An approach to introductory programming

Abhiram Ranade
IIT Bombay

February 24, 2015
Some unique features of computer programming

Computer programming allows students to build something, unlike other courses which are about memorizing and understanding.

You can write programs about anything: math, engineering, commerce, even art. You can explore the world.

Every student can feel that he/she is in control of the computer. This is psychologically liberating.

There is potential for students to fall in love with programming. But do they?

Introductory programming = first course in CS/IT

If students love programming, they will have positive attitude towards rest of CS/IT.
Some unique features of computer programming

- Computer programming allows students to build something, unlike other courses which are about memorizing and understanding.
Some unique features of computer programming

- Computer programming allows students to build something, unlike other courses which are about memorizing and understanding.
- You can write programs about anything: math, engineering, commerce, even art. You can explore the world.
Some unique features of computer programming

▶ Computer programming allows students to build something, unlike other courses which are about memorizing and understanding.
▶ You can write programs about anything: math, engineering, commerce, even art. You can explore the world.
▶ Every student can feel that he/she is in control of the computer. This is psychologically liberating.
Some unique features of computer programming

- Computer programming allows students to build something, unlike other courses which are about memorizing and understanding.
- You can write programs about anything: math, engineering, commerce, even art. You can explore the world.
- Every student can feel that he/she is in control of the computer. This is psychologically liberating.

There is potential for students to fall in love with programming.
Some unique features of computer programming

- Computer programming allows students to build something, unlike other courses which are about memorizing and understanding.
- You can write programs about anything: math, engineering, commerce, even art. You can explore the world.
- Every student can feel that he/she is in control of the computer. This is psychologically liberating.

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

But do they?
Some unique features of computer programming

- Computer programming allows students to build something, unlike other courses which are about memorizing and understanding.
- You can write programs about anything: math, engineering, commerce, even art. You can explore the world.
- Every student can feel that he/she is in control of the computer. This is psychologically liberating.

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

But do they?

Introductory programming = first course in CS/IT
Some unique features of computer programming

- Computer programming allows students to build something, unlike other courses which are about memorizing and understanding.
- You can write programs about anything: math, engineering, commerce, even art. You can explore the world.
- Every student can feel that he/she is in control of the computer. This is psychologically liberating.

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

But do they?

Introductory programming = first course in CS/IT

- If students love programming, they will have positive attitude towards rest of CS/IT.
Our goal

Teach programming in a manner that
Our goal

Teach programming in a manner that

- Students will want to write programs as a hobby, for fun!
Our goal

Teach programming in a manner that

- Students will want to write programs as a hobby, for fun!
- Whenever students see a problem in real life, they will ask, can I write a program to understand this problem better?
Our goal

Teach programming in a manner that

- Students will want to write programs as a hobby, for fun!
- Whenever students see a problem in real life, they will ask, can I write a program to understand this problem better?
  - Programming can help you better understand math, science, engineering.
Our goal

Teach programming in a manner that

- Students will want to write programs as a hobby, for fun!
- Whenever students see a problem in real life, they will ask, can I write a program to understand this problem better?
  - Programming can help you better understand math, science, engineering.
  - Programming can help you explore questions such as “Is this the best way to build a timetable/dam/airport?”
Our goal

Teach programming in a manner that

➤ Students will want to write programs as a hobby, for fun!
➤ Whenever students see a problem in real life, they will ask, can I write a program to understand this problem better?
  ➤ Programming can help you better understand math, science, engineering.
  ➤ Programming can help you explore questions such as “Is this the best way to build a timetable/dam/airport?”
➤ Students appreciate the difficulty and beauty in programming.
The Book

Available in bookstores
Available on-line
Outline

- The prevailing approach to programming education
- General education principles
- Our approach
  - Overview
  - Getting to the essence of programming on day 1
  - "Real life" applications within reach of beginners
- About the book
The prevailing approach to programming education

▶ Too much emphasis on syntax. Learn language X, then language Y...
▶ Learn just one modern language, but learn it well!
▶ Nothing exciting happens in the first 3-4 weeks.
▶ "It is inevitable because students need to understand compiling, declarations, machine models."
Do they really?
▶ One popular book spends the first few chapters on how to print with desired formatting!
▶ Students don’t see any relation between programming and other subjects they study or with their day to day life.
▶ "Not enough time"
Better time management and planning needed.
The prevailing approach to programming education

- Too much emphasis on syntax. Learn language X, then language Y...
The prevailing approach to programming education

- Too much emphasis on syntax. Learn language X, then language Y...
  Learn just one modern language, but learn it well!
The prevailing approach to programming education

- Too much emphasis on syntax. Learn language X, then language Y...
  
  Learn just one modern language, but learn it well!

- Nothing exciting happens in the first 3-4 weeks.
The prevailing approach to programming education

- Too much emphasis on syntax. Learn language X, then language Y...
  
  Learn just one modern language, but learn it well!

- Nothing exciting happens in the first 3-4 weeks.
  
  "It is inevitable because students need to understand compiling, declarations, machine models."

- "Not enough time"
  
  Better time management and planning needed.
The prevailing approach to programming education

- Too much emphasis on syntax. Learn language X, then language Y...
  
  Learn just one modern language, but learn it well!

- Nothing exciting happens in the first 3-4 weeks.
  
  "It is inevitable because students need to understand compiling, declarations, machine models."

  Do they really?
The prevailing approach to programming education

- Too much emphasis on syntax. Learn language X, then language Y...
  
  Learn just one modern language, but learn it well!

- Nothing exciting happens in the first 3-4 weeks.
  - ”It is inevitable because students need to understand compiling, declarations, machine models.”
  
  Do they really?

- One popular book spends the first few chapters on how to print with desired formatting!
The prevailing approach to programming education

- Too much emphasis on syntax. Learn language X, then language Y...
  
  Learn just one modern language, but learn it well!

- Nothing exciting happens in the first 3-4 weeks.
  - "It is inevitable because students need to understand compiling, declarations, machine models."
    
    Do they really?

- One popular book spends the first few chapters on how to print with desired formatting!

- Students don’t see any relation between programming and other subjects they study or with their day to day life.
The prevailing approach to programming education

- Too much emphasis on syntax. Learn language X, then language Y...
  
  **Learn just one modern language, but learn it well!**

- Nothing exciting happens in the first 3-4 weeks.
  - ”It is inevitable because students need to understand compiling, declarations, machine models.”

  **Do they really?**

  - One popular book spends the first few chapters on how to print with desired formatting!

- Students don't see any relation between programming and other subjects they study or with their day to day life.
  - ”Not enough time”
The prevailing approach to programming education

- Too much emphasis on syntax. Learn language X, then language Y...
  
  Learn just one modern language, but learn it well!

- Nothing exciting happens in the first 3-4 weeks.
  - ”It is inevitable because students need to understand compiling, declarations, machine models.”
  
  Do they really?

- One popular book spends the first few chapters on how to print with desired formatting!

- Students don't see any relation between programming and other subjects they study or with their day to day life.
  - ”Not enough time”

  Better time management and planning needed.
Some educational principles (1)

Motivations for learning:

▶ "I cannot get a degree unless I learn this"
▶ "My teacher tells me it will be useful later"
▶ "I like it and I see it is useful"

"If you want to build a ship, don’t drum up people together to collect wood and don’t assign them tasks and work, but rather teach them to long for the endless immensity of the sea."

– Antoine de Saint-Exupéry
Some educational principles (1)

Motivations for learning:

- “I cannot get a degree unless I learn this”
- “My teacher tells me it will be useful later”
- “I like it and I see it is useful”

“If you want to build a ship, don’t drum up people together to collect wood and don’t assign them tasks and work, but rather teach them to long for the endless immensity of the sea.”

