Design and Pedagogy of the Introductory Programming Course

Abhiram Ranade
IIT Bombay

October 12, 2018
ACM Compute, Chitkara University, Chandigrah

Educational scenario in India

Educational scenario in India

Many graduates unemployable

Educational scenario in India

Many graduates unemployable
Much education is based on rote learning

Educational scenario in India

Many graduates unemployable

Much education is based on rote learning

Memorization rather than thought

Educational scenario in India

Many graduates unemployable
Much education is based on rote learning

Memorization rather than thought

A proposal: Intensively reform a few super-important courses

Educational scenario in India

Many graduates unemployable
Much education is based on rote learning

Memorization rather than thought

A proposal: Intensively reform a few super-important courses

- Create more detailed design document, explain rationale.

Educational scenario in India

Many graduates unemployable
Much education is based on rote learning

Memorization rather than thought

A proposal: Intensively reform a few super-important courses

- Create more detailed design document, explain rationale.
- Explain strategies for teaching, assessment, lab work.

Educational scenario in India

Many graduates unemployable
Much education is based on rote learning

Memorization rather than thought

A proposal: Intensively reform a few super-important courses

- Create more detailed design document, explain rationale.
- Explain strategies for teaching, assessment, lab work.
- Monitor implementation, student response.

Educational scenario in India

Many graduates unemployable
Much education is based on rote learning

Memorization rather than thought

A proposal: Intensively reform a few super-important courses

- Create more detailed design document, explain rationale.
- Explain strategies for teaching, assessment, lab work.
- Monitor implementation, student response.

What course to choose?

Educational scenario in India

Many graduates unemployable
Much education is based on rote learning

Memorization rather than thought

A proposal: Intensively reform a few super-important courses

- Create more detailed design document, explain rationale.
- Explain strategies for teaching, assessment, lab work.
- Monitor implementation, student response.

What course to choose?

- Foundational

Educational scenario in India

Many graduates unemployable
Much education is based on rote learning

Memorization rather than thought

A proposal: Intensively reform a few super-important courses

- Create more detailed design document, explain rationale.
- Explain strategies for teaching, assessment, lab work.
- Monitor implementation, student response.

What course to choose?

- Foundational
- Minimum prerequisites

Educational scenario in India

Many graduates unemployable
Much education is based on rote learning

Memorization rather than thought

A proposal: Intensively reform a few super-important courses

- Create more detailed design document, explain rationale.
- Explain strategies for teaching, assessment, lab work.
- Monitor implementation, student response.

What course to choose?

- Foundational
- Minimum prerequisites
- Considered difficult to teach.

Educational scenario in India

Many graduates unemployable
Much education is based on rote learning

Memorization rather than thought

A proposal: Intensively reform a few super-important courses

- Create more detailed design document, explain rationale.
- Explain strategies for teaching, assessment, lab work.
- Monitor implementation, student response.

What course to choose?

- Foundational
- Minimum prerequisites
- Considered difficult to teach.

Introductory Programming?

The introductory programming course

The introductory programming course

- First course in CS curriculum :

The introductory programming course

- First course in CS curriculum :
  - Foundational for all areas of CS.

The introductory programming course

- First course in CS curriculum :
  - Foundational for all areas of CS.
  - Can shape students attitude towards all of CS

The introductory programming course

- First course in CS curriculum :
  - Foundational for all areas of CS.
  - Can shape students attitude towards all of CS
- Important also for non CS majors:

The introductory programming course

- First course in CS curriculum :
  - Foundational for all areas of CS.
  - Can shape students attitude towards all of CS
- Important also for non CS majors:
  - Many find jobs in IT industry

The introductory programming course

- First course in CS curriculum :
  - Foundational for all areas of CS.
  - Can shape students attitude towards all of CS
- Important also for non CS majors:
  - Many find jobs in IT industry
  - Engineering and science uses computers heavily

The introductory programming course

- First course in CS curriculum :
  - Foundational for all areas of CS.
  - Can shape students attitude towards all of CS
- Important also for non CS majors:
  - Many find jobs in IT industry
  - Engineering and science uses computers heavily
- Is theoretical and practical.

The introductory programming course

- First course in CS curriculum :
  - Foundational for all areas of CS.
  - Can shape students attitude towards all of CS
- Important also for non CS majors:
  - Many find jobs in IT industry
  - Engineering and science uses computers heavily
- Is theoretical and practical.
- Touches all aspects of life: science, technology, logistics, fun.

The introductory programming course

- First course in CS curriculum :
  - Foundational for all areas of CS.
  - Can shape students attitude towards all of CS
- Important also for non CS majors:
  - Many find jobs in IT industry
  - Engineering and science uses computers heavily
- Is theoretical and practical.
- Touches all aspects of life: science, technology, logistics, fun.
- Psychologically attractive: students feel in-charge!

The introductory programming course

- First course in CS curriculum :
  - Foundational for all areas of CS.
  - Can shape students attitude towards all of CS
- Important also for non CS majors:
  - Many find jobs in IT industry
  - Engineering and science uses computers heavily
- Is theoretical and practical.
- Touches all aspects of life: science, technology, logistics, fun.
- Psychologically attractive: students feel in-charge!

  Empowering and psychologically liberating if done right!

Current status

Considered difficult to teach worldwide.

Current status

Considered difficult to teach worldwide.

- Failure rates in intro programming course are about 30%.

Current status

Considered difficult to teach worldwide.

- Failure rates in intro programming course are about 30%.
  Watson-Li 14[WL14], Bennedson-Casperson 07[BC07]

Current status

Considered difficult to teach worldwide.

- Failure rates in intro programming course are about 30%.
  Watson-Li 14[WL14], Bennedson-Casperson 07[BC07]
- Students can manually solve very complex problems, but cannot write programs to solve those.

Current status

Considered difficult to teach worldwide.

- Failure rates in intro programming course are about 30%.
  Watson-Li 14[WL14], Bennedson-Casperson 07[BC07]
- Students can manually solve very complex problems, but cannot write programs to solve those. [Win96]

Current status

Considered difficult to teach worldwide.

- Failure rates in intro programming course are about 30%.
  Watson-Li 14[WL14], Bennedson-Casperson 07[BC07]
- Students can manually solve very complex problems, but cannot write programs to solve those. [Win96]
- Programming may be inherently hard.

Current status

Considered difficult to teach worldwide.

- Failure rates in intro programming course are about 30%. Watson-Li 14[WL14], Bennedson-Casperson 07[BC07]
- Students can manually solve very complex problems, but cannot write programs to solve those. [Win96]
- Programming may be inherently hard. [Guz10]

Current status

Considered difficult to teach worldwide.

- Failure rates in intro programming course are about 30%. Watson-Li 14[WL14], Bennedson-Casperson 07[BC07]
- Students can manually solve very complex problems, but cannot write programs to solve those. [Win96]
- Programming may be inherently hard. [Guz10]
- Strong reform is needed.

Current status

Considered difficult to teach worldwide.

- Failure rates in intro programming course are about 30%. Watson-Li 14[WL14], Bennedson-Casperson 07[BC07]
- Students can manually solve very complex problems, but cannot write programs to solve those. [Win96]
- Programming may be inherently hard. [Guz10]
- Strong reform is needed. [LR16]

Current status

Considered difficult to teach worldwide.

- Failure rates in intro programming course are about 30%. Watson-Li 14[WL14], Bennedson-Casperson 07[BC07]
- Students can manually solve very complex problems, but cannot write programs to solve those. [Win96]
- Programming may be inherently hard. [Guz10]
- Strong reform is needed. [LR16]

Indian scenario:

Current status

Considered difficult to teach worldwide.

- Failure rates in intro programming course are about 30%. Watson-Li 14[WL14], Bennedson-Casperson 07[BC07]
- Students can manually solve very complex problems, but cannot write programs to solve those. [Win96]
- Programming may be inherently hard. [Guz10]
- Strong reform is needed. [LR16]

Indian scenario:

- Graduates cannot write simple programs.

Current status

Considered difficult to teach worldwide.

- Failure rates in intro programming course are about 30%. Watson-Li 14[WL14], Bennedson-Casperson 07[BC07]
- Students can manually solve very complex problems, but cannot write programs to solve those. [Win96]
- Programming may be inherently hard. [Guz10]
- Strong reform is needed. [LR16]

Indian scenario:

- Graduates cannot write simple programs. Many surveys

Outline

Outline

- The design of the course

Outline

- The design of the course
- Conveying the spirit of the course

Outline

- The design of the course
- Conveying the spirit of the course
    - Communicating the goals to students

Outline

- The design of the course
- Conveying the spirit of the course
  - Communicating the goals to students
  - Communicating the attractive aspects of programming

Outline

- The design of the course
- Conveying the spirit of the course
  - Communicating the goals to students
  - Communicating the attractive aspects of programming
  - Conveying some fundamental ideas early on

Outline

- The design of the course
- Conveying the spirit of the course
  - Communicating the goals to students
  - Communicating the attractive aspects of programming
  - Conveying some fundamental ideas early on
- Teaching how to design programs

Outline

- The design of the course
- Conveying the spirit of the course
  - Communicating the goals to students
  - Communicating the attractive aspects of programming
  - Conveying some fundamental ideas early on
- Teaching how to design programs
  - A conjecture why programming is found difficult

Outline

- The design of the course
- Conveying the spirit of the course
  - Communicating the goals to students
  - Communicating the attractive aspects of programming
  - Conveying some fundamental ideas early on
- Teaching how to design programs
  - A conjecture why programming is found difficult
  - Pedagogical implications

Outline

- The design of the course
- Conveying the spirit of the course
  - Communicating the goals to students
  - Communicating the attractive aspects of programming
  - Conveying some fundamental ideas early on
- Teaching how to design programs
  - A conjecture why programming is found difficult
  - Pedagogical implications
- Teaching “difficult” language features

Outline

- The design of the course
- Conveying the spirit of the course
  - Communicating the goals to students
  - Communicating the attractive aspects of programming
  - Conveying some fundamental ideas early on
- Teaching how to design programs
  - A conjecture why programming is found difficult
  - Pedagogical implications
- Teaching "difficult" language features
- Experience

Standard designs of introductory programming

Standard designs of introductory programming

Common style of many course descriptions:

Standard designs of introductory programming

Common style of many course descriptions:

- Dry and cryptic

Standard designs of introductory programming

Common style of many course descriptions:

- Dry and cryptic

Like course descriptions of most subjects

Standard designs of introductory programming

Common style of many course descriptions:

- Dry and cryptic

Like course descriptions of most subjects

- "Learn to write programs to solve simple problems."

Standard designs of introductory programming

Common style of many course descriptions:

- Dry and cryptic

  Like course descriptions of most subjects

- “Learn to write programs to solve simple problems.”

  No definition of “simple”. No examples. No details.

