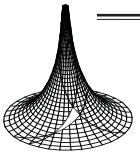

1/1 (2003), 137–152
tmcs@math.klte.hu
http://tmcs.math.klte.hu

Teaching
Mathematics and
Computer Science

A first course in computer science: Languages and goals

Dennis C. Smolarski

Abstract. The College Board Advanced Placement exam in computer science will use the language Java starting in fall 2003. The language chosen for this exam is based on the language commonly taught in introductory computer science courses at the university level. This article reviews the purpose of an introductory course and the various suggestions for the curriculum of introductory courses published by the Association for Computing Machinery. It then proposes that such a course stress foundational concepts over specific language syntax, and then provides a list of such foundational concepts and related topics. Based on this fundamental curriculum, the article recommends C++ as the most appropriate language. An appendix provides a sample syllabus.

Key words and phrases: introductory computer science courses, computer languages, elementary data structures, languages structures, elementary algorithms, data encapsulation, objects, technology and society.

ZDM Subject Classification: B75, D35, M55.

1. Introduction

The College Board is a U.S. corporation known for creating and administering common exams for secondary school students, exams such as the SAT (Scholastic Aptitude Test), the PSAT (Preliminary SAT), and various AP (Advanced Placement) exams. In particular, the AP exams, available for various technical and non-technical subjects, enable students completing their secondary school education to receive credit for university level courses at some institutions, if they score appropriately high on the exams.

Copyright © 2003 by University of Debrecen

The College Board has determined that starting in the 2003–2004 academic year, the AP exams in computer science will use the language JAVA [3]. Since about 1998, the language for this exam had been C++, and prior to that it had been PASCAL [1]. My comments will focus on two major issues: (1) what factors should determine which language is used in a first college-level course in computer science (and thus be the language used for the computer science AP exam)?, and (2) should the syllabus for a first course focus on the syntax of a very powerful language (that may not be widely supported or popular in five years), or on more general and basic concepts, such as problem-solving skills and language-independent structures, which will endure for decades?

2. Programming languages and introductory courses

A quick glance through back issues of computer-related journals such as the *SIGCSE Bulletin,* a publication of the Association for Computing Machinery (ACM) and its Special Interest Group on Computer Science Education (SIGCSE), will reveal that discussions about the first college level course in computer science (e.g., ACM's CS 1 course) and the language appropriate for that course have been numerous and on-going. In a discipline as rapidly developing as computer science, there are bound to be changes, but a perennial question is whether changes in the curriculum of this discipline should be as driven by industry as they often seems to be [7]. In other words, to some it seems that decisions about the language used in an introductory course are made based on what language would be most useful for a student in finding a part-time job, rather than on fundamental skills that would prepare the student for subsequent courses and would make any software being developed by the student well-written and error-free.

At my own university, introductory courses in computer programming are offered in several departments and, over the last 20 years, the language in one or other of these introductory courses has been Basic, Pascal, C++, Ada, Java, C, and Fortran. A nearby university used Scheme (a LISP variant) as the introductory language for several years and other universities used Modula-II for a period of time in the 1980s. For more information, also see surveys about languages used in CS 1, such as [10].

Every time there is a change in the programming language used in any course, an instructor must readjust and, as most experienced faculty in every discipline know, teaching a course the first time with a new book or a new syllabus is never as smooth as one would hope or want. Thus, it is highly likely that in

introductory computer courses, students may experience teaching on the part of the faculty that seems less polished (perhaps due to more uneasiness on the part of the instructor) in comparison with faculty teaching other courses, because the computer faculty have had to adjust the course regularly to new languages. A new language may possibly mean an entirely new approach to programming and dealing with some (and possibly many) peculiarities related to the syntax of the new language and its implementation on hardware accessible to students. Yet how necessary are such changes in languages and what topics should be included in any first course? I am convinced that if an introductory course presents foundational materials well enough, it is relatively easy for a student to shift from one language to another with a minimum of energy and time. This does not mean that someone whose first course was taught in Fortran-95 can become a project manager for a software project written in C++ without some intensive training. But it does mean that a student who is well-grounded in language-independent fundamentals should find the task of shifting to a new language relatively easy, especially when compared to the many difficulties associated with translating a problem statement into working software irrespective of the language.