– Antoine de Saint-Exupéry
Some educational principles (1)

Motivations for learning:

▶ "I cannot get a degree unless I learn this"

– Antoine de Saint-Exupery
Some educational principles (1)

Motivations for learning:
- ”I cannot get a degree unless I learn this”
- ”My teacher tells me it will be useful later”

– Antoine de Saint-Exupéry
Some educational principles (1)

Motivations for learning:

▶ "I cannot get a degree unless I learn this"
▶ "My teacher tells me it will be useful later"
▶ "I like it and I see it is useful"
Some educational principles (1)

Motivations for learning:
- "I cannot get a degree unless I learn this"
- "My teacher tells me it will be useful later"
- "I like it and I see it is useful"

“If you want to build a ship, don’t drum up people together to collect wood and don’t assign them tasks and work, but rather teach them to long for the endless immensity of the sea.”

– Antoine de Saint-Exupery
Some educational principles (2)

How learning is assimilated:
▶ New learning must be reconciled with prior knowledge.
▶ Reconciliation can improve quality of both!
▶ By drawing on prior knowledge, learning can be speeded up.

Prior knowledge:
▶ Mathematics: Arithmetic, algebra, geometry, calculus, statistics.
▶ Physics: Mechanics, optics, circuit theory.
▶ General Knowledge: Networks of various kinds: family trees, web pages, transportation networks.

Students should get a taste of the computation in these areas.
Some educational principles (2)

How learning is assimilated:

- New learning must be reconciled with prior knowledge.
- Reconciliation can improve quality of both!
- By drawing on prior knowledge, learning can be speeded up.

Prior knowledge:

- Physics: Mechanics, optics, circuit theory.
- General Knowledge: Networks of various kinds: family trees, web pages, transportation networks.

Students should get a taste of the computation in these areas.
Some educational principles (2)

How learning is assimilated:

- New learning must be reconciled with prior knowledge.

Prior knowledge:
- Physics: Mechanics, optics, circuit theory.
- General Knowledge: Networks of various kinds: family trees, web pages, transportation networks.

Students should get a taste of the computation in these areas.
Some educational principles (2)

How learning is assimilated:

- New learning must be reconciled with prior knowledge.
- Reconciliation can improve quality of both!
Some educational principles (2)

How learning is assimilated:

- New learning must be reconciled with prior knowledge.
- Reconciliation can improve quality of both!
- By drawing on prior knowledge, learning can be speeded up.
Some educational principles (2)

How learning is assimilated:

▶ New learning must be reconciled with prior knowledge.
▶ Reconciliation can improve quality of both!
▶ By drawing on prior knowledge, learning can be speeded up.

Prior knowledge:

- Physics: Mechanics, optics, circuit theory.
- General Knowledge: Networks of various kinds: family trees, web pages, transportation networks.

Students should get a taste of the computation in these areas.
Some educational principles (2)

How learning is assimilated:

- New learning must be reconciled with prior knowledge.
- Reconciliation can improve quality of both!
- By drawing on prior knowledge, learning can be speeded up.

Prior knowledge:

Some educational principles (2)

How learning is assimilated:

- New learning must be reconciled with prior knowledge.
- Reconciliation can improve quality of both!
- By drawing on prior knowledge, learning can be speeded up.

Prior knowledge:

- Physics: Mechanics, optics, circuit theory.
Some educational principles (2)

How learning is assimilated:
▶ New learning must be reconciled with prior knowledge.
▶ Reconciliation can improve quality of both!
▶ By drawing on prior knowledge, learning can be speeded up.

Prior knowledge:
▶ Mathematics: Arithmetic, algebra, geometry, calculus, statistics.
▶ Physics: Mechanics, optics, circuit theory.
▶ General Knowledge: Networks of various kinds: family trees, web pages, transportation networks.
How learning is assimilated:

- New learning must be reconciled with prior knowledge.
- Reconciliation can improve quality of both!
- By drawing on prior knowledge, learning can be speeded up.

Prior knowledge:

- Physics: Mechanics, optics, circuit theory.
- General Knowledge: Networks of various kinds: family trees, web pages, transportation networks.

Students should get a taste of the computation in these areas.
Some educational principles (3)

Any subject = Principles or Essence + Details

Details must be taught, but they must not overwhelm the principles/essence.

What is the essence of computer programming?
Some educational principles (3)

Any subject = Principles or Essence + Details
Any subject = Principles or Essence + Details

Details must be taught, but they must not overwhelm the principles/essence.
Some educational principles (3)

Any subject = Principles or Essence + Details

Details must be taught, but they must not overwhelm the principles/essence.

What is the essence of computer programming?
Some educational principles (4)

"I hear and I forget. I see and I remember. I do and I understand."
– Confucius

▶ Pictures are important. Animations even more so. Even more interesting if the student herself does the animation.
▶ Doing and experiencing is more important than dry theory.
Some educational principles (4)

How learning happens:

“I hear and I forget. I see and I remember. I do and I understand.”
– Confucious

Pictures are important. Animations even more so. Even more interesting if the student herself does the animation.

Doing and experiencing is more important than dry theory.
Some educational principles (4)

How learning happens:

"I hear and I forget. I see and I remember. I do and I understand."

– Confucious
How learning happens:

"I hear and I forget. I see and I remember. I do and I understand."

– Confucious

- Pictures are important. Animations even more so. Even more interesting if the student herself does the animation.
Some educational principles (4)

How learning happens:

”I hear and I forget. I see and I remember. I do and I understand.”
– Confucious

- Pictures are important. Animations even more so. Even more interesting if the student herself does the animation.
- Doing and experiencing is more important than dry theory.
Our approach: main theses

The essence of programming is to understand the pattern of the computation and express it using the appropriate language construct.

Language learning is relatively easy if there is the opportunity to use what is learnt in a realistic setting. Language learning is boring, and even difficult/stressful, if the only goal is to memorize, and there is no opportunity for use.

Use programming examples from math and science, and even fun and art, to have synergy with the prior knowledge, and career goals of our students.

In the era of cellphones, graphical input and output are very important if the students are to take us seriously. Graphics is important also for science and math visualization.

Learning to reason about and talk about programs, formally and informally, is important.
Our approach: main theses

The essence of programming is to understand the pattern of the computation and express it using the appropriate language construct.

Language learning is relatively easy if there is the opportunity to use what is learnt in a realistic setting. Language learning is boring, and even difficult/stressful, if the only goal is to memorize, and there is no opportunity for use.

Use programming examples from math and science, and even fun and art, to have synergy with the prior knowledge, and career goals of our students.

In the era of cellphones, graphical input and output are very important if the students are to take us seriously. Graphics is important also for science and math visualization.

Learning to reason about and talk about programs, formally and informally, is important.
Our approach: main theses

The essence of programming is to understand the pattern of the computation and express it using the appropriate language construct.

Language learning is relatively easy if there is the opportunity to use what is learnt in a realistic setting. Language learning is boring, and even difficult/stressful, if the only goal is to memorize, and there is no opportunity for use.
Our approach: main theses

The essence of programming is to understand the pattern of the computation and express it using the appropriate language construct.

Language learning is relatively easy if there is the opportunity to use what is learnt in a realistic setting. Language learning is boring, and even difficult/stressful, if the only goal is to memorize, and there is no opportunity for use.

Use programming examples from math and science, and even fun and art, to have synergy with the prior knowledge, and career goals of our students.
Our approach: main theses

The essence of programming is to understand the pattern of the computation and express it using the appropriate language construct.

Language learning is relatively easy if there is the opportunity to use what is learnt in a realistic setting. Language learning is boring, and even difficult/stressful, if the only goal is to memorize, and there is no opportunity for use.

Use programming examples from math and science, and even fun and art, to have synergy with the prior knowledge, and career goals of our students.

In the era of cellphones, graphical input and output are very important if the students are to take us seriously. Graphics is important also for science and math visualization.
Our approach: main theses

The essence of programming is to understand the pattern of the computation and express it using the appropriate language construct.

Language learning is relatively easy if there is the opportunity to use what is learnt in a realistic setting. Language learning is boring, and even difficult/stressful, if the only goal is to memorize, and there is no opportunity for use.

