Report on the design of the introductory programming course

Abhiram Ranade, Venkatesh Chopella, Shrawan Kumar
ACM India

May 18, 2021

Education consists of knowledge, skill, and a confident, enthusiastic, contemplative disposition, as per the recently released draft of the ACM IEEE Computing Curricula 2020 (www.cc2020.net). While Indian Education has generally done well on knowledge, it is often considered lacking on skill/application of knowledge. Furthermore, there is also concern regarding disposition/outlook. CC2020 suggests that cultivating a proper outlook cannot be left as matter of pedagogical style/choice, but it must done explicitly.

In this document we present a design for an introductory programming course which pays more attention to the skill and disposition aspects. We propose a syllabus together with pedagogical and assessment strategies. Courses (nearly) based on our design have been conducted in some universities and on NPTEL. So there is a fair amount of experience and resource created from that. ACM India will be glad to assist other universities desirous of implementing our material.

The rest of this note implicitly refers to 3 resources.

1. An NPTEL course, *An Introduction to Programming through C++*. https://nptel.ac.in/courses/106/101/106101208/
2. An NPTEL course for teachers, *Design and Pedagogy of the Introductory Programming Course*. https://nptel.ac.in/courses/106/101/106101182/
3. A textbook, *An Introduction to Programming through C++*, McGraw Hill Education.

In some sense, this document is an executive summary of the material presented in the above courses and book. There is also some new material, especially when it concerns exercises/assessment.

We have tried somewhat to structure this document using the *competency* framework suggested in the CC 2020 report mentioned above. We have already mentioned three components of a competency: knowledge, skill, and disposition. An additional component is the *Task*, which refers to the purpose served by the learning. The task is the function or the job that the graduate will be capable of after the learning is over. CC 2020 envisages competencies to be used to discuss the entire course as well as the individual topics, sometimes called *knowledge units*. In Section 1 we discuss the framework in more detail and also remark on how to apply it in the Indian context.

In Section 2 we discuss the Task in introductory programming. This is often implicit in common curriculum designs. But as we will see, making a distinction between ends and means is crucial. In Section 3 we discuss the knowledge component, followed by Section 4 about skill and Section 5 about disposition. In Section 6 we focus on assessment. In Section 7 we discuss some scaffolding: a library we use as a learning aid. We believe that scaffolding can go a long way towards speeding up learning, connecting students to interesting problems, and also improving disposition. In Section 9 we propose some ideas on how to take this forward.

1 Competencies and Indian education

As remarked above, the first component of the competency framework is the Task: a clear statement of the purpose of the learning. The task is the end, while the components knowledge and skill are the means. Disposition is an overall mental attitude that ensures care and attention in performing the task.

Knowledge refers to the information, ideas needed for accomplishing the task. Skill refers to the ability and practice of using the ideas. In the context of motor skills, the distinction between knowledge and skill is very clear – there is a difference between knowing the theory of swimming and actually getting into water and floating. Something happens when you get into the water that cannot be conveyed through lectures: something new is encountered and needs to be encountered for the learning to be complete. In

the cognitive domain skill is again the application of knowledge. Even in the cognitive domain the application of knowledge unearths something not experienced while the knowledge was being conveyed.

The four components are mutually reinforcing. If the Task is unclear, then the learning process can become a barren chore. "Why am I learning this, why should I care?" are questions that sometimes do not get answered satisfactorily (at least at the level of individual topics) and do frustrate the learner. Knowledge is necessary in order to apply it to develop skill; but the process of application reinforces knowledge.

> I hear, I forget. I see, I remember. I do, I understand.
>
> – Confucius

Skill creates confidence and energy – important aspects of desired disposition. Only repeated practice can create intuition, another desired aspect. And finally it should be noted that good disposition will invariably help in learning. The education system needs to exploit this synergy.

We feel that overall, the Indian education system is strongest in conveying knowledge to students, but deficient in the other three components. From time to time students struggle through topics not knowing the purpose, studying only because of the examinations. This could be because the teacher himself is unclear and uses the "you will understand it and thank me later" justification. The lack of skill development, or the prevalence of "rote learning" has been remarked quite a bit in many fora. Overall, we do see students who come out frustrated as much as well disposed.