3. Purpose of an introductory CS course

Introductory courses in computer science vary from university to university. At larger schools, there are usually several such courses, each geared toward a particular type of student, for example, one for business majors, another for computer science majors, still others for engineering majors, science majors, or humanities majors. Many smaller schools may only be able to provide one general introductory computing course which must serve computer science majors as well as other science majors and even liberal arts students, some of whom may, perhaps, be interested in obtaining a computer science minor if their initial experience were pleasant enough. The background preparation of students in any introductory course may vary widely. Some may have no programming experience, while others may have been writing programs for several years. In addition, even in larger schools that offer a specialized first course for computer science majors, there is a good possibility that a significant percentage of students enrolled in the first course may not graduate as computer science majors. Thus, it is appropriate that even a course for majors, though preparing those enrolled for subsequent courses, should also be broad enough to benefit the students enrolled who may subsequently decide that their interests and talents lie in other disciplines.

Based on my experience of teaching introductory courses for about 20 years, I support those who advocate emphasizing fundamentals rather than the latest fads [11]. Thus, for me, the language used in a beginning course is secondary to the concepts taught since, in my experience, some of the language syntax is often easier to learn (and understand) than the underlying concepts. In other words, sometimes a student can easily produce a program without any compiler errors (that is, one that is syntactically correct) yet the program gives completely incorrect output. In addition, complicated syntax (leading to numerous compiler errors) distracts students from concentrating on understanding the more fundamental, foundational concepts.

Moreover, it is very difficult to predict what the language of industry will be four years hence or even how a currently popular language (such as Java or C++) may evolve in the near or distant future. (Who in 1970 would have thought that Fortran-95 would include recursion, pointers, and structured variables? Who in 1986 would have foreseen that C++ would eventually include templates [1990] and a boolean data type [1998]?) Schools do a disservice to students to focus on the syntax of a specific language in their first year *at the expense of concepts* that are common to most languages and are core to modern structured programming. In defense of the College Board and the difficulties it faces, however, I do agree that if an AP exam is supposed to reflect a college level course, the language used should also correspond to a language widely-used in introductory computer science courses taught at at universities [6, p. 117]. But it is unclear to me that there is any consensus as to what this language is or should be, or that the difference between Java and C++ is so great as to justify a change. In September 2002, one Department of Computer Engineering I know of actually decided to change the language used in its introductory course to C (not C++) after having used Java for several years.

I also believe that the first course should provide a good foundation but also be taught in such a way that many of the topics will be revisited in greater depth in CS 2. If the first course is taught well enough, it might even entice some uncommitted students to enroll in a second course, providing even non-majors with greater exposure to, and understanding of, the world of computer science and programming.

4. ACM suggestions for a first course

In 1991, the Association for Computing Machinery and the Computer Society of the IEEE issued a report [4] providing a modular approach to a computer curriculum, with subject areas subdivided by knowledge units. It did not provide a single paradigm for various courses commonly found in the curriculum. Instead, it provided sample curricula with sample courses to show how the various knowledge units listed in the report could be organized into sequential courses in various ways.

Although this curricular approach has merits, most schools still organize the foundation courses based on the earlier, 1978 recommendations [2], updated somewhat in 1984 and 1985 [8, 9] for the first two courses, the so-called CS 1 and CS 2 courses. The 1984 CS 1 curriculum recommended including the following major areas: (1) Basic Concepts of Computer Systems, (2) Problem Solving and Algorithm Development, (3) Program Structures, (4) Data Types (both scalar and structured), (5) Program Development—Methods and Style, and (6) Advanced Topics (such as linked lists, searching and sorting algorithms, algorithm analysis and program verification) [8, pp. 999–1000]. It also suggested the use of a language that included (1) structured control statements, (2) structured data types, (3) strong data typing, (4) subprograms with parameters, (5) dynamic storage allocation, and (6) recursion, and it specifically mentioned Pascal, PL/1 and Ada as possible languages [8, p. 1000]. The 1984 CS 1 report also noted that although Algol also satisfied the language requirements, it was not widely used or supported, something that could also be said of Pascal and PL/1 nowadays. It also noted that although Fortran and Basic were widely used, they were regarded as unsuitable because they did not support the desired features, something no longer true when including Fortran 95 or Visual Basic. Oddly enough, C was nowhere mentioned, possibly because it was so frequently associated with more advanced computer users and systems programmers that it was not considered appropriate for a first course (cf. [7]).

