A Mid-Career Review of Teaching Computer Science I

Amruth N. Kumar
Ramapo College of New Jersey
505 Ramapo Valley Road
Mahwah, NJ 07430, USA
1 201 684 7712
amruth@ramapo.edu

ABSTRACT
A mid-career review is presented, of how the teaching of Computer Science I has changed for this instructor over the last two decades. The content of the course has evolved to include algorithm development and program design. Assessment in the course has gone online and moved away from testing how clever the student is, to how much the student has learned in the course. Professional practices are now covered that help students understand and incorporate preferred practices of the discipline. Changes incorporated into the pedagogy include going from using anthropomorphic and ad-hoc to discipline-specific and consistent vocabulary, and from writing code in the class like an experienced programmer to writing it to suit a beginning learner. It is hoped that this review will help new Computer Science I instructors avoid some misconceptions with which this instructor started out.

Categories and Subject Descriptors
K.3.2 [Computer and Information Science Education]: Computer Science Education

General Terms
Documentation, Experimentation

Keywords

1. INTRODUCTION
I have been teaching Computer Science I for two decades now. This seemed like a good time to stop and reflect on my journey as the instructor of this introductory bread-and-butter Computer Science course. It is a journey many instructors have taken before, and many will take in the years to come. What is striking is how much my teaching of the course has changed over these years – its content, assessment, presentation and pedagogy. It is my hope that this documentation of the changes might help freshly minted Computer Science educators skip a step or two in their own inevitable journeys. Over the years, a lot of good educators and researchers have wrung their hands about dismal retention rates in Computer Science I. I have a hunch that some of the reasons for attrition in my own offerings of the course are hiding in plain sight in this article reflecting my journey. It is hoped that cataloging the practices that I have since abandoned might contribute to the discussion on what works and what does not in the teaching of Computer Science I.

This account of the evolution of my Computer Science I course is not meant to be a qualitative or quantitative study of how novices learn - there is abundant literature on how novice programmers learn (e.g., [5]) as well as the differences in the program comprehension of novices versus experts (or instructors) (e.g., [4]). This article presents what makes for effective student learning in Computer Science I from the perspective of one instructor who has taught the course for two decades, and therefore, is in the tradition of phenomenographic studies (e.g., [12]). By necessity, and with apologies to the reader, this account will contain a lot of sentences starting with the first person singular pronoun.

In the following sections, I will present four aspects of teaching Computer Science I: content, assessment, presentation and pedagogy. I will present each aspect in terms of prior practice, a retrospective analysis and current practice.

2. CONTENT
Prior Practice: When I first started teaching Computer Science I, it was all about teaching the syntax and semantics of a programming language in class. The piece de resistance was the sample code written on the board illustrating the programming construct. To be fair, back then, Computer Science I was a purely lecture course, and this seemed like the most logical content to teach – this is what the students did not know coming into the course, and this is what could be taught in a lecture format without recourse to computers. In addition, students were assigned half a dozen programming projects to practice programming constructs. This is where students actually got the opportunity to write a program on a computer and try out the concepts “learned” in class. On the tests, students were expected to write code for a given problem, debug given code, and identify the output of given code, all on paper, without the benefit of a computer or compiler.

In Retrospect: It is now common knowledge that writing a program for a given problem involves at least the following steps:
1. **Problem-Solving**, the process of designing an algorithm for the given problem;

2. **Program Design**, the process of identifying the variables and designing the control constructs needed to implement each stage of the algorithm;

3. **Coding**, the process of writing code to incorporate the variables and control constructs identified during program design;

4. **Debugging and testing.**

In retrospect, in class, I was teaching to step 3, viz., coding, and some amount of step 4, potential pitfalls of programming constructs. The written tests also assessed only these two steps. However, on the projects, the students were expected to execute all four steps. Clearly, there was a mismatch between what was being taught and what was being assessed. Since projects are the most immediate and on-going form of self-assessment for students, i.e., students decided whether they were keeping up in the course or not based on their project performance, the mismatch may have led to a lot of frustration and eventually, attrition among my students.