So the overall message is clear. First, be very clear about the task. Second, find ways to ensure skill development. Third, ensure good disposition. For the second, we need to find ways whereby students get over the phobia of applying knowledge in *unseen* situations. For the third part: there is a simple prescription: make efforts to get the student to like the material, fall in love with the material. This involves careful thought in exposing the power and elegance of what is taught, and getting the student to experience these through problem sets and lab work.

Finally, it must be mentioned that the reform needs to be handled at the level of designing courses. The course designers must carefully plan for each of the components of competencies, and the teachers carry out the plan, albeit with some improvization.

2 The Task

We believe that the task in the introductory programming course is the following.

> *Write programs to perform computations that the student can do manually.*

Notice the task is not "Learn language X" – that is a part of the means and will come later. This distinction is important, because often the introductory programming courses focus too strongly on the language rather than competency in the general programming ideas.

The phrase "computations that the student can do manually" needs some interpretation and elaboration.

1. First, the phrase certainly includes writing programs to perform computational processes that have been already taught in high school or junior college, e.g. arithmetic operations on polynomials, matrices, (arithmetic on numbers with arbitrary number of digits is also interesting). Often students know processes such as finding roots, finding integrals. It is also expected that the student will be able to write programs involving computational processes that may not have been taught, but are implied by what is taught.

2. The second class of important programs that the student can be expected to learn to write are those that model the evolution of a simple system. Here are some examples.

   (a) Library management, bank account management.

   (b) Simulation of mechanical systems, orbiting planets, circuits.

   (c) Simulation of computer execution, train systems.

   (d) Games.

   These systems can be extremely complex; what is envisaged here are systems that non specialized teachers can teach to students in a manner that the students find the mechanisms "common sense". For example, students are taught the laws of Gravitation and motion. Beyond this what is needed is the notion that the velocities or accelerations in a system can be considered to be changing discretely after small time

steps. This principle is the basis of calculus, and is as such familiar to the student. The notions such as two trains cannot be on the same track at the same time unless they are separated by a specific distance (or typically a signal) are also common sense and are attractive to the students' imagination. Games are of course fun and educative.

3. There is an additional important theme. Many of the above applications require somewhat complex input and output. This could be done textually; but in this era, graphical input and output is also interesting. While standard graphical input and output libraries are cumbersome, it is worthwhile to develop a simple library and have the students use that.

Finally, we believe that the introductory programming course should be concerned with correctness and efficiency, but in a minimal manner. The student should be able to argue that the program written by her is correct to the extent that it mimics a certain manual algorithm (whose correctness could be considered obvious). This ability to reason about programs needs to be cultivated. It will be immediately useful to the student in debugging, which is an important skill. Also, the programs should not perform more computation than what natural manual algorithms would. Specifically, while some understanding of correctness and efficiency is expected, sophisticated proofs of correctness or efficiency are beyond the scope.

3 Knowledge

"Knowledge" at the level of the course, corresponds to the bullet list of topics that is often called the "syllabus". These are the topics about which the student needs to be informed. There also are skills associated with the topics, but we discuss them later.

The topics which we consider important are as follows.

1. Basic data types, variables, assignment statements.

2. Conditional execution.

3. Loops. Issues of termination.

4. Basics of program design for a problem involving a loop. Notion of loop invariants and process of debugging.

5. Elementary algorithms for root finding, evaluating math expressions.

6. Functions including recursion. Functional expressions (lambda expressions).

7. Arrays. Uses such as storing sequences, sets, character strings. Sorting including merge sort.

8. Structures/classes

9. Introduction to Memory management (for C++) using the operators new and delete, and issues in automating this using constructors etc.

10. Introduction to standard library classes with good practice for string and vector.

11. Organizing programs by partitioning into functions and classes.

12. Basic understanding of paradigmatic ideas such as a finite state system that evolves as per a set of rules. This should happen through examples accessible to students, e.g. bank accounts, games, elementary physics simulation, simulation of the working of a simplified computer.

13. Basic notions of user interaction: the idea of interpreting commands and executing them. The commands could also be given graphically – this requires understanding some elementary graphical system. See the discussion on Scaffolding, Section 7.