5. Suggested foundational concepts

As I noted earlier, based on personal experience, I suggest that it is more important to focus on foundational concepts in a first course, with examples applicable to a wide range of disciplines, rather than on specialized features of

a high powered programming language. So, before examining specific languages, I would like to propose a list of topics I consider appropriate for a first course, topics not bound to any one programming language, yet topics that can serve both computer science majors and other majors well, whether they take subsequent courses or not. Let me group these topics into four general categories: (1) Topics related to Data, (2) Topics related to Programming Language Structures, (3) Topics related to Algorithms and Examples of Simple Algorithms, (4) General Concerns. I will then mention a few topics not included in any of these four categories.

1. Data Topics

A beginning course needs to cover both *data types* and *elementary data structures.*

Early in the course, the distinction between *numeric data types* (e.g., integer, real, complex) and *non-numeric types* (e.g., boolean, single character, multiple characters) should be presented as well as *precision within a type* (e.g., the C/C++ types of `short, int, long` and `float, double, long double`). One might also note that in some languages certain types are intrinsic (e.g., `COMPLEX` in Fortran) but must be constructed as a user-defined type in other languages.

Although the distinction may be easy to make in theory, remembering the implications when students write programs is another matter. Even after emphasizing the distinction between integers and reals and emphasizing that most languages assume that when arithmetic operations between integers take place the result will be integer (rather than real), such basics are often difficult for students to remember. As a result, in my experience, it is common for students to obtain incorrect results because integer numerators are divided by larger integer denominators, resulting in unexpected quotients of zero! The pervasive use of pocket calculators often blurs for students the distinction between numeric data types that is foundational to numerical programming.

Fundamental to the concept of data is whether one needs to store a single unit of information or multiple units associated together. From this consideration flows the distinction between *scalars* and *arrays.* In my experience, however, the distinction is often simpler to make than to convey to students *when* it is appropriate to use an array and *how* to make use of an array. A recurring tendency I have observed in students is declaring an array and then storing data that will only be used once (e.g., storing all the numbers from an input file to find the sum), when there is no need to retain that information for subsequent use. Similarly,

introductory students regularly find it difficult to determine how to make use of a loop to scan through all elements of an array for some purpose (e.g., searching).

After some basic experience with scalars and arrays, students should also be introduced to *record variables* as an example of a heterogeneous aggregate data structure (e.g., C structures, C++/Java classes, Fortran-95 derived types). Such user-defined structures help emphasize that since arrays must all be of the same data type (a homogeneous data structure), they are inappropriate for problems in which data are not of one uniform type (e.g., a unit of data containing a person's name, bank account number, and bank balance), yet one still would want to store such related information together somehow.

Given that the second introductory computer science course focuses on data structures [9], and also given that there may be students in a CS 1 class who may not take the subsequent course, I suggest that it is important to expose student to aggregate data structures and perhaps include an assignment that includes such data structures, but leave an in-depth study to CS 2.

It is also important to introduce *pointer variables* in CS 1, perhaps, once again, leaving more intensive programming projects with pointers until CS 2. The concept that one can store in computer memory an address (rather than a number or a character), which can then be used to locate other data, is important and is the basis for many other significant topics in computer science. Nevertheless, it is a concept that is difficult for many introductory students to grasp. At minimum, students should be presented with a simply-linked list (at least in diagrammatic form) and even with a binary tree as a two-dimensional extension of a linked list. Tree traversal can be explained in CS 1 in a more mathematical fashion, with the programming details left for CS 2.

