we have decided to adopt ANSI C as our instructional language in the introductory sequence, for reasons which are developed in the next section.

2. THE REASONS FOR STANFORD'S CHOICE

In 1990-91, the faculty in Stanford's Computer Science department held several meetings to address the question of what language should be used in the first course. Since our introductory course serves a large population of non-majors from other engineering departments, that discussion was broadened to include the Engineering School Undergraduate Council, which has curricular authority over courses, like introductory computer science, that are considered to be "engineering fundamentals." As the various options were discussed, the systems faculty in the Computer Science Department and the representatives from other departments in the Engineering School argued strongly for using C in the introductory course. There was also general support for the proposal from the department as a whole. As one of the major proponents of C, I had been prepared to encounter opposition, but the decision turned out to be almost entirely without controversy. With the support of the department, I developed an experimental course using ANSI C, and in 1991-92 offered that course as an alternative to the Pascal-based course. Based on the success of that experience, the department has decided to revise the curriculum so that, starting in 1992-93, all introductory programming courses will be taught in C.

The most important reason for this decision comes from the fact that it has become increasingly important to teach C somewhere in the computer science curriculum. C is used in most upper-level systems courses, and students are expected to know how to use the language by the time they encounter those courses. In previous years, the only instruction that students received in C took place in a sophomore-level "Programming Paradigms" course, which introduced the students to several different programming languages, including C. This course teaches the fundamentals of C syntax and structure, and expects that students will learn everything else they need to know on their own.

Unfortunately, students are better at picking up the syntax and structure of a new language than they are at developing discipline in the use of that language. The introductory sequence at Stanford concentrates—as it should—on instilling in students the importance of good programming style. Unfortunately, at Stanford, and at many other schools, we have been teaching people to become good programmers only in the context of a language that they never again use. Students learn good programming style and methodologies in the context of Pascal, but are forced to develop their own styles and methodologies for the other

languages they encounter. For some students, these critical software engineering skills transfer well. For others, however, many of the concepts seem to get lost in translation, and we are therefore generating a large number of students who program well in an arguably useless language and who program poorly in more useful ones. This is not an ideal state of affairs.

We believe that it is not enough to expect students to learn C on their own; they must receive instruction in C in order to become good C programmers. Turning out another generation of mediocre C programmers is not in anyone's best interest. There is a serious oversupply of such programmers today.

Student demand was also a factor in the decision to adopt ANSI C. In today's computing industry, it is clear that C has become the dominant language for systems development, and our students are not blind to that fact. Given the choice last year between an experimental C section and a well-established Pascal section taught by a popular instructor, students expressed a two-to-one preference for the C section, most often citing C's practicality as the reason for their choice.

The importance of student demand is heightened by the fact that the introductory programming course at Stanford—as at many institutions—must serve the needs of a diverse clientele. Approximately 700 students enroll in the introductory computer science course each year; of these, only 60 will go on to become computer science majors. Most of the students take the course to fulfill the requirements of some other major in the School of Engineering, and there is also a significant enrollment of students from various departments in the School of Humanities and Sciences. For many of these students, the introductory course will be the only computer course they take, and it is important for it to provide a good foundation in the discipline of computing.

Finally, it is important to recognize that the value of following conventional practice has risen in accordance with the increased size of the computer science community and with the increased complexity of programs written today. Twenty years ago, it was easy to have students in the introductory course write programs that seemed exciting to them—programs that seemed as flashy and interesting as the applications they had previously encountered on their own. Today's students have grown up with Macintoshes and fancy graphical interfaces. To them, a program that averages numbers or even one that plays a simple, interactive, text-based game will seem like pretty tame stuff.

It is impossible today to construct from scratch a program that meets the student's heightened sensibility of what a "real" computer program must be. To do so, one must make use of sophisticated software libraries, such as the X11 library or the Macintosh Toolbox. To use such libraries, however, one must program in a way that is

compatible with them. In most cases, this means working with a programming language, such as C, that is implemented on the vast majority of platforms and offers a clean interface to those libraries.

