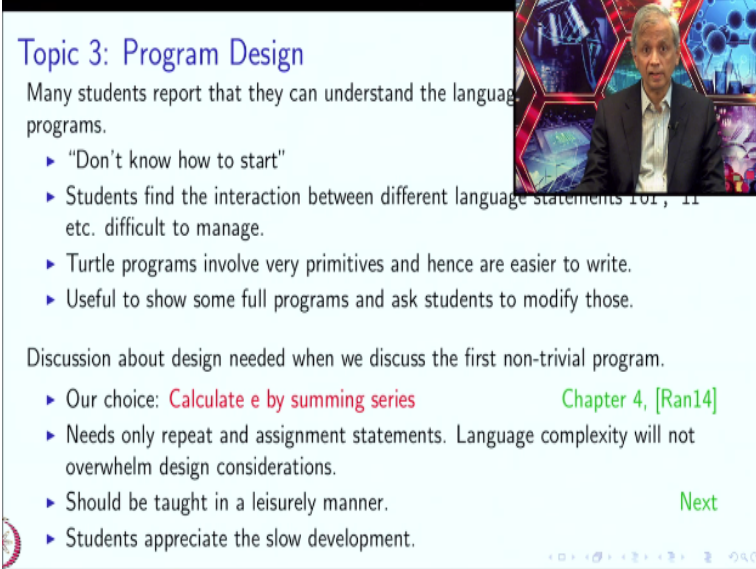


Design and Pedagogy of The Introductory Programming Course
Prof. Abhiram G. Ranade
Department of Computer Science and Engineering
Indian Institute of Technology – Bombay

Lecture - 16
Pedagogy.4: Tour – 3, Conclusion

Welcome back to this sub-lecture in the sequence of lectures on Pedagogy.

(Refer Slide Time: 00:26)



Topic 3: Program Design

Many students report that they can understand the language programs.

- ▶ "Don't know how to start"
- ▶ Students find the interaction between different language statements for, if etc. difficult to manage.
- ▶ Turtle programs involve very primitives and hence are easier to write.
- ▶ Useful to show some full programs and ask students to modify those.

Discussion about design needed when we discuss the first non-trivial program.

- ▶ Our choice: **Calculate e by summing series** Chapter 4, [Ran14]
- ▶ Needs only repeat and assignment statements. Language complexity will not overwhelm design considerations.
- ▶ Should be taught in a leisurely manner. Next
- ▶ Students appreciate the slow development.

So the topic 3 according to our lesson plan is Program Design. So this topic and discussion of this topic at such an early stage is because of the observation in the literature in the education literature that many students report that they can understand the programming language but somehow have a mental block when it comes to even starting to write a program. They do not know how to start, okay.

Student find the interaction between different language statements say the 'for' statement and the 'if' statement difficult to manage. So if they know many statements they just get more confused, okay. So should I use the 'for' or should I use the 'if', should the 'if' go outside; should the 'for' go outside whatever it is. So knowing too much is also a cause for confusion. Now, when we Turtle programs the primitives are very few.

So you have the turtle movement, turning and maybe not even pen up, pen down, okay and repeat, so there really a two things that we need to worry about, and it is clear that the repeat is going to outside of the turtle movement. I cannot say turn and inside the turn have put a repeat, okay. So it is very clear that the turning will go inside the repeat rather than the other way around. So the number of choices the number of design choices in writing the programs is very small.

And therefore, perhaps at early stage students might get might find it easier to write program so that is the motivation, okay. So while many students complain or it is their problem that we do not know to start, they are just overwhelmed by the language; the way we should look at is okay they are overwhelmed at the language then let us get them to write programs early on when they only a very small number of statements.

So assignment statements are enough but repeat makes it a little bit more interesting and so even that repeat even without assignment statements actually just the turtle primitives, we find that students are at completely at home. So the Turtle toning programs, programs for drawing pictures using the turtle are the easiest program for the children to write, that has been my experience.

Okay, so as far as program design is concerned it is useful to show some full programs to students and ask them to modify those as well. But again, everything goes nicely if the, if in the beginning we only use a small number of constructs. So the moment we talk about the first non-trivial program we should do the discussion about program design, okay. So our choice regarding this is calculation of e by summing the series.

So this can be written using just repeats and assignments and therefore it is a natural point at which to do this discussion okay so it is done in Chapter 4, and it is the third topic of the course. So to calculate e the language complexity will not be a problem; a student will not get confused about “Oh, what should I use, I have so many things and which one is right one to use.” No, they only have the repeat and the assignment statements.

But nevertheless, this should be taught in a leisurely manner. So we discussed earlier how to teach summing of the series for e , we discussed it at length, so remember what we discussed. We said first figure out what you do by hand; figure out the structure, okay so is there loop—what do you thing happen inside the loop, do all that exercise, okay. So do that exercise in a leisurely manner and then write the program, okay.

So students actually appreciate the slow development. This is something that I have asked student, students have read my book as well as students in my class room. Do you want me to go over this slowly or should I just omit it or should I just go very fast? So bright student, as well as not so bright students have told me, “Please go over it slowly. This discussion helps us because it tells us what a full program is.”

A full program is sort of a different beast as compare to 10 individual statements as well. A 3-line full program is different from understanding three statements in isolation. So please go over this slowly and that would be my suggestion to you as well.

(Refer Slide Time: 05:58)

Program Design (continued)

Our recipe for program design

- ▶ Understand problem: create input output examples.
- ▶ Understand how you would solve the problem manually
 - ▶ State what **values** you calculate in each iteration.