I suggest that students should also be introduced to the concept of *objects* in an introductory class, given the contemporary emphasis on object-oriented programming. But, once again, there is a difference between introducing such a concept and an in-depth study. Typically, I introduce students to a *stack* as one example of an abstract data type (ADT) in which the storage structure for data (i.e., the stack implemented as an array), and the special functions associated with inserting and deleting elements (i.e., "push" and "pop") are so linked together that it is best to join them together as a single programming reality, the "class." The pervasive world of object-oriented programming recommends that students be exposed to objects very early in a computer science curriculum, so that they start thinking about the interconnection between data storage and operations on data early in their computing careers. Yet in-depth programming experience with

objects can easily be delayed to a second course, after students have had more experience with fundamentals.

2. Topics related to Programming Language Structures

For many students, dealing with a programming language is the first experience they have had that forces them to organize their thought processes. Many struggle with the basics of breaking a problem into a number of steps performed sequentially. Many are subsequently frustrated when an inanimate machine keeps telling them they have made "errors." The basic three programming concepts described as integral to contemporary "structured programming" by Edsger W. Dijkstra, namely, (1) sequencing, (2) repetition, and (3) choice, should be foundational to any introductory computer science course [5, pp. 16–23]. Thus, in addition to arithmetic computations and assignment statements performed sequentially, students should be exposed to *repetition structures* (e.g., `for` and `while` loops), a *simple choice statement* (i.e., `if ... else` statement), a *multiple choice statement* (i.e., `switch` statement). The use of a conditional loop or an if-statement also means exposing students to *boolean expressions* and relational and boolean operators.

Since modularity is emphasized in contemporary programming practice, early in the course, students should be shown how to create self-standing *functions* (including "void" functions, also called *procedures* in Pascal or *subroutines* in Fortran). It would also be good to make students aware of various *parameter passing schemes* (e.g., C++/Pascal call-by-value versus call-by-address schemes) used within a language or in different languages.

Once again, experience has shown that some distinctions are difficult for novice programmers to make. For example, a recurring phenomenon is the confusion between an if-statement and a while-loop. Some students realize they need to perform an action or computation based on a condition, but often they make use of the if-statement when they really need to use a looping structure such as a while-loop to repeat the desired action or computation.

3. Topics related to Algorithms and Examples of Simple Algorithms

Probably the easiest classes of sample algorithms to present and discuss in a first computer science course are simple *sorting* and *searching* algorithms for arrays. Conceptually it is easy for students to understand putting a set of information into some kind of order or searching to find something stored. Typically, the sorting algorithms presented are bubblesort, selection sort, or insertion sort.

More efficient (and complicated) algorithms based on recursive techniques (e.g., quicksort) or advanced data structures (e.g., heapsort) are better deferred to the CS 2 course or a course on algorithms. Presenting several elementary sorting and searching algorithms, however, can help students to appreciate various approaches to solving a single problem and emphasize that, often, there is no one right way to write a program. Occasionally introducing a more advanced concept such as a "stable" sort or "index" sorting or the complexity of an algorithm (e.g., that linear search is $O(n)$ while binary search is $O(\lg n)$) can add a level of sophistication to an otherwise elementary topic while not significantly complicating the level of programming skills needed.

Throughout the first course, various problem solving techniques need to be mentioned and discussed. How does one break a problem down? What are the global steps that can each be coded as separate functions or procedures? Can a problem be solved by subdividing it into simpler cases (the so-called "divide and conquer" approach)? Students should be exposed to the distinction between *iteration* and *recursion* and that these are two different approaches to problem solving when repetition is required. Sample recursive code should be examined and simple recursive functions should be evaluated by hand. Solutions to simple problems could be presented in both an iterative and a recursive version for comparison. Once again, a more in-depth study of recursion can often better be deferred to a CS 2 course.

4. General Concerns

Of course, it would be useless to write code and not be able to retrieve computed data, so *input-output commands* need to be presented, along with a minimum number of related topics, such as a few formatting features and ways to make use of files. Since input-output commands vary so widely from language to language, my own approach is to minimize exposure to such topics until students become comfortable with the more important concepts and the more fundamental language features.

