

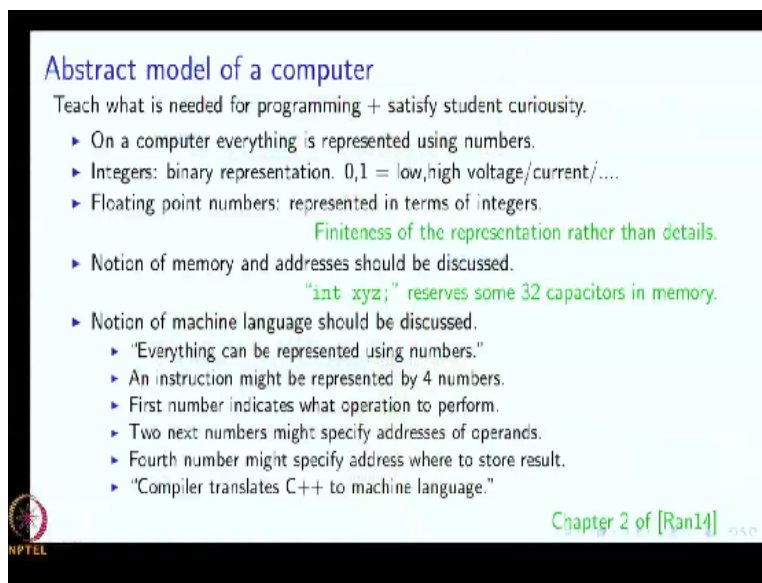
Design and Pedagogy of The introductory Programming Course
Prof. Abhiram G. Ranade
Department of Compute Science and Engineering
Indian Institute of Technology - Bombay

Lecture – 10

Basic Ideas in Our Approach. 5: Remarks on Individual Topics -1

Hello and welcome again. In the last lecture, we saw a list of proposed cores of terms for the course that we are designing. In this lecture, we are going to see some more detail. So when we say students need to understand the abstract model of a computer, we are going to see what exactly do they need to understand. We are not really going to talk so much about how exactly it is to be taught. Our focus in this is going to be what it is that they need to know? How to teach, we will talk about a little bit later.

(Refer Slide Time: 00:59)



Abstract model of a computer

Teach what is needed for programming + satisfy student curiosity.

- ▶ On a computer everything is represented using numbers.
- ▶ Integers: binary representation. 0,1 = low,high voltage/current/....
- ▶ Floating point numbers: represented in terms of integers.
Finiteness of the representation rather than details.
- ▶ Notion of memory and addresses should be discussed.
"int xyz;" reserves some 32 capacitors in memory.
- ▶ Notion of machine language should be discussed.
 - ▶ "Everything can be represented using numbers."
 - ▶ An instruction might be represented by 4 numbers.
 - ▶ First number indicates what operation to perform.
 - ▶ Two next numbers might specify addresses of operands.
 - ▶ Fourth number might specify address where to store result.
 - ▶ "Compiler translates C++ to machine language."

NPTEL Chapter 2 of [Ran14]

Okay, so the first topic is Abstract model of a computer. In one line we should teach what is needed for programming plus what is needed to satisfy student curiosity? So one thing that is needed for programming is that on a computer, everything is represented using numbers. So our student should understand that. So for example, integers are represented in binary and 0 is represented by a low voltage, 1 is represented by the high voltage.

So our students are not going to do anything directly with this but this is something that satisfies their curiosity and will see it plays an additional role, okay. So floating point numbers, they are

going to be represented in terms of integers and these 2, these 2 bullets essentially are same that the representation on a computer is finite. So when I talk about an integer, I usually mean a 32 bit integer or something, something like that.

So the natural way is that representation is finite. If I talk about a floating point number, then it means that we are going to have finite of precision or the exponent is going to be finite or something like that. Then student should know what memory is? Okay, and what addresses are? For example, if I write down `int xyz`, it would be nice if students have a mental image of what happens on a computer.

