Design and Pedagogy of the Introductory Programming Course

Abhiram Ranade
IIT Bombay

Overview

Introductory programming course is important

- First course in CS curriculum :
  - Programming = Foundation of CS
  - Can shape students attitude towards all of CS
- Important also for non CS majors:
  - Many find jobs in IT industry
  - Most engineering and science uses computers heavily
  - "Understanding X = writing programs related to X" [Knuth]
- Computer programming allows students to build something, unlike other courses which are about memorizing and understanding.
- You can write programs about anything: math, engineering, commerce, even art. You can explore the world.
- Every student can feel that he/she is in control of the computer. This is psychologically liberating.

There is potential for students to fall in love with programming.

The current state: we have a crisis!

- The international scenario:
  - Watson Li 2014 [WL14]
    - Survey of 15 countries, 161 courses.
    - Language: C, Python, C++, Java, VB, Fortran, ...
    - Failure rate: $\approx$ 30 % over all countries, languages.
  - Bennedson Casperson 2007 [BC07]: similar findings
- The Indian Scenario:
  - aspiringminds.com surveys
    - Many graduates cannot write simple programs,
    - Overall, many graduates considered unemployable
  - University failure rates not available, but university exams are considered to be based on rote learning.

We should step back and introspect...

- Is programming so hard?
  [Guz10] “Heres a possibility: Its inherently hard.”
- Do we need to slow down our courses?
  [LR16] effectively says this.
- Should we teach differently?
  [Gri74] “But what do we really teach? We describe the tools the student has at his disposal (the do-loop, goto, declarations, etc.), give a few examples, and then tell him to write programs. Almost no word on how to begin, how to find ideas, how to structure his thoughts, and how to arrive at a well-structured, well-written, readable program.”
- Are we able to motivate students to study?
  [RMMH^{+}09] Graphics and animation seem to motivate children well.
- Are our exams fair?
  Over time, tests used by researchers for measuring programming competency seem to have become easier..

Goals of this course

- Examine how the introductory programming course has been taught over the years.
- Examine what the educational literature says are the difficulties.
- Design the course objectives
- Suggest teaching strategies
- Suggest how to motivate students better
- Suggest how to create fair exams

Course Overview

Introduction and Survey: Traditional approaches to programming education. Non-traditional approaches. Experience. Challenges.

Basic ideas in our approach: Encouraging students to become aware of manual algorithms they already know. Strategies for translation from manual algorithms to programs. Some generic problem solving strategies. Programming language C++ : with or without Object Orientation.

Pedagogical strategies: Graphics as a teaching aid and for fun! "Repeat" loops as scaffolding for standard loops and to enable exciting programs to be written early on. Role of motivational examples in explaining concepts.

Issues in designing medium size programs: Use of standard libraries. Object oriented programming. Dynamic memory allocation.

Design of Exams: Types of questions. Estimating difficulty using Bloom's Taxonomy.

Administrative issues:

Grading: Weekly Assignment: 25%, Final Exam: 75%

Reading:

- Slides
- Uploaded papers
- Some material will refer to “An Introduction to Computer Programming through C++”, Abhiram Ranade, McGraw Hill 2014.

Teaching Assistant: Apoorv Garg, Research Scholar in IIT Bombay.

Discussion group: Please ask questions. We will attempt to answer them as quickly and in as much detail as possible.

Introduction and Survey

Outine

Wikipedia definition of programming

- "... a process that leads from an original formulation of a computing problem to executable computer programs."
- "... analysis, developing understanding, generating algorithms, verification of requirements of algorithms including their correctness and resources consumption, and implementation in a target programming language."
- "Related tasks include testing, debugging, and maintaining the source code, implementation of the build system, and management of derived artifacts such as machine code of computer programs."

Some intro prog. books and courses attempt to cover all..
But perhaps we should be selective?

- Requirements analysis — Requires experience and maturity?
- Design of algorithms including those requiring domain knowledge.
  Algo design = specialized course in 3rd year
- Coding
- Software engineering — Requires experience and maturity?