We should mention here that all the topics above are meant to be at Bloom level *apply* and above (except where we say introduction). This aspect will be discussed in the section on skills. Notice that the depth of understanding is also implied by what we specified as the Task.

In principle, almost any programming language is acceptable. However, there are some advantages to choosing a mainstream language. First the (sane subsets of) mainstream languages are often conceptually simpler, e.g. they rely less on recursion. Second, many of the important ideas from the theoretically sophisticated languages such as higher order functions, lambda expressions, object orientation, have been substantially taken up by mainstream languages.

4 Skill

The key question in skill development is: What practice problem should we give after each concept is taught? We have already said that the practice problems need something not directly discussed in the class. We believe that the notion of problem families provides just the right amount of novelty to build up the confidence of the students.

A problem family is a set of problems which can be solved using somewhat similar programs. It should be large so that the teacher can do a few in class, leave many as drill, and yet have many for use in examinations. Math is full of problem families, e.g. after learning about integration the teacher can create hundreds of integration problems. Design problems in engineering (e.g. design an amplifier for the following design parameters) are often natural problem families.

As much as possible, drill problems should have a real world connection. This often makes them attractive to students, and thus helps in cultivating disposition. This of course changes when students mature – mature students can see the real life connections themselves, and may often appreciate crisp, abstracted problem statements.

We give examples of problem family based drills for 4 important topics in the introductory programming course.

4.1 Drill for conditional execution

Some may say that conditional execution is too easy and any explicit drill is not needed. However, drill problems will help the weak students. Also, since our family is somewhat interesting, it might amuse the other students too. Our family is:

> *Write a program to play a guessing game. The user selects an object from a taxonomy specified as a part of the problem. The program must ask a series of questions such as "Is your chosen animal a mammal?" and eventually guess the selected object.*

This is an exercise in nesting if statements. It can be made more interesting by demanding that the program ask a minimum number of questions. Further, the teacher can create taxonomies very easily (e.g. a taxonomy of vehicles, or food items).

The exercise can neatly lead into standard binary search.

4.2 Drill for iteration

The main concepts to be understood in iteration are (a) maintaining state between iterations and the notion of invariants, (b) ensuring termination, (c) nesting loops, (d) the use of control variables in not just counting iterations but affecting what goes on in each iteration. Our first drill family is:

> *Write a program that evaluates the sum of a given series to n terms or till some error bound is met (e.g. $1+1/1!+1/2!+\ldots$) The key point is that the ith term of the series should not be calculated from scratch but using the $i-1$th term calculated earlier.*

It should be clear that calculating the ith term ($1/i!$) from the $i-1$th ($1/(i-1)!$) is typically done very economically (division by i for the example series). But this improved efficiency comes at the cost of having to manage the state from one iteration to next. This can be more or less complex depending upon the series. It should be noted that there are a huge number of natural series summing problems, based on Taylor series or other recurrence based calculations such as those in probability theory. Note that the significance of the series need not be explained. Also other infinitary expressions such as continued fractions or nested square roots could also be used.

Another family of drill problems comes from drawing parameterized iterative pictures. This requires a picture drawing library, which is useful in many ways (Section 7). Some of figures to be drawn are shown in Figure 1(left). Each figure is specified using a parameter, e.g. the number of steps in the the staircase in each arm of the top left figure. The parameter has to be related to the loop count and also possibly to turning angles etc. Making such relationships is an important programming skill, in addition to the skill of nesting loops appropriately.

4.3 Drill for recursion

Recursion is considered a difficult topic in programming. It is indeed difficult, if you expect students to design interesting recursive algorithms. Structural recursion is easier, and far more important for many practitioners. So we propose *drawing recursive pictures* as a drill problem for recursion. Figure 1(right) gives a number of figures having recursive structure. The recursive structure is visually obvious. Furthermore, the student gets direct visual feedback as the drawing unfolds – in the order in which the recursive

calls execute. So it is felt that drill with such figures will go a long way in cementing the execution mechanics of recursion in the minds of the students.

4.4 Drill for arrays

We propose the implementation of board games as a drill for arrays. As an example, consider the common board game of snakes and ladders. The simplest implementation involves using an array in which the ith element indicates the action needed if a player lands in square i. Each player's position is simply an array index. Array indices change by throws of the dice, and at each step it is necessary to index into the board array to determine what to do. It is not too hard for the teacher to invent board games – and each can become a drill problem.