In order to lay a good foundation, even in a first course *data encapsulation* should be discussed (in conjunction with the presentation of "objects"). Issues related to program style (e.g., modularity, indentation, mnemonic identifiers) and to documentation also need to be emphasized, along with some discussion of debugging and of a concern for program correctness.

Some courses may also wish to include a bit of *history of computing and societal issues*, such as a discussion of the impact of computerization on the

workforce, the annoyance of email spam, computer related crimes, dependence on possibly imperfect software in critical places (hospitals, air traffic control, etc.). Such topics regularly appear in professional journals and are referred to in newspaper articles, and may lead to lively discussions after students have had some hands-on experience in writing and debugging code.

Instructors of elementary courses should also give consideration to classroom techniques. Some instructors can make excellent use of a computer classroom and appropriate software can prevent students from "surfing the net" during class time. Other instructors prefer using a data projector and notes posted on a course web site, or power point or similar video presentations. Still others have used a combination of overhead slides and blackboard techniques. One difficulty experienced by many students in introductory courses is having inaccurate examples in notes they take in classes. Try as they might, students find it difficult to transcribe every set of braces and every semi-colon accurately from their instructor's notes to their own notes. In addition, instructors can easily, though unintentionally, present incorrect coding examples when unexpected questions arise, code that has not been debugged with the aid of a compiler. Thus the use of prepared slides or web pages, to which the students have access either before or after class, can assist the learning process by reducing transcription errors and allowing students to focus on the content of the material being presented rather than on the accuracy of their note-taking skills. In my experience, students very much appreciate having access to printed notes or notes on web pages, but also appreciate the regular use of "blackboard" examples in response to their questions or to explain prepared examples in greater depth.

5. Omitted Topics

There are a number of topics not listed under the previous four categories. For example, although I think it is very important to expose students to a Unix or Linux programming environment and a widely-used editor such as `vi` or emacs, some schools or faculty may find it to be just as appropriate to introduce students to computer science and programming using software run on an individual machine. Common Unix/Linux program development tools such as makefiles are very helpful, but a discussion of such tools can takes valuable time away from more important material that should be covered in a first course. Similarly, the more recent and specialized features of a language (such as the use of user-defined header files in C, C++, or Java, various namespaces or standard templates in C++, or

various predefined classes in Java) can often sidetrack and frustrate beginning students.

Struggling with the operating system or with advanced features of a language does not seem, in my experience, to be beneficial for beginning students if it compromises their learning of foundational concepts about programming languages and computer science.

6. Appropriate Computer Language

Given a basic list of desirable topics for an introductory course, one is faced with the question of which language can best support the pedagogical goals, without, however, completely ignoring the problems associated with support of this language and the desirability of a language widely-used outside of academia.

Pascal was designed as an academic introductory language, but it does not include object-oriented programming concepts and it is no longer widely supported. Similarly, C, though long established, widely supported, and available in a "standard" version, also does not include object-oriented programming concepts (nor does it have multiple parameter passing schemes for subprograms), and it has been argued that C's power can be counterproductive in an introductory course [7]. Java, though widely available, is so object-oriented that it is difficult to write a simple "Hello, world!" program. In addition, the input-output features that ideally should be easy for a new user to use, are far from transparent in Java. Such a difficulty may be seen as an opportunity for a committed computer science major, but for other students who may be in an introductory course, the complicated syntax of a simple Java output statement may, in fact, have significant negative consequences. Scheme may have many supporters from those who advocate a functional approach to computer languages and from the artificial intelligence community, but many students enrolled in an introductory computer science course are concurrently trying to understand functions in an introductory calculus course and may find Scheme (or other Lisp-related languages) as counter-intuitive as they find maximum-minimum or related rates problems in a differential calculus course. And non-computer science majors may find Scheme not at all useful when trying to solve problems arising from mathematics or physics courses.