So you could say or you should be in a position to say that a statement like `int xyz` reserves sum 32 capacitors in memory. This is a very crisp description. Okay, so this, there is no ambiguity about this, okay. So therefore, it is desirable and remember that our students are science and engineering students predominantly who have completed 10+2. So for those students, certainly an engineering level description, a description which tells, gives detail about what actually happens is desirable.

Now the idea that everything is represented using numbers, has to be driven more and one of the ways in which it can be driven home is by talking about machine language. So everything can be represented using numbers. An instruction might be represented by 4 numbers. A first number indicates what operation to perform. Next numbers might indicate addresses or where in memory those numbers reside and the fourth number might specify where the result goes.

So the point is that we are not going to tell them that the first number if it is 70, it means addition or whatever, that is not the point. The point is that student should get a general idea of how a computer works, that data lives in memory. There is something which fetches data, which performs operations on it and is put back into memory, and the result is put back into memory. And a compiler is going to translate whatever C++ code we write to machine language.

So 2 points are important over here. The first point is that there is this step. So we are, when we are going to teach, talk about compilation, then students will know at least a little bit about what

is going to happen. So this is what is going to happen. We are going to translate our high level C++ program into a sequence of numbers which the machine can, so to say, understand. And this is discussed in chapter 2 of our textbook.

There is one more point I want to mention over here which is that if I describe everything as happening in terms of numbers, there is a certain kind of precision that I am hinting at, okay. So it says that everything has to be very crisply defined. If you want a multiplication, you have to say a multiplication. The computer is not like a human being which will guess what is in your mind.

So the notion that there is machine language that you have to write down these codes essentially is also going to nudge the student into understanding that the computer is, is a very precise beast. It does not try to help you out. It does what it is told and it has to be told something very very precisely. The next thing that I want to talk about is, how do we teach the language?

(Refer Slide Time: 06:41)

The slide is titled "Teaching the Language" in blue. It contains several lines of text in black and green. The text includes references to [Gri74] and [SEBJ82], discusses student difficulties with loops and expression syntax, and provides advice on teaching operators and switch statements. It also includes an analogy to spoken language and a note about discussing pedagogy. The NPTEL logo is in the bottom left corner, and navigation icons are in the bottom right.

Teaching the Language

[Gri74] already suggests teaching minimal language elements.

[SEBJ82] say that students have difficulties with loops

Students have difficulty with expression syntax of ++, +=

Discourage these operators, at least expression syntax

Avoid asking: What is the effect of i++++j?

May forget break in switch statement

Discourage switch statement

Analogy to spoken language:
Good writing uses simple words and simple grammar.

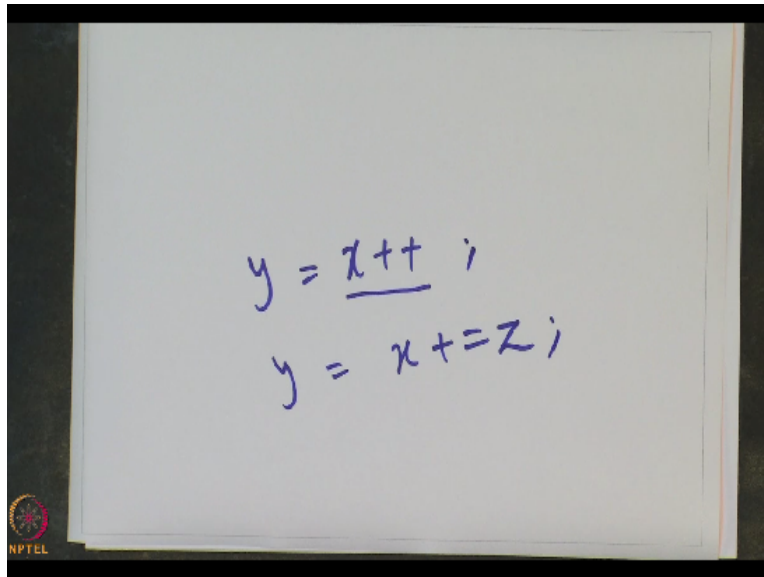
More when we discuss pedagogy

NPTEL

Now David Giles, David Grice, I suppose, in 1974 already suggested that we should be teaching a very minimal language. Already even then I think people who are seeing that there were very complicated language elements, okay. Now this is important in light of something which we have seen earlier that students have difficulties with loops.