Use programming examples from math and science, and even fun and art, to have synergy with the prior knowledge, and career goals of our students.

In the era of cellphones, graphical input and output are very important if the students are to take us seriously. Graphics is important also for science and math visualization.

Learning to reason about and talk about programs, formally and informally, is important.
Our approach: implementation

Realistic application driven language learning:

▶ "Consider problem X. X cannot be solved using what you know so far. So we will learn a language construct."

▶ For every language construct, e.g. loops, functions, arrays, OOP, we discuss many detailed realistic applications.

Formal and informal reasoning about each language construct.

Programming applications from many areas:

▶ Math, engineering, art, logistics, ... games, fun ...

A teaching tool: 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
Our approach: implementation

Realistic application driven language learning:

▶ "Consider problem X. X cannot be solved using what you know so far. So we will learn a language construct."
▶ For every language construct, e.g. loops, functions, arrays, OOP, we discuss many detailed realistic applications.
▶ Formal and informal reasoning about each language construct.

Programming applications from many areas:
▶ Math, engineering, art, logistics, ... games, fun ...

A teaching tool: 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
Our approach: implementation

Realistic application driven language learning:

▶ "Consider problem X. X cannot be solved using what you know so far. So we will learn a language construct."

▶ For every language construct, e.g. loops, functions, arrays, OOP, we discuss many detailed realistic applications.

▶ Formal and informal reasoning about each language construct.

Programming applications from many areas:

▶ Math, engineering, art, logistics, ... games, fun ...

A teaching tool: 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
Our approach: implementation

Realistic application driven language learning:

▶ "Consider problem X. X cannot be solved using what you know so far. So we will learn a language construct."

▶ For every language construct, e.g. loops, functions, arrays, OOP, we discuss many detailed realistic applications.
Our approach: implementation

Realistic application driven language learning:

▶ ”Consider problem X. X cannot be solved using what you know so far. So we will learn a language construct.”

▶ For every language construct, e.g. loops, functions, arrays, OOP, we discuss many detailed realistic applications.

▶ Formal and informal reasoning about each language construct.
Our approach: implementation

Realistic application driven language learning:

► ”Consider problem X. X cannot be solved using what you know so far. So we will learn a language construct.”

► For every language construct, e.g. loops, functions, arrays, OOP, we discuss many detailed realistic applications.

► Formal and informal reasoning about each language construct.

Programming applications from many areas:
Our approach: implementation

Realistic application driven language learning:

- "Consider problem X. X cannot be solved using what you know so far. So we will learn a language construct."
- For every language construct, e.g. loops, functions, arrays, OOP, we discuss many detailed realistic applications.
- Formal and informal reasoning about each language construct.

Programming applications from many areas:

- Math, engineering, art, logistics, ... games, fun ...
Our approach: implementation

Realistic application driven language learning:

- ”Consider problem X. X cannot be solved using what you know so far. So we will learn a language construct.”
- For every language construct, e.g. loops, functions, arrays, OOP, we discuss many detailed realistic applications.
- Formal and informal reasoning about each language construct.

Programming applications from many areas:

- Math, engineering, art, logistics, ... games, fun ...

A teaching tool: Simplecpp
Our approach: implementation

Realistic application driven language learning:

▶ ”Consider problem X. X cannot be solved using what you know so far. So we will learn a language construct.”
▶ For every language construct, e.g. loops, functions, arrays, OOP, we discuss many detailed realistic applications.
▶ Formal and informal reasoning about each language construct.

Programming applications from many areas:

▶ Math, engineering, art, logistics, ... games, fun ...

A teaching tool: Simplecpp

▶ Turtle graphics, Coordinate based 2D graphics
Our approach: implementation

Realistic application driven language learning:

▷ ”Consider problem X. X cannot be solved using what you know so far. So we will learn a language construct.”

▷ For every language construct, e.g. loops, functions, arrays, OOP, we discuss many detailed realistic applications.

▷ Formal and informal reasoning about each language construct.

Programming applications from many areas:

▷ Math, engineering, art, logistics, ... games, fun ...

A teaching tool: Simplecpp

▷ Turtle graphics, Coordinate based 2D graphics

▷ Graphical input, Elementary animation
Our approach: implementation

Realistic application driven language learning:

▶ ”Consider problem X. X cannot be solved using what you know so far. So we will learn a language construct.”
▶ For every language construct, e.g. loops, functions, arrays, OOP, we discuss many detailed realistic applications.
▶ Formal and informal reasoning about each language construct.

Programming applications from many areas:

▶ Math, engineering, art, logistics, ... games, fun ...

A teaching tool: Simplecpp

▶ Turtle graphics, Coordinate based 2D graphics
▶ Graphical input, Elementary animation
▶ Very easy to use; “alternative to <iostream>”
Our approach: implementation

Realistic application driven language learning:

▶ "Consider problem X. X cannot be solved using what you know so far. So we will learn a language construct."

▶ For every language construct, e.g. loops, functions, arrays, OOP, we discuss many detailed realistic applications.

▶ Formal and informal reasoning about each language construct.

Programming applications from many areas:

▶ Math, engineering, art, logistics, ... games, fun ...

A teaching tool: 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
Our approach: implementation

Realistic application driven language learning:

▶ ”Consider problem X. X cannot be solved using what you know so far. So we will learn a language construct.”
▶ For every language construct, e.g. loops, functions, arrays, OOP, we discuss many detailed realistic applications.
▶ Formal and informal reasoning about each language construct.

Programming applications from many areas:

▶ Math, engineering, art, logistics, ... games, fun ...

A teaching tool: 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, ...
Our approach: implementation

Realistic application driven language learning:

▶ ”Consider problem X. X cannot be solved using what you know so far. So we will learn a language construct.”
▶ For every language construct, e.g. loops, functions, arrays, OOP, we discuss many detailed realistic applications.
▶ Formal and informal reasoning about each language construct.

Programming applications from many areas:

▶ Math, engineering, art, logistics, ... games, fun ...

A teaching tool: 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`
Choice of Language: Pick C++ over C

String processing: C++ string class is much safer than null terminated char arrays of C. Possibility of many bugs is eliminated.

Initialization: By using constructors, proper initialization can be forced in C++. Less danger of forgetting.

Standard (Template) library: provides many features that you want, e.g. lists, priority queues, sorting. Also customizable algorithms, e.g. sorting.

Dynamic memory management: Reduce bugs by using standard idioms. Even better: avoid entirely by using standard library.

Object oriented features: Encapsulation, inheritance, polymorphism indeed make it easier and safer to write and extend programs.

Even without object orientation, teach C++ rather than C.
Choice of Language: Pick C++ over C

**String processing:** C++ string class is much safer than null terminated char arrays of C. Possibility of many bugs is eliminated.
Choice of Language: Pick C++ over C

**String processing:** C++ string class is much safer than null terminated char arrays of C. Possibility of many bugs is eliminated.

**Initialization:** By using constructors, proper initialization can be forced in C++. Less danger of forgetting.
Choice of Language: Pick C++ over C

String processing: C++ string class is much safer than null terminated char arrays of C. Possibility of many bugs is eliminated.

Initialization: By using constructors, proper initialization can be forced in C++. Less danger of forgetting.

Standard (Template) library: provides many features that you want, e.g. lists, priority queues, sorting. Also customizable algorithms, e.g. sorting.
Choice of Language: Pick C++ over C

**String processing:** C++ string class is much safer than null terminated char arrays of C. Possibility of many bugs is eliminated.

**Initialization:** By using constructors, proper initialization can be forced in C++. Less danger of forgetting.

**Standard (Template) library:** provides many features that you want, e.g. lists, priority queues, sorting. Also customizable algorithms, e.g. sorting.

**Dynamic memory management:** Reduce bugs by using standard idioms. Even better: avoid entirely by using standard library.
Choice of Language: Pick C++ over C

**String processing:** C++ string class is much safer than null terminated char arrays of C. Possibility of many bugs is eliminated.

