A CONCEPT-FIRST APPROACH FOR AN INTRODUCTORY COMPUTER SCIENCE COURSE*

Linda P. DuHadway, Stephen W. Clyde, Mimi M. Recker, Donald H. Cooley
Computer Science Department
Utah State University
4205 Old Main Hill
Logan, UT 84322-4205
435-797-2451
lindad@cc.usu.edu
swc@cs.usu.edu
mimi.recker@usu.edu
cooley@don.cs.usu.edu

ABSTRACT

Several weaknesses have been identified to the programming-first approach often used in introductory computer science courses. Despite these weaknesses, programming continues to be the central focus in CS1 for many institutions. This paper proposes a concept-first approach that can be integrated into existing programming-first curriculum.

The approach is based on three principles: a) drawing from the students' everyday experiences to introduce new ideas and skills; b) allowing students time to acquire a foundation in these concepts before introducing a high-level programming language; and c) separating fundamental concepts from language syntax. A feasibility study for integrating this new approach in a CS1 class at USU is described.

INTRODUCTION

The introductory programming series for an undergraduate degree in Computer Science at Utah State University consists of three required courses based on ACM's Computing Curricula guidelines for CS1, CS2, and CS3. USU has been using C++ in these courses for about eight years. Prior to that, C was the language of choice. In addition, instructors have traditionally followed an imperative-first approach [2], which is a programming-first model with emphasis given to control structures in the first semester. Although these courses are required for CS majors, they are designed to accommodate students from other fields who want or need to learn basic programming skills. On the average, only 12% of the students taking CS1 are CS majors. The averages jump to 20% and 65% for CS2 and CS3, respectively.

USU's CS Department also offers an optional course in "Problem Solving with Computers" (CS1050) that is designed to teach general problem solving skills and provide an overview of computer science. Most CS majors do not take CS1050 and if they do, it would typically be before CS1. Using a breadth-first approach [2], CS1050 provides students with non-technical introductions to algorithms, program performance, operating systems, assembly languages, architecture, programming concepts and software development. It also includes a 2-3 week section on C++ that comes towards the end of the semester. Consistent with the breadth-first approach, students obtain an overview in CS before having to deal with any particular programming language.

The authors have observed that the background obtained in the first part of CS1050 facilitates the learning of programming skills. The programming section in this course is brief, but students are able to solve problems by writing C++ programs that include variable declarations, mathematical calculations, I/O, conditional statements, and looping structures. The students seem comfortable learning the syntax of the language and the development environment.

A recent survey found that a student's comfort level was the best indicator of success in an introductory computer science course [9]. Based on informal observations in tutor labs, the authors have noticed a difference in the comfort level between the students in CS1 and those in CS1050. The students in CS1 seemed more frustrated and uncertain about which direction to go. The CS1050 students were challenged by having to write entire programs but seemed more comfortable with the process. This difference may be attributed to the fact that CS1050 students were given a foundation in programming concepts prior to having to code.

CS1 tends to be demanding due to the amount of material covered and the workload. If students have not had previous programming experience, they are faced with the challenge of learning programming concepts, C++, and a development environment simultaneously. Having to learn in three different domains simultaneously often leads to cognitive overload. When the quantity of new information is too large, students may be unable to synthesize all the information. They are then left with gaps in their understanding that make it more difficult to succeed as new material is presented. William Fone, claimed that "If the learning space is not constrained the learner may experience considerable cognitive overload and that in turn may well be counter productive" [5]. Duane Buck and David Stucki describe the impact of early

coding as leaving students " ... overwhelmed, uncertain of how to begin, and grasping at the air. Often, this leads them to the self-destructive tendency to do experimental programming, where they just randomly throw things in to see if it helps. Experimental programming has proven to waste a huge amount of students' time, in our experience, with little actual learning" [1]. This early lack of understanding can cause difficulties for the students as they continue through the course.