So it is not that students understand language very easily and given that we say that we have a crisis in education. I think it is important that we stick to easy language constructs so long as we have adequate language constructs. So for example, students have difficulty with the expression syntax of ++. What do I mean by that?

(Refer Slide Time: 07:31)



So suppose I write `x++`. So you would think that this means increment value of `x`. it does mean that but I can also assign this as `y`. So this is going to produce a change in `x` but that expression itself is going to have a value and that value is going to be placed in `y`. So in this case and this is rather tricky in my opinion that the original value of `x` gets placed over here and then the value is incremented, okay.

Now this expression syntax might have been useful long ago when compilers were not that good. Today, the requirements have changed. We have good compilers. So it is much better to use simple expression, okay, so simple, simple language, so that nobody has to think twice when they read a sentence like `y=x++`, okay. `+=` is similar, `+=` also has an expression syntax. So I can write something like `y=x+=z`, okay.

Now there is this expression syntax. It is written in books. It is written in the textbook that we are vaguely following but my recommendation is that these kinds of expressions should not really be used by students. Why? Because they are very tricky, okay. You should use simple language,

language which anybody is reading can understand very easily. If you read that, you read what you write, later on you should not have any difficulty.

You should not have to scratch your head and say look, what is this complicated stuff that I wrote myself? So at least I would say discourage even `x++`, okay. But at least discourage the expression syntax because I have seen students go wrong with `x++`. `x++` students think of as viewing `x+1`. So they make a change but the value that they get is the old value. So it is a complicated operator.

And it is more, more importantly it is inconsistent with the mathematical knowledge that we have and 50 years ago when compilers were weak, these operators might have been very useful but today compilers are very powerful and therefore, these operators are not needed at all. You should be using very simple programming language, the programming constructs. This means conversely that we should not think of these operators as good opportunities for asking tricky questions.

So for example, here is a tricky question that I have seen people ask. What is the effect of `i+++++j`? This actually was a question on some examination, would you believe it? I think it is a terrible question because it says that knowing all these things, knowing complex expressions is important. No, it is not. It is much more important to write programs which anybody can understand at a glance.

Writing the complex expression like this is serving no one. So when you say, when you want to emphasize, emphasize a point, your examination should be very consistent with whatever you are emphasizing. Now there are some language constructs like the switch statement in which after each case, it is expected that there will be a break. Now this statement has been criticised because many people will forget to put a break.

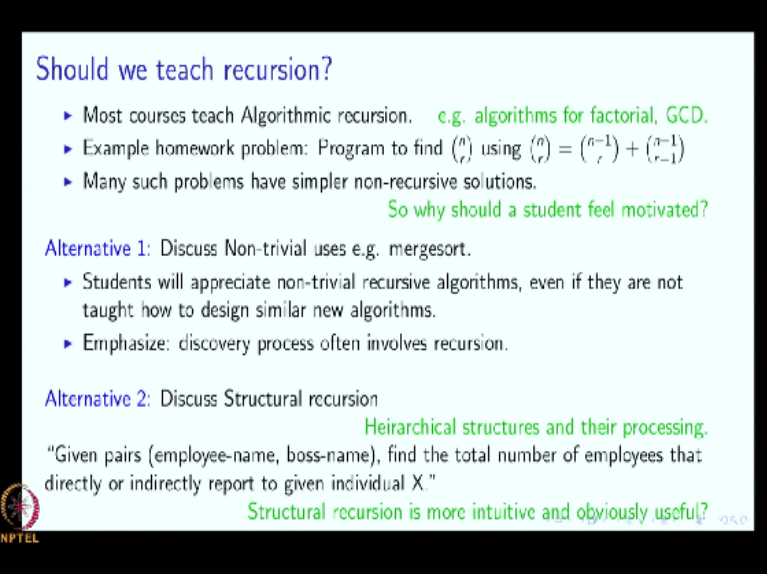
So therefore, I feel that we should discourage the switch statement. Does not matter really if you write it as the sequence of if and else, okay. If, sequence of with `()` (11:54) several then clauses but there at least you are not in danger that oh, I just forgot something and now my program is