Is it really necessary to explicitly teach steps 1 and 2 – problem-solving and program design? Computer Science I textbooks generally do not cover these topics at all or in any great detail. Personally, I do not recall ever being taught how to solve real-life problems to make computing more interesting and eventually, attrition among my students.

Going back to my own experience as a student decades ago, I recall being perplexed, not quite sure how to come up with an algorithm for a given problem. So, when in doubt (or lost), my practice was to just start coding, hoping the solution would transpire as I went along – so much for top-down decomposition! In my experience, average students continue to have trouble designing the algorithm for a given problem. This stands to reason – when confronted with a problem in everyday life, human beings go into an iterative plan-act sequence (e.g., want to get gas? Drive to the nearest gas station. Station is closed? Drive onwards to the next station). In contrast, writing an algorithm involves sketching out all the steps, not just the next one, and taking into account all the possibilities, not just the one immediately confronting us (e.g., what if the gas station has run out of the type of gas I want?). This is further complicated by the push in educational circles to use real-life problems to make computing more interesting and relevant to students (e.g., [13]) - coming up with the algorithm for a real-life problem can be even more daunting to students, who must now also resolve the ambiguities inherent in the problem statement of a real-life problem (e.g., does Ace count as 1 or 10 in Blackjack?).

Students not only have problems coming up with the algorithm, but also struggle with program design, as is clear from reading the code they write – programs with superfluous variables, unnecessary nesting of if-else statements, conjunctive conditions where simple conditions would have sufficed; read-only parameters passed by reference to functions, an entire array passed as parameter to a function that only references one element in it, variables referenced in a single member function declared as class variables, etc.

**Current Practice:** My conclusion was, yes, it is necessary to explicitly teach problem-solving and program design techniques in Computer Science I. I now cover these topics in every class, in the context of every programming construct. In each class, after assigning a new problem, I tease out the algorithm for the problem in collaboration with the students, and write down the algorithm on the board. This is followed by program design in consultation with students:

- Data analysis, i.e., analysis of the variables needed for the program and their desired data types and forms
- Control analysis, i.e., determination of the control structures needed for each step in the algorithm. Control analysis includes showing the students how to translate information in the problem statement into tables and fork diagrams [11] that help them properly nest if-else statements; how to design loops [10] and introduce accumulator variables into a loop after it is designed; and how to pick the signature of a function based on its desired input/output characteristic.

- In later classes, control flow analysis, i.e., the desirable order in which functions should be called, and data flow analysis, i.e., the movement of data as parameters among functions and classes.

All of these help students write a program for a given problem from scratch. Explicitly covering them and consistently repeating them for each problem help students develop, practice and internalize algorithm development and program design – the first two steps in programming that I was not covering earlier. Similar program design techniques have been advocated by the very successful Program By Design project [16].

Finally, we now use closed labs [7] in Computer Science I, as do many other schools. Closed labs offer the opportunity for students to practice all four steps of programming listed earlier, under the supervision of the instructor. We go through the first two steps – problem-solving and program design – during the lecture, with me taking the lead on the board and students assisting me. Students complete the last two steps – coding, and debugging and testing – in the closed lab – they take the lead (individually) and I assist them when necessary. In other words, the class is now “flipped” [3].

3. **Assessment**

**Prior Practice:** When I started out, the tests in my course were written, not online. Students were expected to write code for a given problem, debug given code, and identify the output of given code, all on paper, without the benefit of a computer or compiler.