With such noticeable differences between students in CS1050 and those in CS1, it seems reasonable that CS1 can be improved to increase students' learning. This paper describes an idea for a concept-first approach to teaching CS1 and presents how we tested its feasibility. Section 2 describes the concept-first approach and principles. To test the idea, we wanted to integrate it into a CS1 class. However, since CS1 is the first of a three-part series, there are limitations to the changes that could be made. Specifically, we could not eliminate any topics; only rearrange how the material is presented. Section 3 describes what we actually did for the feasible study. Observations of the feasibility study are presented in Section 4. Section 5 contains a summary and proposed future work.

2. THE CONCEPT-FIRST APPROACH

The concept-first approach is based on three principles:

1. drawing on the students' everyday experiences when introducing the principles of computer science;
2. allowing the students to work within a single domain for a period of time before adding a second or third one; and
3. separating computer science concepts from language syntax.

2.1 Draw from Students' Experience

The first principle is to use everyday experiences from the students' background to introduce new programming concepts. Using common experiences can aid in bridging the gap between the background students bring with them and this new world of programming. Creating an environment where instruction can benefit from previous experience is a concept with much theoretical and empirical support within a particular learning theory, called Schema Theory. Marcy Driscoll discusses this aspect of Schema theory:

> Most learners already know something about any new topic they are asked to study, or they can make meaningful connections between what they know and what they are being asked to learn. However, possessing relevant prior knowledge is no guarantee that learners will activate and use it appropriately. In an instructional situation, then, the activation of prior knowledge should not be left to chance. To assure that meaningful learning takes place, instructors and designers can employ a variety of strategies to help learners relate their prior knowledge to new information they are to acquire. [3]

2.2 Focus on a Single Domain

The second principle is to allow the student to gain familiarity with one of the domains of CS before adding a second or third domain. The combined domains for a curriculum represent a "learning space" [5]. For an introductory programming course that follows an imperative-first approach, the learning space is large and complex. It typically consists of three domains: general programming concepts, a programming language, and a development environment. It takes time for a student to assimilate new material from any one these domains. Expecting them to learn new material from all three domains simultaneously may be too much for many students. When students become overwhelmed, their learning efficiency and comfort levels drop. On the other hand, if the learning space is kept as focused as possible, students are more likely to succeed.

2.3 Separating Concepts from Syntax

When instructors use language syntax to teach computer science concepts, the concepts are frequently overshadowed. The ACM Curriculum guidelines state:

> Programming courses often focus on syntax and the particular characteristics of a programming language, leading students to concentrate on these relatively unimportant details rather than the underlying algorithmic skills. This focus on details means that many students fail to comprehend the essential algorithmic model that transcends particular programming languages. Moreover, concentrating on the mechanistic details of programming constructs often leaves students to figure out the essential character of programming through an ad hoc process of trial and error. Such courses thus risk leaving students who are at the very beginning of their academic careers to flounder on their own with respect to the complex activity of programming. [2]

By separating concepts from their implementation in a high-level programming language, the concept-first approach provides a way for students to learn and understand the underlying principles as stand alone ideas that exist outside any single programming language. Separating concepts from language syntax helps build a cognitive framework that gives students a structure on which they can hang new ideas. Also, spending time on a programming concept independent its realization in any particular language will help students understand that the concept is significant and more universal than just what's found in a single language. Finally, when a student gains understanding of a fundamental programming concept beyond its use in a single language, then that concept is more likely to become part of a meaningful foundation for future studies.

This separation between concept and syntax needs to be a separation in time as well as presentation. It is possible to discuss a concept during the first part of class and talk about the syntax used to implement the concept in the same class period. Even though there is a separation in the discussion, there may not be enough separation in time for the students to assimilate the concept before having to deal with the syntax. Consequently, the concept may still be lost.

2.4 Benefits of the Concept-First Approach

Although benefits may be obtained by following any one of the above principles, the most significant benefits come from adhering to all three. These anticipated benefits include the following:

- An increase in the efficiency of learning syntax. With a foundation in concepts, the students will be able to learn the syntax of a programming language more quickly.
- Improved quality in the students understanding of programming constructs. The programming constructs would be employed more effectively because the students have an increased awareness of the principles underlying them.
- There would be an improvement in the comfort level of the students taking CS1. Often the comfort level of students is not considered significant. In fact, most studies that research what factors impact success in introductory programming classes do not even address this factor. But one survey that asked the question, found "Comfort level in the computer science class was the best predictor of success in the course" [9]. Even more significant than math background.