**Initialization:** By using constructors, proper initialization can be forced in C++. Less danger of forgetting.

**Standard (Template) library:** provides many features that you want, e.g. lists, priority queues, sorting. Also customizable algorithms, e.g. sorting.

**Dynamic memory management:** Reduce bugs by using standard idioms. Even better: avoid entirely by using standard library.

**Object oriented features:** Encapsulation, inheritance, polymorphism indeed make it easier and safer to write and extend programs.
Choice of Language: Pick C++ over C

String processing: C++ string class is much safer than null terminated char arrays of C. Possibility of many bugs is eliminated.

Initialization: By using constructors, proper initialization can be forced in C++. Less danger of forgetting.

Standard (Template) library: provides many features that you want, e.g. lists, priority queues, sorting. Also customizable algorithms, e.g. sorting.

Dynamic memory management: Reduce bugs by using standard idioms. Even better: avoid entirely by using standard library.

Object oriented features: Encapsulation, inheritance, polymorphism indeed make it easier and safer to write and extend programs.

Even without object orientation, teach C++ rather than C.
Getting to the essence of programming on day 1

Introduction using "Turtle Geometry":

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.
Getting to the essence of programming on day 1

Introduction using "Turtle Geometry":

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.
Getting to the essence of programming on day 1

Introduction using "Turtle Geometry":
Invented in 1960s by Seymour Pappert, as part of the Logo programming language for teaching programming to children.
Introduction using "Turtle Geometry":
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.
Introduction using "Turtle Geometry":
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.
Introduction using "Turtle Geometry":
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.
Introduction using "Turtle Geometry":
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.
A simple program fragment

turtleSim();   // Start turtle graphics

forward(100);
right(90);

forward(100);
right(90);

forward(100);
right(90);

forward(100);
A simple program fragment

turtleSim(); // Start turtle graphics

forward(100);
right(90);

forward(100);
right(90);

forward(100);
right(90);

forward(100);
right(90);

forward(100);

What does this draw?
A simple program fragment

turtleSim(); // Start turtle graphics

forward(100);
right(90);

forward(100);
right(90);

forward(100);
right(90);

forward(100);

What does this draw?

This program can be discussed in the first lecture.
The real day 1 program

```cpp
#include <simplecpp>

main_program{
    turtleSim();
    forward(100); wait(0.5); right(90); wait(0.5);
    forward(100); wait(0.5); right(90); wait(0.5);
    forward(100); wait(0.5); right(90); wait(0.5);
    forward(100); wait(0.5); right(90); wait(0.5);
    forward(100);
    wait(5);
}
```
The real day 1 program

```cpp
#include <simplecpp>
main_program{
  turtleSim();
  forward(100); wait(0.5); right(90); wait(0.5);
  forward(100); wait(0.5); right(90); wait(0.5);
  forward(100); wait(0.5); right(90); wait(0.5);
  forward(100); wait(0.5); right(90); wait(0.5);
  forward(100);
  wait(5);
}
```

- `<simplecpp>`: For graphics functionality. Also includes "using namespace std;", and `<iostream>`. 
The real day 1 program

```cpp
#include <simplecpp>
main_program{
    turtleSim();
    forward(100); wait(0.5); right(90); wait(0.5);
    forward(100); wait(0.5); right(90); wait(0.5);
    forward(100); wait(0.5); right(90); wait(0.5);
    forward(100); wait(0.5); right(90); wait(0.5);
    forward(100);
    wait(5);
}
```

- `<simplecpp>`: For graphics functionality. Also includes “using namespace std;”, and `<iostream>.
- Students must eventually understand namespaces, iostreams, int main, but not on day 1.
The real day 1 program

```cpp
#include <simplecpp>
main_program{
    turtleSim();
    forward(100); wait(0.5); right(90); wait(0.5);
    forward(100); wait(0.5); right(90); wait(0.5);
    forward(100); wait(0.5); right(90); wait(0.5);
    forward(100); wait(0.5); right(90); wait(0.5);
    forward(100);
    wait(5);
}
```

- `<simplecpp>`: For graphics functionality. Also includes “using namespace std;”, and `<iostream>`.
- Students must eventually understand namespaces, iostreams, `int main`, but not on day 1.
- `main_program`: C++ preprocessor macro. Automatically gets translated to `int main()`
The real day 1 program

```cpp
#include <simplecpp>

main_program{
    turtleSim();
    forward(100); wait(0.5); right(90); wait(0.5);
    forward(100); wait(0.5); right(90); wait(0.5);
    forward(100); wait(0.5); right(90); wait(0.5);
    forward(100);
    wait(5);
}
```

- `<simplecpp>`: For graphics functionality. Also includes “using namespace std;”, and `<iostream>`.
- Students must eventually understand namespaces, iostreams, `int main`, but not on day 1.
- `main_program`: C++ preprocessor macro. Automatically gets translated to `int main()`.
- `wait`: So that we have time to see..
The second program for day 1

```cpp
#include <simplecpp>
main_program{
    turtleSim();
    repeat(4){
        forward(100); right(90);
    }
}
```
#include <simplecpp>
main_program{
  turtleSim();
  repeat(4){
    forward(100); right(90);
  }
}

“New statement”: repeat

repeat (count) { statements to be repeated }
The second program for day 1

```cpp
#include <simplecpp>
main_program{
  turtleSim();
  repeat(4){
    forward(100); right(90);
  }
}
```

“New statement”: repeat

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

Introduced to enable interesting programs from day 1.
The second program for day 1

```cpp
#include <simplecpp>
main_program{
    turtleSim();
    repeat(4){
        forward(100); right(90);
    }
}
```

“New statement”: repeat

repeat (count) { statements to be repeated }

Introduced to enable interesting programs from day 1. Students understand it instantly.
The second program for day 1

```cpp
#include <simplecpp>
main_program{
  turtleSim();
  repeat(4){
    forward(100); right(90);
  }
}
```

“New statement”: repeat

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

Introduced to enable interesting programs from day 1. Students understand it instantly. Implemented using C++ macros. Students can be told later.
Another day 1 program

main_program{
    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!
Another day 1 program

main_program{
    turtleSim();
    repeat(10){
        repeat(4){
            forward(100); right(90);
        }
        right(10);
    }
    wait(10);
}

“What do you think it does?”
Another day 1 program

```
main_program{
    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

```plaintext
main_program{
    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?

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 100 statements to draw a 100 sided polygon!

Very important activity while designing programs!

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.
What have students learnt on day 1?

1. Control flow

Very important activity while designing programs!

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.
What have students learnt on day 1?

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

Very important activity while designing programs!

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.
What have students learnt on day 1?

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

Very important activity while designing programs!

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.
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 100 statements to draw a 100 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 100 statements to draw a 100 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 100 statements to draw a 100 sided polygon!

Very important activity while designing programs!

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 100 statements to draw a 100 sided polygon!

Very important activity while designing programs!

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 100 statements to draw a 100 sided polygon!

Very important activity while designing programs!

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.
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.

But students already know "repeat"!

▶ Can draw more interesting pictures.

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

▶ Can evaluate series, calculate averages.
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..

But students already know ”repeat”!
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.

But students already know ”repeat”!

- Can 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..

But students already know ”repeat”!

- Can draw more interesting pictures.
  ```java
  repeat(10){forward(i); right(90); i=i+10;}
  ```
- Can evaluate series, calculate averages.
Week 3: Coordinate based graphics: Creation, Manipulation, Graphical input

Circle c1(200,225,25); // c1 = circle, center at (200,225), r = 25
Rectangle r1(200,200,50,70); // center coordinates, width, height
r1.right(45); // rotate 45 degrees
c1.move(10,20); // move by 10,20 in x,y directions

int z = getClick(); // z = 65536 * x coordinate of cursor + y coordinate

Excitement value: Elementary animation can be done.

Educational goals acheived: Implicit introduction to constructor and member access syntax. Practice with repetition, which will be needed to do interesting animations.