3. AVOIDING THE PITFALLS

For the most part, the advantages of using C discussed in the preceding section are fundamentally strategic in nature. We believe that introducing C at the introductory level will yield long-term advantages as students apply the knowledge and engineering discipline they have learned to more advanced courses or projects that require a knowledge of C. Viewed from the perspective of the introductory sequence alone, it is clear that using C presents a variety of tactical problems. I do not deny for a moment that it is harder to teach C to beginning students than it is to teach Pascal; my claim is rather that the long-term benefits more than compensate for the short-term costs. I also believe that it is possible to minimize the problems associated with teaching C at the introductory level by applying standard software engineering strategies—procedural abstraction, modular decomposition, and information hiding—to good pedagogical effect.

For example, one of the most common complaints about teaching C is that certain conceptually simple operations are implemented in conceptually sophisticated ways [Mody91]. Often, C's design follows an underlying logic that is only meaningful to the experienced programmer. To the novice, that logic is entirely obscure, and the student is given no conceptual framework for understanding the mechanism.

As an example, consider the operation of reading an integer from the console and storing it in a variable—an operation which is likely to occur in conversational programs very early in the introductory course. The ANSI libraries include a function for this purpose, and the experienced programmer would know to code this operation as follows:

```
scanf("%d", &value);
```

Unfortunately, the use of **scanf** is extremely confusing to new programmers. In order to achieve the effect of call-by-reference, **scanf** passes a pointer to the variable used to store the input value. Explaining this mechanism in detail requires students to comprehend pointers and a great deal more about parameter passing than one wants to introduce at the beginning of a course. As it is, the **scanf** syntax has to be presented as a linguistic rule: every variable that appears in a **scanf** call (except for character arrays, which turn up later and multiply the confusion) must be preceded with an ampersand. This approach might be tolerable if C compilers were able to enforce this syntactic rule. Given C's structure and semantics, however, the compiler cannot detect the extremely common error of leaving out the ampersand. The student writes

```
scanf("%d", value);
```

the compiler doesn't complain, the value entered on the console gets written into deep space, the program gives the wrong answer (assuming that it doesn't crash), and the student has absolutely no clue as to what went wrong.

In addition to the problem of requiring an obscure calling convention, **scanf** has several other properties that tend to be at least as confusing. One significant problem is that **scanf** makes error recovery difficult to perform. The return value from **scanf** indicates only the number of matching fields, and it is impossible to determine easily, for example, whether there was extra text on the input line. If the result is zero, indicating that no value has been read, how does the programmer recover from that condition, particularly with respect to clearing text from the current line? A second problem is that using **scanf** requires the student to have a much more detailed knowledge of how the input system handles special characters. If **scanf** is used to read an integer from the console, and the user terminates that input by typing the newline character, **scanf** will read the terminating newline and then put it back in the input buffer using **ungetc**. If the student then tries to perform character input, that newline will be the first character read. This behavior is extremely difficult to explain to the novice. In fact, all of these problems are beyond the scope of an introductory treatment, and there is no way to give beginning students a complete explanation.

In order to understand how to solve this problem in a classroom setting, the key is to recognize what experienced programmers do when they encounter similar problems in the process of developing software. As programmers, we often encounter situations in which the details necessary to implement a conceptually simple operation turn out to be messy. When this occurs, the standard approach is to hide the complexity by defining a new function or procedure that encapsulates the complexity and provides to its caller a clean and simple interface.

I believe that this same approach offers an ideal mechanism for deferring discussion of C's more confusing features until students are able to make sense of them. In the specific case of **scanf**, for example, one can mask the complexity by defining a new library that provides a simplified I/O capability. In the course at Stanford, the interface to that library was provided by the header file "simpio.h", which gave students access to the functions **GetInteger** and **GetFloat**, which read and return an integer and floating point value, respectively. Armed with this library, students read integer values by writing

```
value = GetInteger();
```

This call takes the place of the earlier **scanf** construction and has several advantages from the point of view of student understanding. First, it is consistent with every other type of function that they see, and its use involves no special syntactic "rules" that the compiler cannot check. Second, the library implementation offers much better error checking and built-in recovery operations. Third, the

implementation is defined so that `GetInteger` always reads a complete line, and there is therefore never any confusion about whether the terminating newline character remains in the input pipeline. All in all, students understand this mechanism much more easily, and can use it without any significant difficulty. The `scanf` function is then introduced at a more appropriate point in the course, when it is also possible to expose the implementation of the "simpio.h" interface.