**In Retrospect:** In retrospect, both the form and content of the tests were off the mark. I thought a good test question was one whose answer was not obvious. Now, I understand that a good test question is not about finding out how clever a student is, but how much the student has learned in the class. For example, consider the following C++ problem that I would have assigned to the student to assess knowledge of parameter passing – the student is asked to identify the output of the code:

```cpp
void insert (int value, int data[], int & size )
```

```cpp
int main()
{
    int array [3] = {1, 2, 3};
    int size = 3;
    insert (5, array, size);
    return 0;
}
```

- Data analysis, i.e., analysis of the variables needed for the program and their desired data types and forms
- Control analysis, i.e., determination of the control structures needed for each step in the algorithm. Control analysis includes showing the students how to translate information in the problem statement into tables and fork diagrams [11] that help them properly nest if-else statements; how to design loops [10] and introduce accumulator variables into a loop after it is designed; and how to pick the signature of a function based on its desired input/output characteristic.

- In later classes, control flow analysis, i.e., the desirable order in which functions should be called, and data flow analysis, i.e., the movement of data as parameters among functions and classes.

All of these help students write a program for a given problem from scratch. Explicitly covering them and consistently repeating them for each problem help students develop, practice and internalize algorithm development and program design – the first two steps in programming that I was not covering earlier. Similar program design techniques have been advocated by the very successful Program By Design project [16].

Finally, we now use closed labs [7] in Computer Science I, as do many other schools. Closed labs offer the opportunity for students to practice all four steps of programming listed earlier, under the supervision of the instructor. We go through the first two steps – problem-solving and program design – during the lecture, with me taking the lead on the board and students assisting me. Students complete the last two steps – coding, and debugging and testing – in the closed lab – they take the lead (individually) and I assist them when necessary. In other words, the class is now “flipped” [3].

3. **Assessment**

**Prior Practice:** When I started out, the tests in my course were written, not online. Students were expected to write code for a given problem, debug given code, and identify the output of given code, all on paper, without the benefit of a computer or compiler.

**In Retrospect:** In retrospect, both the form and content of the tests were off the mark. I thought a good test question was one whose answer was not obvious. Now, I understand that a good test question is not about finding out how clever a student is, but how much the student has learned in the class. For example, consider the following C++ problem that I would have assigned to the student to assess knowledge of parameter passing – the student is asked to identify the output of the code:

```cpp
void insert (int value, int data[], int & size )
```
In Retrospect: That, they did. For example, when writing an English essay, one might plough through a draft of the entire essay before stopping to check the cohesion of its narrative. Students often assume that the process of writing a program is similar – write the entire program, call it “mostly done”, and then try to compile it. When writing an English essay, one would write the paragraphs and sentences within each paragraph in a linear order. Students assume that a program is written the same way – in a linear order top to bottom, an order that reflects neither the problem-solving process nor the program design process. I found that it was necessary to explicitly clarify the process of programming and proactively disabuse students of misconceptions borrowed from other disciplines.

Current Practice: When students write a program in each class, I insist that they edit, compile, run and test their program after each step in the algorithm. For good measure, in addition to writing the steps in the algorithm on the board, I number those steps in the order in which the student must attempt them, and insist that students attempt the steps in this order. The advantages of enumeration and incremental compilation are numerous:

- Any new bugs can be localized to the most recently developed step in the algorithm. This is similar to the concept of Programmer Test (itself a modification of Unit Test) in Extreme Programming [1].
- Students get positive reinforcement from seeing their partial program work, which builds their self-confidence in a virtuous cycle.
- Students do not realize that the order in which they attempt steps in the algorithm is important. In my experience, if students feel they know the code for a step, they will attempt it first, regardless of whether it is the first or the last step in the algorithm. For example, they might attempt to print the value of a variable before assigning to it or in some cases, even declaring it! When they do, and inevitably get stuck, guiding them through the process of climbing out of the mess on their own without just giving away the desired code is very frustrating and time-consuming. (Yet, in a closed lab, when the rest of the class is bored and rearing to go to the next task, it is too tempting to throw in the towel and simply give the student the desired code). Thanks to the ordering of the steps, I now simply ask them to roll back their program.

4. PRESENTING PROFESSIONAL PRACTICES

Prior Practice: Even after I began to cover problem-solving and program-design techniques in class, and started using closed labs in the course, it did not occur to me that it was important to discuss preferred programming practices and that in the absence of such discussion, students would bring to bear, models learned in other disciplines that were inadequate or wrong for programming.