Standard designs of introductory programming

Common style of many course descriptions:

- Dry and cryptic

  Like course descriptions of most subjects

- "Learn to write programs to solve simple problems."

  No definition of "simple". No examples. No details.

- Learn language X.

Standard designs of introductory programming

Common style of many course descriptions:

- Dry and cryptic

  Like course descriptions of most subjects

- "Learn to write programs to solve simple problems."

  No definition of "simple". No examples. No details.

- Learn language X.

  Usually profuse description.

Standard designs of introductory programming

Common style of many course descriptions:

- Dry and cryptic

  Like course descriptions of most subjects
- "Learn to write programs to solve simple problems."

  No definition of "simple". No examples. No details.
- Learn language X.

  Usually profuse description.

Inevitable result...

Standard designs of introductory programming

Common style of many course descriptions:

- Dry and cryptic

  Like course descriptions of most subjects

- "Learn to write programs to solve simple problems."

  No definition of "simple". No examples. No details.

- Learn language X.

  Usually profuse description.

Inevitable result...

- Students write few programs to solve unseen problems.

Standard designs of introductory programming

Common style of many course descriptions:

- Dry and cryptic

  Like course descriptions of most subjects
- "Learn to write programs to solve simple problems."

  No definition of "simple". No examples. No details.
- Learn language X.

  Usually profuse description.

Inevitable result...

- Students write few programs to solve unseen problems.
- Most course time is spent on language learning.

What should a design document contain?

What should a design document contain?

- High level course goal

What should a design document contain?

- High level course goal

  What student will be able to do after the course.

What should a design document contain?

- High level course goal

  What student will be able to do after the course.
  What makes the effort worthwhile.

What should a design document contain?

- High level course goal

What student will be able to do after the course.
What makes the effort worthwhile.
State in the language of laymen if possible.

What should a design document contain?

- High level course goal

  What student will be able to do after the course.

  What makes the effort worthwhile.

  State in the language of laymen if possible.

- Detailed learning objectives

What should a design document contain?

- High level course goal

  What student will be able to do after the course.
  What makes the effort worthwhile.
  State in the language of laymen if possible.

- Detailed learning objectives

State minimum accomplishments expected.

What should a design document contain?

- High level course goal

  What student will be able to do after the course.
  What makes the effort worthwhile.
  State in the language of laymen if possible.

- Detailed learning objectives

State minimum accomplishments expected.

Allow individual universities to decide how many hours/courses, but not fall below minimum.

What should a design document contain?

- High level course goal

  What student will be able to do after the course.
  What makes the effort worthwhile.
  State in the language of laymen if possible.

- Detailed learning objectives

State minimum accomplishments expected.

Allow individual universities to decide how many hours/courses, but not fall below minimum.

Define terms, do not say "solve simple problems".

What should a design document contain?

- High level course goal

What student will be able to do after the course.
What makes the effort worthwhile.
State in the language of laymen if possible.

- Detailed learning objectives

State minimum accomplishments expected.

Allow individual universities to decide how many hours/courses, but not fall below minimum.

Define terms, do not say "solve simple problems".

Give examples.

What should a design document contain?

- High level course goal

  What student will be able to do after the course.
  What makes the effort worthwhile.
  State in the language of laymen if possible.

- Detailed learning objectives

State minimum accomplishments expected.

Allow individual universities to decide how many hours/courses, but not fall below minimum.

Define terms, do not say “solve simple problems”.

Give examples.

Discuss evaluation and pedagogy strategies.

Course design sketch: Main course goal

Course design sketch: Main course goal

Write programs to perform all calculations you can do **manually.**

Course design sketch: Main course goal

Write programs to perform all calculations you can do **manually.**

Lots of practice with short programs: Arithmetic on numbers, matrices, polynomials, searching, root finding, ...

Course design sketch: Main course goal

Write programs to perform all calculations you can do **manually.**

Lots of practice with short programs: Arithmetic on numbers, matrices, polynomials, searching, root finding, ...

Medium sized (150 lines) programs: that model a system with state + evolution rules + user interaction.

Course design sketch: Main course goal

Write programs to perform all calculations you can do **manually.**

Lots of practice with short programs: Arithmetic on numbers, matrices, polynomials, searching, root finding, ...

Medium sized (150 lines) programs: that model a system with state + evolution rules + user interaction.

- Simulation of mechanical systems, orbiting planets, circuits.

Course design sketch: Main course goal

Write programs to perform all calculations you can do **manually.**

Lots of practice with short programs: Arithmetic on numbers, matrices, polynomials, searching, root finding, ...

Medium sized (150 lines) programs: that model a system with state + evolution rules + user interaction.

- Simulation of mechanical systems, orbiting planets, circuits.
- Simulation of computer execution, train systems.

Course design sketch: Main course goal

Write programs to perform all calculations you can do **manually.**

Lots of practice with short programs: Arithmetic on numbers, matrices, polynomials, searching, root finding, ...

Medium sized (150 lines) programs: that model a system with state + evolution rules + user interaction.

- Simulation of mechanical systems, orbiting planets, circuits.
- Simulation of computer execution, train systems.
- Games.

Course design sketch: Main course goal

Write programs to perform all calculations you can do **manually.**

Lots of practice with short programs: Arithmetic on numbers, matrices, polynomials, searching, root finding, ...

Medium sized (150 lines) programs: that model a system with state + evolution rules + user interaction.

- Simulation of mechanical systems, orbiting planets, circuits.
- Simulation of computer execution, train systems.
- Games.
- Library management, bank account management.

Course design sketch: Main course goal

Write programs to perform all calculations you can do **manually.**

Lots of practice with short programs: Arithmetic on numbers, matrices, polynomials, searching, root finding, ...

Medium sized (150 lines) programs: that model a system with state + evolution rules + user interaction.

- Simulation of mechanical systems, orbiting planets, circuits.
- Simulation of computer execution, train systems.
- Games.
- Library management, bank account management.

Clearly indicate depth of learning expected.

Course design sketch: Main course goal

Write programs to perform all calculations you can do **manually.**

Lots of practice with short programs: Arithmetic on numbers, matrices, polynomials, searching, root finding, ...

Medium sized (150 lines) programs: that model a system with state + evolution rules + user interaction.

- Simulation of mechanical systems, orbiting planets, circuits.
- Simulation of computer execution, train systems.
- Games.
- Library management, bank account management.

Clearly indicate depth of learning expected.

Efficiency issues: At least as efficient as **manual** computation.

Course design sketch: Main course goal

Write programs to perform all calculations you can do **manually.**

Lots of practice with short programs: Arithmetic on numbers, matrices, polynomials, searching, root finding, ...

Medium sized (150 lines) programs: that model a system with state + evolution rules + user interaction.

- Simulation of mechanical systems, orbiting planets, circuits.
- Simulation of computer execution, train systems.
- Games.
- Library management, bank account management.

Clearly indicate depth of learning expected.

Efficiency issues: At least as efficient as **manual** computation.

Correctness issues: Correctly mimic **manual** algorithm.

Course design sketch: Learning objectives:

Course design sketch: Learning objectives:

How a computer works:

Course design sketch: Learning objectives:

How a computer works:

- Binary representation for numbers

Course design sketch: Learning objectives:

How a computer works:

- Binary representation for numbers
- Representation of text, images, ... using numbers

Course design sketch: Learning objectives:

How a computer works:

- Binary representation for numbers
- Representation of text, images, ... using numbers
- Block diagram of CPU

Course design sketch: Learning objectives:

How a computer works:

- Binary representation for numbers
- Representation of text, images, ... using numbers
- Block diagram of CPU
- Memory and addresses

Course design sketch: Learning objectives:

How a computer works:

- Binary representation for numbers
- Representation of text, images, ... using numbers
- Block diagram of CPU
- Memory and addresses

Programming language syntax and semantics

Course design sketch: Learning objectives:

How a computer works:

- Binary representation for numbers
- Representation of text, images, ... using numbers
- Block diagram of CPU
- Memory and addresses

Programming language syntax and semantics

- Data types, variables, assignment

Course design sketch: Learning objectives:

How a computer works:

- Binary representation for numbers
- Representation of text, images, ... using numbers
- Block diagram of CPU
- Memory and addresses

Programming language syntax and semantics

- Data types, variables, assignment
- Conditional execution, Iteration

Course design sketch: Learning objectives:

How a computer works:

- Binary representation for numbers
- Representation of text, images, ... using numbers
- Block diagram of CPU
- Memory and addresses

Programming language syntax and semantics

- Data types, variables, assignment
- Conditional execution, Iteration
- Functions and recursion

Course design sketch: Learning objectives:

How a computer works:

- Binary representation for numbers
- Representation of text, images, ... using numbers
- Block diagram of CPU
- Memory and addresses

Programming language syntax and semantics

- Data types, variables, assignment
- Conditional execution, Iteration
- Functions and recursion
- Arrays and classes

Course design sketch: Learning objectives:

How a computer works:

- Binary representation for numbers
- Representation of text, images, ... using numbers
- Block diagram of CPU
- Memory and addresses

Programming language syntax and semantics

- Data types, variables, assignment
- Conditional execution, Iteration
- Functions and recursion
- Arrays and classes

Running time analysis: Understand time taken by nested loops.

Learning objectives (contd.)

Learning objectives (contd.)

Algorithm/Program Design:

Learning objectives (contd.)

Algorithm/Program Design:

- Design an algorithm for solving the problem manually.

Learning objectives (contd.)

Algorithm/Program Design:

- Design an algorithm for solving the problem manually.

  Using techniques learned prior to programming.

Learning objectives (contd.)

Algorithm/Program Design:

- Design an algorithm for solving the problem manually.

  Using techniques learned prior to programming.

- Understand the structure of the manual algorithm.

Learning objectives (contd.)

Algorithm/Program Design:

- Design an algorithm for solving the problem manually.

  Using techniques learned prior to programming.
- Understand the structure of the manual algorithm.
- Translate manual algorithm to computer program.

Learning objectives (contd.)

Algorithm/Program Design:

- Design an algorithm for solving the problem manually.
  Using techniques learned prior to programming.
- Understand the structure of the manual algorithm.
- Translate manual algorithm to computer program.
- Efficiency: as much as natural manual computation.

Learning objectives (contd.)

Algorithm/Program Design:

- Design an algorithm for solving the problem manually.
  Using techniques learned prior to programming.
- Understand the structure of the manual algorithm.
- Translate manual algorithm to computer program.
- Efficiency: as much as natural manual computation.

Program correctness:

Learning objectives (contd.)

Algorithm/Program Design:

- Design an algorithm for solving the problem manually.
  Using techniques learned prior to programming.
- Understand the structure of the manual algorithm.
- Translate manual algorithm to computer program.
- Efficiency: as much as natural manual computation.