Process of writing a program [MAD$^+$01]

1. Abstract the problem from its description
2. Generate sub-problems
3. Transform sub-problems into sub-solutions
4. Re-compose the sub-solutions into a working program
5. Evaluate and iterate

What you need in order to program [dB89]

- General orientation: what programs are for and what can be done with them
- The notional machine: a model of the computer as it relates to executing programs;
- Notation: the syntax and semantics of a particular programming language
- Structures: Schemas/plans used for designing programs.

  library of previously learnt programs

  Strategies for composing elementary programs
- Pragmatics: Skills of planning, developing, testing, debugging, and so on.

AICTE Programming Course Learning Objectives [AIC18]

The course will enable the students to

- Formulate simple algorithms for arithmetic and logical problems
  What is "simple"? What do we teach?
- Translate the algorithms to programs (in C language)
- Test and execute the programs and correct syntax and logical errors
- Implement conditional branching, iteration and recursion
- Decompose a problem into functions and synthesize a complete program using divide and conquer approach
- Use arrays, pointers and structures to formulate algorithms and programs
- Apply programming to solve matrix addition and multiplication problems and searching and sorting problems
- Apply programming to solve simple numerical method problems, namely root finding of function, differentiation of function and simple integration

AICTE Programming Course Lesson Plan

Introduction to Programming (Flow chart/pseudocode, compilation etc.), Variables (including data types)	(2 hrs)
Arithmetic expressions and precedence	(2 hrs)
Conditional Branching and Loops	(8 hrs)
Arrays (1-D, 2-D), Character arrays and Strings	(6 hrs)
Basic Algorithms: Searching, Basic Sorting Algorithms, Finding roots of equations, idea of time complexity	(6 hrs)
Functions (including built in libraries) and Recursion with examples such as Quick sort, Ackerman function	(8 hrs)
Structure and Pointers (including self referential structures e.g., linked list, notional introduction)	(6 hrs)
File handling	(2 hrs)

Tutorial and Lab: (total 4 contact hours per week)

University of Virginia/Bloomfield [Blo18]

Learning Outcomes

- Understand fundamentals of programming such as variables, conditional and iterative execution, methods, etc.
- Understand fundamentals of object-oriented programming in Java, including defining classes, invoking methods, using class libraries, etc.
- Be aware of the important topics and principles of software development.
- Have the ability to write a computer program to solve specified problems.

  Knowing language should be sufficient!
- Be able to use the Java SDK environment to create, debug and run simple Java programs.

Pune Engineering College [Pun18]

Course Outcomes:

1. Makes students gain a broad perspective about the uses of computers in engineering industry.
2. Develops basic understanding of computers, the concept of algorithm and algorithmic thinking.
3. Develops the ability to analyze a problem, develop an algorithm to solve it.
4. Develops the use of the C programming language to implement various algorithms, and develops the basic concepts and terminology of programming in general.
5. Introduces the more advanced features of the C language

The most common model of teaching

The model which seems to work in math/physics
Teacher teaches basic tools: Key theorems, conservation principles.
Teacher solves problems. In class/tutorials. Solved problems in book.
Student solves problems. "End of chapter problems",...

As applied to programming:
Teacher teaches basic tools: Language constructs
Teacher solves problems. Writes programs involving the constructs.
Student solves problems. Asked to write programs.

Hope: By learning the basic tools, and by emulating the teacher, and by using street smarts, the student should be able to write programs to solve unseen problems.

Remarks

Most courses want similar outcomes

- Learn Language elements: variables, assignment statements, conditionals, loops, functions including recursion, arrays, structures.
- Learn to develop algorithms

  Confusing. Algorithm design is an advanced course.
- Learn to write simple programs

  "Simple" is not defined.
- Learn specific algorithms, e.g. sorting, root finding.

Result:

- Teachers focus on the non-ambiguous parts of their mandate: Teaching language elements, teaching specific algorithms.
- Students cannot write "Simple Programs" (Next)

The Rainfall Problem. [SEBJ82, VTL09]

“Read rainfall measurements specified as a sequence of integers terminated by 99999. Output the average of the numbers encountered before 99999.”