Note however, that if we allow the actions in each square to be interpreted in a very general manner, then we can actually encode machine language programming as a board game! For example, the rule for square i could be something like:

1. Let p, q, r denote the numbers in squares $i + 1, i + 2, i + 3$.
2. Add the numbers in squares p, q of the board, and store the result in square r of the board.
3. Then move to square $i + 4$.

We believe that this drill adequately tests the ability to perform array operations. It also fires up the imagination of the students.

4.5 Reasoning about programs, testing and debugging

Bugs are a fact of life in programming: programmers will make mistakes and as a result programs will typically not run correctly on the first attempt. How to recover from this is an important skill.

This skill can be cultivated through some explicit instruction and subsequent practice. First, the student be able to identify important test cases (e.g. random, "corner"). Second, the student must develop some facility in figuring out what values the different variables will have at different points in execution. This could be expressed as: if the input is x then variable y will have value z, either symbolically in terms of x, or for a specific value of

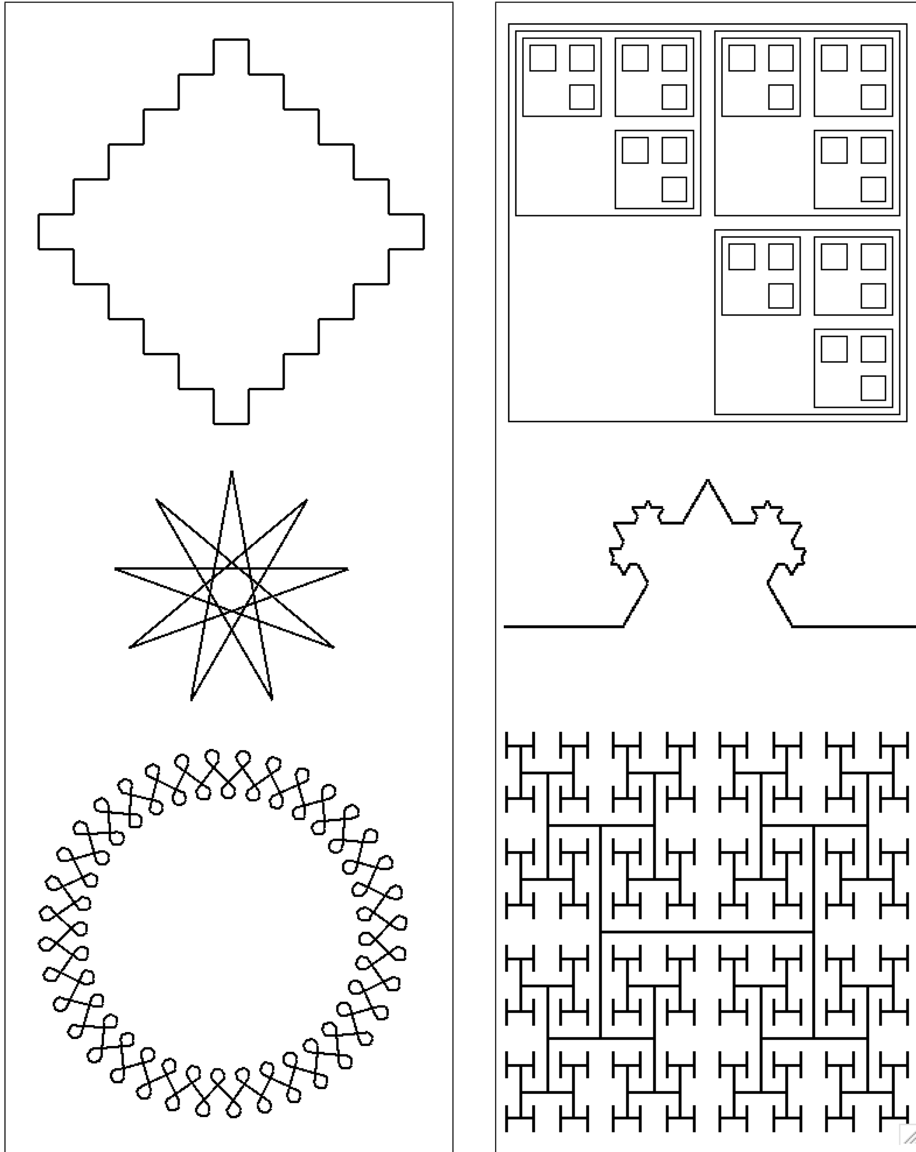

Figure 1: Drills for iteration and recursion

x. Third, the student must be explicitly told to put print statements in the program and check whether the values of variables are as expected.