In i th iteration i th term t_i of series is calculated...

- ▶ Make your program mirror the manual computation.

Statement of what happens in iteration i : **Plan**
Stating plans is very important for debugging.

- ▶ Identify what variables you will need.

A variable to hold each value that needs to be remembered.

Emphasize the distinction between what to compute and how to compute it.

What: Plan. States what values will be calculated.

How: Code that does the computation. Comes later.

Homeworks: Programs to (approximately) evaluate other infinite sums and products.



So what is our recipe for program design, which we discussed last week, understand the problem: creating input output examples, understand how you solve the problem manually, okay state what values you calculate in each iteration, so in the i th iteration, i th term, t_i of the series is

calculated, okay. So all these you have to do, we have discussed it last week, and accordingly write the program, okay. So discuss it slowly in the classroom, okay.

Encourage students to write down the plan. What happens in i th iteration? t_i , it was one over i factor is calculated or $t_i =$ whatever value you have associated with t_i gets calculated. What happens in each variable? State them, tell them that stating plans is important for debugging. Should your program not work correctly? You should be able to figure out where things went wrong and for that writing down the plan is really important, okay.

The way the recipe goes, identify the variables you will need, okay. And one point that we already emphasized at even at the stage is the distinction between what to compute and how to compute it. What to compute is given by the plan. The plan says—well, I should clarify that. The plan will say something like t_i will be calculated. So the plan will talk about what values are going to be calculated, okay in which iteration that is true.

But it will they will name what values are going to be calculated, so that is the what, by what I mean what values. How is the Code? So the students should be aware about both of these things. They should give names to the values, that is what it means to be aware. So what value is calculated in the i th iteration is distinct from where that value stored. I might have the name t for the variable and t_i for the value.

And if t 's are same but again the subscript says that this is a value, so there is a distinction between values and variables. And we talked, we should be clear about are we talking about the value or are we talking about the variable. Now, in this case homeworks are fairly natural. There are other infinite sums and products very nice sums and products. And by the way several of these sums and products have an Indian origin.


There were Indian mathematicians from about 2000 years old as well who have invented or certainly 1000 years' old who have invented very nice formulae for lots of things that we thing today are very modern. So do tell your students that there is an Indian concoction.

(Refer Slide Time: 09:10)

Remarks

Program design issues should continue to get discussed throughout the course.

- ▶ Students should be frequently reminded to design manually before writing programs.
- ▶ Writing down precise plans for all important loops should be expected.
- ▶ Plans should be like invariants, and they should contain all the information required to see that the program is correct.
Should not demand formal proof of correctness.
- ▶ When you find students debugging code in labs, see if they have written a plan. Remind them that this will help in debugging.
- ▶ Likewise for every function it is important to clearly state what it is supposed to do, before writing the function – also useful in debugging.



Okay, so I think programming discussion design issues should get started early on and you should continue to discuss them throughout the course. Continue to emphasize throughout the course that manual algorithm is designed first before writing programs before writing program. Be very clear how you solve the problem manually. Write down plans and variants at least for all the important rules.

And yeah so the plan should be like invariants and they should contain all the information that will help us in deciding whether the program is correct or not, okay. So this take some thinking but at least you can ask the students, is this variable changing, then why is not that change reflected in the plan that you have put in. We cannot demand a formal proof of correctness because that is quite laborious, okay.

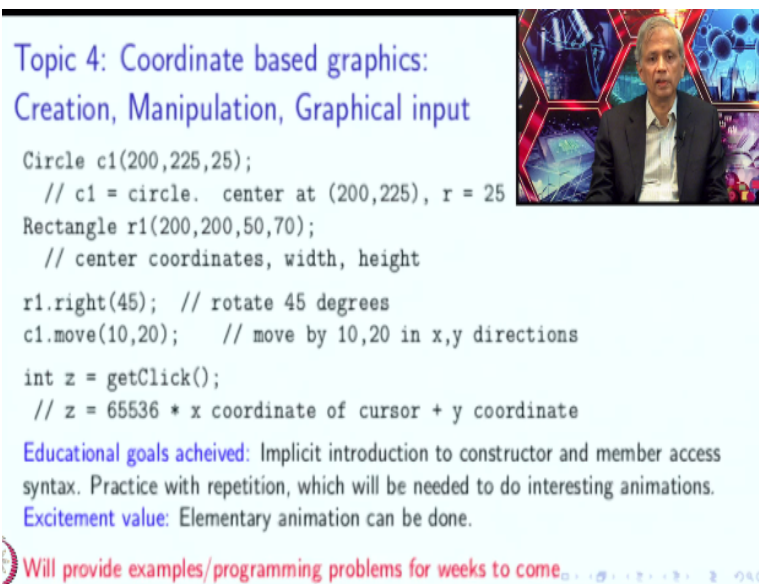
But at least thought that went into the design. The thought must have been something like that in this iteration I will calculate this. Instead of letting the thought remain in the minds of the student in some intuitive nebulous form ask the student, may the student get into the habit of putting that write down on paper. And when you find student debugging code in labs, see if they have written the plan. Remind them that this will help them in debugging, okay.

Now, when you come functions the documentation changes slightly. The plan changes slightly. So here what you should tell them is that before writing the body of the function you should state

very clearly what the function will be written in relation to the inputs the arguments of the function. So not just for the cases you think are going to get used but also for other cases, or at least you could say, that look, in these cases the function is going to be written this.

For these values the function, we do something funny but I do not care about these values, so at least upfront you will be stating that there is something wrong if the function is called with these values. Such statements are going to be very useful in debugging your code.

