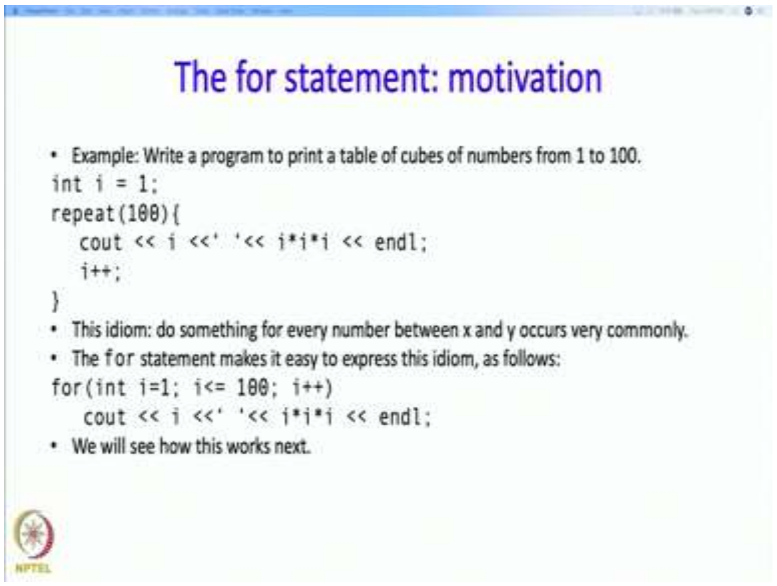
Introduction to Programming through C++
Professor Abhiram G. Ranade
Department of Computer Science and Engineering
Indian Institute of Technology Bombay
Lecture No. 7 Part – 4
Looping Statements
The for statement

(Refer Slide Time: 0:17)



The for statement: motivation

- Example: Write a program to print a table of cubes of numbers from 1 to 100.

```
int i = 1;
repeat(100){
   cout << i <<' '<< i*i*i << endl;
   i++;
}
```

- This idiom: do something for every number between x and y occurs very commonly.
- The for statement makes it easy to express this idiom, as follows:

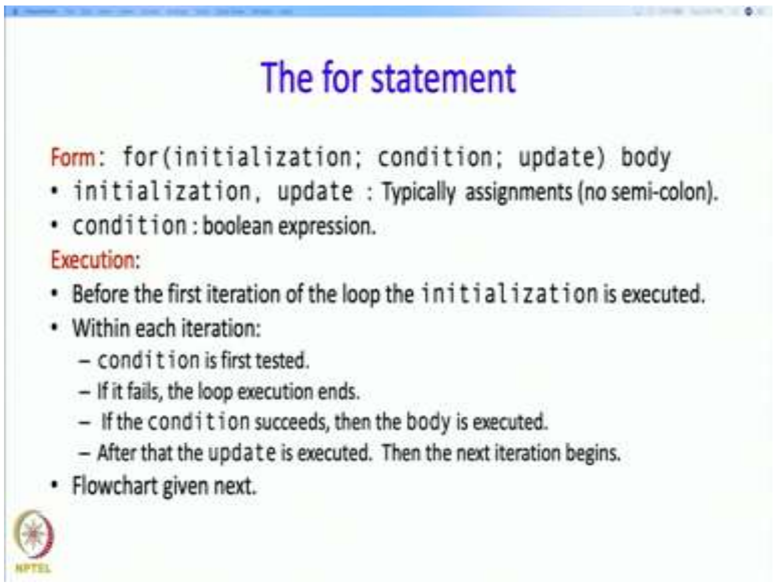
```
for(int i=1; i<= 100; i++)
    cout << i <<' '<< i*i*i << endl;
```

- We will see how this works next.

In the last segment we discuss the break statement and the continue statement. Now, we are going to discuss another statement called the for statement, this is a bit of a complicated statement and so let me state a problem which will sort of need the statement and where the statement will fit very nicely. So here is a problem, we would like to write a program to print a table of cubes of the numbers from 1 to 100, so what do we want? We want on the first-line 1 and 1, on the second line 2 and 8, on the third line 3 and 27, the fourth line 4 and 64, 5 and 125 and so on, all the way till 100. Now of course you can write this using the while or you can also write it, even using the repeat, because you know that you are going to do exactly 100 iterations, so there is no real mystery about that. So, what would you do? So for example you will write this int I=1, repeat 100 times, cout<<I<<' '<<I*I*I<<endl; have a space in between. And of course then you would have to increment I, so that would be the program or that would be the program fragment which will do what we want.

Now, if so happens that this idiom and by that I mean doing something for every number between some X and Y, so this idiom occurs very frequently and therefore, C++ provides you a way to describe it very succinctly and that is exactly the for statement. So this is how the statement works, so you want things to go between 1 and 100 and we want that variable to be called I, so we are going to write for(int I=1, I<=100; I++) and we want the value of I to change by I++, so we also specify that right here. And then what we actually want to happen, we are specify after this and this is the body of the for, so we will get the exact statement in a minute. But this code is equivalent to the four lines, I have written earlier and this code is very stylised and as a programmer, you will get very, very familiar with it and just by looking at it, you will know that oh I is going from 1 to 100 and this is going to be done for each value, whereas in the previous statement these limits and this systematic update is kind of lost, whereas over here it is an exactly a predictable place and therefore this makes for better readability. So exactly how this work, we will see in a minute.