In Retrospect: Pen and paper questions can be properly designed to test whether the student understands programming concepts. But they do not test whether the student can turn around and write a program involving those concepts. Since the objective of introductory Computer Science is problem-solving and programming, I now use online tests in the course [9]. In the online tests, students design, write, and test a program for a given problem. By offering hints during the test, and awarding partial credit for incomplete programs, I try to reduce the stress associated with writing a program under deadline. Online tests reward students who complete the most number of projects and therefore, have independently practiced problem-solving and program design techniques [8].

Current Practice: Presenting professional practices

Prefered programming practices

- Any new bugs can be localized to the most recently developed step in the algorithm. This is similar to the concept of Programmer Test (itself a modification of Unit Test) in Extreme Programming [1].
- Students get positive reinforcement from seeing their partial program work, which builds their self-confidence in a virtuous cycle.
- Students do not realize that the order in which they attempt steps in the algorithm is important. In my experience, if students feel they know the code for a step, they will attempt it first, regardless of whether it is the first or the last step in the algorithm. For example, they might attempt to print the value of a variable before assigning to it or in some cases, even declaring it! When they do, and inevitably get stuck, guiding them through the process of climbing out of the mess on their own without just giving away the desired code is very frustrating and time-consuming. (Yet, in a closed lab, when the rest of the class is bored and rearing to go to the next task, it is too tempting to throw in the towel and simply give the student the desired code). Thanks to the ordering of the steps, I now simply ask them to roll back their program.

4. PRESENTING PROFESSIONAL PRACTICES

Prior Practice: Even after I began to cover problem-solving and program-design techniques in class, and started using closed labs in the course, it did not occur to me that it was important to discuss preferred programming practices and that in the absence of such discussion, students would bring to bear, models learned in other disciplines that were inadequate or wrong for programming.

In Retrospect: That, they did. For example, when writing an English essay, one might plough through a draft of the entire essay before stopping to check the cohesion of its narrative. Students often assume that the process of writing a program is similar – write the entire program, call it “mostly done”, and then try to compile it. When writing an English essay, one would write the paragraphs and sentences within each paragraph in a linear order. Students assume that a program is written the same way – in a linear order top to bottom, an order that reflects neither the problem-solving process nor the program design process. I found that it was necessary to explicitly clarify the process of programming and proactively disabuse students of misconceptions borrowed from other disciplines.

Current Practice: When students write a program in each class, I insist that they edit, compile, run and test their program after each step in the algorithm. For good measure, in addition to writing the steps in the algorithm on the board, I number those steps in the order in which the student must attempt them, and insist that students attempt the steps in this order. The advantages of enumeration and incremental compilation are numerous:

- Any new bugs can be localized to the most recently developed step in the algorithm. This is similar to the concept of Programmer Test (itself a modification of Unit Test) in Extreme Programming [1].
- Students get positive reinforcement from seeing their partial program work, which builds their self-confidence in a virtuous cycle.
- Students do not realize that the order in which they attempt steps in the algorithm is important. In my experience, if students feel they know the code for a step, they will attempt it first, regardless of whether it is the first or the last step in the algorithm. For example, they might attempt to print the value of a variable before assigning to it or in some cases, even declaring it! When they do, and inevitably get stuck, guiding them through the process of climbing out of the mess on their own without just giving away the desired code is very frustrating and time-consuming. (Yet, in a closed lab, when the rest of the class is bored and rearing to go to the next task, it is too tempting to throw in the towel and simply give the student the desired code). Thanks to the ordering of the steps, I now simply ask them to roll back their program.

4. PRESENTING PROFESSIONAL PRACTICES

Prior Practice: Even after I began to cover problem-solving and program-design techniques in class, and started using closed labs in the course, it did not occur to me that it was important to discuss preferred programming practices and that in the absence of such discussion, students would bring to bear, models learned in other disciplines that were inadequate or wrong for programming.