Program correctness:

- Does your program do what you would do manually?

Learning objectives (contd.)

Algorithm/Program Design:

- Design an algorithm for solving the problem manually.
  Using techniques learned prior to programming.
- Understand the structure of the manual algorithm.
- Translate manual algorithm to computer program.
- Efficiency: as much as natural manual computation.

Program correctness:

- Does your program do what you would do manually?
- Loop invariants

Learning objectives (contd.)

Algorithm/Program Design:

- Design an algorithm for solving the problem manually.
  Using techniques learned prior to programming.
- Understand the structure of the manual algorithm.
- Translate manual algorithm to computer program.
- Efficiency: as much as natural manual computation.

Program correctness:

- Does your program do what you would do manually?
- Loop invariants

Software engineering:

Learning objectives (contd.)

Algorithm/Program Design:

- Design an algorithm for solving the problem manually.
  Using techniques learned prior to programming.
- Understand the structure of the manual algorithm.
- Translate manual algorithm to computer program.
- Efficiency: as much as natural manual computation.

Program correctness:

- Does your program do what you would do manually?
- Loop invariants

Software engineering:

How to breakup code into functions, classes..

Learning objectives (contd.)

Algorithm/Program Design:

- Design an algorithm for solving the problem manually.
  Using techniques learned prior to programming.
- Understand the structure of the manual algorithm.
- Translate manual algorithm to computer program.
- Efficiency: as much as natural manual computation.

Program correctness:

- Does your program do what you would do manually?
- Loop invariants

Software engineering:
How to breakup code into functions, classes..

Standard Library

Course design sketch: Additional advice

Course design sketch: Additional advice

- How to motivate students

Course design sketch: Additional advice

- How to motivate students
- How to convey the spirit of the course on day 1

Course design sketch: Additional advice

- How to motivate students
- How to convey the spirit of the course on day 1
- Which topics are difficult and how to teach them

Course design sketch: Additional advice

- How to motivate students
- How to convey the spirit of the course on day 1
- Which topics are difficult and how to teach them
- Strategies for setting exams

Course design sketch: Additional advice

- How to motivate students
- How to convey the spirit of the course on day 1
- Which topics are difficult and how to teach them
- Strategies for setting exams

Suggestive rather than mandatory

Course design sketch: Additional advice

- How to motivate students
- How to convey the spirit of the course on day 1
- Which topics are difficult and how to teach them
- Strategies for setting exams

Suggestive rather than mandatory

Many instructors will appreciate guidance.

Conveying the spirit of the course

What is the spirit of introductory programming?

What is the spirit of introductory programming?

- Programming is a powerful tool.

What is the spirit of introductory programming?

- Programming is a powerful tool.
- Programming contains intellectual challenge.

What is the spirit of introductory programming?

- Programming is a powerful tool.
- Programming contains intellectual challenge.
- Programming touches all aspects of life.

What is the spirit of introductory programming?

- Programming is a powerful tool.
- Programming contains intellectual challenge.
- Programming touches all aspects of life.

Should convey this early on..

What is the spirit of introductory programming?

- Programming is a powerful tool.
- Programming contains intellectual challenge.
- Programming touches all aspects of life.

Should convey this early on..

To get better student motivation.

What is the spirit of introductory programming?

- Programming is a powerful tool.
- Programming contains intellectual challenge.
- Programming touches all aspects of life.

Should convey this early on..

To get better student motivation.

“Convey” $\neq$ “State”

What is the spirit of introductory programming?

- Programming is a powerful tool.
- Programming contains intellectual challenge.
- Programming touches all aspects of life.

Should convey this early on..

To get better student motivation.

"Convey" $\neq$ "State"

Demos and examples have more impact.

A teaching tool: Simplecpp

A teaching tool: Simplecpp

www.cse.iitb.ac.in/~ranade/simplecpp

A teaching tool: Simplecpp

`www.cse.iitb.ac.in/~ranade/simplecpp`

- *Turtle* graphics

A teaching tool: Simplecpp

`www.cse.iitb.ac.in/~ranade/simplecpp`

- *Turtle* graphics
- Coordinate based 2D graphics

A teaching tool: Simplecpp

`www.cse.iitb.ac.in/~ranade/simplecpp`

- *Turtle* graphics
- Coordinate based 2D graphics
- Graphical input, Elementary animation

A teaching tool: Simplecpp

`www.cse.iitb.ac.in/~ranade/simplecpp`

- *Turtle* graphics
- Coordinate based 2D graphics
- Graphical input, Elementary animation
- Very easy to use; "alternative to `<iostream>`"

A teaching tool: Simplecpp

`www.cse.iitb.ac.in/~ranade/simplecpp`

- *Turtle* graphics
- Coordinate based 2D graphics
- Graphical input, Elementary animation
- Very easy to use; "alternative to `<iostream>`"
- Appropriate for the era of touch screens and cell phones

A teaching tool: Simplecpp

`www.cse.iitb.ac.in/~ranade/simplecpp`

- *Turtle* graphics
- Coordinate based 2D graphics
- Graphical input, Elementary animation
- Very easy to use; "alternative to `<iostream>`"
- Appropriate for the era of touch screens and cell phones
- Useful for illustrating recursion, classes, ...

A teaching tool: Simplecpp

`www.cse.iitb.ac.in/~ranade/simplecpp`

- *Turtle* graphics
- Coordinate based 2D graphics
- Graphical input, Elementary animation
- Very easy to use; "alternative to `<iostream>`"
- Appropriate for the era of touch screens and cell phones
- Useful for illustrating recursion, classes, ...
- "New statement": `repeat`

Getting to the essence of programming on day 1

Getting to the essence of programming on day 1

First impressions are very important

Getting to the essence of programming on day 1

First impressions are very important

- Convey the spirit, beauty, power of your material

Getting to the essence of programming on day 1

First impressions are very important

- Convey the spirit, beauty, power of your material
- Students are fresh and more alert on day 1

Getting to the essence of programming on day 1

First impressions are very important

- Convey the spirit, beauty, power of your material
- Students are fresh and more alert on day 1

Introduce programming using “Turtle Graphics”:

Getting to the essence of programming on day 1

First impressions are very important

- Convey the spirit, beauty, power of your material
- Students are fresh and more alert on day 1

Introduce programming using "Turtle Graphics":

- Invented in 1960s by Seymour Pappert, as part of the Logo programming language for teaching programming to children.

Getting to the essence of programming on day 1

First impressions are very important

- Convey the spirit, beauty, power of your material
- Students are fresh and more alert on day 1

Introduce programming using "Turtle Graphics":

- Invented in 1960s by Seymour Pappert, as part of the Logo programming language for teaching programming to children.
- Turtle:

Getting to the essence of programming on day 1

First impressions are very important

- Convey the spirit, beauty, power of your material
- Students are fresh and more alert on day 1

Introduce programming using “Turtle Graphics”:

- Invented in 1960s by Seymour Pappert, as part of the Logo programming language for teaching programming to children.
- Turtle:
  - A symbolic animal that lives on the screen.

Getting to the essence of programming on day 1

First impressions are very important

- Convey the spirit, beauty, power of your material
- Students are fresh and more alert on day 1

Introduce programming using "Turtle Graphics":

- Invented in 1960s by Seymour Pappert, as part of the Logo programming language for teaching programming to children.
- Turtle:
  - A symbolic animal that lives on the screen.
  - Moves as per commands issued by the program.

Getting to the essence of programming on day 1

First impressions are very important

- Convey the spirit, beauty, power of your material
- Students are fresh and more alert on day 1

Introduce programming using "Turtle Graphics":

- Invented in 1960s by Seymour Pappert, as part of the Logo programming language for teaching programming to children.
- Turtle:
  - A symbolic animal that lives on the screen.
  - Moves as per commands issued by the program.
  - Has a pen, which draws on the screen as the turtle moves.

Getting to the essence of programming on day 1

First impressions are very important

- Convey the spirit, beauty, power of your material
- Students are fresh and more alert on day 1

Introduce programming using "Turtle Graphics":

- Invented in 1960s by Seymour Pappert, as part of the Logo programming language for teaching programming to children.
- Turtle:
  - A symbolic animal that lives on the screen.
  - Moves as per commands issued by the program.
  - Has a pen, which draws on the screen as the turtle moves.
- Goal of turtle graphics: Draw interesting pictures on the screen.

The "Hello World" program

The "Hello World" program

```
#include <simplecpp>        // also loads iostream ...
```

The "Hello World" program

```
#include <simplecpp>          // also loads iostream ...
int main(){
```

The “Hello World” program

```
#include <simplecpp>          // also loads iostream ...
int main(){
  turtleSim();                 // Start turtle simulator
```

The “Hello World” program

```
#include <simplecpp>          // also loads iostream ...
int main(){
  turtleSim();                    // Start turtle simulator
  forward(100);   // Turtle to move 100 pixels forward
```

The “Hello World” program

```
#include <simplecpp>        // also loads iostream ...
int main(){
  turtleSim();                     // Start turtle simulator
  forward(100);   // Turtle to move 100 pixels forward
  right(120);      // Turtle to turn right 120 degrees
```

The "Hello World" program

```
#include <simplecpp>        // also loads iostream ...
int main(){
  turtleSim();                   // Start turtle simulator
  forward(100);   // Turtle to move 100 pixels forward
  right(120);      // Turtle to turn right 120 degrees
  forward(100);
```

The “Hello World” program

```
#include <simplecpp>          // also loads iostream ...
int main(){
  turtleSim();                    // Start turtle simulator
  forward(100);   // Turtle to move 100 pixels forward
  right(120);      // Turtle to turn right 120 degrees
  forward(100);
  right(120);
```

The "Hello World" program

```
#include <simplecpp>          // also loads iostream ...
int main(){
  turtleSim();                          // Start turtle simulator
  forward(100);   // Turtle to move 100 pixels forward
  right(120);      // Turtle to turn right 120 degrees
  forward(100);
  right(120);
  forward(100);
```

The "Hello World" program

```
#include <simplecpp>          // also loads iostream ...
int main(){
  turtleSim();                   // Start turtle simulator
  forward(100);   // Turtle to move 100 pixels forward
  right(120);      // Turtle to turn right 120 degrees
  forward(100);
  right(120);
  forward(100);
}
```

The second program for day 1

```
int main(){
  turtleSim();
  repeat(10){
    forward(100); right(36);
  }
}
```

The second program for day 1

```
int main(){
  turtleSim();
  repeat(10){
    forward(100); right(36);
  }
}
```

“New statement”: `repeat`

```
repeat (count) { statements to be repeated }
```

The second program for day 1

```
int main(){
  turtleSim();
  repeat(10){
    forward(100); right(36);
  }
}
```

"New statement": `repeat`

```
repeat (count) { statements to be repeated }
```

Implemented using C++ macros. Students can be told later.