wrong and therefore, who knows what catastrophe it might cause. In all this, there is an analogy to spoken language, okay.

Good writing or good speaking for that matter uses simple words and simple grammar. If you are fond of using complex words and complex long sentences, you may think that you are very talented but many people may just think that oh, that guy is just pompous or more importantly people will have difficulty in understanding you. So in a similar manner, write your programs using simple constructs.

Compilers are very good today and they will take care and they will generate the right kind of machine language code, even if you do not specify that I want $x=x+1$ by writing the `++`, okay. So, and you will have more to say about this when we discuss pedagogy but definitely when we talk about language, this is what we have in mind. The focus should be on the simple elements and not, and even we should discourage complex elements. Now should we teach recursion? There are 2 schools of thought, okay.

(Refer Slide Time: 13:41)



Should we teach recursion?

- ▶ Most courses teach Algorithmic recursion. e.g. algorithms for factorial, GCD.
- ▶ Example homework problem: Program to find $\binom{n}{r}$ using $\binom{n}{r} = \binom{n-1}{r} + \binom{n-1}{r-1}$
- ▶ Many such problems have simpler non-recursive solutions.
So why should a student feel motivated?

Alternative 1: Discuss Non-trivial uses e.g. mergesort.

- ▶ Students will appreciate non-trivial recursive algorithms, even if they are not taught how to design similar new algorithms.
- ▶ Emphasize: discovery process often involves recursion.

Alternative 2: Discuss Structural recursion

Heirarchical structures and their processing.

"Given pairs (employee-name, boss-name), find the total number of employees that directly or indirectly report to given individual X."

Structural recursion is more intuitive and obviously useful?

NPTEL

And I think most courses will teach algorithmic recursion and I, meaning I put this recursion in Tier 2 core but I suspect that it is probably a core topic by now. 50 years ago, or you can do a lot without recursion but really, it is becoming a core topic. So as I said if the topics in amongst what I have listed as Tier 2 core, this is probably the most compulsory. This really is something

that you should consider putting in core.

But there is one more reason why I put it in Tier 2 core because unless you teach it well, unless you teach it adequately, students will not, will just learn it superficially and will not really get much out of it. Okay, so how do we teach it? As I said, most courses teach algorithmic recursion. What does it mean? Algorithms were factorial. The recursive algorithms were factorial, GCD, things like that.

The example homework problems might be program to find n choose r , using the identity n choose $r = n-1$ choose $r+n-1$ choose $r-1$, okay. Now the difficulty in this way of teaching is that many such problems have simpler non-recursive solutions. So then the student might naturally ask if that is the case, why should I even bother to know about these things. I mean, I can do n choose r by knowing the formula, right.

So I have a formula I can just write down that formula, why should I have to calculate in this manner. So how do we, how do we motivate the student? Well, there are 2 alternatives. One alternative is to discuss non-trivial uses, for example, mergesort. A mergesort is very tricky to implement without recursion. So you should discuss it and that would be a good reason to say to students look here is why you should be learning recursion.