In Retrospect: That, they did. For example, when writing an English essay, one might plough through a draft of the entire essay before stopping to check the cohesion of its narrative. Students often assume that the process of writing a program is similar – write the entire program, call it “mostly done”, and then try to compile it. When writing an English essay, one would write the paragraphs and sentences within each paragraph in a linear order. Students assume that a program is written the same way – in a linear order top to bottom, an order that reflects neither the problem-solving process nor the program design process. I found that it was necessary to explicitly clarify the process of programming and proactively disabuse students of misconceptions borrowed from other disciplines.

Current Practice: When students write a program in each class, I insist that they edit, compile, run and test their program after each step in the algorithm. For good measure, in addition to writing the steps in the algorithm on the board, I number those steps in the order in which the student must attempt them, and insist that students attempt the steps in this order. The advantages of enumeration and incremental compilation are numerous:

- Any new bugs can be localized to the most recently developed step in the algorithm. This is similar to the concept of Programmer Test (itself a modification of Unit Test) in Extreme Programming [1].
- Students get positive reinforcement from seeing their partial program work, which builds their self-confidence in a virtuous cycle.
- Students do not realize that the order in which they attempt steps in the algorithm is important. In my experience, if students feel they know the code for a step, they will attempt it first, regardless of whether it is the first or the last step in the algorithm. For example, they might attempt to print the value of a variable before assigning to it or in some cases, even declaring it! When they do, and inevitably get stuck, guiding them through the process of climbing out of the mess on their own without just giving away the desired code is very frustrating and time-consuming. (Yet, in a closed lab, when the rest of the class is bored and rearing to go to the next task, it is too tempting to throw in the towel and simply give the student the desired code). Thanks to the ordering of the steps, I now simply ask them to roll back their program.
to the earliest step they skipped, either deleting or commenting out the code for all the subsequent steps. This “rollback recovery” (applied here to novice programming) borrows from similar techniques used in computing literature to recover from hardware and software faults (e.g., [6]).

So, in my current practice, I emphasize that the process is just as important as the product of programming, a sentiment echoed by other educators (e.g., [2]). I insist that they practice the process just as much as everything else.

When writing control statements such as if-else or while loop, my students are required to write the shell (language-mandated syntax) first, before filling in details such as the condition expression, if-clause and else-clause. This practice minimizes bugs resulting from mismatched or misplaced closing statements (braces, end statements). While this practice is obvious to experienced programmers, it is not obvious to novices who have only seen programs in their finished form, and not as they are being written. So, in addition to explicitly mentioning the practice during the lecture, I compulsively reinforce its use during the laboratory session, reminding students that writing code hierarchically rather than linearly will avoid a lot of gratuitous bugs and save them a lot of frustration and time.

When debugging, professional programmers draw upon the wealth of their experience to quickly identify the problem and fix it in the code. For novices though, debugging, i.e., localizing, identifying and fixing bugs is a very time-consuming process. Localizing the bug is aided by the practice of testing code after each step in the algorithm. Identifying and fixing the bug on the other hand, are skills learned through practice. We all know how unhelpful compiler errors can be – if a student fails to enclose a character literal constant (say x) within single quotes, the compiler complains that x has not been declared! This is when a beginner dutifully proceeds to declare x as a variable, which leads the programmer and the program further astray. When students are working on their own after class on a programming project, they could easily spend hours agonizing over such bugs without being the better for it. In order to help students learn from their past mistakes and save time in the process, I now instruct students to maintain a “bug library” – the symptom of each bug and its cause. During the lecture, I point out typical bugs that students should add to the library, e.g., how a semi-colon after the else keyword results in a semantic error in most popular languages. Students are allowed to bring the bug library to their online tests.