(Refer Slide Time: 11:48)



```
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
```

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

Excitement value: Elementary animation can be done.

Will provide examples/programming problems for weeks to come.

Once you have given a description of program design we continue on and we continue with one more lecture in graphics because this lecture is going to enable us to do interesting things for the rest of the semester, okay, so this lecture is going to be about creation, manipulation and graphical input. Creation of graphical objects and graphical input. So the first statement over here Circle c1 200, 225, 25 is going to create a circle. Its center is going to be at 200, 225 and its radius is 25.

The second statement is going to be creating a rectangle at center 200, 200; its width and height will be respectively 50 and 70. Now, of course I can use different names and I will get different rectangles. So if I want to create 10 rectangles I can use 10 such statements, okay. Now, introduce these statements as special statements for creating rectangles right now. But if you are familiar with C++ you will realize that these are actually constructors.

So you do not have to tell that because that is the technical term. The first few weeks avoid using complicated technical terms. Constructor sounds really complicated, so just tell them that this statement is useful for creating rectangles. Now if I want a rectangle to be rotated right I can issue the command `r1. right`; I can write `c1.move 10, 20` to move by 10 and 20 in the x, y direction, okay. So here now we are keeping track of the coordinate system.

So there is a coordinate frame, it is located its origin is located at the top left corner as usual, okay. And here is an input statement. So if I write `getClick()` it is like reading in values, your program instead of waiting for something to be typed is going to be waiting for a click to happen. And `z` will get 65536 times the x coordinate of the cursor at the time of the click + the y coordinate.

You may see that these are actually just simply saying that the most significant 16 bits should be the x coordinate and the least significant bit should be the y coordinate. And if the student wants to get back x and y all he or she has to do is divide by 65536 and take the quotient as well as the remainder. Okay. If you do this what have your complex? So first of all it will be an implicit introduction to a constructor and member access context.

Introduction performed reasonably painlessly because students understand this very, very precisely. And moreover one student can create circles, squares and move them around. They will be able to do interesting animation. And to do interesting animation they will have to use `repeats` and so it will be practicing; they will get an occasion to practice, repeat and that is certainly core activity in computer science.

And this is immense excitement value, because elementary animation can be done already. This will provide examples and programming problems for weeks to come. And you will have pictorial analogs of everything that you study, and that will be very easy; that will make it very easy to understand many difficult concepts.

(Refer Slide Time: 15:50)

Topic 4 example 1: Projectile motion

```
initCanvas("Projectile motion", 500,500); // start 2d graphics

int start = getClick(); // launch point

Circle sp(start /65536, start % 65536, 5);
sp.penDown();

double vx=1,vy=-5; // initial velocity
repeat(100){
  sp.move(vx,vy);
  vy = vy + 0.1;
  wait(0.1);
}
getClick(); // just so that you can see
```

So along with this lecture you can have a Projectile motion example. So here it goes. So you are going to so far 2D graphics instead of saying Turtle Sent, you are going to have squall initCanvas. So this will create a window of 500/500 pixels and the title will be Projectile motion. So you will get the starting point of where the projectile is going to be launch from by doing getClick().

And at that point you will create a circle. So sp is that starting point, sorry sp is the circle that you created, so its center is going to be at the getClick() that is what you get by dividing 65536 and also taking the remainder and the circle will have the radius 5. And you are going to put the pen for that sp down, okay. Now, there is going to be a velocity x and y velocities and you are going to move that sp by the x and y velocities vx and vy.

But notice that vy is changing, vy is increasing whereas vx remains the same. So that is exactly like how projectiles moves. The y velocity reduces because of gravity. Here it is increasing because of gravity because remember that in our graphical system the y coordinative going downwards. So anyway the y velocity is increasing and the x velocity is decreasing so it will look like the way a projectile moves, okay

And it will draw that projectile it will make the movement for 100 steps and at the end of it, it will stop, and then picture will not go away until you click one more time, so that is the last

click, the last click is where just so that the picture does not go away and you get to see the picture as long as you want to. Now this example on one hand serves as a demonstration of two-dimensional graphic non-turtle based graphics.

At the same time, it is hinting at one very important big idea in scientific computing. And what is that big idea? The big idea is about solving problems in the following manner. So at the beginning I know what the state of the world is. I know how the world was at each step. And by knowing how the world evolves in a tiny fraction of time in this case our tiny fraction of time is one step of that repeat loop and that step might mean one microsecond for all we know.

Okay, so in one microsecond we know how different things change. So the velocity changes-- we do not know directly how the position changes, but we know how the velocity changes. So as a result we can update a state for a world and then we can sure how our world is going to evolve. So this is a very simple system, just a single projectile, just a single ball thrown in the air. But, you should point out that this is how many complex systems actually get simulated.

And this is what might in fancy language we called the Euler's method okay Euler iteration or First order Euler iteration not 0th order, okay. And indeed it is an important idea although it is very simple. You can also do least square fit, okay so we are given points and we want to fit a line to that.

(Refer Slide Time: 19:53)

Topic 4 example 2: Least squares fit

T2C

$(x_1, y_1), \dots, (x_n, y_n)$: points to fit line to

Best fit line is $y = mx + c$ where

$$m = \frac{nr - qs}{np - q^2} \quad c = \frac{ps - qr}{np - q^2}$$

where $p = \sum_i x_i^2$, $q = \sum_i x_i$, $r = \sum_i x_i y_i$, and $s = \sum_i y_i$.

All calculations can be done using a simple repeat loop!