Will provide examples/programming problems for weeks to come
Circle c1(200,225,25);
   // c1 = circle. center at (200,225), r = 25
Rectangle r1(200,200,50,70);
   // center coordinates, width, height
Week 3: Coordinate based graphics: Creation, Manipulation, Graphical input

Circle c1(200,225,25);
    // c1 = circle. center at (200,225), r = 25
Rectangle r1(200,200,50,70);
    // center coordinates, width, height

r1.right(45); // rotate 45 degrees
c1.move(10,20); // move by 10,20 in x,y directions
Circle c1(200,225,25);
  // c1 = circle. center at (200,225), r = 25
Rectangle r1(200,200,50,70);
  // center coordinates, width, height

r1.right(45); // rotate 45 degrees
c1.move(10,20); // move by 10,20 in x,y directions

int z = getClick();
  // z = 65536 * x coordinate of cursor + y coordinate
Circle c1(200,225,25);  
   // c1 = circle. center at (200,225), r = 25  
Rectangle r1(200,200,50,70);  
   // center coordinates, width, height  
r1.right(45); // rotate 45 degrees  
c1.move(10,20); // move by 10,20 in x,y directions  

int z = getClick();  
// z = 65536 * x coordinate of cursor + y coordinate  

Excitement value: Elementary animation can be done.
Circle c1(200,225,25);
   // c1 = circle. center at (200,225), r = 25
Rectangle r1(200,200,50,70);
   // center coordinates, width, height

r1.right(45);    // rotate 45 degrees
c1.move(10,20);  // move by 10,20 in x,y directions