Students rarely test their code for anything other than the expected values of program input. This can be attributed to two problems: 1) Students do not know how to come up with the correct test cases; 2) Students lack or do not appreciate the discipline of comprehensively testing their code after each stage of development. The first problem is addressed by introducing the concept of boundary values, and using the number line to illustrate conjunctive and disjunctive conditions of if-else statements and loops as follows – an arrow is drawn on the number line originating at each boundary value and pointing in the direction prompted by the relational operator, e.g., hour >= 1 is drawn as an arrow originating at 1 and pointing right; hour <= 12 is drawn as an arrow originating at 12 and pointing left. Conjunctive conditions are true for the values on the number line that fall under both the arrows, and disjunctive conditions are true for the values that fall under either arrow. So, test cases must include, in addition to boundary values, at least one value on the number line under either, and both / neither arrow. This technique also helps illustrate why a condition such as hour >= 1 || hour <= 12 is always true and hour < 1 && hour > 12 is always false.

The second problem, i.e., instilling the discipline of comprehensive testing is much harder since Computer Science I students are usually less interested in taking ownership of their code than they are of meeting the benchmarks established for correct completion of the course project. One option is to record an observation journal - consisting of the test cases used and the code behavior observed – as an integral part of the comments preceding each section of code.

Iterative development of code, writing constructs hierarchically rather than linearly, maintaining a bug library, developing the discipline of testing code – these are some of the professional practices that are not self-evident, but are very helpful to Computer Science I students. While many of these ideas are traditionally covered in depth in Software Engineering course, introducing them in some rudimentary form in Computer Science I is not just a good idea, it is necessary.

5. PEDAGOGY

Prior Practice: As a beginning programmer and later, as a beginning instructor, I remember anthropomorphizing programming: “this function grabs the input, shoots its result to this other function, which then decides whether to keep it or toss it. Huh? I see my students do it now when describing their project, and wonder if they really know what they are talking about. More importantly, I doubt very much that my Computer Science I students understood what I was talking about as a beginning instructor.

In Retrospect: Anthropomorphic verbs hide insecurities. Insecurity about whether a program works (my students describing their project), how the program works (someone with a shallow knowledge of a new technology talking about it), or may be how to explain how the program works (I as a beginning instructor). Beginning students cannot replicate grabbing, shooting, deciding and tossing unless they see the code behind each of these verbs. If so, why not simply call these actions what they stand for in the code - reading input, returning value, and using selection statement? Each of these terms corresponds to specific syntax in the code and is unambiguous.

Current Practice: At the risk of sounding staid and boring to my students, I no longer use anthropomorphic verbs. Instead, I started using more appropriate computing terms. But, over the years, I found that this was not sufficient; it was also necessary to be consistent. A case in point is the phrase “returning a value by reference”, as applied to functions. Technically, returning has a specific meaning in functions, so, it should be passing a value by reference, or more generically and evocatively, sending a value back by reference. I have found that using words and phrases to consistently mean the same concept every time in class is challenging, but worth it in terms of the clarity of the presentation. It also helps students develop their own professional vocabulary.
**Prior Practice:** When writing code in class on the board, I would write it as an experienced programmer, not as an experienced teacher. The distinction is important: an experienced programmer expediently combines multiple steps in an algorithm into the same line of code. This conflation often confuses and sometimes misdirects a beginning programmer. An experienced teacher might choose to present code step-by-step, in a manner understandable and replicable by the beginning student.

**In Retrospect:** When we help a baby learn to walk or a child learn to ride a bike, we stoop to the learner’s level. The same should be true of helping novices learn to write programs. Students develop a mental model of programming based on the examples presented to them. They are just as likely to develop incorrect models as correct models. So, the examples presented to them and their sequence are both very important. For example, one of the first programs presented one semester was about calculating simple interest. In the program, unfortunately, I combined two algorithm steps, viz., calculating interest and printing it, into a single statement. Sure enough, several students inferred that all calculations had to be done in print statements! It took me weeks to disabuse them of this notion.