Next

And the best fit line is $y = mx + c$, so we have to determine m and c and we can determine c by following these formulae and the exact values are calculated according to this. This described in chapter 4. And it is an interesting; it is an interesting example; it is an important example in these days of machine learning. But I will still say that this not core material this is tear to core. So decide whether you have time for it and whether your students are up to it, okay.

But the important point is that all these material is accessible, even right now, we do not need 'if' statements, just a simple repeat loop is adequate, okay. So these are the formulae.

(Refer Slide Time: 20:50)

Week 3 assignments 2: Least squares fit

```
initCanvas("Fitting a line to data",500,500);
double p=0, q=0, r=0, s=0;
int n; cin >> n;
repeat(n){
    int cPos = getClick(); double x = cPos/65536, y = cPos % 65536;
    Circle(x,y,5).imprint();

    p = p + x*x;
    q = q + x;
    r = r + x*y;
    s = s + y;
}
double m = (n*r - q*s)/(n*p - q*q);
double c = (p*s - q*r)/(n*p - q*q);
Line l(0,c, 500, 500*m+c);
```

So this is the code, okay. So let me know delve over the code but the code is going to put circles where you click and then it is going to calculate those p, q, r, s as we decided earlier and for that only a repeat loop is going to suffice. And then at the end of it a line l, the last in the, the last line l is being constructed, so that line will get drawn. So the beauty of this is that you will actually be able to click points on the screen and then a line will be fitted to what you click.

Without graphics the way this example would work is that you would have to input the values of x and y and then the coefficients m and c of the line will get printed. Now that is hard is not very easy to check even approximately whether those m and c are correct where at few years you will get a graphical demonstration. So it shows the power of graphics and it also gives you an introduction to a very important area.

(Refer Slide Time: 21:53)

The if statement

All lectures should begin by stating the motivation for what will be taught

"Here is a problem you cannot program with what you already know."

Then teach whatever language feature that is needed for the program.

Motivation for the if statement:

"Suppose you want to calculate tax according to the following rules: (1) If your annual income is less than Rs 180000 you pay no tax, (2) If your annual income is between Rs 180001 and Rs 500000 you pay 10 % of your income above Rs. 180000, (3) If your income is between ..."

You may have other favourite examples

This example can be used to develop many programs.

- ▶ One if statement per tax bracket.
- ▶ One large if-then-elseif-... statement with simpler consecutive conditions.

Use flowcharts to help students understand the difference.

After this you should be talking about the 'if' statement, okay. So again, we should start all lectures by beginning why are we teaching this, okay. Here is the problem you cannot program without the new statement or using what you just what you know already, okay. After that students are receptive to hearing you teach a new language feature. Okay, so what is the motivation for the 'if' statement?

Here is the motivation. You could say suppose you want to calculate tax according to the following rules. If your annual income is $< 1,80,000$ you pay no tax, if your annual income is

between 1,81,001 and 5,00,000 you pay 10% of your income above 1,80,000 and so on. Okay, so there are various rules. Now, as a tax payer which rule fits you requires us to know an 'if' statement or this is something that cannot be calculated using just repeats. And so therefore, you need an 'if' statement.

This tax calculation is somehow a good example, but you could have your own favorite examples. And this example can be used to develop many programs. So for every tax bracket you could have separator statement or you would have one large if-then-elseif statement for where each, each elseif is for a different bracket. But there is some difference between the two the conditions that you using the two statements.

In the second, in the second option the conditions are going to be simpler, because the second- the elseif clause is going to be executed only if the previous clauses have not been executed, okay. So it is important to write a program in several ways, especially 'if' is not an easy statement, so you can – you should show the students several ways of writing the same logic, okay. And here chart will definitely help students understand the different between the two ways of writing the same logic.

So this is where flow charts I will recommend using but this is for understanding a program not really deriving the program. Okay, after that it is natural to talk about loops.

(Refer Slide Time: 24:34)

Loops

Motivation for while loop: In repeat you need to know upfront how many times the loop executes.

- ▶ Soloway's rainfall problem: Given a sequence of numbers terminated by 99999 (sentinel), find the average of all numbers excepting 99999.

Infinite looping possible: Need to argue termination.

Example of difficult termination argument: Euclid's GCD

Rainfall problem loop: (read, test, process)

Can be implemented using code copying or break statement.

Nicely explained using flowcharts.

for loop: Must use when there is a natural "control variable"

Students should be able to transform while to for and vice versa.

Example programs: Many from scientific computing, physical simulation.

Tell students that repeat is implemented using for

So for—again you need to talk about motivation. So for the 'for' loop you should say why is not a repeat loop adequate. Well, you should say in a repeat loop, you need to know upfront, how many times the loop executes. Now, here Soloway's rainfall problem that we have talked about earlier can come in handy. So the problem is given a sequence of numbers terminated by 99999 which is a sentinel, find the average of all numbers except for 99999.

So we do not know how many numbers are going to be there, and therefore the repeat cannot be used so a 'while' is needed. But, with power with great power comes great responsibility as well. So infinite looping is possible with a while loop, while loop gives you power but it also gives you the responsibility of making sure that you are not going to loop an infinite number of times. So we need to argue termination.

That is the responsibility that comes with the power of the i. Then, termination arguments maybe easy or difficult, a slightly difficult termination argument is for Euclid's GCD. And I will strongly recommend doing this. It is a classic algorithm and it can be used as a theme throughout the course. So it can be used in re-correction as well. Okay, so coming back to the Rainfall problem.