int z = getClick();
   // z = 65536 * x coordinate of cursor + y coordinate

**Excitement value:** Elementary animation can be done.
**Educational goals acheived:** Implicit introduction to constructor and member access syntax. Practice with repetition, which will be needed to do interesting animations.
Week 3: Coordinate based graphics: 
Creation, Manipulation, Graphical input

```c
Circle c1(200, 225, 25);
   // c1 = circle. center at (200, 225), r = 25
Rectangle r1(200, 200, 50, 70);
   // center coordinates, width, height
r1.right(45);  // rotate 45 degrees
c1.move(10, 20);  // move by 10,20 in x,y directions

int z = getClick();
   // z = 65536 * x coordinate of cursor + y coordinate

Excitement value: Elementary animation can be done.
Educational goals acheived: Implicit introduction to constructor and member access syntax. Practice with repetition, which will be needed to do interesting animations.

Will provide examples/programming problems for weeks to come.
Demos
Week 4 and beyond: Other language constructs

- Conditional Statement: if, select
  - Tax calculation
  - Implementing "buttons": Deciding whether the user has clicked inside a rectangle
  - Whether a number is prime

- Loops: while, do, break, for
  - Mark averaging
  - Computing the number of digits in a number
  - Whether a number is prime
  - Greatest common divisor. (Euclid). Correctness proof.

- Functions:
  - Euclid's GCD algorithm. Nonrecursive and recursive.
  - LCM (nested function calls)
  - Functions to draw polygons, drawing text
  - Specification of function, precondition, post condition
  - Drawing trees ("visible" recursion)
Week 4 and beyond: Other language constructs

Conditional Statement: if, select

- Tax calculation
- Implementing "buttons": Deciding whether the user has clicked inside a rectangle
- Whether a number is prime
Week 4 and beyond: Other language constructs

Conditional Statement: if, select
  - Tax calculation
  - Implementing “buttons”: Deciding whether the user has clicked inside a rectangle
  - Whether a number is prime

Loops: while, do, break, for
  - Mark averaging
  - Computing the number of digits in a number
  - Whether a number is prime
  - Greatest common divisor. (Euclid). Correctness proof.
Week 4 and beyond: Other language constructs

Conditional Statement: if, select
- Tax calculation
- Implementing ”buttons”: Deciding whether the user has clicked inside a rectangle
- Whether a number is prime

Loops: while, do, break, for
- Mark averaging
- Computing the number of digits in a number
- Whether a number is prime
- Greatest common divisor. (Euclid). Correctness proof.

Functions:
- Euclid’s GCD algorithm. Nonrecursive and recursive.
- LCM (nested function calls)
- Functions to draw polygons, drawing text
- Specification of function, precondition, post condition
- Drawing trees (”visible” recursion)
Application: Gravitational simulation:

Input: Current positions, velocities, masses of stars.

Output: How do they move over time.

You learned this in Physics. Why not see it for yourself?

Fundamental problem in development of Science, including computing.

Introduction to Euler methods: marching over time.

\[ \text{next position} = f(\text{current position}, \text{velocity}) \]
\[ \text{next velocity} = g(\text{current velocity}, \text{current positions}) \]

3d vectors: Structure that can be used for representing displacements, velocities, accelerations.

Overloading operators so that you can write vector assignments like

\[ s = u \times t + a \times t \times t / 2; \]
Application: Gravitational simulation:

**Input:** Current positions, velocities, masses of stars.

**Fundamental problem in development of Science, including computing.**

**Introduction to Euler methods: marching over time.**

\[
\text{next position} = f(\text{current position}, \text{velocity})
\]

\[
\text{next velocity} = g(\text{current velocity}, \text{current positions})
\]

**3d vectors:** Structure that can be used for representing displacements, velocities, accelerations.

**Overloading operators** so that you can write vector assignments like

\[
s = u \times t + a \times t \times t / 2;
\]
Application: Gravitational simulation:

**Input:** Current positions, velocities, masses of stars.

**Output:** How do they move over time.

Fundamental problem in development of Science, including computing.

Introduction to Euler methods: marching over time.

next position = f(current position, velocity)

next velocity = g(current velocity, current positions)

3d vectors: Structure that can be used for representing displacements, velocities, accelerations.

Overloading operators so that you can write vector assignments like

s = u*t + a*t*t/2;
Application: Gravitational simulation:

**Input:** Current positions, velocities, masses of stars.
**Output:** How do they move over time.

- You learned this in Physics. Why not see it for yourself?
Application: Gravitational simulation:

**Input:** Current positions, velocities, masses of stars.

**Output:** How do they move over time.

- You learned this in Physics. Why not see it for yourself?
- Fundamental problem in development of Science, including computing.
Application: Gravitational simulation:

**Input:** Current positions, velocities, masses of stars.

**Output:** How do they move over time.

- You learned this in Physics. Why not see it for yourself?
- Fundamental problem in development of Science, including computing.
- Introduction to Euler methods: marching over time.
  
  \[
  \text{next position} = f(\text{current position, velocity}) \\
  \text{next velocity} = g(\text{current velocity, current positions})
  \]
Application: Gravitational simulation:

**Input:** Current positions, velocities, masses of stars.

**Output:** How do they move over time.

- You learned this in Physics. Why not see it for yourself?
- Fundamental problem in development of Science, including computing.
- Introduction to Euler methods: marching over time.
  
  \[
  \text{next position} = f(\text{current position}, \text{velocity})
  \]
  
  \[
  \text{next velocity} = g(\text{current velocity}, \text{current positions})
  \]

- 3d vectors: Structure that can be used for representing displacements, velocities, accelerations.
Application: Gravitational simulation:

**Input:** Current positions, velocities, masses of stars.

**Output:** How do they move over time.

- You learned this in Physics. Why not see it for yourself?
- Fundamental problem in development of Science, including computing.
- Introduction to Euler methods: marching over time.
  
  \[
  \text{next position} = f(\text{current position, velocity})
  \]
  
  \[
  \text{next velocity} = g(\text{current velocity, current positions})
  \]

- 3d vectors: Structure that can be used for representing displacements, velocities, accelerations.

- Overloading operators so that you can write vector assignments like \( s = u*t + a*t*t/2 \);
Application: Layout of math expressions:

Very simple parsing: recursion.

Recursive algorithm to calculate positions.

Good example of tree based representation.

Ideas useful for symbolic computation including differentiation, integration.

Implementation without inheritance.

Implementation with inheritance, and extended to include exponents.
Application: Layout of math expressions:

Input: math expression e.g. "(1+1/(2+1/(2+(1/2)))))"
Application: Layout of math expressions:

Input: math expression e.g. "(1+(1/(2+(1/(2+(1/2))))))"

Output: Drawing:

\[
1 + \frac{1}{2 + \frac{1}{2 + \frac{1}{2}}}
\]
Application: Layout of math expressions:

Input: math expression e.g. "(1+(1/(2+(1/(2+(1/2)))))))"

Output: Drawing:

\[ 1 + \frac{1}{2 + \frac{1}{2 + \frac{1}{2}}} \]

- Very simple parsing: recursion.
Application: Layout of math expressions:

Input: math expression e.g. "(1+(1/(2+(1/(2+(1/2)))))))"

Output: Drawing:

\[ 1 + \frac{1}{2 + \frac{1}{2 + \frac{1}{2}}} \]

- Very simple parsing: recursion.
- Recursive algorithm to calculate positions
Application: Layout of math expressions:

Input: math expression e.g. "\((1+(1/(2+(1/(2+(1/2)))))))\)"

Output: Drawing:

\[
1 + \frac{1}{2 + \frac{1}{2 + \frac{1}{2}}}
\]

- Very simple parsing: recursion.
- Recursive algorithm to calculate positions
- Good example of tree based representation.
Application: Layout of math expressions:

**Input:** math expression e.g. "(1+(1/(2+(1/(2+(1/2))))))"

**Output:** Drawing:

\[
1 + \frac{1}{2 + \frac{1}{2 + \frac{1}{2}}}
\]

- Very simple parsing: recursion.
- Recursive algorithm to calculate positions
- Good example of tree based representation.
- Ideas useful for symbolic computation including differentiation, integration.
Application: Layout of math expressions:

Input: math expression e.g. 
"(1+(1/(2+(1/(2+(1/2))))))"

Output: Drawing:

\[
1 + \frac{1}{2 + \frac{1}{2 + \frac{1}{2}}}
\]

- Very simple parsing: recursion.
- Recursive algorithm to calculate positions.
- Good example of tree based representation.
- Ideas useful for symbolic computation including differentiation, integration.
- Implementation without inheritance.
Application: Layout of math expressions:

Input: math expression e.g. "(1+(1/(2+(1/(2+(1/2))))))"

Output: Drawing:

\[
1 + \frac{1}{2 + \frac{1}{\frac{1}{2}}}
\]

- Very simple parsing: recursion.
- Recursive algorithm to calculate positions
- Good example of tree based representation.
- Ideas useful for symbolic computation including differentiation, integration.
- Implementation without inheritance
- Implementation with inheritance, and extended to include exponents.
Application: Discrete event simulation

Main idea: When designing any system, make a mathematical model of it and its stimuli. See how it behaves under random stimuli. Does it remain stable, do very long queues build up?

Examples: Toll plaza, restaurants. Airport: Considered at some detail.

▶ Aircraft should not get too close on runways/taxiways.
▶ At any time there must be at most 1 aircraft at any "gate".
▶ Central question: How does delay vary with traffic?

Implementation issues:
▶ Events to happen in the future are placed in a "priority queue".
▶ Event = function. "Lambda Expression/C++11"
▶ Resource management and deadlocks are inherent part of such simulations.

Messenger cyclist simulation implements Dijkstra's shortest path algorithm
Application: Discrete event simulation

Main idea: When designing any system, make a mathematical model of it and its stimuli. See how it behaves under random stimuli. Does it remain stable, do very long queues build up?

Examples: Toll plaza, restaurants. Airport: Considered at some detail.

▶ Aircraft should not get too close on runways/taxiways.
▶ At any time there must be at most 1 aircraft at any “gate”.
▶ Central question: How does delay vary with traffic?

Implementation issues:
▶ Events to happen in the future are placed in a “priority queue”.
▶ Event = function. “Lambda Expression/C++11”
▶ Resource management and deadlocks are inherent part of such simulations.

Messenger cyclist simulation implements Dijkstra’s shortest path algorithm
Application: Discrete event simulation

Main idea: When designing any system, make a mathematical model of it and its stimuli. See how it behaves under random stimuli. Does it remain stable, do very long queues build up?

Examples: Toll plaza, restaurants. Airport: Considered at some detail.
Application: Discrete event simulation

Main idea: When designing any system, make a mathematical model of it and its stimuli. See how it behaves under random stimuli. Does it remain stable, do very long queues build up?