Now students will appreciate non-trivial recursive algorithms if they are clever even if they are not taught how to design similar new algorithms. Because designing something like mergesort, designing another algorithm which is similar to mergesort is not what we are going to cover in the course, but that is okay. So we have, we have covered recursion and we have now said that look here is another, here is an algorithm, mergesort which uses this recursion.

So students will be very happy to see clever algorithms. Now you can also emphasize that recursion is not just a program expression strategy but it is also an aide to discovery. So, so just staying with that for a second. You could probably argue that you could discover his GCD algorithm probably by thinking recursive. So he says, he perhaps or he very rightly said to himself, oh, I need to find the GCD of 5718 and 19261.

Well, can I find smaller numbers whose GCD is the same or somehow smaller numbers whose GCD would help. So that aspect of recursion should perhaps be stressed. So GCD is a very good, good problem on which to develop this C. Another way to motivate the need for recursion is to discuss structural recursion, okay. What does it mean? That very often we see structures which are hierarchical.

So for example, family trees and we need to process them. So here is a typical problem. Say given pairs of employee, name and boss name, find the total number of employees that directly or indirectly report to a given individual X. So here you, that indirect reporting, the number of people who report indirectly would naturally have to be counted using recursion and the point here is that discovering this, is fairly straightforward.

Because we have said, it is indirect reporting, students will know that look, I am going to look at the employee, the people who report to X and then count how many people report to the people who report to x and so on. So, so this recursion in a way is more obvious, okay. And it is also very more common. So if you want to teach recursion in my opinion, it cannot be just an algorithm for factorial.

That is something that students will learn for may be a week and forget and they will not really appreciate the importance of recursion. So I am going to suggest that when you say you want to teach recursion, you should make a little bit of a bigger story and you should teach that bigger story because only if you make a bigger story, will students actually understand the power and will remember things, okay. So structural recursion is more intuitive and useful which is what I said earlier.

(Refer Slide Time: 19:48)

Pointers

Guiding principles:

- ▶ Fundamental concept, must be taught
- ▶ Students should understand relationship to arrays
- ▶ Students should understand pointer calculation and `[]` as an operator on pointers.

Why indexing into an array happens in constant time.

- ▶ **Avoid explicit pointer arithmetic**
 - ▶ Use `A[i]` rather than `*(A+i)`
 - ▶ Pass entire array rather than starting from an offset Error prone
- ▶ **Minimize use of pointers**
 - ▶ Use standard library `vector` or `list` rather than build your own.
 - ▶ Avoid null terminated char arrays, instead use `string` class.

Vastly safer and more elegant. Also powerful
Possible in C++, but not in C.

NPTEL

Alright, Pointers. Do we need to teach pointers. So here are the guiding principles. Pointers are a fundamental concept and so they must be taught. Student should understand the relationships to arrays. So in C++ or in C, the name of an array happens to be a, happens to be a pointer, happens to be of type pointer. It is a constant pointer or it is an address but they should understand that, okay. They should understand that pointer arithmetic and they should understand the indexing operator with the indexing operation as an operator on pointers.

(Refer Slide Time: 20:44)

The image shows a whiteboard with handwritten text in blue ink. The top line reads `x[i]` followed by an arrow pointing to the left. The bottom line reads `x [] i`. This illustrates the concept of array indexing as an operation between the array name and the index.

NPTEL

So one way of saying this is they should realize that in a programming language when I write `x` of `i`, this really is an operation that happens between `x` and `i`. So I might as well have written, might as well written, have written this as `x operator i` but because of tradition, this form is

preferred, okay. So this is necessary because it is sort of, sort of says what is sort of the philosophy of program, okay.

So we do not want to make anything into a special case. Indexing is not really a special case but it is an operator just as $x+y$ is result of an operator, x index i is also an operator. It is a complicated operator and student should understand exactly what it does that the value of i is added to the address x appropriately, okay. So all this is needed because students need to understand why indexing into an array happens in constant time. So they need to understand there is pointer arithmetic or address arithmetic, okay.