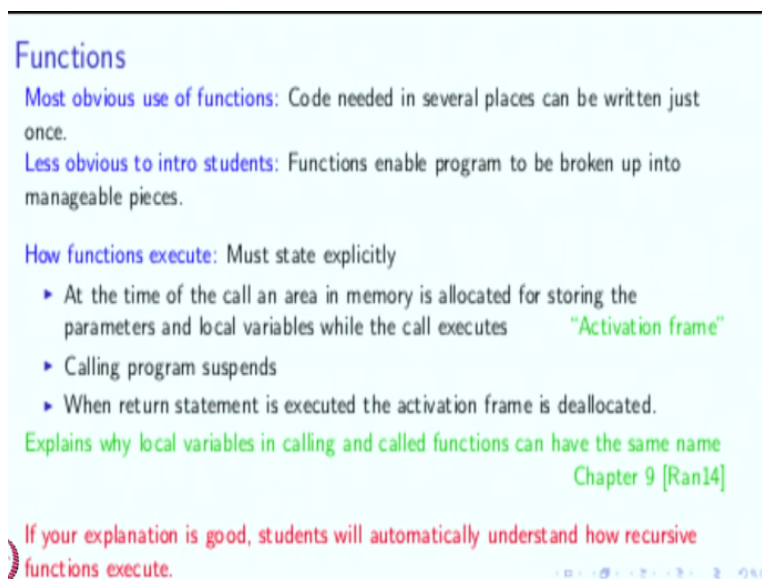
The Rainfall problem loop usually is read, test and process, so the test is in the middle. So which is typically implemented in two ways by using code copying or by the break statement, so both

of these base should be talk about, okay. So and again both of these ways are nicely explained in flow chart. It is given in the book that we are talking about. And next come the ‘for loop’ and the key point of the ‘for loop’ is that you should use it whenever there is a natural control variable.

If there is a control variable tell your students, do not use the .while loop’, make it your style, make it your programming style that the ‘for loop’ must be used if there is a control variable. And when you write a ‘while loop’ vaguely I know that there is not a nice single control variable associated with this repetition that is being discussed. And student should be, should have this skill or should have the ability to transform a program which uses a ‘while loop’ to a ‘for loop’ and vice versa, okay.

Many examples of loops are possible, from scientific computing, physical simulation, okay these are really great examples. And at this point you can tell them that the repeat statement is implemented using the ‘for’ statement. And this loops are discussed in chapter 7 and 8 of the textbook that we have been talking about. The next topic is Functions. And how do you motive functions, okay. So the most obvious use is that if you need the same code in several places.

(Refer Slide Time: 27:58)



Functions

Most obvious use of functions: Code needed in several places can be written just once.

Less obvious to intro students: Functions enable program to be broken up into manageable pieces.

How functions execute: Must state explicitly

- ▶ At the time of the call an area in memory is allocated for storing the parameters and local variables while the call executes "Activation frame"
- ▶ Calling program suspends
- ▶ When return statement is executed the activation frame is deallocated.

Explains why local variables in calling and called functions can have the same name

Chapter 9 [Ran14]

If your explanation is good, students will automatically understand how recursive functions execute.

If you want the same set of operations to be performed in different parts of your program on different data, then functions give you a way of writing the code just ones. Another but less obvious use to introductory students is that functions enable the program to be broken up into

manageable pieces. I would say going with the first one as the main motivation and then introducing the second one later.

How function execute is and is a very important topic in my opinion. And I think you should explain this very, very clearly and very, very slowly. So for example, you should say that at the time of the call an area in memory is allocated for storing the parameters and local variables that are needed while the call executes. So taught is very precise term that area is called “Activation frame.” This is a compiler term. It is a technical term, but this is an important concept.

So if it is an important concept, use a technical term. It signals to the students that something important is being discussed, and so the student is required to pay attention and give some care then the calling programs suspense, okay copying happens and when the return statement is executed, activation frame is reallocated. So I am not giving you all detail over here. You know those details and they are discussed in the textbook as well, okay.

But, why is this needed? So this is needed because it explains, why local variables in the calling and called functions can have the same name. Because even if I say i, if I write it inside the function then that i does not mean the variable in the main program, okay. So the notions of scope and lifetime get very nicely understood if you talk very clearly about activation frame. The notion of copying and the notion of pass by reference also get understood very nicely.

So this is discussed in chapter 9. And a Byproduct of this is that if your explanation is good, students will automatically understand how recursive functions executed.

(Refer Slide Time: 30:24)

Functions: references and pointers

References and pointers are useful in writing functions that

- ▶ Return more than one result.
- ▶ Modify their arguments.



References are natural to be taught with functions.

Teaching of pointers can be deferred to arrays or structures, but:

- ▶ When taught with arrays there is additional complexity of index calculation.
- ▶ When taught with structures, there is additional complexity of references to elements of the same type.
- ▶ When taught with functions we need discuss only the operators address of, & and dereference, *.