Studied by many groups over many years.

Relatively standard idiom: sequence being terminated by a sentinel.

Number of students completing successfully: 15-20 %

McCracken Working Group 01 Study [MAD+01]

“Evaluate a multiterm arithmetic expression.”

- Version 1: Expression given in post fix notation.
- Version 2: Expression given in infix notation. However, there is no operator precedence. Operators evaluated in left to right order.
- Version 3: Infix notation. No operator precedence. But parentheses may be present, which must be respected in evaluation.

“After 1 year of college”, 2 programming courses?

216 students, 4 universities, Average Score: 23/110

McCartney et al 13 Study [MBE^{+}13]

Simplification of [MAD^{+}01]

- Infix expression evaluation, without precedence and without parentheses.

  Easiest problem from [MAD^{+}01]
- Code supplied for tokenization, skeleton main program.
- Given during the second programming course.
- 40 students, average 68/110. (Scoring is different)

  Much better performance than [MAD^{+}01]

But in spite of all simplification, 68/110 is only 60 %.

Utting et al 13 Study [UTM^{+}13]

“Complete the implementation of a 24-hour clock by implementing methods for advancing time, adding and subtracting time and comparing time.”

- 418 students with at least one programming course.
- Some students were given “test harness”, i.e. a code that tests correctness.
- Students with test harness completed more than 3 out of 4 methods.
- Students without test harness completed less than 1.

Much easier than previous studies, yet performance is not great.

Performance improves if we ask for small pieces of code.

Performance improves if we provide test harness.

Observations

What do students find hard?

- Syntax is easiest.
- Semantics, e.g. how a loop will execute, is harder.
- Designing a complete program/algorithm is hardest.

[Win96]: “Almost any undergraduate can add a set of numbers or compute an average of a set of numbers; why can't over half of them write a loop to do the same operations?”

Less Common Approaches to Programming and Programming Education

Will programming be easier in another programming paradigm?

Programming paradigm = model of computation + language

Hope: Perhaps there is a simpler model of computation which students will be able to program more easily...

Functional Programming

Program = math expression, Execution = expression simplification

Functional programming using the Scheme language:

Expressions must be specified in prefix syntax:

```
(operator operands ...)
(function-name arguments ...)
```

Example: `(* (+ 5 3) (- 7 4))`

"Simplify wherever possible"

becomes: `(* 8 (- 7 4))`

becomes: `(* 8 3)`

becomes: 24 Result of program

Programs can contain names denoting values, functions..

- Values/function definitions are substituted in place of names as needed.
- Argument values replace parameters during substitution.

A more complex Scheme expression/program

```
(define (fact n)   ;; factorial function
   (if (= n 1)
       1
       (* n (fact (- n 1)))
   )
)
```

"if" function: `(if test consequent alternate)` evaluates to `consequent` if `test` is true, and `alternate` otherwise.

Example: `(fact 2)`
becomes: `(if (= 2 1) 1 (* 2 (fact (- 2 1))))`
becomes: `(* 2 (fact 1))`
becomes: `(* 2 (if = 1 1) 1 (* 1 (fact (- 1 1))))`
becomes: `(* 2 1)`
becomes: 2 Result of program

Remarks

- FP execution model is very simple and familiar: expression simplification.
- FP syntax is very simple: expression syntax.
- It is often easy to reason about FP programs, because values of names do not change. No "`i = i + 1;`"
- FP uses recursion heavily.

  Often corresponds to natural math definitions.

  But recursion can also be difficult to understand.
- Many programs written using FP are very elegant.

  Depth first search etc. are hard to express.
- Imperative programs are usually much faster in practice.

Conclusion: Use functional principles wherever possible.

- avoid global variables
- higher order functions

  "lambda" expressions inducted into C++, Java

Dijkstra's approach [DFG83, Dij88]

Proving the correctness of programs is considered very important.

Also deriving correct programs.

Simple loop based programs. Recursion considered less.

Often tricky fast algorithms ($O(n)$) are derived for problems for which simple slow ($O(n^2)$) time algorithms exist.