Examples: Toll plaza, restaurants. Airport: Considered at some detail.

- Aircraft should not get too close on runways/taxiways.
Application: Discrete event simulation

Main idea: When designing any system, make a mathematical model of it and its stimuli. See how it behaves under random stimuli. Does it remain stable, do very long queues build up?

Examples: Toll plaza, restaurants. Airport: Considered at some detail.

- Aircraft should not get too close on runways/taxiways.
- At any time there must be at most 1 aircraft at any “gate”.

Implementation issues:
- Events to happen in the future are placed in a “priority queue”.
- Event = function. “Lambda Expression/C++11”
- Resource management and deadlocks are inherent part of such simulations.

Messenger cyclist simulation implements Dijkstra’s shortest path algorithm
Application: Discrete event simulation

Main idea: When designing any system, make a mathematical model of it and its stimuli. See how it behaves under random stimuli. Does it remain stable, do very long queues build up?

Examples: Toll plaza, restaurants. Airport: Considered at some detail.

- Aircraft should not get too close on runways/taxiways.
- At any time there must be at most 1 aircraft at any "gate".
- Central question: How does delay vary with traffic?

Implementation issues:
- Events to happen in the future are placed in a "priority queue".
- Event = function. "Lambda Expression/C++11"
- Resource management and deadlocks are inherent part of such simulations.

Messenger cyclist simulation implements Dijkstra's shortest path algorithm
Application: Discrete event simulation

Main idea: When designing any system, make a mathematical model of it and its stimuli. See how it behaves under random stimuli. Does it remain stable, do very long queues build up?

Examples: Toll plaza, restaurants. Airport: Considered at some detail.
  ▶ Aircraft should not get too close on runways/taxiways.
  ▶ At any time there must be at most 1 aircraft at any ”gate”.
  ▶ Central question: How does delay vary with traffic?

Implementation issues:
Application: Discrete event simulation

Main idea: When designing any system, make a mathematical model of it and its stimuli. See how it behaves under random stimuli. Does it remain stable, do very long queues build up?

Examples: Toll plaza, restaurants. Airport: Considered at some detail.

- Aircraft should not get too close on runways/taxiways.
- At any time there must be at most 1 aircraft at any ”gate”.
- Central question: How does delay vary with traffic?

Implementation issues:

- Events to happen in the future are placed in a ”priority queue”.

Messenger cyclist simulation implements Dijkstra’s shortest path algorithm.
Application: Discrete event simulation

Main idea: When designing any system, make a mathematical model of it and its stimuli. See how it behaves under random stimuli. Does it remain stable, do very long queues build up?

Examples: Toll plaza, restaurants. Airport: Considered at some detail.
- Aircraft should not get too close on runways/taxiways.
- At any time there must be at most 1 aircraft at any ”gate”.
- Central question: How does delay vary with traffic?

Implementation issues:
- Events to happen in the future are placed in a ”priority queue”.
- Event = function. ”Lambda Expression/C++11”
Application: Discrete event simulation

Main idea: When designing any system, make a mathematical model of it and its stimuli. See how it behaves under random stimuli. **Does it remain stable, do very long queues build up?**

Examples: Toll plaza, restaurants. Airport: Considered at some detail.
- Aircraft should not get too close on runways/taxiways.
- At any time there must be at most 1 aircraft at any ”gate”.
- Central question: How does delay vary with traffic?

Implementation issues:
- Events to happen in the future are placed in a ”priority queue”.
- Event = function. **”Lambda Expression/C++11”**
- Resource management and deadlocks are inherent part of such simulations.
Application: Discrete event simulation

Main idea: When designing any system, make a mathematical model of it and its stimuli. See how it behaves under random stimuli. Does it remain stable, do very long queues build up?

Examples: Toll plaza, restaurants. Airport: Considered at some detail.

- Aircraft should not get too close on runways/taxiways.
- At any time there must be at most 1 aircraft at any "gate".
- Central question: How does delay vary with traffic?

Implementation issues:

- Events to happen in the future are placed in a "priority queue".
- Event = function. “Lambda Expression/C++11”
- Resource management and deadlocks are inherent part of such simulations.

Messenger cyclist simulation implements Dijkstra’s shortest path algorithm
Other applications and algorithms discussed

Computer Science
- Merge sort, Binary search trees
- Back track search, Game tree search.
- Reference counting. (C++11)

Other areas:
- Solving circuits, page rank.
- Newton-Raphson algorithm for multiple dimensions: use in solving a mechanical system. "Necklace".
Book adoption and support

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

Used in an IITB-EdX MOOC being taught by Prof. D. B. Phatak and Prof. S. Chakraborty of IITB.

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.
Book adoption and support

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

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

- Used in the introductory programming course at IITB during development and after publication.
- Used in an IITB-EdX MOOC being taught by Prof. D. B. Phatak and Prof. S. Chakraborty of IITB.
- Students have used simplecpp graphics to do many exciting projects: games, puzzles, graphical editors, ...
Book adoption and support

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

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

- Pictorially observing the effect of iteration, recursion drives home the concepts effortlessly.
Concluding remarks

- Pictorially observing the effect of iteration, recursion drives home the concepts effortlessly.
- Students reported that after programming math problems their understanding of math and programming both improved.
Concluding remarks

- Pictorially observing the effect of iteration, recursion drives home the concepts effortlessly.
- Students reported that after programming math problems their understanding of math and programming both improved.
- Substantial discussion of style: different ways of writing same logic, readability.
Concluding remarks

- Pictorially observing the effect of iteration, recursion drives home the concepts effortlessly.
- Students reported that after programming math problems their understanding of math and programming both improved.
- Substantial discussion of style: different ways of writing same logic, readability.
- To become a good programmer, you must write many, many, programs. Many interesting exercises are given after each chapter.
Concluding remarks

- Pictorially observing the effect of iteration, recursion drives home the concepts effortlessly.
- Students reported that after programming math problems their understanding of math and programming both improved.
- Substantial discussion of style: different ways of writing same logic, readability.
- To become a good programmer, you must write many, many, programs. Many interesting exercises are given after each chapter.
- Book contains a lot of applications which will help students for doing projects, stimulate their imagination.
“What is the central core of the subject [computer science]? What is it that distinguishes it from the separate subjects with which it is related? What is the linking thread which gathers these disparate branches into a single discipline. My answer to these questions is simple – it is the art of programming a computer. It is the art of designing efficient and elegant methods of getting a computer to solve problems, theoretical or practical, small or large, simple or complex. It is the art of translating this design into an effective and accurate computer program.

– C.A.R. Hoare
Graphics as example of inheritance

Sprite: Class of all graphics objects

Rectangle, Circle, Polygon,...: Subclasses of Sprite.

Turtle: Subclass of Polygon.

Sprite holds attributes common to all graphics objects.

▶ Data members: on-screen position coordinates. Orientation. Colour...

▶ Function members: move, right, left

Rectangle class holds

▶ Data members: width and height

▶ Function member: function that actually draws the rectangle on the screen.
Graphics as example of inheritance

**Sprite**: Class of all graphics objects
Graphics as example of inheritance

**Sprite**: Class of all graphics objects

**Rectangle, Circle, Polygon,...**: Subclasses of Sprite.
Graphics as example of inheritance

**Sprite**: Class of all graphics objects

**Rectangle, Circle, Polygon, etc.**: Subclasses of Sprite.

**Turtle**: Subclass of Polygon.
Graphics as example of inheritance

**Sprite**: Class of all graphics objects

**Rectangle, Circle, Polygon, ...**: Subclasses of Sprite.

**Turtle**: Subclass of Polygon.

Sprite holds attributes common to all graphics objects.
Graphics as example of inheritance

**Sprite**: Class of all graphics objects

**Rectangle, Circle, Polygon, ...**: Subclasses of Sprite.

**Turtle**: Subclass of Polygon.

Sprite holds attributes common to all graphics objects.
- Data members: on-screen position coordinates. Orientation. Colour...
Graphics as example of inheritance

**Sprite**: Class of all graphics objects

**Rectangle, Circle, Polygon, ...**: Subclasses of Sprite.

**Turtle**: Subclass of Polygon.

Sprite holds attributes common to all graphics objects.
- Data members: on-screen position coordinates. Orientation. Colour...
- Function members: move, right, left
Graphics as example of inheritance

**Sprite:** Class of all graphics objects

**Rectangle, Circle, Polygon, ...:** Subclasses of Sprite.

**Turtle:** Subclass of Polygon.

Sprite holds attributes common to all graphics objects.

- Data members: on-screen position coordinates, Orientation, Colour...
- Function members: move, right, left

Rectangle class holds
**Graphics as example of inheritance**

*Sprite*: Class of all graphics objects

*Rectangle, Circle, Polygon,*...*: Subclasses of Sprite.

*Turtle*: Subclass of Polygon.

Sprite holds attributes common to all graphics objects.
- Data members: on-screen position coordinates. Orientation. Colour...
- Function members: move, right, left

Rectangle class holds
- Data members: width and height
Graphics as example of inheritance

**Sprite**: Class of all graphics objects

**Rectangle, Circle, Polygon,**...: Subclasses of Sprite.

**Turtle**: Subclass of Polygon.

Sprite holds attributes common to all graphics objects.
- Data members: on-screen position coordinates. Orientation. Colour...
- Function members: move, right, left

Rectangle class holds
- Data members: width and height
- Function member: function that actually draws the rectangle on the screen.
Other uses of graphics


Visualizing output of programs. Fitting line to points. Gravitational simulation. Airport simulation.

Visualization is important also because it can help discover programming errors more quickly.
Other uses of graphics

Other uses of graphics

Other uses of graphics

- Visualization is important also because it can help discover programming errors more quickly.
Some traditional issues in introductory programming

- Program design vs. algorithm design