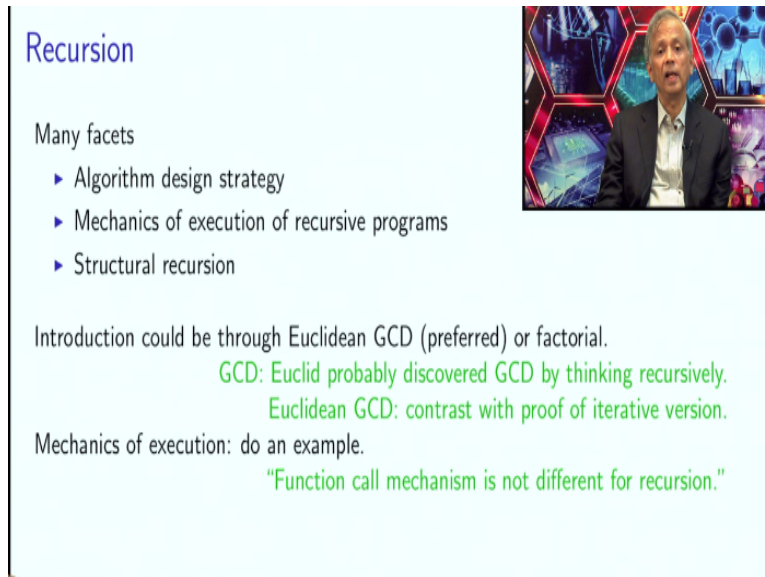
I think along with functions you should talk about references and pointers as well. References and Pointers are useful in writing functions that written more than one result, okay. So a very natural way of doing that is to write reference arguments or pass arguments by pointers. Or if you want a function to modify the arguments then you have to pass a reference or a pointer. Now I feel that references are natural to be taught in functions and I think most people will agree.

Now teaching of pointers can be deferred to arrays or structures, but when taught with arrays there is additional complexity of index calculation. So when you are introducing a concept introduce it at the point at which you can talk about it with more simplicity. And therefore, I suggest talking about pointers along with functions. And if also introduce them with structures but there is this whole business about referencing elements of the same type.

So again instead of that I would say taught about pointers as a part of functions, although the two are not directly related, okay. So if you say that they are not related but you can use pointers if you want functions which modify their arguments or which return more than one result. When you teach functions and when you discuss pointers then you only talk about the address of operator and the dereference operator, okay. So these are two simple operators.

You do not have to talk about the indexing operators. And these two simple operators can be discussed in isolation, okay. Now recursion is an important subject whether you should teach it in the first course or not has been discussed but I should point out that it has many facets.

(Refer Slide Time: 32:35)



Recursion

Many facets

- ▶ Algorithm design strategy
- ▶ Mechanics of execution of recursive programs
- ▶ Structural recursion

Introduction could be through Euclidean GCD (preferred) or factorial.
GCD: Euclid probably discovered GCD by thinking recursively.
Euclidean GCD: contrast with proof of iterative version.

Mechanics of execution: do an example.
"Function call mechanism is not different for recursion."

So first of all it is an Algorithm design strategy, okay. Euclid probably invented GCD algorithm by thinking recursively. But recursion also has a certain mechanics of execution. So more and more frames are created. So students must become familiar with both of these facets. And to make matters more exciting there is structural recursion as well. So which of these things you should talk about?

Well, introduction could be through Euclidean GCD which I prefer or you could use factorial because it is simpler, okay. And Euclidean GCD if you talk about then it is proof of correctness or proof of termination can be contrasted with the proof of the correctness of the iterative version or the plan you write in the iterative version, okay. Mechanics of execution; actually show the activation frames that get created and encourage students to be able to show activation frames as exercise, okay.

Again, if you do this you will be able to tell student that the function call mechanism is really the same, recursive functions do not have any different mechanism, okay. So that is the beauty of

explaining that mechanism because you explain it ones and it will come in handy in explaining this supposedly more complex concept of recursion, and this is discussed in chapter 10.

(Refer Slide Time: 34:27)

Structural recursion

Structural recursion is more known to student, e.g. family trees.

Canonical problem class that can be studied:

Persons are numbered 0 through $n - 1$. Array `Father[0..n-1]` contains values s.t.

- ▶ If `Father[i] = -1`, then the father of person i is not known.
- ▶ If `Father[i] ≥ 0`, then `Father[i]` is the father of person i .

"Write a program that reads a number j and finds the oldest ancestor of j ."

"Write a program that reads a number j and finds the number of descendants of j ."

- ▶ Many such relationship questions can be asked.
- ▶ Students will be able to solve them, while getting practice with index manipulation and recursive thinking.

Classical structural recursion is discussed in Chapter 24, [Ran14]

I want to say a little bit about structural recursion because structural recursion is more known to student because students know family trees, because student know that there can be a parent but the parent will himself of herself have a parent and so on. So there is some sort of a natural notion of recursive relationship. So given such familiar structures okay familiar relationships you can ask interesting problem.

So for example you could say, Persons are numbered 0 through $n-1$. And say for example, Array `Father` with n elements contains value such that if `Father` is `-1` then the `Father` of Person i is not known. If `Father` or sum i is > 0 then `Father` of i actually the Id of the `Father` of Person i . So for a Person i you either the array either tells you that look you do not know who the `Father` is or you or the array tell you which number is the `Father` of the Person.

So now you can ask something like, write a program that reads a number j and finds the oldest ancestor f j . Will a student be able to solve this problem if the array is given in the bold? Definitely, try it, give it to students and ask them to solve it. And now can we do it using the C++ program, or whatever programming language you use. For that a natural thing will be to use recursion.

Well, recursion will be either more natural for this program where you say, find the number of descendants of j so the natural idea will be, just do not look at all direct descendants then look at all the indirect descendants. So the indirect descendants will be most naturally found by using recursion. So many such relations questions can ask; it could be parents, it could be bosses okay.

Now students will be able to call them while in the case of the array they will get practice with manipulating indexes and also recursive thinking. Classical recursion—structural recursion is discussed in Chapter 24, but you really do not need to do that. You can just do this example. And —because this is such a familiar example, students will follow it and it will also drive home the notion of recursion. You can also have pictorial recursion.