The second program for day 1

```
int main(){
  turtleSim();
  repeat(10){
    forward(100); right(36);
  }
}
```

"New statement": `repeat`

```
repeat (count) { statements to be repeated }
```

Implemented using C++ macros. Students can be told later.
Statement is very easy to understand.

The second program for day 1

```
int main(){
  turtleSim();
  repeat(10){
    forward(100); right(36);
  }
}
```

"New statement": `repeat`

```
repeat (count) { statements to be repeated }
```

Implemented using C++ macros. Students can be told later.
Statement is very easy to understand.
Introduced to enable interesting programs from day 1.

Another day 1 program

```
int main(){
  turtleSim();
  repeat(10){
    repeat(4){
      forward(100); right(90);
    }
    right(10);
  }
  wait(10);
}
```

Another day 1 program

```
int main(){
  turtleSim();
  repeat(10){
    repeat(4){
      forward(100); right(90);
    }
    right(10);
  }
  wait(10);
}
```

“What do you think it does?”

Another day 1 program

```
int main(){
  turtleSim();
  repeat(10){
    repeat(4){
      forward(100); right(90);
    }
    right(10);
  }
  wait(10);
}
```

"What do you think it does?"

This is what I ask students. Most figure it out!

Another day 1 program

```
int main(){
  turtleSim();
  repeat(10){
    repeat(4){
      forward(100); right(90);
    }
    right(10);
  }
  wait(10);
}
```

"What do you think it does?"

This is what I ask students. Most figure it out!

Why? Because it is an interesting challenge!

What have students learnt on day 1?

What have students learnt on day 1?

1. Control flow

What have students learnt on day 1?

1. Control flow
2. Elementary iteration, including nested iteration

What have students learnt on day 1?

1. Control flow
2. Elementary iteration, including nested iteration
3. Basic ideas of syntax, spaces, indentation

What have students learnt on day 1?

1. Control flow
2. Elementary iteration, including nested iteration
3. Basic ideas of syntax, spaces, indentation
4. Importance of observing patterns in what is to be accomplished, and expressing them using `repeat`

What have students learnt on day 1?

1. Control flow
2. Elementary iteration, including nested iteration
3. Basic ideas of syntax, spaces, indentation
4. Importance of observing patterns in what is to be accomplished, and expressing them using `repeat`
   Do not write 10 statements to draw a 10 sided polygon!

What have students learnt on day 1?

1. Control flow
2. Elementary iteration, including nested iteration
3. Basic ideas of syntax, spaces, indentation
4. Importance of observing patterns in what is to be accomplished, and expressing them using `repeat`
   Do not write 10 statements to draw a 10 sided polygon!
   Very important activity while designing programs!

What have students learnt on day 1?

1. Control flow
2. Elementary iteration, including nested iteration
3. Basic ideas of syntax, spaces, indentation
4. Importance of observing patterns in what is to be accomplished, and expressing them using `repeat`
   Do not write 10 statements to draw a 10 sided polygon!
   Very important activity while designing programs!

   Essence of programming?

What have students learnt on day 1?

1. Control flow
2. Elementary iteration, including nested iteration
3. Basic ideas of syntax, spaces, indentation
4. Importance of observing patterns in what is to be accomplished, and expressing them using `repeat`
   Do not write 10 statements to draw a 10 sided polygon!
   Very important activity while designing programs!

Essence of programming?

Homework on day 1: draw chessboard, draw circles (as limit of n sided polygon). Draw 5 sided star.

What have students learnt on day 1?

1. Control flow
2. Elementary iteration, including nested iteration
3. Basic ideas of syntax, spaces, indentation
4. Importance of observing patterns in what is to be accomplished, and expressing them using `repeat`
   Do not write 10 statements to draw a 10 sided polygon!
   Very important activity while designing programs!

   Essence of programming?

Homework on day 1: draw chessboard, draw circles (as limit of n sided polygon). Draw 5 sided star.

Need some high school geometry. However, most programming needs some domain knowledge.

What have students learnt on day 1?

1. Control flow
2. Elementary iteration, including nested iteration
3. Basic ideas of syntax, spaces, indentation
4. Importance of observing patterns in what is to be accomplished, and expressing them using `repeat`
   Do not write 10 statements to draw a 10 sided polygon!
   Very important activity while designing programs!

Essence of programming?

Homework on day 1: draw chessboard, draw circles (as limit of n sided polygon). Draw 5 sided star.

Need some high school geometry. However, most programming needs some domain knowledge.

Students are happy to do this because they can see interesting things happening, they can feel the power.

Final day 1 activity: demo

Final day 1 activity: demo

Give a demo that shows off some of the things that can be done in the course.

Final day 1 activity: demo

Give a demo that shows off some of the things that can be done in the course.

- Drawing interesting patterns: Needs careful calculation

Final day 1 activity: demo

Give a demo that shows off some of the things that can be done in the course.

- Drawing interesting patterns: Needs careful calculation
- Drawing trees: exercise in recursion.

Final day 1 activity: demo

Give a demo that shows off some of the things that can be done in the course.

- Drawing interesting patterns: Needs careful calculation
- Drawing trees: exercise in recursion.
- Bouncing Balls, planetary motion: simulation of physical systems

Final day 1 activity: demo

Give a demo that shows off some of the things that can be done in the course.

- Drawing interesting patterns: Needs careful calculation
- Drawing trees: exercise in recursion.
- Bouncing Balls, planetary motion: simulation of physical systems
- Airport simulation

Final day 1 activity: demo

Give a demo that shows off some of the things that can be done in the course.

- Drawing interesting patterns: Needs careful calculation
- Drawing trees: exercise in recursion.
- Bouncing Balls, planetary motion: simulation of physical systems
- Airport simulation
- Cars: exercise in composing graphics objects

Final day 1 activity: demo

Give a demo that shows off some of the things that can be done in the course.

- Drawing interesting patterns: Needs careful calculation
- Drawing trees: exercise in recursion.
- Bouncing Balls, planetary motion: simulation of physical systems
- Airport simulation
- Cars: exercise in composing graphics objects

Demo may inspire students to do something for its sake, rather than for an exam.

Final day 1 activity: demo

Give a demo that shows off some of the things that can be done in the course.

- Drawing interesting patterns: Needs careful calculation
- Drawing trees: exercise in recursion.
- Bouncing Balls, planetary motion: simulation of physical systems
- Airport simulation
- Cars: exercise in composing graphics objects

Demo may inspire students to do something for its sake, rather than for an exam.

Demo conveys what we expect, what is possible after the course.

Sustaining the excitement after day 1

Sustaining the excitement after day 1

Topics for day 2/week 2: "How a computer works?". Data types and variables. Assignment statements.

Sustaining the excitement after day 1

Topics for day 2/week 2: "How a computer works?". Data types and variables. Assignment statements.

Standard teaching style: Information overload about number representation, assignment statements and its facets: truncation..

Sustaining the excitement after day 1

Topics for day 2/week 2: “How a computer works?”. Data types and variables. Assignment statements.

Standard teaching style: Information overload about number representation, assignment statements and its facets: truncation..

Can we make this interesting and active?

Sustaining the excitement after day 1

Topics for day 2/week 2: "How a computer works?". Data types and variables. Assignment statements.

Standard teaching style: Information overload about number representation, assignment statements and its facets: truncation..

Can we make this interesting and active?

`repeat` can bring out power of assignment statement

Sustaining the excitement after day 1

Topics for day 2/week 2: "How a computer works?". Data types and variables. Assignment statements.

Standard teaching style: Information overload about number representation, assignment statements and its facets: truncation..

Can we make this interesting and active?

`repeat` can bring out power of assignment statement

- Draw more interesting pictures.

Sustaining the excitement after day 1

Topics for day 2/week 2: "How a computer works?". Data types and variables. Assignment statements.

Standard teaching style: Information overload about number representation, assignment statements and its facets: truncation..

Can we make this interesting and active?

`repeat` can bring out power of assignment statement

- Draw more interesting pictures.

```
repeat(10){forward(i); right(90); i = i + 10;}
```

Sustaining the excitement after day 1

Topics for day 2/week 2: "How a computer works?". Data types and variables. Assignment statements.

Standard teaching style: Information overload about number representation, assignment statements and its facets: truncation..

Can we make this interesting and active?

`repeat` can bring out power of assignment statement

- Draw more interesting pictures.

  ```
  repeat(10){forward(i); right(90); i = i + 10;}
  ```
- Discuss accumulation and sequence generation idioms

Sustaining the excitement after day 1

Topics for day 2/week 2: "How a computer works?". Data types and variables. Assignment statements.

Standard teaching style: Information overload about number representation, assignment statements and its facets: truncation..

Can we make this interesting and active?

`repeat` can bring out power of assignment statement

- Draw more interesting pictures.

  ```
  repeat(10){forward(i); right(90); i = i + 10;}
  ```

- Discuss accumulation and sequence generation idioms

```
int i=1, sum=0, val=0;
repeat(10){ cin >> val; sum = sum + val; }  // accum.
repeat(10){ cout << i << endl; i = i * 2;}  //seq gen.
```

Sustaining the excitement after day 1

Topics for day 2/week 2: "How a computer works?". Data types and variables. Assignment statements.

Standard teaching style: Information overload about number representation, assignment statements and its facets: truncation..

Can we make this interesting and active?

`repeat` can bring out power of assignment statement

- Draw more interesting pictures.

  `repeat(10){forward(i); right(90); i = i + 10;}`
- Discuss accumulation and sequence generation idioms

```
int i=1, sum=0, val=0;
repeat(10){ cin >> val; sum = sum + val; }  // accum.
repeat(10){ cout << i << endl; i = i * 2;}  //seq gen.
```

These idioms would normally be written using `while`/`for`, which are too complicated for week 2.

Teaching program Design

Conventional wisdom: Programming is hard

Conventional wisdom: Programming is hard

Fact: In standards 1-12, students learn sophisticated algorithms for solving many problems manually.

Conventional wisdom: Programming is hard

Fact: In standards 1-12, students learn sophisticated algorithms for solving many problems manually.

- Integer arithmetic, factoring, GCD, ... (from primary school!)

Conventional wisdom: Programming is hard

Fact: In standards 1-12, students learn sophisticated algorithms for solving many problems manually.

- Integer arithmetic, factoring, GCD, ... (from primary school!)
- Arithmetic on matrices, polynomials

Conventional wisdom: Programming is hard

Fact: In standards 1-12, students learn sophisticated algorithms for solving many problems manually.

- Integer arithmetic, factoring, GCD, ... (from primary school!)
- Arithmetic on matrices, polynomials
- Calculus: integration, differentiation

Conventional wisdom: Programming is hard

Fact: In standards 1-12, students learn sophisticated algorithms for solving many problems manually.