If a simple debugger and interactive development environment (IDE) are available they could be used. Stepping through a program can give a feeling of “looking inside the computer”.

The ability to write and understand invariants – crisp statements about values taken by variables at different points in the program – is valuable in a different way. Students can first be given the invariants and then asked to develop code that implements the invariant. This is useful for developing familiarity with invariants as well as as a way to give hints about how to develop a program.

5 Disposition

CC 2020 expects graduates to have an enthusiastic, confident, thoughtful disposition towards their subject. We feel this translates to saying, for practical purpose, “Students should like whatever it is that they are learning!” Someone who is taking up a subject as a profession had better like it; liking it will also help in learning.

Liking anything of course has an irrational subjective component. But we can help along by highlighting and showcasing the power as well as the elegance of our subject. This is easier said than done; in the daily grind of teaching there is a certain urgency to “cover material”. Furthermore, it is not easy to crisply articulate the elegance and the power. Simple articulation is usually not enough; it must be backed by demonstrations and opportunities for the student to experience the ideas herself. This can be done by creating exercises that enable the student to explore the stated elegant/powerful ideas. This part is greatly facilitated by using graphics as scaffolding (Sections 4.2, 4.3, 7).

It should also be noted that the first lecture is crucial for inducing love at first sight. It needs to begin with a topic which shows of the power and the elegance. A demo of what exciting programs can be written can also be very useful. The course mentioned at the beginning of this document provides some suggestions.

Examples play an extremely important role in showing off elegant/powerful ideas. For example, a very short recursive procedure is enough to draw a very complex, realistic looking tree on the screen. Such demonstrations have to

happen at different levels: first at the level of introducing the course. Then at the level of teaching each topic. Indeed CC 2020 envisages the education at each level to involve a clear statement of the task, imparting of the relevant knowledge, development of skill, and creating a warm disposition in the student.

One important idea in this is to introduce every new idea by a so called *killer application* – an application which cannot be implemented (well) using the ideas the student knows so far, and hence the new idea is essential. If a new idea is a variation on an idea learnt earlier, then the incremental benefits, which might be apparent only in certain settings, should then be discussed, e.g. multiple statements for looping.

Do note that cultivation of disposition also happens through the assignments given to the student. Students will get excited if you give assignments that have real application rather than synthetic assignments. Students will get excited if the assignments are fun. Note however that students appreciate difficult assignments too – if they see that the end goal is worthwhile.

6 Assessment

We believe that the assessment in introductory programming should focus on writing programs. The best way today is to have an online programming test, where sample inputs and outputs are given, and the program is tested on hidden inputs. This requires some initial investment in getting the student to become familiar with the set up of submitting programs, having them compiled and run automatically. But it is worth the effort because this is a skill of use in all subsequent learning.

There should be some simple programs from the drill problem families discussed above. In addition there could be harder programs, so long as the harder problems are also of the type discussed in the classroom or solved in labs.

In addition to program design, there can be some other types of problems too. The most important amongst these deals with invariants: complete the program given so that it upholds the given invariant. This tests the ability to reason about code, and also introduces a student to ideas that are needed for proving correctness and even designing clever algorithms.

Other types of questions are also fine, so long as they are drilled in the classroom.

7 Scaffolding

The term scaffolding refers to libraries or code that is given to the student as a black box which helps in learning. The contents of the black box are not expected to be understood. We use two kinds of scaffolding.

First, we use a library for two dimensional graphics. The library supports simple coordinate graphics as well as turtle graphics as pioneered by the Logo programming language. We have found graphics to be invaluable for creating drill problems as has been noted above. In fact our simple system is adequate even to create elementary animation as well as graphical simulators and editors. These are often easier to program than text based systems, and thus enable the student to go farther in building interesting programs. Finally, students seem to love graphics – clearly a great help in cultivating disposition. We built our own library in C++; such libraries are available in Java and Python also.