(Refer Slide Time: 37:13)

Pictorial Recursion

Tree = trunk + two small trees

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

So for example, I can think of a tree as a Trunk two small trees. And this code puts that observation into practice. So it says that if I want to draw a tree with levels > 0 then it consists of a forward a line obtained by moving the turtle forward and that is the trunk. Then the turtle turns left and then draws a tree with one less level or it draws the first smaller tree. When the turtle comes back and now you turn again this time you compensate for the 15 degrees you turn left and you draw further draw, further turn 15 degrees to the right and you draw a second smaller tree.

So then you come back and then you come back down your trunk—okay because somehow you want, you are sort going to tell the user that look, this function is going to draw a tree, the turtle starts at the bottom of the trunk; it draws everything and then it comes back again to the bottom of the trunk, okay. So this code actually does draw a tree. But look at how simple it is. Look at the power of recursion.

So it helps in understanding recursion and it is also a good source of problems. Okay, well as far as arrays are concerned there are many uses of arrays, unordered list or set;

(Refer Slide Time: 38:44)

Arrays

Many uses of arrays:

- ▶ Unordered list/set Roll nos of students in a class
- ▶ Queue Passengers waiting for taxis
- ▶ Map from integers $0, \dots, n - 1$ to array elements Histogram

Index calculation:

- ▶ $A[i]$ calculated in constant time.
- ▶ Why $A[i]$ might go out of bounds.

Roll numbers of students in a class need to be put in unordered list or set. It can be ordered list. So it can be used for Queue, so it can be passengers waiting for a taxis for example or it could be a map, and array can serve as a map. So it maps the number 0 and -1 into array elements and this is how it gets used in calculating histograms. So Index calculation deciding which actual variable gets used requires the computer to decide an index calculation or to determine what $A[i]$ remains.

So if we talk about pointers then we can say that this requires an address calculation and it can be done in constant time. And it also can be used to explain why $A[i]$ might go out of bounds if i is very large, okay. So the pointer interpretation of A is quite important especially because you want

to pass A2 functions. So these points need to be studied, okay. So, Multidimensional arrays and representation of matrices should be discussed.

But probably you do not have too much time and probably it is better discussed as vectors of vectors so after you do the standard library. And null terminated character arrays should also be discussed, but I again I will recommend not going overboard with both of these topics because they are better than using the standard libraries of C++. Structures are the starting point of object oriented programming and it should just about state.

Structures and classes are essentially the same in C++, okay the only there is only a small difference between the two things.

(Refer Slide Time: 40:47)

Structures: starting point of OOP

- ▶ Structures and classes are essentially same in C++.
Members are public in structures and private in classes by default.
- ▶ Motivation for structures: aggregation/composition
Gathering together the data/code associated with an entity.
Represent every interesting entity using a structure.
- ▶ Member functions are natural and can be motivated.
- ▶ Constructors can be motivated as a means to "not forget to initialize".

```
struct queue{int elts[10],next; queue(){next = 0;}}
```
- ▶ Copy constructor, destructor, are harder to motivate. T2C
- ▶ Operator overloading is not hard, but perhaps not worth the effort as core. T2C
- ▶ Encapsulation, inheritance and polymorphism are slightly harder to motivate.

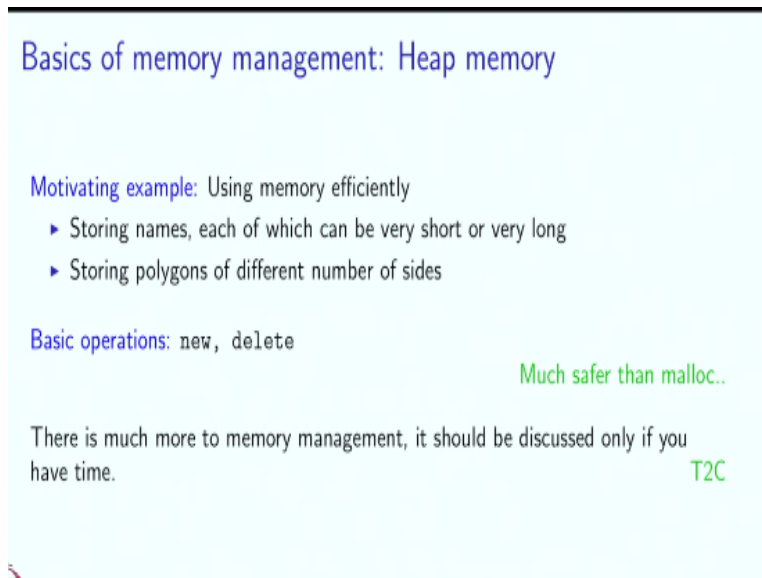
And the motivations for structure is aggregation or composition which is really an OOP idea. Gathering together the data or code associated with an entity. And from a programming philosophy point of view what this is going to say is that you should represent every interesting entity using the structure, this kind of the OOP principle. Member functions are natural and can be motivated. Constructor are motivated as a means to "not forget to initialize."

So therefore, I would like to put this as a core topic, okay. So for example, I am having a queue, if I put a constructor then this next element will get initialized to 0. So the queue will sort of be

well behaved and you will not have to remember initialize it. Copy constructor, destructor etcetera are harder to motivate and that is definitely not core material, it is tear to core. Operator overloading is not hard, but perhaps it is not worth the effort. So you could put it as tear to core as well.

Encapsulation, inheritance and polymorphism are definitely harder to motivate and therefore they are tear to core.

(Refer Slide Time: 41:55)



Basics of memory management: Heap memory

Motivating example: Using memory efficiently

- ▶ Storing names, each of which can be very short or very long
- ▶ Storing polygons of different number of sides

Basic operations: new, delete

Much safer than malloc..