3. FEASIBILITY STUDY

The existing CS1 curriculum covers the first ten chapters of *Starting Out With C++* [6]. Table 1 lists the topics. Each chapter is taught by working though coding examples. Students complete homework assignments that require them to write their own code.

Table 2 – The Topics

The Topics

- Declaration and types
- Mathematical Calculations
- Input/Output
- Conditional Statements

To test the feasibility of the concept-first approach, the authors applied the principles described in Section 2 to develop a new curriculum for this course and then taught one of the four sections offered during the Fall 2001 semester using this curriculum.

The new curriculum consisted of two weeks that focused exclusively on programming concepts and thirteen weeks covering the topics outlined above. This two-week introductory section was a direct implementation of the separating of concepts from syntax and allowing the students to focus on the concept domain exclusively. It also facilitated drawing on the students' experience. Examples familiar to the students could be more directly associated with the concepts than the topics. For example, it was easier to relate an activity that used repeated steps (practice 15 lay ups and then do 20 free throws) to the concept of counted loops than directly to a for-loop.

Table 3 – The Concepts

The Concepts

- Problem solving in a step-by-step fashion
- Sequential operations
- Making choices based on a condition
- Repeating steps using a counted loop
- Repeating steps using a conditional loop
- Reading and interpreting problem statements
- Writing instructions using a specific language
- Working with variables

As each concept was taught, examples were used that were familiar to most of the students. There was discussion about what the students already knew and how that knowledge could be applied specifically to CS. During this time a common background was created among the students. Together they learned these fundamental principles. This commonality became a foundation on which to teach problem solving using a programming language.

Students applied these concepts using "Our Psuedocode" (see Appendix), a simple psuedocode language consisting of only seven commands and few rules. This language is brief and written in understandable language making it easy to learn and master. Problem statements were given to students on a daily basis. These problems increased in complexity. They were expected to write solutions using "Our Psuedocode". The assignments required them to apply class discussion to resolve the problem statements. In this way students gained hands-on experience of creating a solution, in a specific language to a specific problem.

We share one example. The fourth class period was spent on conditional loops. We presented common examples of repeating an action until a condition was met such as take bites until your plate is empty or until you are full. Then we presented the one statement in Our Psuedocode that performs such a task: While (condition) remains true do … End of while. Critical issues relating to the use of this structure like priming the loop, changing the condition inside the loop, being accurate in the statement of the condition, and the possibility of an infinite loop were discussed in detail. Several examples of programs that used this looping structure were presented and the students were sent home with two problems of their own to solve. Following are those problems:

1. Write a program with a loop that lets the user enter a series of integers. The user should enter –99 to signal the end of the series. After all the numbers have been entered, the program should display the sum of all the numbers entered.

2. Write a program that does division. Ask the user for two numbers. Have the program calculate the answer for the first number divided by the second number. If the user tries to divide by zero, write an error message. Make the program loop so that the user can enter as many pairs of numbers as they would like. For each pair of numbers, perform the calculation. Have the user enter a "C" to continue and a "Q" to quit [4].

This assignment used the concepts the students had already learned and added the new feature of using conditional looping. Solving these problems was well within the grasp of the students. They could use their understanding of conditional loops without worrying about syntax issues like what type of variables needed to be used or what header files needed to be included.

By the time the curriculum covers the C++ implementation of the while loop, the issues learned in this class period are familiar and have been used in a variety of settings. Adding the syntax requirements of C++ is an easy step.

As the class moved into the topic section of the course, the first assignment was to write a simple C++ program The students were expected to write this program in a standard word processor which is how they had been writing the psuedocode solutions. Then the development environment was introduced and they began coding solutions using that environment. These two domains, the language and the development environment could not effectively be separated more than a day or two. Once a student is familiar with the environment, it becomes an aid to learning the language. But this single assignment gave the students a chance to write C++ in a familiar environment before adding the third domain.