- Integer arithmetic, factoring, GCD, ... (from primary school!)
- Arithmetic on matrices, polynomials
- Calculus: integration, differentiation
- Geometric constructions

Conventional wisdom: Programming is hard

Fact: In standards 1-12, students learn sophisticated algorithms for solving many problems manually.

- Integer arithmetic, factoring, GCD, ... (from primary school!)
- Arithmetic on matrices, polynomials
- Calculus: integration, differentiation
- Geometric constructions
- Solving physics problems

Conventional wisdom: Programming is hard

Fact: In standards 1-12, students learn sophisticated algorithms for solving many problems manually.

- Integer arithmetic, factoring, GCD, ... (from primary school!)
- Arithmetic on matrices, polynomials
- Calculus: integration, differentiation
- Geometric constructions
- Solving physics problems
- Balancing chemical equations

Conventional wisdom: Programming is hard

Fact: In standards 1-12, students learn sophisticated algorithms for solving many problems manually.

- Integer arithmetic, factoring, GCD, ... (from primary school!)
- Arithmetic on matrices, polynomials
- Calculus: integration, differentiation
- Geometric constructions
- Solving physics problems
- Balancing chemical equations
- Tax calculation, elementary commerce

Conventional wisdom: Programming is hard

Fact: In standards 1-12, students learn sophisticated algorithms for solving many problems manually.

- Integer arithmetic, factoring, GCD, ... (from primary school!)
- Arithmetic on matrices, polynomials
- Calculus: integration, differentiation
- Geometric constructions
- Solving physics problems
- Balancing chemical equations
- Tax calculation, elementary commerce

Observation: The programming exercises we ask in intro programming are typically much simpler than above problems!

Conventional wisdom: Programming is hard

Fact: In standards 1-12, students learn sophisticated algorithms for solving many problems manually.

- Integer arithmetic, factoring, GCD, ... (from primary school!)
- Arithmetic on matrices, polynomials
- Calculus: integration, differentiation
- Geometric constructions
- Solving physics problems
- Balancing chemical equations
- Tax calculation, elementary commerce

Observation: The programming exercises we ask in intro programming are typically much simpler than above problems!

Our students can execute complex algorithms

Conventional wisdom: Programming is hard

Fact: In standards 1-12, students learn sophisticated algorithms for solving many problems manually.

- Integer arithmetic, factoring, GCD, ... (from primary school!)
- Arithmetic on matrices, polynomials
- Calculus: integration, differentiation
- Geometric constructions
- Solving physics problems
- Balancing chemical equations
- Tax calculation, elementary commerce

Observation: The programming exercises we ask in intro programming are typically much simpler than above problems!

Our students can execute complex algorithms

But they cannot write programs based on simple algorithms!

A proposol to resolve the mystery

A proposol to resolve the mystery

"Students understand algorithms intuitively, not algebraically"

A proposol to resolve the mystery

"Students understand algorithms intuitively, not algebraically"

Example: Integer multiplication

A proposol to resolve the mystery

"Students understand algorithms intuitively, not algebraically"

Example: Integer multiplication

- Our students were not taught: "In the jth subiteration of the ith iteration, multiply the jth digit of the multiplier by the ith digit of the multiplicand"

A proposol to resolve the mystery

"Students understand algorithms intuitively, not algebraically"

Example: Integer multiplication

- Our students were not taught: "In the jth subiteration of the ith iteration, multiply the jth digit of the multiplier by the ith digit of the multiplicand"
- Our students know the algorithm as a geometric tableau – how you arrange the partial products in a staggered manner.

A proposol to resolve the mystery

"Students understand algorithms intuitively, not algebraically"

Example: Integer multiplication

- Our students were not taught: "In the jth subiteration of the ith iteration, multiply the jth digit of the multiplier by the ith digit of the multiplicand"
- Our students know the algorithm as a geometric tableau – how you arrange the partial products in a staggered manner.

What is needed for programming:

A proposol to resolve the mystery

"Students understand algorithms intuitively, not algebraically"

Example: Integer multiplication

- Our students were not taught: "In the jth subiteration of the ith iteration, multiply the jth digit of the multiplier by the ith digit of the multiplicand"
- Our students know the algorithm as a geometric tableau – how you arrange the partial products in a staggered manner.

What is needed for programming:

- An algebraic description of the computation.

A proposol to resolve the mystery

"Students understand algorithms intuitively, not algebraically"

Example: Integer multiplication

- Our students were not taught: "In the jth subiteration of the ith iteration, multiply the jth digit of the multiplier by the ith digit of the multiplicand"
- Our students know the algorithm as a geometric tableau – how you arrange the partial products in a staggered manner.

What is needed for programming:

- An algebraic description of the computation.

Hypothesis: Students have difficulty in translating from their intuitive/geometric understanding to an algebraic representation.

Towards resolving the crisis

Towards resolving the crisis

Pedagogy Proposal 1: Our general advice to students should be:

Towards resolving the crisis

Pedagogy Proposal 1: Our general advice to students should be:

1. First think of how you solve the problem manually.

Towards resolving the crisis

Pedagogy Proposal 1: Our general advice to students should be:

1. First think of how you solve the problem manually.

Assume for now: student can solve problem manually.

Towards resolving the crisis

Pedagogy Proposal 1: Our general advice to students should be:

1. First think of how you solve the problem manually.

   Assume for now: student can solve problem manually.

2. Introspect over the manual method.

Towards resolving the crisis

Pedagogy Proposal 1: Our general advice to students should be:

1. First think of how you solve the problem manually.

   Assume for now: student can solve problem manually.

2. Introspect over the manual method.

   Become aware of your own thought process..

Towards resolving the crisis

Pedagogy Proposal 1: Our general advice to students should be:

1. First think of how you solve the problem manually.

   Assume for now: student can solve problem manually.

2. Introspect over the manual method.

   Become aware of your own thought process..

3. Find patterns in what you are doing in the manual algorithm and write down the patterns algebraically.

Towards resolving the crisis

Pedagogy Proposal 1: Our general advice to students should be:

1. First think of how you solve the problem manually.

   Assume for now: student can solve problem manually.

2. Introspect over the manual method.

   Become aware of your own thought process..

3. Find patterns in what you are doing in the manual algorithm and write down the patterns algebraically.

Pedagogy Proposal 2: We should explicitly teach how to translate from human computation to computer computation

Towards resolving the crisis

Pedagogy Proposal 1: Our general advice to students should be:

1. First think of how you solve the problem manually.

   Assume for now: student can solve problem manually.

2. Introspect over the manual method.

   Become aware of your own thought process..

3. Find patterns in what you are doing in the manual algorithm and write down the patterns algebraically.

Pedagogy Proposal 2: We should explicitly teach how to translate from human computation to computer computation

- Human computation does not have "variables".

Towards resolving the crisis

Pedagogy Proposal 1: Our general advice to students should be:

1. First think of how you solve the problem manually.

   Assume for now: student can solve problem manually.

2. Introspect over the manual method.

   Become aware of your own thought process..

3. Find patterns in what you are doing in the manual algorithm and write down the patterns algebraically.

Pedagogy Proposal 2: We should explicitly teach how to translate from human computation to computer computation

- Human computation does not have "variables".
  Students have difficulty in forming and manipulating variables.

Towards resolving the crisis

Pedagogy Proposal 1: Our general advice to students should be:

1. First think of how you solve the problem manually.
   Assume for now: student can solve problem manually.
2. Introspect over the manual method.
   Become aware of your own thought process..
3. Find patterns in what you are doing in the manual algorithm and write down the patterns algebraically.

Pedagogy Proposal 2: We should explicitly teach how to translate from human computation to computer computation

- Human computation does not have "variables".
  Students have difficulty in forming and manipulating variables.
- Humans seem to see everything in a glance.

Towards resolving the crisis

Pedagogy Proposal 1: Our general advice to students should be:

1. First think of how you solve the problem manually.
   Assume for now: student can solve problem manually.
2. Introspect over the manual method.
   Become aware of your own thought process..
3. Find patterns in what you are doing in the manual algorithm and write down the patterns algebraically.

Pedagogy Proposal 2: We should explicitly teach how to translate from human computation to computer computation

- Human computation does not have "variables".
  Students have difficulty in forming and manipulating variables.
- Humans seem to see everything in a glance.
  A computer operates on few variables at a time.

Example 1: Computing the value of e

Example 1: Computing the value of e

Write a program to compute $e = \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \ldots$ by adding n terms.

Example 1: Computing the value of e

Write a program to compute $e = \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \ldots$ by adding n terms.

Our students can surely do this manually!

Example 1: Computing the value of e

Write a program to compute $e = \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \ldots$ by adding n terms.

Our students can surely do this manually!

But for writing a program, they will have many questions..

Example 1: Computing the value of e

Write a program to compute $e = \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \ldots$ by adding n terms.

Our students can surely do this manually!

But for writing a program, they will have many questions..

- Do we need a loop?

Example 1: Computing the value of e

Write a program to compute $e = \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \ldots$ by adding n terms.

Our students can surely do this manually!

But for writing a program, they will have many questions..

- Do we need a loop?
- How many iterations will it run?

Example 1: Computing the value of e

Write a program to compute $e = \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \ldots$ by adding n terms.

Our students can surely do this manually!

But for writing a program, they will have many questions..

- Do we need a loop?
- How many iterations will it run?
- What variables to use?

Example 1: Computing the value of e

Write a program to compute $e = \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \ldots$ by adding n terms.

Our students can surely do this manually!

But for writing a program, they will have many questions..

- Do we need a loop?
- How many iterations will it run?
- What variables to use?
- How to update the variables in each iteration?

Example 1: Computing the value of e

Write a program to compute $e = \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \ldots$ by adding n terms.

Our students can surely do this manually!

But for writing a program, they will have many questions..

- Do we need a loop?
- How many iterations will it run?
- What variables to use?
- How to update the variables in each iteration?

Students actually like this explained...

Computing the value of e (contd.)

Computing the value of e (contd.)

Our suggested strategy: First observe what you do manually

Computing the value of e (contd.)

Our suggested strategy: First observe what you do manually

- (Manual) computation has $n - 1$ phases.

Computing the value of e (contd.)

Our suggested strategy: First observe what you do manually

- (Manual) computation has $n - 1$ phases.
- In ith phase, you calculate the value of $\frac{1}{i!}$

Computing the value of e (contd.)

Our suggested strategy: First observe what you do manually

- (Manual) computation has $n-1$ phases.
- In ith phase, you calculate the value of $\frac{1}{i!}$

  Value calculated in ith phase $= \dfrac{\text{value in } i-1 \text{ th phase}}{i}$.

Computing the value of e (contd.)