And that is how array indexing happens and that is how most importantly even if my index i is very large, I can get to that i th element of the array in constant amount of time, in, just the time needed to that addition, okay. However, pointer arithmetic should be avoided explicitly. So for example, students should use A of i rather than write something which is equivalent but quite ugly, $*A+I$, right.

So pointer arithmetic is to be avoided but you should know it. It is kind of like have you learn tables in, multiplication tables in school. As we grow older, we do not use multiplication tables, we use calculators. Why? Because most of us will probably make a mistake if we do long multiplications all the time by hand. We will, to eliminate mistakes, we will spend too much time.

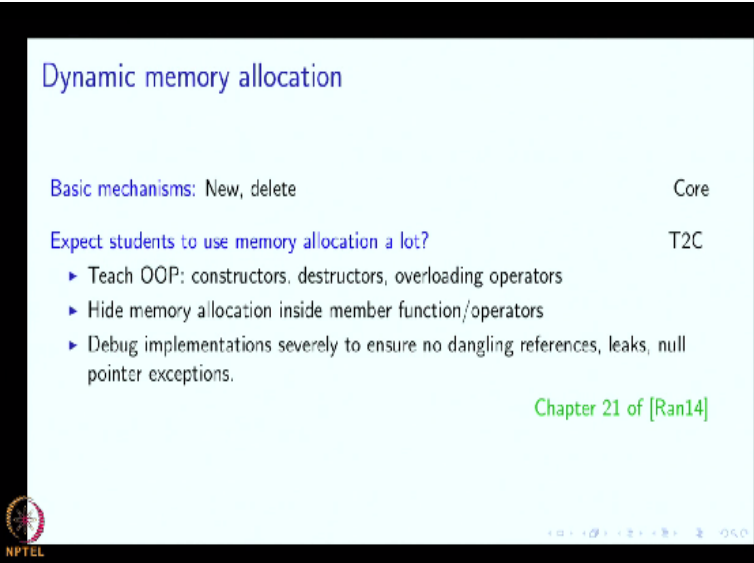
On the other hand, we should know, we should understand what multiplication is. It is exactly the same principle. So we should understand what exactly is going on when we write A of i . So for that purpose, we should have, we should understand pointers, okay. Then you can use pointers to pass arrays starting from an offset. But I would recommend not doing that. Because it could lead to errors and I could, I would say that 10 students do not do this as much as possible, okay.

Well it is up to you of course. So in general, minimize use of pointers because they are a very low level idea but you should know about them. So use standard library vector or list rather than

build your own. So in C, you can, you have to build your own data structures. In C++, do not build your own data structures if the standard library suffices. Avoid null terminated character arrays which are passed by pointers, okay and in C++, use the string class.

The string class can be used without talking about pointers. Let me put it this way. Pointers are really dangerous in the sense that they can lead to all kinds of errors like dangling pointers, okay, like memory leaks, okay. So we should be, we should be avoiding them. They are like live electrical wires. We, and we should understand what live electrical wires are but does not mean we should touch them all the time or we should even have them in our hands all the time. So the standard library and C++ provides strings which are vastly safer, so use those.

(Refer Slide Time: 25:11)



The slide is titled "Dynamic memory allocation" in blue text. It lists "Basic mechanisms: New, delete" as "Core". It then asks "Expect students to use memory allocation a lot?" and lists "T2C" as the answer. Below this, there are three bullet points: "Teach OOP: constructors, destructors, overloading operators", "Hide memory allocation inside member function/operators", and "Debug implementations severely to ensure no dangling references, leaks, null pointer exceptions." The slide also includes a green footer "Chapter 21 of [Ran14]" and an NPTEL logo in the bottom left corner.

Basic mechanisms: New, delete	Core
Expect students to use memory allocation a lot?	T2C

- ▶ Teach OOP: constructors, destructors, overloading operators
- ▶ Hide memory allocation inside member function/operators
- ▶ Debug implementations severely to ensure no dangling references, leaks, null pointer exceptions.