There is much more to memory management, it should be discussed only if you have time. T2C

Basics of memory management, Heap memory. Okay, using memory efficiently using the storing names, can be very short or very long, so you do not know how long their name is so you cannot declare an array very easily. Storing polygons are of different number of sides. So you should teach the core area as a core topic new and delete, okay. And by the way this is much safer than malloc.

There is much more to memory management, but it should be discussed only if you have time and it is really T2C, tear to core, okay. So things about memory leads and how to height behind constructors and destructors and things like that.

(Refer Slide Time: 42:43)

The string and vector classes

string class should be considered **core** and discussed well.

- ▶ Manipulating null terminated strings is very error prone.
- ▶ If you do not introduce string class your students will continue to use null terminated strings.

vector class should also be considered **core**.

- ▶ If you don't teach this your students will somehow learn linked lists and use those.
- ▶ Vectors are better than lists in most use cases.
- ▶ Convenient even otherwise: no need to pass length.
- ▶ Library function for sorting.

"These classes use memory allocation but that is an advanced topic."

The string and vector classes are great and therefore I consider them to be core. So manipulating null terminated strings is really error prone. If you do not introduce string class your students will continue to use null terminated strings and that would be a great petty. So your—so you should use string class. Vector class should also be considered core, okay. If you do not teach again your students will learn linked lists for example and use those.

And instead of learn linked list the vectors are much easier are very easy and by enlarge they can be used in most places that linked lists are used. And they are convenient even otherwise, they are convenient even more than arrays, because if you have a vector it has a member function for size and you do not have to pass the size when you pass a vector to a function. Then there is a Library function for sorting which is also useful.

So can say that these classes use memory allocation and we are not going to talk about how that happens, that is an advanced topic, and it could be taught as a tear 2 core topic. So it has been discussed in the book, but it is tear 2 core material. So the working environment choices are it could be a shell some kind of a unique shell or some shell.

(Refer Slide Time: 44:19)

Working environment

Choices: shell + editor. Choose editor that indents.
Codeblocks IDE: modified to allow single file program.

And an editor, if you an editor then pick one that can do indention, okay. Encourage students to look at programs only after they are indented otherwise you will not spot wrong nesting and things like that. We modified the code blocks IDE so you can use it. So-- it has been modified to allow single file programs. So I am going to conclude in a few minutes but I want to point out that graphics and repeat turn out to be really useful in this entire exercise.

So they enable discussing interesting problems from day 1.

(Refer Slide Time: 45:02)

Utility of graphics and repeat

- ▶ Graphics and repeat enable discussing interesting problems from day 1.
- ▶ Graphics is useful throughout the course for generating interesting problems.
- ▶ repeat is useful until standard looping statements are taught, which might happen as much as a month into the course.
- ▶ Having good programming examples is very useful to make students remember concepts.
- ▶ Graphics is useful for explaining concepts also, e.g. recursion.
- ▶ Repeat is halfway to looping constructs, so breaks down the difficulty of understanding looping constructs.
- ▶ Graphics and repeat are very easy to learn: ideal scaffolding.

Like training wheels on children's bikes.

It is useful throughout the course for generating interesting homeworks and classroom problems. Repeat s useful until standard looping statements are taught, which might happen as late as a

month into the course. So instead of doing dull examples, with repeat you can have very interesting examples. Having good programming examples is very useful to make students to remember concepts and that is accomplished by using graphics and repeat.

So graphics is also useful for explaining concepts, you can use it to draw trees and trees and sort of recursion action. So you can see that “Oh, I am going to draw the left side first”, and indeed the left side appears first in your recursive core. So the correspondence between the drawing and the program is very interesting to students. Repeat is halfway to a standard looping constructs, so it breaks down the difficulty of understanding looping constructs.

The graphics and repeat are very easy to learn and so in my opinion their idea is scaffolding, okay. Just like training wheels on children’s bike. My experience has been that even the weaker students understand turtle movement and repeat. And this understanding gives them confident to write programs, and so please, please, please use this; your students will love you for it.

(Refer Slide Time: 46:23)

Concluding remarks

- ▶ Basic ideas behind our pedagogy were presented.
- ▶ Pedagogy for the core syllabus was presented.

So concluding, to conclude what we did was we presented the basic ideas behind our pedagogy, okay, and we only talked about the core syllabus. We will talk about the other part the T2C part a little bit later. Pedagogy for the advanced topics and Design of examinations is what we will talk later on.

(Refer Slide Time: 47:04)

- 📖 B. du Boulay, *Some difficulties of learning to program*, Studying the novice programmer (E. Soloway and J. Spohrer, eds.), Lawrence Erlbaum, Hillsdale, NJ, 1989, pp. 283–299.
- 📖 Tobias Kohn, *Variable evaluation: An exploration of novice programmers' understanding and common misconceptions*, Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education (New York, NY, USA), SIGCSE '17, ACM, 2017, pp. 345–350.
- 📖 Abhiram Ranade, *An Introduction to Programming through C++*, McGraw Hill Education (India) Private Limited, New Delhi, 2014, <http://www.cse.iitb.ac.in/~ranade/book.html>.
- 📖 Simon, *Assignment and sequence: Why some students can't recognise a simple swap*, Proceedings of the 11th Koli Calling International Conference on Computing Education Research (New York, NY, USA), Koli Calling '11, ACM, 2011, pp. 10–15.

These are the references and-- so that brings us to the end of this sequence of lectures. Thank you.