Our suggested strategy: First observe what you do manually

- (Manual) computation has $n - 1$ phases.
- In ith phase, you calculate the value of $\frac{1}{i!}$
  Value calculated in ith phase $= \dfrac{\text{value in } i - 1 \text{ th phase}}{i}$.
- At this point you might realize it is better to give names,
  t_i = term calculated in ith iteration.
  s_i = sum calculated in ith iteration.

Computing the value of e (contd.)

Our suggested strategy: First observe what you do manually

- (Manual) computation has $n-1$ phases.
- In ith phase, you calculate the value of $\frac{1}{i!}$
  Value calculated in ith phase $= \dfrac{\text{value in } i-1 \text{ th phase}}{i}$.
- At this point you might realize it is better to give names,
  $t_i =$ term calculated in ith iteration.
  $s_i =$ sum calculated in ith iteration.
- Above observation: $t_i = t_{i-1}/i$.

Computing the value of e (contd.)

Our suggested strategy: First observe what you do manually

- (Manual) computation has $n - 1$ phases.
- In ith phase, you calculate the value of $\frac{1}{i!}$
  Value calculated in ith phase $= \dfrac{\text{value in } i-1 \text{ th phase}}{i}$.
- At this point you might realize it is better to give names,
  t_i = term calculated in ith iteration.
  s_i = sum calculated in ith iteration.
- Above observation: $t_i = t_{i-1}/i$. Also $s_i = s_{i-1} + t_i$

Computing the value of e (contd.)

Our suggested strategy: First observe what you do manually

- (Manual) computation has $n - 1$ phases.
- In ith phase, you calculate the value of $\frac{1}{i!}$

  Value calculated in ith phase $= \dfrac{\text{value in } i-1 \text{ th phase}}{i}$.
- At this point you might realize it is better to give names,

  t_i = term calculated in ith iteration.

  s_i = sum calculated in ith iteration.
- Above observation: $t_i = t_{i-1}/i$. Also $s_i = s_{i-1} + t_i$

  Different from pseudocode/flowchat: s_i, t_i are names of values.

Computing the value of e (contd.)

Our suggested strategy: First observe what you do manually

- (Manual) computation has $n-1$ phases.
- In ith phase, you calculate the value of $\frac{1}{i!}$

  Value calculated in ith phase $= \dfrac{\text{value in } i-1 \text{ th phase}}{i}$.
- At this point you might realize it is better to give names,

  $t_i =$ term calculated in ith iteration.

  $s_i =$ sum calculated in ith iteration.
- Above observation: $t_i = t_{i-1}/i$. Also $s_i = s_{i-1} + t_i$

  Different from pseudocode/flowchat: s_i, t_i are names of values.

"Next write the program"

Computing the value of e (contd.)

Our suggested strategy: First observe what you do manually

- (Manual) computation has $n-1$ phases.
- In ith phase, you calculate the value of $\frac{1}{i!}$

  Value calculated in ith phase $= \dfrac{\text{value in } i-1 \text{ th phase}}{i}$.
- At this point you might realize it is better to give names,

  $t_i =$ term calculated in ith iteration.

  $s_i =$ sum calculated in ith iteration.
- Above observation: $t_i = t_{i-1}/i$. Also $s_i = s_{i-1} + t_i$

  Different from pseudocode/flowchat: s_i, t_i are names of values.

"Next write the program"

- $n-1$ phases $\longrightarrow$ loop that runs $n-1$ times

Computing the value of e (contd.)

Our suggested strategy: First observe what you do manually

- (Manual) computation has $n - 1$ phases.
- In ith phase, you calculate the value of $\frac{1}{i!}$

  Value calculated in ith phase $= \dfrac{\text{value in } i - 1 \text{ th phase}}{i}$.
- At this point you might realize it is better to give names,

  $t_i =$ term calculated in ith iteration.

  $s_i =$ sum calculated in ith iteration.
- Above observation: $t_i = t_{i-1}/i$. Also $s_i = s_{i-1} + t_i$

  Different from pseudocode/flowchat: s_i, t_i are names of values.

"Next write the program"

- $n - 1$ phases $\longrightarrow$ loop that runs $n - 1$ times
- Use variables `s,t` to store the values s_i, t_i.

Computing the value of e (contd.)

Our suggested strategy: First observe what you do manually

- (Manual) computation has $n-1$ phases.
- In ith phase, you calculate the value of $\frac{1}{i!}$

  Value calculated in ith phase $= \dfrac{\text{value in } i-1 \text{ th phase}}{i}$.
- At this point you might realize it is better to give names,

  $t_i =$ term calculated in ith iteration.

  $s_i =$ sum calculated in ith iteration.
- Above observation: $t_i = t_{i-1}/i$. Also $s_i = s_{i-1} + t_i$

  Different from pseudocode/flowchat: s_i, t_i are names of values.

"Next write the program"

- $n-1$ phases $\longrightarrow$ loop that runs $n-1$ times
- Use variables `s,t` to store the values s_i, t_i.

```
double s=1, t=1;  int i=1, n; cin >> n;
repeat(n-1){
   t = t / i;  s = s + t; i = i + 1;
}
```

Remarks

Remarks

- Encourage students to write *precise* comments:

Remarks

- Encourage students to write *precise* comments:
  "At the beginning of ith iteration

Remarks

- Encourage students to write *precise* comments:
  "At the beginning of ith iteration
  s will hold $s_{i-1} = \frac{1}{0!} + \ldots, \frac{1}{i-1!}$

Remarks

- Encourage students to write *precise* comments:
  "At the beginning of ith iteration
  `s` will hold $s_{i-1} = \frac{1}{0!} + \ldots, \frac{1}{i-1!}$
  `t` will hold $t_{i-1} = \frac{1}{i-1!}$"

Remarks

- Encourage students to write *precise* comments:
  "At the beginning of ith iteration
  `s` will hold $s_{i-1} = \frac{1}{0!} + \ldots, \frac{1}{i-1!}$
  `t` will hold $t_{i-1} = \frac{1}{i-1!}$"

Invariants..

Remarks

- Encourage students to write *precise* comments:
  "At the beginning of ith iteration
  `s` will hold $s_{i-1} = \frac{1}{0!} + \ldots, \frac{1}{i-1!}$
  `t` will hold $t_{i-1} = \frac{1}{i-1!}$"

  Invariants..

- Program design can be overwhelming if you have too many choices.

Remarks

- Encourage students to write *precise* comments:
  "At the beginning of ith iteration
  `s` will hold $s_{i-1} = \frac{1}{0!} + \ldots, \frac{1}{i-1!}$
  `t` will hold $t_{i-1} = \frac{1}{i-1!}$"

  Invariants..

- Program design can be overwhelming if you have too many choices.

  "I know all statements, but I dont know which to select!"

Remarks

- Encourage students to write *precise* comments:
  "At the beginning of ith iteration
  `s` will hold $s_{i-1} = \frac{1}{0!} + \ldots, \frac{1}{i-1!}$
  `t` will hold $t_{i-1} = \frac{1}{i-1!}$"

  Invariants..

- Program design can be overwhelming if you have too many choices.

  "I know all statements, but I dont know which to select!"
  Knowing too many, very complex statements is confusing.

Remarks

- Encourage students to write *precise* comments:
  "At the beginning of ith iteration
  `s` will hold $s_{i-1} = \frac{1}{0!} + \ldots, \frac{1}{i-1!}$
  `t` will hold $t_{i-1} = \frac{1}{i-1!}$"

  Invariants..

- Program design can be overwhelming if you have too many choices.

  "I know all statements, but I dont know which to select!"
  Knowing too many, very complex statements is confusing.
  `while`, `for` are confusing to novices.

Remarks

- Encourage students to write *precise* comments:
  "At the beginning of ith iteration
  `s` will hold $s_{i-1} = \frac{1}{0!} + \ldots, \frac{1}{i-1!}$
  `t` will hold $t_{i-1} = \frac{1}{i-1!}$"

  Invariants..

- Program design can be overwhelming if you have too many choices.

  "I know all statements, but I dont know which to select!"
  Knowing too many, very complex statements is confusing.
  `while`, `for` are confusing to novices.
  `if` is also taught before `while`, `for`.

Remarks

- Encourage students to write *precise* comments:
  "At the beginning of ith iteration
  `s` will hold $s_{i-1} = \frac{1}{0!} + \ldots, \frac{1}{i-1!}$
  `t` will hold $t_{i-1} = \frac{1}{i-1!}$"

  Invariants..

- Program design can be overwhelming if you have too many choices.

  "I know all statements, but I dont know which to select!"
  Knowing too many, very complex statements is confusing.
  `while`, `for` are confusing to novices.
  `if` is also taught before `while`, `for`.

- When we teach program design:

Remarks

- Encourage students to write *precise* comments:
  "At the beginning of ith iteration
  `s` will hold $s_{i-1} = \frac{1}{0!} + \ldots, \frac{1}{i-1!}$
  `t` will hold $t_{i-1} = \frac{1}{i-1!}$"

  Invariants..

- Program design can be overwhelming if you have too many choices.

  "I know all statements, but I dont know which to select!"
  Knowing too many, very complex statements is confusing.
  `while`, `for` are confusing to novices.
  `if` is also taught before `while`, `for`.

- When we teach program design:
  students know repeat + assignment.

Remarks

- Encourage students to write *precise* comments:
  "At the beginning of ith iteration
  `s` will hold $s_{i-1} = \frac{1}{0!} + \ldots, \frac{1}{i-1!}$
  `t` will hold $t_{i-1} = \frac{1}{i-1!}$"

  Invariants..

- Program design can be overwhelming if you have too many choices.

  "I know all statements, but I dont know which to select!"
  Knowing too many, very complex statements is confusing.
  `while, for` are confusing to novices.
  `if` is also taught before `while, for`.

- When we teach program design:
  students know repeat + assignment.

  Fewer choices, less confusion.

Other revealing example of manual vs. computer computation

Other revealing example of manual vs. computer computation

- Remove extra spaces from a line of text.

Other revealing example of manual vs. computer computation

- Remove extra spaces from a line of text.
- Given a 2d array of bits, count number of objects (group of contiguous 1s)

Other revealing example of manual vs. computer computation

- Remove extra spaces from a line of text.
- Given a 2d array of bits, count number of objects (group of contiguous 1s)
- Determining whether a sequence of parentheses is balanced.

Other revealing example of manual vs. computer computation

- Remove extra spaces from a line of text.
- Given a 2d array of bits, count number of objects (group of contiguous 1s)
- Determining whether a sequence of parentheses is balanced.
- Place 8 queens on a chessboard so that no queen captures another.

Teaching difficult topics

Difficult topic: Recursion

Difficult topic: Recursion

With graphics: pictorial recursion

Difficult topic: Recursion

With graphics: pictorial recursion

Tree = trunk + two small trees

```
void tree(int levels){
  if(levels > 0){
    forward(levels*25);            // trunk
    left(15);
    tree(levels-1);                // first small tree
    right(30);
    tree(levels-1);                // second small tree
    left(15);
    forward(-levels*25);
  }
}
```

Difficult topic: Inheritance

Difficult topic: Inheritance

How to construct good motivating examples?