I also used the course libraries to remedy other pedagogical problems that tend to arise when using C. For example, C includes no Boolean type—an omission that has serious negative implications for teaching. Although the use of integers for this purpose might be acceptable for experienced programmers, it has the effect of muddying completely one of the central concepts that introductory students must master. In my course, I solved the dilemma by defining the type `bool` as an enumeration consisting of the constants `FALSE` and `TRUE`, and then putting that definition in a standard library that students use from the very beginning of the course. When the topic of Boolean data was first mentioned, I noted in passing that the type `bool` is an extension provided by the library. Once that was said, I didn't mention it again for the rest of the quarter. For my students, there is quite definitely a Boolean type. Every condition they see, and every condition they write has a proper Boolean form; the traditional C shortcuts that depend on the equivalence of `FALSE` and zero never arise. This approach leads to a much better conceptual understanding of how to use Booleans and improves programming style and practice.

The standard library used with the course also includes an error-handling facility that makes it easy for students to think about error-checking and other defensive programming strategies early in the course. In the beginning, the error-handling facility is used simply to provide a uniform mechanism for reporting errors and then terminating the program. Later in the course, I introduce more advanced facilities of the error-handling package that allow students to trap those errors and specify recovery actions, without changing the basic discipline. The facility is based on a more general exception-handling mechanism for C [Roberts87].

4.   REACTIONS TO THE LIBRARY APPROACH

In reading through reviews of an introductory text based on this approach, I was interested to find that the early introduction of libraries, such as the "simpio.h" facility outlined in the preceding section, proved to be rather controversial. Several reviewers responded very favorably, noting that no other textbook addresses the problem that students are not prepared to handle C constructs in their full generality. Other reviewers, however, expressed strongly negative reactions to my tactic of introducing specialized abstractions, such as the use of `GetInteger` and `GetFloat` to mask the complexity of `scanf`. These reviewers argued that I was changing the language or that I

was likely to distract the students by providing a mechanism that they would "remember after the course is over, instead of remembering `scanf`." One reader felt sufficiently strongly about this question to set an entire sentence in italics: "*If you're going to teach ANSI C, teach ANSI C, not some modified local version of the language!*"

I believe that the last comment cuts to the heart of the controversy. I was not trying to teach ANSI C. I was trying to teach programming. I just happened to be using ANSI C because I believe that students who learn programming using ANSI C will be better practitioners in the C environment after they complete the course.

CS1 is not a language course alone. Although learning a programming language is an essential component of the introductory presentation, there are many aspects of an introductory course that are entirely independent of language choice. Of these, one of the most important is procedural abstraction as a technique for managing complexity. Students need to learn how to take a complex operation and encapsulate it as a procedure call with a simple and consistent interface. The fact that our course follows its own advice with respect to library development encourages students to do the same.

Last year's experimental course was extremely effective in getting students to recognize the value of abstraction. We introduced separate compilation early and emphasized the importance of well-specified *interfaces* as the link between the *implementer* of a module, who understands the details, and the *client* of that module, who understands only the abstract effect. Our students had the standard troubles with algorithms, problem solving, implementation details, and debugging, but they did come to master the notion of an interface and recognize its importance. And the concept seemed to stay with them. One of my students wrote to me a year later:

> I want to let you know that I've applied with great success your libraries in the courses I've taken . . .; it's a great concept of yours to deal with abstractions so that when you design a package, you, the "client" are relieved of the underlying implementation and can concentrate on the particular algorithms of your particular program.

I don't, of course, claim credit for the concept, but I do believe that the pedagogical emphasis on abstraction has paid off.