Now, we could use examples from our study of the concepts to relate the students' past to their current learning. This increased the available set of background experiences to draw from. These experiences were more valuable because they were recent, closely related to the current subject and common to the entire class. During the thirteen-week portion of the class, the topics in table 1 were covered with an emphasis on drawing on the students' background, both the background they had brought to the course as well as the background developed during the first two-week introductory section.

4. OBSERVATIONS

This model provided evidence that it is possible to take two weeks out of the standard fifteen weeks of instruction, teach concepts separate from a high-level language, and still cover the topics that are covered in a more traditional presentation. This was one of the main purposes of the feasibility study. When this concept-first approach was originally presented, the greatest concern was that it would not leave adequate time to cover all the required topics. However, the background in concepts had increased the efficiency of learning syntax and thirteen weeks was a satisfactory amount of time to teach those topics.

This study was not sufficient to identify if the programming constructs had been employed more effectively. But the speed at which students picked up some of the constructs was an indication that they had an understanding of the fundamental concepts. A few times the class

was able to cover in a single day what had historically taken an entire week. One of the greatest benefits occurred when we reached the if/else structure. Aware of the class' background in condition statements, the lesson was written to cover several sections in one class period. After completing the prepared information with time still remaining, the students looked as though they were still waiting for something new. The class, as a whole, had a background in the concept of conditional statements. They had applied the concept using "Our Psuedocode". Now, when presented with the topic of if/else structures in C++, they were comfortable and readily learned the syntax to apply their knowledge in this new setting. This experience of finding new C++ topics familiar happened several times throughout the semester.

Again, this study did not control enough factors to measure the comfort level of students learning with the concept-first approach compared to their counterparts being instructed by the more traditional approach. It was the observation of the instructor that the students responded well to this approach. Several students commented that they appreciated the introduction before being expected to write programs.

Another insight was the role that semantics played in this approach. Semantics were not explicitly taught with the concepts but because they are inherently part of both the concepts and the syntax, they served as a link between the two domains. When the student learns the C++ syntax of something they have already learned conceptually, the commonality of the semantics helps the student connect the syntax with the concept.

It is a fundamental principle that instruction benefits from using examples that are familiar to students [3][7]. It was interesting to find how much easier it was to come up with real world examples at the conceptual level. Originally we anticipated that most of what would be introduced to the students would be new information. What we discovered was that students already had experience with many of the fundamental programming concepts we are teaching. For example, they have made decisions based on certain conditions. They have followed sets of instructions. They already have solved story problems in math and followed grammar rules in English. Each student comes with experiences that can be associated with the programming concepts being taught. Identifying for the student what they already know and what is to be learned helps them to understand that they have already acquired knowledge and skills that will be useful when learning computer programming. It also reminds them to use their background as they learn this new subject. As examples were presented, the instructor pointed out similarities with previous experience and identified how new concepts were different. In this way, students were able to use analogies from their real-world experiences to help them understand computer science topics. Indeed, research in cognitive psychology suggests that analogical thinking can be a powerful means for understanding novel concepts [3].

Background from the students' educational training and daily living provide a valuable resource from which to develop concepts and lay a foundation for the study of computer programming. As an instructor explicitly identifies some experiences common to the majority of students and applies them to programming, students gain a clearer understanding of how their background has prepared them for the study of computer science. For the students who feel they are starting from scratch, this exposure can provide a more comfortable entry into the

world of CS and can increase their confidence. They will have a clearer understanding of what they bring to the table.

5. SUMMARY AND FUTURE WORK

This model provided support for the feasibility of the concept-first approach. It would be interesting to set up an experiment where the differences between sections were controlled. One section would cover the core curriculum as outlined in this paper and the other section would take the traditional approach. Learning effectiveness, efficiency, and quality could then be assessed via outcome measures and a satisfaction survey. This study would provide quantifiable information about the effectiveness of this approach.