It is true that a simple enumerated loop, such as

```
sum = 0;
for (int i=0, i < 10, i++)
    {
       sum = sum + i;
    }
```

is actually identical in C, C++, and Java. But how such a structure fits into a larger program and how one outputs information from a program including such a loop is where significant differences lie. Yet it is these auxiliary considerations that can shift the balance toward one direction rather than another when deciding upon a programming language.

Taking into account the various considerations, I see C++ being the logical choice. Yet, C++ itself continues to evolve and its more recently-introduced features, if used, could be detrimental to the goals of an introductory course. It also shares with its parent language, C, a lack of good compiler error-checking, a feature that made Pascal such a wonderful language to use in beginning courses.

Given various pros and cons, at this moment in history, to me it still seems best to make use of a subset of C++ as the language of instruction in an introductory university level course. C++'s inclusion of a `class` (along with features such as inheritance and polymorphism) prepares the way for computer science majors to study object-oriented programming in greater depth in subsequent courses without having to learn a completely new language. C++'s similarity to C enables other students to make use of a widely-used contemporary language for basic programming. C++'s simplicity for a simple program and of input and output enables students to focus on content rather than form. Of course, as languages evolve and new languages are created, other languages in the years ahead may supplant C++. Yet, like Fortran, the parent language C will probably remain popular for several decades, at least among certain users, and since it is relatively easy to move from a basic version of C++ to C, learning a subset of C++ would probably be of benefit to most students in an elementary course, and will provide a solid foundation for more eager students to learn "cousin" languages such as Java or Perl.

7. Conclusion

A typical U.S. academic term is 15 weeks (a semester) or 10 weeks (a quarter) long, with 150 minutes of class time per week (usually subdivided into three 50-minute classes). (This constitutes a common "3 unit course" in either system.) Unfortunately, many faculty often find that there is not enough time to cover all the desired topics for any course. Moreover, in technical courses the intensity of the material usually means that the instructor and students alike are not at their mental best during the last weeks of a term.

The list of topics I enumerated earlier is more than enough for a typical term. (For discussion and reference purposes I have provided a sample syllabus as an appendix.) Foundational material is also best learned by repetition. Thus, students in elementary courses may benefit more by having several assignments making use of relatively low-level techniques in different ways rather than being exposed to more complicated material without having first mastered the foundational topics. Often programming projects can be so arranged that a subsequent assignment incorporates an earlier project, adding a feature not required in the earlier assignment, while introducing increasingly more significant concepts.

There are a number of topics that I suggest might be introduced in CS 1, but gone over in more depth in CS 2, for example, pointer variables, lists, trees, abstract tree traversals, advanced sorting techniques, recursion, "objects," among others. (Also, in CS 2, a more thorough introduction to operating system techniques and language dependent features, such as Unix makefiles and C++ header files, might also be very appropriate.) From my perspective, a wide variety of undergraduate students would benefit from being exposed to such topics and knowing of their existence and this, in my mind, is one reason why such topics should be introduced and discussed, at least minimally, in an introductory course. But given the vast array of topics in which computer science majors need to become proficient, I feel that, pedagogically, it is best to examine such topics in greater depth in a CS 2 course, and then, perhaps yet again, from a different perspective, in a course such as one dealing with the analysis of algorithms. Returning to the same topic over and over again, in different courses and from different perspectives, each time in a little more depth, will probably make a much more lasting impression than a single exhaustive presentation. Such repetition of topics through a series of courses can be as important to learning any subject as repetition within a single course.

My purpose in this essay has been to offer a few thoughts based on my personal experience, as well as on the experience of others involved in laying a foundation for university students of computer science. My experience is shaped by the U.S. educational system and may not translate well to other systems. Yet, I feel that throughout the world most introductory courses are faced with the same dilemma: how to introduce students to computer science with a programming language that is useful yet simple, is powerful yet does not overwhelm, supports the best of modern programming insights yet does not frustrate a non-technical student. The ideal language may still need to be created!

8. Appendix

Sample Syllabus

The following syllabus suggests 30 50-minute classes for covering the indicated foundational material. Additional topics are indicated below.