Difficult topic: Inheritance

How to construct good motivating examples?

Simplecpp provides `Composite` class for creating composite objects

Difficult topic: Inheritance

How to construct good motivating examples?

Simplecpp provides `Composite` class for creating composite objects

```
class Wheel : public Composite{
  Circle *rim;
  Line *spoke[10];
public:
  Wheel(double x, double y, Composite* owner=NULL) :
  Composite(x,y,owner){
    rim = new Circle(0,0,RADIUS,this);
    for(int i=0; i<10; i++){
      spoke[i] = new Line(0, 0, RADIUS*cos(i*PI/5),
      RADIUS*sin(i*PI/5), this);
    }
  }
};
```

Difficult topic: Inheritance

How to construct good motivating examples?

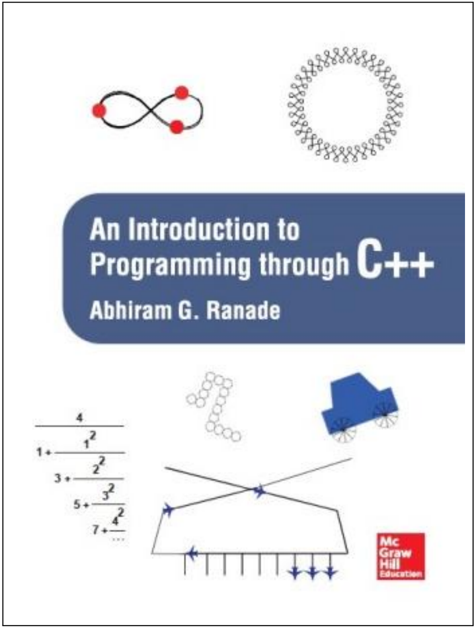
Simplecpp provides `Composite` class for creating composite objects

```
class Wheel : public Composite{
  Circle *rim;
  Line *spoke[10];
public:
  Wheel(double x, double y, Composite* owner=NULL) :
  Composite(x,y,owner){
    rim = new Circle(0,0,RADIUS,this);
    for(int i=0; i<10; i++){
      spoke[i] = new Line(0, 0, RADIUS*cos(i*PI/5),
      RADIUS*sin(i*PI/5), this);
    }
  }
};
```

Wheels can be created, moved, rotated, just like ordinary graphics objects

Our approach has been documented and tested!

Available in bookstores
Available on-line



Key features

Key features

- Develops and presents the teaching approach described so far.

Key features

- Develops and presents the teaching approach described so far.
- Substantial programming exercises

Key features

- Develops and presents the teaching approach described so far.
- Substantial programming exercises
- Project ideas

Key features

- Develops and presents the teaching approach described so far.
- Substantial programming exercises
- Project ideas
- Advanced topics in programming, numerical and graph algorithms,...

Key features

- Develops and presents the teaching approach described so far.
- Substantial programming exercises
- Project ideas
- Advanced topics in programming, numerical and graph algorithms,...

Book vs. MOOC?

Key features

- Develops and presents the teaching approach described so far.
- Substantial programming exercises
- Project ideas
- Advanced topics in programming, numerical and graph algorithms,...

Book vs. MOOC?

- Labs, homeworks need handholding.

Key features

- Develops and presents the teaching approach described so far.
- Substantial programming exercises
- Project ideas
- Advanced topics in programming, numerical and graph algorithms,...

Book vs. MOOC?

- Labs, homeworks need handholding.
- There has to be a knowledgeable teacher locally.

Key features

- Develops and presents the teaching approach described so far.
- Substantial programming exercises
- Project ideas
- Advanced topics in programming, numerical and graph algorithms,...

Book vs. MOOC?

- Labs, homeworks need handholding.
- There has to be a knowledgeable teacher locally.
- Book is an invaluable safety net for weak students.

Key features

- Develops and presents the teaching approach described so far.
- Substantial programming exercises
- Project ideas
- Advanced topics in programming, numerical and graph algorithms,...

Book vs. MOOC?

- Labs, homeworks need handholding.
- There has to be a knowledgeable teacher locally.
- Book is an invaluable safety net for weak students.
- Also provides challenge material to bright students.

Key features

- Develops and presents the teaching approach described so far.
- Substantial programming exercises
- Project ideas
- Advanced topics in programming, numerical and graph algorithms,...

Book vs. MOOC?

- Labs, homeworks need handholding.
- There has to be a knowledgeable teacher locally.
- Book is an invaluable safety net for weak students.
- Also provides challenge material to bright students.

Book + MOOC!

Experience

Experience

- Used in the introductory programming course at IITB during development and after publication.

Experience

- Used in the introductory programming course at IITB during development and after publication.
- Used in an IITB-EdX MOOC taught by Prof. D. B. Phatak and Prof. S. Chakraborty of IITB.

Experience

- Used in the introductory programming course at IITB during development and after publication.
- Used in an IITB-EdX MOOC taught by Prof. D. B. Phatak and Prof. S. Chakraborty of IITB.
- Currently used in the introductory programming course at

Experience

- Used in the introductory programming course at IITB during development and after publication.
- Used in an IITB-EdX MOOC taught by Prof. D. B. Phatak and Prof. S. Chakraborty of IITB.
- Currently used in the introductory programming course at
  - IIT Bombay, IIT Goa, IIT Dharwad

Experience

- Used in the introductory programming course at IITB during development and after publication.
- Used in an IITB-EdX MOOC taught by Prof. D. B. Phatak and Prof. S. Chakraborty of IITB.
- Currently used in the introductory programming course at
  - IIT Bombay, IIT Goa, IIT Dharwad
  - Vishwakarma Institute of Technology, Pune

Experience

- Used in the introductory programming course at IITB during development and after publication.
- Used in an IITB-EdX MOOC taught by Prof. D. B. Phatak and Prof. S. Chakraborty of IITB.
- Currently used in the introductory programming course at
  - IIT Bombay, IIT Goa, IIT Dharwad
  - Vishwakarma Institute of Technology, Pune
  - Department of Computer Science, Goa University.

Experience

- Used in the introductory programming course at IITB during development and after publication.
- Used in an IITB-EdX MOOC taught by Prof. D. B. Phatak and Prof. S. Chakraborty of IITB.
- Currently used in the introductory programming course at
  - IIT Bombay, IIT Goa, IIT Dharwad
  - Vishwakarma Institute of Technology, Pune
  - Department of Computer Science, Goa University.
- NPTEL course on "Design and Pedogy of the Introductory Programming Course".

Experience

- Used in the introductory programming course at IITB during development and after publication.
- Used in an IITB-EdX MOOC taught by Prof. D. B. Phatak and Prof. S. Chakraborty of IITB.
- Currently used in the introductory programming course at
  - IIT Bombay, IIT Goa, IIT Dharwad
  - Vishwakarma Institute of Technology, Pune
  - Department of Computer Science, Goa University.
- NPTEL course on "Design and Pedogy of the Introductory Programming Course".
- Students have used simplecpp graphics to do many exciting projects: games, puzzles, graphical editors, ...

Experience

- Used in the introductory programming course at IITB during development and after publication.
- Used in an IITB-EdX MOOC taught by Prof. D. B. Phatak and Prof. S. Chakraborty of IITB.
- Currently used in the introductory programming course at
  - IIT Bombay, IIT Goa, IIT Dharwad
  - Vishwakarma Institute of Technology, Pune
  - Department of Computer Science, Goa University.
- NPTEL course on "Design and Pedogy of the Introductory Programming Course".
- Students have used simplecpp graphics to do many exciting projects: games, puzzles, graphical editors, ...
- Simplecpp: available as library for unix, and embedded in an IDE for Windows/unix. Search the net.

Experience

- Used in the introductory programming course at IITB during development and after publication.
- Used in an IITB-EdX MOOC taught by Prof. D. B. Phatak and Prof. S. Chakraborty of IITB.
- Currently used in the introductory programming course at
  - IIT Bombay, IIT Goa, IIT Dharwad
  - Vishwakarma Institute of Technology, Pune
  - Department of Computer Science, Goa University.
- NPTEL course on "Design and Pedogy of the Introductory Programming Course".
- Students have used simplecpp graphics to do many exciting projects: games, puzzles, graphical editors, ...
- Simplecpp: available as library for unix, and embedded in an IDE for Windows/unix. Search the net.
- Chapterwise slides available to instructors.
  Contact McGraw Hill representative.

Summary: Introductory programming

Summary: Introductory programming

- Our students already know algorithms.

Summary: Introductory programming

- Our students already know algorithms.

  We help them translate what they know into programs.

Summary: Introductory programming

- Our students already know algorithms.

  We help them translate what they know into programs.
- Must write many small programs and a few large programs.

Summary: Introductory programming

- Our students already know algorithms.

  We help them translate what they know into programs.

- Must write many small programs and a few large programs.

  Programming can be understood only through practice.

Summary: Introductory programming

- Our students already know algorithms.

  We help them translate what they know into programs.

- Must write many small programs and a few large programs.

  Programming can be understood only through practice.

  Programs must do interesting things.

Summary: Introductory programming

- Our students already know algorithms.
  We help them translate what they know into programs.
- Must write many small programs and a few large programs.
  Programming can be understood only through practice.
  Programs must do interesting things.
- Graphics helps in grabbing student attention,

Summary: Introductory programming

- Our students already know algorithms.

  We help them translate what they know into programs.
- Must write many small programs and a few large programs.

  Programming can be understood only through practice.

  Programs must do interesting things.
- Graphics helps in grabbing student attention,

  Useful to explain concepts.

Summary: Introductory programming

- Our students already know algorithms.
  We help them translate what they know into programs.
- Must write many small programs and a few large programs.
  Programming can be understood only through practice.
  Programs must do interesting things.
- Graphics helps in grabbing student attention,
  Useful to explain concepts.
  Graphical input/output is useful in science and technology.

Summary: Introductory programming

- Our students already know algorithms.

  We help them translate what they know into programs.
- Must write many small programs and a few large programs.

  Programming can be understood only through practice.

  Programs must do interesting things.
- Graphics helps in grabbing student attention,

  Useful to explain concepts.

  Graphical input/output is useful in science and technology.
- `repeat` statement is understandable on day 1.

Summary: Introductory programming

- Our students already know algorithms.
  We help them translate what they know into programs.
- Must write many small programs and a few large programs.
  Programming can be understood only through practice.
  Programs must do interesting things.
- Graphics helps in grabbing student attention,
  Useful to explain concepts.
  Graphical input/output is useful in science and technology.
- `repeat` statement is understandable on day 1.
  Speeds up learning.