This concept-first approach can be implemented within an existing timeframe and provides subject matter coverage that prepares the student for upcoming courses in the introductory CS series. It improved the efficiency of learning language syntax. It provides the functional skills hoped for by a programming-first approach. There are perceived and theoretical benefits to this approach. This separation of concepts from syntax may provide a variety of benefits such as limiting the amount of new information the student is exposed to at one time thereby reducing the risk of overload. Another benefit is the creation of a common background within the class. This background can be developed during the introductory unit and then used as a platform from which to continue the study of CS using a programming language.. The lessons learned in this feasibility study, during the process of developing and implementing this alternative curriculum, and what may be learned from future research can be useful in designing and improving similar courses.

APPENDIX - OUR PSEUDOCODE

Our Pseudocode

computation:
  Set value of *variable_name* to *arithmetic_operation*
Input/Output:
  Get a value for *variable_name* , *variable_name* ,…
  Print value of *variable_name* , *variable_name* „…
  Print the message " *text* "
Conditional:
  if (*condition*) is true then
    operation
    .
    .
    .
  else (else is optional)
    operation
    .
    .
    .
  end of if
Iterative:
  Repeat *number* times
    operation
    .
    .
    .
  End of repeat
While (*condition*) remains true do
    operation
    .
    .
    .
  End of while
Replace italicized text.
You can use the following mathematical operators:
+ - / * ( ) < > <= >= = !=
Indent operations that are part of conditional or iterative statements.
End program with the statement Stop.
Enclose conditions in ( ).
Enclose message text in double quotes.
Variable names are only one word long, and consist of letters, digits, and the underscore. They can start with a letter or the underscore.
  Ex: height, length_of_side, grossWage
Variable names need to be descriptive.
Adapted from "An Invitation to Computer Science" [8]

Example:
Write an algorithm that allows the user to input 3 numbers and print out the average of those numbers.

Solution 1:
Print the message "Please enter a number"
Get a value for first_number
Print the message "Please enter a number"
Get a value for second_number
Print the message "Please enter a number"
Get a value for third_number
Set value of average to (first_number + second_number
  + third_number) / 3
Print the message "The average is "
Print value of average
Stop

Solution 2:
Print the message "input 3 numbers"
Get a value for number1, number2, number3
Set value of average to (number1 + number2
  + number3) / 3
Print the message "The average of the numbers is "
Print value of average
Stop

Solution 3:
Set value of total to 0
Repeat 3 times
  Print the message "input a number"
  Get a value for number
  Set value of total to total + number
End of repeat
Set value of average to total / 3
Print the message "The average of the 3 numbers is "
Print value of average
Stop

REFERENCES

[1] Buck, D. and Stucki, D.J. "JkarelRobot: A Case Study in Supporting Levels of Cognitive Development in the Computer Science Curriculum", Proceedings of SIGSCE Symposium, 2001, 16-20.

[2] Computing Curricula 2001, Computer Science Volume, Chapter 7 Introductory Courses. December 15, 2001. http://www.acm.org/sigcse/cc2001/cs-introductory-courses.html

[3] Driscoll, M.P., *Psychology of Learning For Instruction,* Allyn & Bacon, 1999.

[4] DuHadway, L.P., Dinerstein, K., and Earl, A. C., CS1700 Course Website, http://eagle.cs.usu.edu/cs1700/

[5] Fone, W. "Using a Familiar Package to Demonstrate a Difficult Concept: Using an Excel Spreadsheet Model to Explain the Concepts of Neural Networks to Undergraduates." Proceedings of ITISCE, 2001, 165-168.

[6] Gaddis, T., *Standard Version of Starting Out with C++, Third Edition,* Scott Jones Publishers, 2001.

[7] Rising, L. "Teaching by example is the only kind of teaching!", OOPSLA Educators Symposium, 1999.

[8] Schneider, G.M. and Gersting, J.L., *An Invitation to Computer Science,* PWS Publishing 1998.

[9] Wilson, B.C. and Shrock, S. "Contributing to Success in an Introductory Computer Science Course: A Study of Twelve Factors", SIGCSE 2001, 184-188.