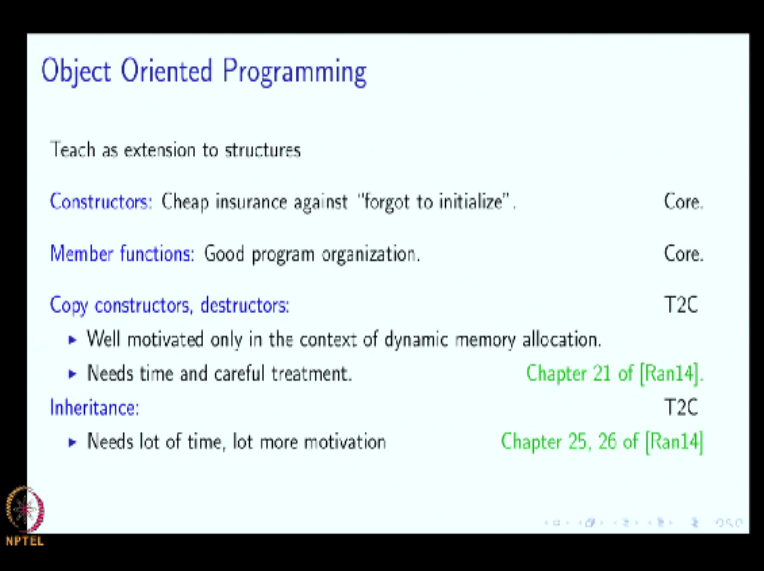
Chapter 21 of [Ran14]

Dynamic memory allocation. The basic mechanisms, new and delete, I think are core, okay. If students are going to use memory allocation a lot, then they should learn more things and those, in my opinion, might not be core but, but they might be Tier 2 core. So Objective Oriented Programming should be taught and constructors, destructors, overloading operators should be used if, to hide damage correction, to hide memory allocation under the hood, okay.

So, inside member functions or operators, okay. And again you should stress, you should tell the students that look you know how to allocate memory and now you know how to do it safely, use object oriented programming or rather use constructors, destructors and things like that and we

will come to this in greater detail and debug implementations so that in day to day programming, you do not have to worry about pointers or memory, memory allocation, okay. So this is discussed in chapter 21 quite in detail.

(Refer Slide Time: 26:24)



The slide is titled "Object Oriented Programming" and contains a table of topics and their core status. The table is as follows:

Topic	Core Status
Teach as extension to structures	
Constructors: Cheap insurance against "forgot to initialize".	Core.
Member functions: Good program organization.	Core.
Copy constructors, destructors:	T2C
▶ Well motivated only in the context of dynamic memory allocation.	
▶ Needs time and careful treatment.	Chapter 21 of [Ran14].
Inheritance:	T2C
▶ Needs lot of time, lot more motivation	Chapter 25, 26 of [Ran14]

The slide also features the NPTEL logo in the bottom left corner and navigation icons in the bottom right corner.

Object Oriented Programming. So it should be taught as an extension to structures. I think constructors can be thought of as core, okay. Constructors are cheap insurance against this problem of I forgot to initialize something, okay. Member functions are good for program organization and in some sense, these could be core. These are also fairly simple. Copy constructors and destructors are definitely not core, okay.

So unless you are going to build objects or, or software components as you might call them, you should not teach them and you should teach them only as Tier 2 core, okay. So they are well motivated only in the context of dynamic memory allocation, mostly, mostly well motivated. And they need time and careful treatment. So that is given in chapter 21 but this material is not core. Inheritance, definitely is Tier 2 core. Needs a lot of time and a lot more motivation which is provided but not core.

(Refer Slide Time: 27:37)

Teaching specific algorithms

Some clever algorithms must be taught for creating excitement! Core

- ▶ Euclid's GCD
- ▶ Newton-Raphson. Babylonians used this for square roots 3500 years ago!
- ▶ Binary search
- ▶ Calculate math functions using series...