Unit 1 (3 classes): History of Programming Languages, Introduction to Programming and Problem Solving, Technical Details (e.g., how to telnet, how to use an editor, how to compile a sample program), general programming concerns (e.g., documentation, commenting of code, indentation, useful identifier names), skeleton C++ program.

Unit 2 (3 classes): Data types, scalar variables, arithmetic.

Unit 3 (1 class): Simple input and output (and the use of files).

Unit 4 (1 class): Counted repetition (`for` loops).

Unit 5 (3 classes): Boolean expressions, `if` statements, Conditional loops (`while` loops).

Unit 6 (4 classes): Modular programming, library functions, user-defined functions, parameter passing schemes.

Unit 7 (4 classes): Arrays, sorting, searching.

Unit 8 (2 classes): Records variables, structures and classes.

Unit 9 (2 classes): Abstract Data Types (particularly those involving both data and member functions in an object), objects and classes, the "stack" (with a sample implementation using an array).

Unit 10 (2 classes): Recursion, mathematical and programming examples (e.g., fibonacci numbers, factorials, powers).

Unit 11 (3 classes): Pointer variables, linked lists, trees, tree traversals.

Unit 12 (2 classes): Technology and Society, Ethical Concerns.

Depending on the number of additional classes available in the academic term and the language used, appropriate additional topics may include: character data and strings, class constructors and destructors, overloading functions and operators, inheritance, templates, dynamically allocatable arrays, C's standard input/output functions (i.e., `stdio.h`), other C/C++ structures and features (constant identifiers, `switch` structure, conditional operator, friend member functions, static variables, `const` parameter attribute, header files, parameters for `main`), additional Unix features (use of a makefile, separate compilation of files, file redirection), alternative implementation of a stack using a linked list, overview of computer architecture and machine level programming (what does a compiler really do?).

As explained above, some additional topics may be appropriately presented for the sake of non-computing majors who will not take a subsequent computing course. Many of the additional topics should be revisited and covered in greater depth in a CS 2 course geared specifically for computing majors and minors.

References

[1] Hal Abelson *et al.*, Technical Opinion: The First-Course Conundrum, *Communications of the ACM* **38**, no. 6 (June 1995), 116–117.

[2] ACM Curriculum Committee on Computer Science Curriculum '78-Recommendations for the undergraduate program in computer science, *Communications of the ACM* **22**, no. 3 (1979), 147–166.

[3] The College Board, Advanced Placement Program Course Descriptions: Computer Science, May 2002.

[4] Computing Curricula 1991, Report of the ACM/IEEE-CS Joint Curriculum Task Force, (Allen B. Tucker, ed.), ACM Press, Baltimore, 1991.

[5] O.-J. Dahl, E. W. Dijkstra and C. A. R. Hoare, *Structured Programming*, Academic Press, London, 1972.

[6] S. Horwitz *et al.*, Technical Opinion: Why Change?, *Communications of the ACM* **38**, no. 6 (June 1995), 117–118.

[7] L. F. Johnson, Technical Opinion: C in the First Course Considered Harmful, *Communications of the ACM* **38**, no. 5 (May 1995), 99–101.

[8] E. B. Koffman *et al.*, Recommended Curriculum for CS1, 1984, *Communications of the ACM* **27**, no. 10 (Oct 1984), 998–1001.

[9] E. B. Koffman *et al.*, Recommended Curriculum for CS2, 1984, *Communications of the ACM* **28**, no. 8 (Aug 1985), 815–818.

[10] S. P. Levy, Computer Language Usage in CS 1: Survey Results, *SIGCSE Bulletin* **27**, no. 3 (Sept 1995), 21–26.

[11] D. L. Parnas, Education for Computing Professionals, *Computer* (IEEE Computer Society) **23**, no. 1 (Jan 1990), 17–22.

DENNIS C. SMOLARSKI
DEPARTMENT OF MATHEMATICS AND COMPUTER SCIENCE
SANTA CLARA UNIVERSITY
SANTA CLARA, CALIFORNIA USA

E-mail: dsmolarski@math.scu.edu

(Received May 6, 2002)