5.   EMPHASIZING GOOD PRACTICE

A second common concern is that C—by virtue of the very power and flexibility that have ensured its commercial success—is far too great a temptation for the beginning student. C has a reputation as the language of the hacker, a domain in which all concern for good programming style or sound engineering practice are inevitably displaced by a passion for writing the shortest possible program or the

program that generates the most efficient possible code. This view is sometimes encouraged by the pedagogical treatment. In the years that C was introduced in the "Programming Paradigms" course, the focus was on the nitty-gritty details of C, such as pointer/array equivalence, the increment/decrement operators, and the various syntactic shortcuts for which C is notorious—precisely the features students are most likely to abuse. One of the traditional handouts used in the course was titled "C Gets Ugly," and it is not hard to imagine that some students would turn this assessment into a self-fulfilling prophecy.

My own view is that the power of C is most destructive precisely when we teach students to use good programming techniques in some other language and then expect them to pick up C on their own. It is at this point that the temptation of power seems most seductive. By teaching them good programming style and emphasizing the importance of discipline as we teach them about the language, we can produce solid programmers who carry these skills forward to more advanced work.

Brian Reid, who was a professor at Stanford for several years, captured the dilemma presented by the power of C in the simple observation: "power tools can kill." In designing a new curriculum, a department has the choice of (1) avoiding the use of such power tools until students reach upper-level courses or (2) offering courses that include appropriate "power tool safety" components at the introductory level. We decided to opt for the second approach.

I believe it is useful to remember the following observation made by Larry Flon, then at Carnegie Mellon [Flon75]:

> There does not now, nor will there ever, exist a programming language in which it is the least bit hard to write bad programs.

The key to writing good programs, just as much today as in 1975, is good discipline. If we teach that discipline, we will produce good programmers. If we do not, the choice of language barely matters.

6. CONCLUSIONS

Over a two-year period, the Stanford Department of Computer Science has redesigned its introductory course to use ANSI C, replacing Pascal as the language of instruction. To avoid some of the problems that ordinarily accompany the use of C, we have adopted three strategies. First, the course defers the introduction of some of the more difficult features of the language by using libraries to encapsulate the complexity of those features so that the student sees them only through a simple abstract interface. Second, we emphasize the idea of procedural abstraction and modular interfaces from the very beginning of the course, so that students themselves develop the skills necessary to manage complexity when it arises. Third, we focus throughout the course on the discipline of software engineering, so that students come to recognize—and then to demonstrate by following our example—that C programs can be written so that they are elegant, clearly presented, well-documented, and easily maintained.

REFERENCES

[Abelson85] Harold Abelson and Gerald Sussman, with Julie Sussman, *Structure and Interpretation of Computer Programs*, Cambridge: MIT Press, 1985.

[Flon75] Lawrence Flon, "On research in structured programming," *SIGPLAN Notices*, Vol. 10, No. 10, October 1975.

[Frank90] Thomas S. Frank and James F. Smith, "Ada as a CS1-CS2 language," *SIGCSE Bulletin*, Vol. 22, No. 2, June 1990.

[Gabrini86] Philippe Gabrini, J. Mack Adams, Barry Kurtz, "Converting from Pascal to Modula-2 in the Undergraduate Curriculum, *Proceedings of the Seventeenth SIGCSE Technical Symposium on Computer Science Education*, February 1986.

[Kernighan88] Brian Kernighan and Dennis Ritchie, *The C Programming Language*, second edition, Englewood Cliffs: Prentice Hall, 1988.

[Koffman84] Elliot B. Koffman, Philip Miller, and Caroline Wardle, "Recommended Curriculum for CS1, 1984," *Communications of the ACM*, Vol. 27, No. 10, October 1984.

[Koffman88] Elliot B. Koffman, "The case for Modula-2 in CS1 and CS2," *Proceedings of the Nineteenth SIGCSE Technical Symposium on Computer Science Education*, February 1988.

[Mody91] R. P. Mody, "C in Education and Software Engineering," *SIGCSE Bulletin*, Vol. 23, No. 3, September 1991.

[Roberts89] Eric Roberts, "Implementing Exceptions in C," Research Report #40, Digital Equipment Corporation/Systems Research Center, Palo Alto, California, March 1989.

[Winslow89] Leon Winslow and Joseph Lang, "Ada in CS1," *Proceedings of the Twentieth SIGCSE Technical Symposium on Computer Science Education*, February 1989.