A Generic algorithm:

"Try all possibilities" for constraint satisfaction problems T2C

- ▶ "Find x_1, \dots, x_n satisfying specified constraints"
- ▶ By using recursion we can go over all plausible assignments of values to x_1, \dots, x_n can take as a group, and check whether a given assignment satisfies all constraints. Somewhat involved use of recursion.
- ▶ Once the algorithm is understood in general, it can be used for many constraint satisfaction problems, e.g. 8 queens, cryptarithms.

NPTEL

Teaching specific algorithms. Clever algorithms must be created, must be taught for creating excitement. So mergesort is a good idea. Euclid's GCD. Newton-Raphson. Newton-Raphson's method for finding the roots of an equation is actually fairly simple. There is a nice geometric view of it which everybody understands at one glance, okay. But another important point about Newton-Raphson algorithm is, it is history.

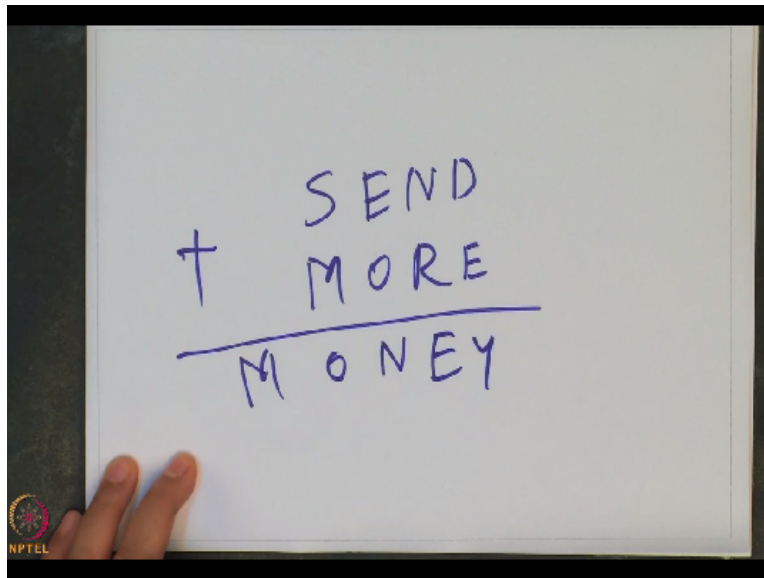
Babylonians actually invented it 3500 years ago. Can you believe it for computing square roots? So it is actually a simple but yet very elegant algorithm and it is shocking that somebody could invent it and, and that somebody could invent it, invent it 3500 years ago. So you could in fact argue that history of computing is really the history of human kind. Certainly the square root algorithm is extremely important in the history of computing.

Binary search is very elegant, so it is core. You should show it somewhere either in the problems or either have some excuse to teach it. Calculating math functions using series is useful. People are surprised that in finding out how to compute sines, cosines and things like that. Now we have also talked about try all possibilities for constraint satisfaction problems. It is a generic algorithm and you could teach it.

But I would say it is Tier 2 core. So the general algorithm is find x_1 to x_n satisfying specified constraints and by going, by doing a recursion, we can go over all possible, possible assignments

and then printout which assignments satisfies all the constraints. Okay, once the algorithm is understood by the students, it can be used for many constraint satisfaction problems including 8 queens problems which we talked about and also cryptarithms. Cryptarithms are problems where you are given something like send more money and send, you have to replace the digits in send and more with numbers so that the equation makes sense.

(Refer Slide Time: 30:07)



So something like this. So every letter must be replaced by the same digit and then this addition should make sense, okay. So these kinds of problems can be solved if you know how to, if you know that generic algorithm that we have talked about. Okay, so we will stop this lecture over here. We are coming to the important topic about what exactly do we want to teach when we talk about writing programs and we will take that in the next lecture. Thank you.