Beginners find it dry.

Conclusion:

- Basic notions such as invariants are important even in introductory programming.
- Other steps could be left informal.
- Clever algorithms should not be expected in intro prog.

“Objects first” approach

Object oriented programming is arguably the dominant programming paradigm today.

Proponents of the paradigm believe it should be taught ”from day 1” so that ”you dont get corrupted by other approaches”.

Books have been written about this: typical application domains considered in the books = designing user interfaces, drawing pictures on screen.

Tends to be heavy on syntax. Appears to give undue importance to minor issues: how do you change the colour of a textbox.

Many concerns: how can you learn member functions without learning ordinary functions?

Not clear OOP lives up to its promise [Dét06]

Logo/Scratch: programming for children

Logo [dA81, Pap80], Scratch [RMMH+09], and others

- Teaches programming using graphics/geometry
  Grabs attention.
- Usage paradigm 1: Narrative
  - Create graphical characters, animate/choreograph their movements and tell a story.
  - Avenue for expression of artistic talent.
  - Not full range of programming skills, somewhat like web page design.
- Usage paradigm 2: General
  - Draw precise, patterned, intricate geometric figures.
  - Solve standard programming problems.

Logo/Scratch are very successful

Created hope/confidence that children can program.

Conclusions: Graphics builds on what students know/like.

“Scaffolding”

Which assignment is more exciting:

- “Write a program to multiply two polynomials.”
- “Program this robot so that it dances.”

A dancing robot may be more attractive but

- Making a robot dance requires learning some robotics.
- Making a robot dance requires a lot of code.

Solution: Teach domain specific concepts and provide domain specific code!
(Scaffolding: a temporary structure on the outside of a building, made of wooden planks and metal poles, used by workmen while building, repairing, or cleaning the building.)

Scaffolding has been developed for many subjects:
robotics, data science, graphics, Geographical Information Systems

Learning overhead + all students may not like chosen scaffolding

Summary

- Worldwide failure rates in programming courses are $> 30\%$.
- Indian situation appears similar.
  - Probably failure rates are not that low, because exams are typically much less challenging, based on “rote learning”.
  - Competence of graduates is low.
- Should be considered a Crisis.

Main disappointment: Students are not able to write programs to solve simple (unseen) problems.

No formal definition of “simple problem”.

Summary (contd.)

- Experts have used tests of wildly varying difficulty to test programming aptitudes of students
  - No consensus about what programming is?
  - Are they lowering standards in light of student performance?
- Yet lots of excitement about teaching programming to children.
  - Children are given simpler problems?
  - Graphics/animations provide good motivation?
- Few approaches worry about formal correctness.
- Confusion about how to/whether to/how much to teach algorithm design a.k.a. "problem solving".

Conclusion: The questions still remain...

- Is programming so hard?
  [Guz10] "Heres a possibility: Its inherently hard."
- Do we need to slow down our courses?
  [LR16] effectively says this.
- Should we teach differently?
  [Gri74] "But what do we really teach? We describe the tools the student has at his disposal (the do-loop, goto, declarations, etc.), give a few examples, and then tell him to write programs. Almost no word on how to begin, how to find ideas, how to structure his thoughts, and how to arrive at a well-structured, well-written, readable program."
- Are we able to motivate students to study?
  [RMMH$^+$09] Graphics and animation seem to motivate children well.
- Are our exams fair?
  Over time, tests used by researchers for measuring programming competency seem to have become easier..

Model curriculum for first year undergraduate degree courses in engineering and technology, Accessed July 2018, `https://www.aicte-india.org/sites/default/files/model%20curriculum%20for%201st%20year%20ug.compressed.pdf`.

Jens Bennedsen and Michael E. Caspersen, *Failure rates in introductory programming*, SIGCSE Bull. **39** (2007), no. 2, 32–36.

Alan Bloomfield, *CS 101 Intro to Computing: Course objectives and Review*, Accessed July 2018, `https://www.cs.virginia.edu/~asb/portfolio/eocms/cs101-fall05-eocm.pdf`.