**Current Practice:** Now, my practice is to let the algorithm dictate the code. I no longer conflate steps in the algorithm, even when it seems very natural to do so. As an example, consider a program to calculate the total cost of a purchase. An experienced programmer might write a statement such as:

```
totalCost = quantity * costPerItem * (1 + TAX_RATE);
```

In the algorithm, though, this corresponds to three steps:

```
// Calculate the cost of all the items
// Calculate the taxes
// Calculate the total cost as the sum of the cost of all the items and taxes
```

When developing code in class, I write a separate statement for each of these three steps. Moreover, I declare two additional variables to hold the cost of all the items and the taxes. The advantages of such deliberative coding in class are many:

- Since we do not take shortcuts, we tend to repeat the same process of program design over and over again. Repetitive practice makes perfect.
- Math-phobic students find it easier to understand when each calculation involves only one operator.
- It reduces the likelihood of students developing incorrect mental models of programming, such as the one described earlier.

Of course, the faster learners quickly progress to conflating code for multiple steps in the algorithm and eliminating the need for variables to hold intermediate values in expressions. As an added bonus, they feel more empowered because they figured this out on their own rather than being instructed or shown how to do it! At the same time, the slower learners do not feel confused, overwhelmed or lost in class. So, deliberative coding is a win-win for everyone.

**Prior Practice:** As a beginning Computer Science I instructor, I thought it was very important to present in class, every subtlety and nuance of each programming construct. I felt it was important to be comprehensive in my coverage. It made logical sense to present everything related to a programming concept or construct at the same time.

**In Retrospect:** Presenting everything about a programming construct all at once quickly discharged my responsibility as a teacher, but not my obligation to help my students learn. Cognitive Load Theory recommends that information should be properly segmented and sequenced to avoid overload in learners [14]. Presenting all the nuances, exceptions and subtleties at once meant that information was being neither segmented nor sequenced.

**Current Practice:** I introduce the most general case of any programming concept or construct first, and allow students time to practice with it for a week before introducing special cases, exceptions, etc. Often, I present alternatives as either exceptions or special cases, e.g., functions returning void as a special case of non-void functions; simple statement as a special case for the body of a loop, compound statement being the general/default case; and empty statement as the exception to what can be the if- clause in an if-else statement. Pedagogically, this makes sense, since students tend to think that what is introduced first is the most typical case, and build out every other case with respect to it, as predicted by anchoring bias [15]. They are better prepared because the general version they learn first is the version applicable to the majority of the programs they will write.

Incidentally, “curse of knowledge” is the term recently coined by Harvard psychologist Steven Pinker for the last two prior practices – when an instructor assumes a certain level of knowledge on the part of the student, and forgets that the student is trying to learn the material for the first time!

**Current Practice:** Whether it is arrays, parameter passing by reference, or pointers, once a programming construct is introduced, students feel compelled to use it in all subsequent programs, often ending up with greatly convoluted and unreadable programs. So, I now find it necessary to remind my students of the “law of the instrument”, or a variation of it: just because they bought a hammer does not mean everything is a nail. In other words, it is not sufficient to tell students when to use a given programming construct, it is also necessary to tell them when not to. This was so counter-intuitive to me that I do not have a prior practice or retrospection to offer for this current practice.

6. DISCUSSION

In retrospect, the content, assessment, presentation and pedagogy of my Computer Science I course have changed significantly over the years. What did all this mean to the learning of students in my course?

In my experience, the benefits of most educational interventions do not accrue to the A-grade students who would have done well even without them. This probably explains why there are always some educators who dismiss most interventions as something their students do not need and would not benefit from – because their student population is skewed towards A-grade. Educational innovations do not often benefit F-grade students either, most of whom would fail regardless of the course pedagogy. This probably explains why numerous solutions have been offered for improving retention in Computer Science I, but most have failed to deliver on the promise. It has been my experience that
educational interventions primarily benefit the students in the middle – their grade improves by up to a letter grade, thanks to the intervention. I believe, the same has been true for all the changes that have occurred in my Computer Science I course as it evolved over the years.

Looking back at myself two decades ago, I would be hard pressed to give myself a passing grade as the instructor of Computer Science I. I am curious whether I will feel the same way in two decades about my current practices. Only time will tell.

7. REFERENCES
[16] www.programbydesign.org