Second, we find it useful to teach a very elementary looping construct on the first day. We manufactured this in C++, using the macro feature to create a `repeat (count)` { `body` } statement which has no conditions, nor any control variables, but simply causes the `body` to be executed `count` many times. This is the second part of our scaffolding. The macro feature is missing in Java or Python, so the standard statement could be used without explaining its details, e.g. in Python on day 1 you may say, "If you want to repeat statement `xyzzy` 10 times, write it as `for repetitions in range(10) : xyzzy`". The detailed semantics of the `range` construct or the fact that `repetitions` is really a variable can be explained in due course.

Our experience is that a loop which runs a fixed number of times is very easily understood even in the first lecture of the course. In fact students are even able to understand nested loops. This enables us to get to interesting programs, e.g. drawing iterative pictures as in Figure 1. We find this fires up students – a big victory on the dispositional front. Note that in the standard teaching order it would be several weeks before loops are taught; we cannot afford to delay exciting assignments this long.

8 Concluding Remarks

It could be said that traditionally the Task component is left implicit, the knowledge component is specified as the syllabus by the course designers,

while the skill and disposition are left to the teachers. We feel that this state is not acceptable; the course design needs to have a discussion of all 4 components as we have provided here.

Of course, the level of preparation, morale, maturity and alignment of the students to the course does determine the actual amount of time spent in the course on the different components of competencies. For example, certain student populations may come in with a very positive attitude and for them cultivating the disposition may happen automatically. In the present climate in Indian universities we believe however that very strong emphasis and class room time needs to be given to cultivating disposition. Second, our focus in this document has been on somewhat elementary skill development. The drill problems we have suggested are meant as small but definite steps towards writing programs to solve *unseen* problems. They are meant to give confidence to students to move out of their current rote learning mindset without scaring them off with harder problems. Once they take the baby steps, they can be given harder problems for sure.

The CC 2020 notion of disposition includes *ethical behaviour*. It is clearly an important attribute of graduates. Universities can cultivate this as much by explicit discussion and courses on ethics as by adopting high ethical standards in their own conduct. For example, cheating in examinations needs to be eradicated; without that discussion of ethics might sound theoretical and even hypocritical. Within the introductory programming course, ethical questions could be raised through imaginary dialogues with employees/clients. The key question here would be: "How can you assure that your program is correct?" This is a scientific question as well as ethical, and students often have notions such as "I should get as many marks as the number of test cases passed by my program." The fact that an erroneous program can lead to monetary losses and even loss of life, and that a "no error whatsoever" promise is made in delivering a program needs to be emphasized.

Finally, we should mention that CC 2020 framework is very extensive. In this report we have only picked up broad points and interpreted them in the context of introductory programming.

9 Deploying these ideas

Through this document, and the resources mentioned, we have provided substantial guidance and help for the teaching of the introductory programming

course. So one possibility is just to recommend all this as a resource on the AICTE website. It should be helpful for all introductory programming courses because most of the ideas are language independent.

But we feel more could be done. We have pointed out some specific flaws in the current system: lack of an articulation (and often a misunderstanding) of the task, lack of skill development plans, and also lack of plans to inculcate disposition. We feel that these flaws need to be corrected.

One step in this regard would be to ask course designers (Board of studies members) to prepare a course conduct document describing these three aspects (including assessment). It could be at the same level of detail as Sections 2 through 6 of this document. Teachers could be given the freedom to deviate, but then they would need to be ready with the rationale for the deviation. Furthermore, we feel that course designers should accept feedback from teachers, discuss it appropriately in meetings, and modify courses accordingly.

We feel that such a protocol will be useful in all courses, not just this one. But to evaluate the practicality of this proposal, perhaps this protocol: that course designers should provide a course conduct document and that they should be in dialogue with teachers, could be first tried out for the introductory programming course.

ACM India will be willing to support all the efforts mentioned above, through development of additional resources and holding workshops as needed.

Acknowledgements

We are grateful to several individuals including members of the ACM Education Committee for giving comments on this report and general discussions.