Andrea diSessa and Harold Abelson, *Turtle geometry: the computer as a medium for exploring mathematics*, MIT Press, Cambridge, MA, USA, 1981.

B. du Boulay, *Some difficulties of learning to program*, Studying the novice programmer (E. Soloway and J. Spohrer, eds.), Lawrence Erlbaum, Hillsdale, NJ, 1989, pp. 283–299.

Françoise Détienne, *Assessing the cognitive consequences of the object-oriented approach: a survey of empirical research on* *object-oriented design by individuals and teams*, CoRR **abs/cs/0611154** (2006), Earlier version in Interacting with Computers, 9:1 47-72.

E. Dijkstra, W. Fiejen, and A. Gasteren, *Derivation of a termination detection algorithm for distributed computations*, Information Processing Letters **16** (1983), 217–219.

E. W. Dijkstra, *On the cruelty of really teaching computing science*, 1988, EWD-1036.

David Gries, *What should we teach in an introductory programming course?*, Proceedings of the Fourth SIGCSE Technical Symposium on Computer Science Education (New York, NY, USA), SIGCSE '74, ACM, 1974, pp. 81–89.

M. Guzdial, *Is learning to program inherently hard?*, April 2010, Retrieved from: https://computinged.wordpress.com/2010/04/14/is-learning-to-program-inherently-hard/.

Andrew Luxton-Reilly, *Learning to program is easy*, Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education, ITiCSE '16, ACM, 2016, pp. 284–289.

M. McCracken, V. Almstrum, D. Diaz, M. Guzdial, D. Hagan, Y.B.-D. Kolikant, C. Laxer, L. Thomas, I. Utting, and T. Wilusz, *A multinational, multi-institutional study of assessment of programming skills of first-year CS students*, ACM SIGCSE Bulletin **33** (2001), no. 4, 182–196.

Robert McCartney, Jonas Boustedt, Anna Eckerdal, Kate Sanders, and Carol Zander, *Can first-year students program yet?: A study revisited*, Proceedings of the Ninth Annual International ACM Conference on International Computing Education Research (New York, NY, USA), ICER '13, ACM, 2013, pp. 91–98.

Seymour Papert, *Mindstorms: Children, computers, and powerful ideas*, Basic Books, Inc., New York, NY, USA, 1980.

College of Engineering Pune, Course Plan, Accessed July 2018, `http://www.coep.org.in/page_assets/594/fy.pdf`.

Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and Yasmin Kafai, *Scratch: Programming for all*, Communications of the ACM **52** (2009), no. 11, 60–67.

E. Soloway, K. Ehrlich, J. Bonar, and Greenspan. J., *What do novices know about programming?*, In Directions in Human-Computer Interactions (A. Badre and B. Shneiderman, eds.), Ablex. New York, 1982.

Ian Utting, Allison Elliott Tew, Mike McCracken, Lynda Thomas, Dennis Bouvier, Roger Frye, James Paterson, Michael Caspersen, Yifat Ben-David Kolikant, Juha Sorva, and Tadeusz Wilusz, *A fresh look at novice programmers' performance and their teachers' expectations*, Proceedings of the ITiCSE Working Group Reports Conference on Innovation and Technology in Computer

Science Education-working Group Reports (New York, NY, USA), ITiCSE -WGR '13, ACM, 2013, pp. 15–32.

Anne Venables, Grace Tan, and Raymond Lister, *A closer look at tracing, explaining and code writing skills in the novice programmer*, Proceedings of the Fifth International Workshop on Computing Education Research Workshop (New York, NY, USA), ICER '09, ACM, 2009, pp. 117–128.

Leon E. Winslow, *Programming pedagogy – a psychological overview*, SIGCSE Bull. **28** (1996), no. 3, 17–22.

Christopher Watson and Frederick W.B. Li, *Failure rates in introductory programming revisited*, Proceedings of the 2014 Conference on Innovation and Technology in Computer Science Education (New York, NY, USA), ITiCSE '14, ACM, 2014, pp. 39–44.