Summary: Introductory programming

- Our students already know algorithms.
  We help them translate what they know into programs.
- Must write many small programs and a few large programs.
  Programming can be understood only through practice.
  Programs must do interesting things.
- Graphics helps in grabbing student attention,
  Useful to explain concepts.
  Graphical input/output is useful in science and technology.
- `repeat` statement is understandable on day 1.
  Speeds up learning.
  Good first step towards standard looping statements.

Summary: Course design of specially selected courses

Summary: Course design of specially selected courses

- Course designs should be more detailed

Summary: Course design of specially selected courses

- Course designs should be more detailed
  - Specify the bottom line in a layman's language.

Summary: Course design of specially selected courses

- Course designs should be more detailed
  - Specify the bottom line in a layman's language.

    "What great things can I do at the end"

Summary: Course design of specially selected courses

- Course designs should be more detailed
  - Specify the bottom line in a layman's language.

    "What great things can I do at the end"
  - Course goals should take clear position on important issues.

Summary: Course design of specially selected courses

- Course designs should be more detailed
  - Specify the bottom line in a layman's language.

    "What great things can I do at the end"
  - Course goals should take clear position on important issues.

    How much algorithm design/hardware/software engg? ...

Summary: Course design of specially selected courses

- Course designs should be more detailed
  - Specify the bottom line in a layman's language.
    "What great things can I do at the end"
  - Course goals should take clear position on important issues.
    How much algorithm design/hardware/software engg? ...
  - Educational research literature should be consulted

Summary: Course design of specially selected courses

- Course designs should be more detailed
  - Specify the bottom line in a layman's language.

    "What great things can I do at the end"
  - Course goals should take clear position on important issues.

    How much algorithm design/hardware/software engg? ...
  - Educational research literature should be consulted

    "Standard difficulties"? How to overcome?

Summary: Course design of specially selected courses

- Course designs should be more detailed
  - Specify the bottom line in a layman's language.
    "What great things can I do at the end"
  - Course goals should take clear position on important issues.
    How much algorithm design/hardware/software engg? ...
  - Educational research literature should be consulted
    "Standard difficulties"? How to overcome?
  - Course designers should suggest pedagogy

Summary: Course design of specially selected courses

- Course designs should be more detailed
  - Specify the bottom line in a layman's language.
    "What great things can I do at the end"
  - Course goals should take clear position on important issues.
    How much algorithm design/hardware/software engg? ...
  - Educational research literature should be consulted
    "Standard difficulties"? How to overcome?
  - Course designers should suggest pedagogy
    How to convey spirit, exams, ...

Summary: Course design of specially selected courses

- Course designs should be more detailed
  - Specify the bottom line in a layman's language.

    "What great things can I do at the end"
  - Course goals should take clear position on important issues.

    How much algorithm design/hardware/software engg? ...
  - Educational research literature should be consulted

    "Standard difficulties"? How to overcome?
  - Course designers should suggest pedagogy

    How to convey spirit, exams, ...
- Course design should be a well-paid consultancy project?

Summary: Course design of specially selected courses

- Course designs should be more detailed
  - Specify the bottom line in a layman's language.
    "What great things can I do at the end"
  - Course goals should take clear position on important issues.
    How much algorithm design/hardware/software engg? ...
  - Educational research literature should be consulted
    "Standard difficulties"? How to overcome?
  - Course designers should suggest pedagogy
    How to convey spirit, exams, ...
- Course design should be a well-paid consultancy project?
  Expect professionalism but not altruism.

Summary: Course design of specially selected courses

- Course designs should be more detailed
  - Specify the bottom line in a layman's language.
    "What great things can I do at the end"
  - Course goals should take clear position on important issues.
    How much algorithm design/hardware/software engg? ...
  - Educational research literature should be consulted
    "Standard difficulties"? How to overcome?
  - Course designers should suggest pedagogy
    How to convey spirit, exams, ...
- Course design should be a well-paid consultancy project?
  Expect professionalism but not altruism.
- Regulatory bodies should monitor, ask compliance?

Summary: Course design of specially selected courses

- Course designs should be more detailed
  - Specify the bottom line in a layman's language.
    "What great things can I do at the end"
  - Course goals should take clear position on important issues.
    How much algorithm design/hardware/software engg? ...
  - Educational research literature should be consulted
    "Standard difficulties"? How to overcome?
  - Course designers should suggest pedagogy
    How to convey spirit, exams, ...
- Course design should be a well-paid consultancy project?
  Expect professionalism but not altruism.
- Regulatory bodies should monitor, ask compliance?
  Strong medicine, but only for chosen courses.

Summary: Course design of specially selected courses

- Course designs should be more detailed
  - Specify the bottom line in a layman's language.
    "What great things can I do at the end"
  - Course goals should take clear position on important issues.
    How much algorithm design/hardware/software engg? ...
  - Educational research literature should be consulted
    "Standard difficulties"? How to overcome?
  - Course designers should suggest pedagogy
    How to convey spirit, exams, ...
- Course design should be a well-paid consultancy project?
  Expect professionalism but not altruism.
- Regulatory bodies should monitor, ask compliance?
  Strong medicine, but only for chosen courses.
- Nation-wide electronic discussion forum for teachers?

N. Alzahrani, F. Vahid, A. Edgcomb, K. Nguyen, and R. Lysecky, *Python versus c++: An analysis of student struggle on small coding exercises in introductory programming courses*, Proceedings of the 2018 ACM SIGCSE Technical Symposium on Computer Science Education (New York, NY, USA), SIGCSE '18, ACM, 2018, pp. 86–91.

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

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.

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.

Choice of Programming Language

Choice of Programming Language

Very high level languages: Scheme, Haskel, Python, ...

Choice of Programming Language

Very high level languages: Scheme, Haskel, Python, ...

- Programming is cleaner, easier.

Choice of Programming Language

Very high level languages: Scheme, Haskel, Python, ...

- Programming is cleaner, easier.

But also see [AVE+18] for a contrary view

Choice of Programming Language

Very high level languages: Scheme, Haskel, Python, ...

- Programming is cleaner, easier.

  But also see [AVE$^+$18] for a contrary view
- Too abstract: hard to relate to "what really happens in the computer"

Choice of Programming Language

Very high level languages: Scheme, Haskel, Python, ...

- Programming is cleaner, easier.

  But also see [AVE+18] for a contrary view
- Too abstract: hard to relate to "what really happens in the computer"
- Hard to convincingly say: "`A[i]` can be accessed in constant time independent of `i`"

Choice of Programming Language

Very high level languages: Scheme, Haskel, Python, ...

- Programming is cleaner, easier.

  But also see [AVE+18] for a contrary view
- Too abstract: hard to relate to "what really happens in the computer"
- Hard to convincingly say: "`A[i]` can be accessed in constant time independent of `i`"

Low level languages: C

Choice of Programming Language

Very high level languages: Scheme, Haskel, Python, ...

- Programming is cleaner, easier.
  But also see [AVE+18] for a contrary view
- Too abstract: hard to relate to "what really happens in the computer"
- Hard to convincingly say: "`A[i]` can be accessed in constant time independent of `i`"

Low level languages: C

- Students can understand more easily "what really happens in the computer"

Choice of Programming Language

Very high level languages: Scheme, Haskel, Python, ...

- Programming is cleaner, easier.

  But also see [AVE+18] for a contrary view
- Too abstract: hard to relate to "what really happens in the computer"
- Hard to convincingly say: "`A[i]` can be accessed in constant time independent of `i`"

Low level languages: C

- Students can understand more easily "what really happens in the computer"
- Programming is harder: too much pointer manipulation

Choice of Programming Language

Very high level languages: Scheme, Haskel, Python, ...

- Programming is cleaner, easier.

  But also see [AVE+18] for a contrary view
- Too abstract: hard to relate to "what really happens in the computer"
- Hard to convincingly say: "`A[i]` can be accessed in constant time independent of `i`"

Low level languages: C

- Students can understand more easily "what really happens in the computer"
- Programming is harder: too much pointer manipulation
- Code replication needed: generic coding not possible

Our choice: C++

Our choice: C++

- Start with sane C subset

Our choice: C++

- Start with sane C subset

  Omit outdated features e.g. `i+++++j; x++ += y++;`

Our choice: C++

- Start with sane C subset

  Omit outdated features e.g. `i+++++j; x++ += y++;`

- Teach address arithmetic, memory management

Our choice: C++

- Start with sane C subset

  Omit outdated features e.g. `i+++++j; x++ += y++;`

- Teach address arithmetic, memory management

  Help understanding what happens in a computer, efficiency

Our choice: C++

- Start with sane C subset
  Omit outdated features e.g. `i+++++j; x++ += y++;`
- Teach address arithmetic, memory management
  Help understanding what happens in a computer, efficiency
- Teach `string`, `vector` classes which enable high level programming.

Our choice: C++

- Start with sane C subset
  Omit outdated features e.g. `i+++++j; x++ += y++;`
- Teach address arithmetic, memory management
  Help understanding what happens in a computer, efficiency
- Teach `string`, `vector` classes which enable high level programming.
- Key point: At once low level and high level.

Our choice: C++

- Start with sane C subset

  Omit outdated features e.g. `i+++++j; x++ += y++;`
- Teach address arithmetic, memory management

  Help understanding what happens in a computer, efficiency
- Teach `string`, `vector` classes which enable high level programming.
- Key point: At once low level and high level.
  - Students should be aware of pointers/memory management but should use standard library classes which hide these.

Our choice: C++

- Start with sane C subset

  Omit outdated features e.g. `i+++++j; x++ += y++;`
- Teach address arithmetic, memory management

  Help understanding what happens in a computer, efficiency
- Teach `string`, `vector` classes which enable high level programming.
- Key point: At once low level and high level.
  - Students should be aware of pointers/memory management but should use standard library classes which hide these.
  - Also other template classes e.g. map, queue.

Our choice: C++

- Start with sane C subset
  Omit outdated features e.g. `i+++++j; x++ += y++;`
- Teach address arithmetic, memory management
  Help understanding what happens in a computer, efficiency
- Teach `string`, `vector` classes which enable high level programming.
- Key point: At once low level and high level.
  - Students should be aware of pointers/memory management but should use standard library classes which hide these.
  - Also other template classes e.g. map, queue.
- Much safer and convenient than C, even without OOP.

Our choice: C++

- Start with sane C subset

  Omit outdated features e.g. `i+++++j; x++ += y++;`
- Teach address arithmetic, memory management

  Help understanding what happens in a computer, efficiency
- Teach `string`, `vector` classes which enable high level programming.
- Key point: At once low level and high level.
  - Students should be aware of pointers/memory management but should use standard library classes which hide these.
  - Also other template classes e.g. map, queue.
- Much safer and convenient than C, even without OOP.
- Growth possibility: features such as lambda expressions..