- How much software engineering should you teach?
- How much methodology/OOP?
- How much history of computing?
- How much hardware?
- How much of STL?
Some traditional issues in introductory programming

- Program design vs. algorithm design
- How much software engineering should you teach?
- How much methodology/OOP?
- How much history of computing?
- How much hardware?
- How much of STL?
Program design vs. Algorithm design

Program design: Writing a program when you know how to solve the problem using paper and pencil.

Algorithm design: Deciding what to calculate to solve a problem.

Main focus: Program design

However, Recursion is covered well.

Many good algorithms are described:
- Root finding, curve fitting
- Sorting including mergesort
- Solving linear systems of equations
- Gravitational simulation
- Backtracking, e.g. 8 queens.
- A few advanced algorithms to pique the interest for projects, further study.

Coding some of these is left for exercises.
Program design vs. Algorithm design

Program design: Writing a program when you know how to solve the problem using paper and pencil.
Program design vs. Algorithm design

Program design: Writing a program when you know how to solve the problem using paper and pencil.
Algorithm design: Deciding what to calculate to solve a problem.

Main focus: Program design
However, Recursion is covered well.
Many good algorithms are described:
- Root finding, curve fitting
- Sorting including mergesort
- Solving linear systems of equations
- Gravitational simulation
- Backtracking, e.g. 8 queens.
- A few advanced algorithms to pique the interest for projects, further study.

Coding some of these is left for exercises.
Program design vs. Algorithm design

**Program design:** Writing a program when you know how to solve the problem using paper and pencil.

**Algorithm design:** Deciding what to calculate to solve a problem.

Main focus: Program design

- Many good algorithms are described
  - Root finding, curve fitting
  - Sorting including mergesort
  - Solving linear systems of equations
  - Gravitational simulation
  - Backtracking, e.g. 8 queens.
  - A few advanced algorithms to pique the interest for projects, further study.

Coding some of these is left for exercises.
Program design vs. Algorithm design

Program design: Writing a program when you know how to solve the problem using paper and pencil.
Algorithm design: Deciding what to calculate to solve a problem.

Main focus: Program design

However, Recursion is covered well.
Program design vs. Algorithm design

Program design: Writing a program when you know how to solve the problem using paper and pencil.
Algorithm design: Deciding what to calculate to solve a problem.

Main focus: Program design

However, Recursion is covered well.

Many good algorithms are described
Program design vs. Algorithm design

Program design: Writing a program when you know how to solve the problem using paper and pencil.
Algorithm design: Deciding what to calculate to solve a problem.

Main focus: Program design

However, Recursion is covered well.

Many good algorithms are described
  - Root finding, curve fitting
Program design vs. Algorithm design

Program design: Writing a program when you know how to solve the problem using paper and pencil.
Algorithm design: Deciding what to calculate to solve a problem.

Main focus: Program design

However, Recursion is covered well.

Many good algorithms are described

▶ Root finding, curve fitting
▶ Sorting including mergesort
Program design vs. Algorithm design

**Program design**: Writing a program when you know how to solve the problem using paper and pencil.

**Algorithm design**: Deciding what to calculate to solve a problem.

Main focus: Program design

However, **Recursion** is covered well.

Many good algorithms are described

- Root finding, curve fitting
- Sorting including mergesort
- Solving linear systems of equations

Coding some of these is left for exercises.
Program design vs. Algorithm design

Program design: Writing a program when you know how to solve the problem using paper and pencil.
Algorithm design: Deciding what to calculate to solve a problem.

Main focus: Program design

However, Recursion is covered well.

Many good algorithms are described
- Root finding, curve fitting
- Sorting including mergesort
- Solving linear systems of equations
- Gravitational simulation
Program design vs. Algorithm design

Program design: Writing a program when you know how to solve the problem using paper and pencil.
Algorithm design: Deciding what to calculate to solve a problem.

Main focus: Program design

However, Recursion is covered well.

Many good algorithms are described
- Root finding, curve fitting
- Sorting including mergesort
- Solving linear systems of equations
- Gravitational simulation
- Backtracking, e.g. 8 queens.
Program design vs. Algorithm design

Program design: Writing a program when you know how to solve the problem using paper and pencil.
Algorithm design: Deciding what to calculate to solve a problem.

Main focus: Program design

However, Recursion is covered well.

Many good algorithms are described
- Root finding, curve fitting
- Sorting including mergesort
- Solving linear systems of equations
- Gravitational simulation
- Backtracking, e.g. 8 queens.
- A few advanced algorithms to pique the interest for projects, further study.
Program design vs. Algorithm design

**Program design**: Writing a program when you know how to solve the problem using paper and pencil.

**Algorithm design**: Deciding what to calculate to solve a problem.

Main focus: Program design

However, **Recursion** is covered well.

Many good algorithms are described

- Root finding, curve fitting
- Sorting including mergesort
- Solving linear systems of equations
- Gravitational simulation
- Backtracking, e.g. 8 queens.
- A few advanced algorithms to pique the interest for projects, further study.

Coding some of these is left for exercises.
Some introductory programming courses teach OOP on day 1. My strategy: Pose progressively harder programming problems to the student, and teach whatever is needed to solve those. Need for OOP can be made to evolve. The book has substantial discussion, but not at the beginning. Chapter on "inheritance based design".

▶ A substantial application developed earlier without inheritance is rewritten with inheritance.
▶ New implementation is easier to extend. A non-trivial extension is carried out.

Perfectly fine to keep OOP in second course.
Programming methodologies

Some introductory programming courses teach OOP on day 1.
Programming methodologies

Some introductory programming courses teach OOP on day 1.

My strategy: Pose progressively harder programming problems to the student, and teach whatever is needed to solve those.
Programming methodologies

Some introductory programming courses teach OOP on day 1.

My strategy: Pose progressively harder programming problems to the student, and teach whatever is needed to solve those.

Need for OOP can be made to evolve. The book has substantial discussion, but not at the beginning.
Programming methodologies

Some introductory programming courses teach OOP on day 1.

My strategy: Pose progressively harder programming problems to the student, and teach whatever is needed to solve those.

Need for OOP can be made to evolve. The book has substantial discussion, but not at the beginning.

Chapter on “inheritance based design”.
Programming methodologies

Some introductory programming courses teach OOP on day 1.

My strategy: Pose progressively harder programming problems to the student, and teach whatever is needed to solve those.

Need for OOP can be made to evolve. The book has substantial discussion, but not at the beginning.

Chapter on “inheritance based design”.
- A substantial application developed earlier without inheritance is rewritten with inheritance.
Some introductory programming courses teach OOP on day 1.

My strategy: Pose progressively harder programming problems to the student, and teach whatever is needed to solve those.

Need for OOP can be made to evolve. The book has substantial discussion, but not at the beginning.

Chapter on “inheritance based design”.

- A substantial application developed earlier without inheritance is rewritten with inheritance.
- New implementation is easier to extend. A non-trivial extension is carried out.
Some introductory programming courses teach OOP on day 1.

My strategy: Pose progressively harder programming problems to the student, and teach whatever is needed to solve those.

Need for OOP can be made to evolve. The book has substantial discussion, but not at the beginning.

Chapter on “inheritance based design”.

- A substantial application developed earlier without inheritance is rewritten with inheritance.
- New implementation is easier to extend. A non-trivial extension is carried out.

Perfectly fine to keep OOP in second course.
Software engineering

[9x251]Software engineering

Writing specs: Must be stressed. The book has some discussion. Specs for most non-trivial programs have been carefully developed. Invariants/program proving: Substantial discussion. Necessary because students usually try a few cases and declare their program to be correct. Invariants are useful even for explaining algorithms. Testing: Some discussion of how to generate test cases. Also some discussion of code coverage.
Writing specs: Must be stressed. The book has some discussion. Specs for most non-trivial programs have been carefully developed.
Writing specs: Must be stressed. The book has some discussion. Specs for most non-trivial programs have been carefully developed.

Invariants/program proving: Substantial discussion. Necessary because students usually try a few cases and declare their program to be correct.
Writing specs: Must be stressed. The book has some discussion. Specs for most non-trivial programs have been carefully developed.

Invariants/program proving: Substantial discussion. Necessary because students usually try a few cases and declare their program to be correct.

Invariants are useful even for explaining algorithms.
Software engineering

**Writing specs:** Must be stressed. The book has some discussion. Specs for most non-trivial programs have been carefully developed.

**Invariants/program proving:** Substantial discussion. Necessary because students usually try a few cases and declare their program to be correct.

**Invariants are useful even for explaining algorithms.**

**Testing:** Some discussion of how to generate test cases. Also some discussion of code coverage.
Libraries, especially STL

String, Vector, Map classes covered in depth.
A baby version of the String class is developed to illustrate the idea of class design + memory management.
The priority queue class is used in the chapter on simulation.
Libraries, especially STL

String, Vector, Map classes covered in depth.
Libraries, especially STL

String, Vector, Map classes covered in depth.

A baby version of String class is developed to illustrate idea of class design + memory management.
Libraries, especially STL

String, Vector, Map classes covered in depth.

A baby version of String class is developed to illustrate idea of class design + memory management.

Priority queue class is used in chapter on simulation.
Unusual/Advanced topics

- Lambda expressions (C++ 11)
- Memory management, including reference counting
- Simulation. Gravitational simulation, airport simulation. Dijkstra’s algorithm as a simulation. A general framework is developed.