(Refer Slide Time: 3:23)



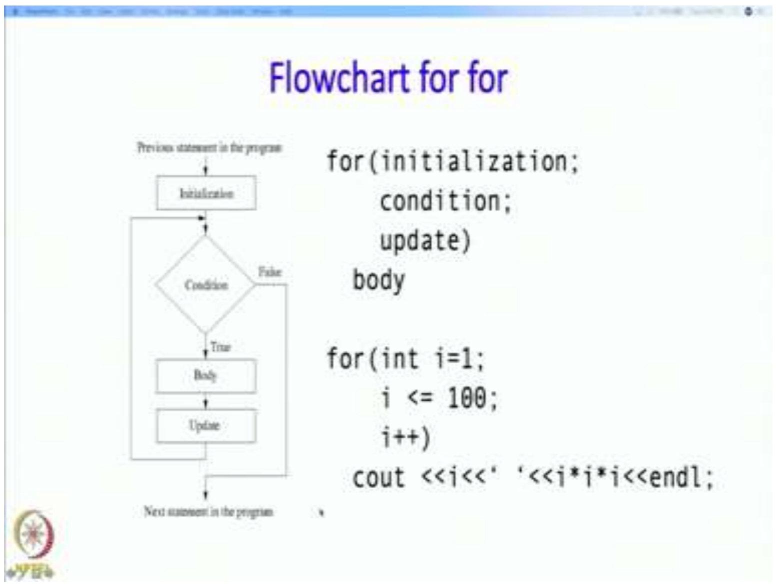
The for statement

Form: `for(initialization; condition; update) body`
- `initialization`, `update` : Typically assignments (no semi-colon).
- `condition` : boolean expression.

Execution:
- Before the first iteration of the loop the `initialization` is executed.
- Within each iteration:
  - `condition` is first tested.
  - If it fails, the loop execution ends.
  - If the `condition` succeeds, then the body is executed.
  - After that the `update` is executed. Then the next iteration begins.
- Flowchart given next.

So the general structure of the for statement, so it is used like a following, so for then inside parenthesis first we have the initialization, initialization followed by semicolon, followed by a condition, followed by a update and then the body, the initialization and the update are typically assignments, they could be other things, but they could be typically assignments and of course the semicolons are not included, of course there is an semicolons immediately after the initialization, but after the update, there is no semicolon, so that something that you have to keep in mind. Condition as usual is a Boolean expression and the body, well the body could be a single statement or it could be a block, alright, so how does this execute? So, before the first iteration of the loop the initialization is executed. So we will see the example in a minute, but first the initialization is executed, then the iteration begins, then at the beginning of each iteration the condition is tested, if the condition fails, if the condition turns out to be false then the rest of the iteration, which includes the body is not going to be executed, and in fact the loop will end, the loop execution will end and you will go to execute the next statement following the for statement. If on the other hand, the condition turns out to be true, then the body is executed and after the body is executed, the update is also executed, then after the execution of the update the next iteration begins, what does that mean? Again the condition is going to be tested, again, if it fails the loop ends.

(Refer Slide Time: 5:25)



Flowchart for for

Previous statement in the program
Initialization
Condition
False
True
Body
Update
Next statement in the program

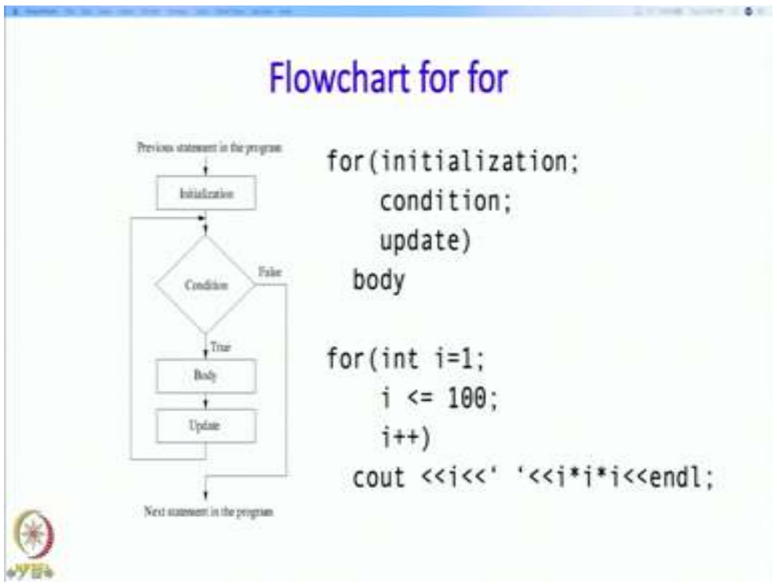
```
for(initialization;
    condition;
    update)
  body

for(int i=1;
    i <= 100;
    i++)
  cout <<i<<' '<<i*i*i<<endl;
```

Alright, so let us look at the flowchart. So shown here is the flowchart for the for, then the for statement itself and an example. So what is happening over here? Well, this is the for statement, so the initialization is executed first as in the flowchart, then the condition is checked, if the condition is false, then the statement goes to the next statement in the program, if the condition is true then the body is executed, so this body is executed and then if after the body is executed, the update is executed. So the update it is stated earlier, but it is executed at this point and after that again, the entire condition testing happens and so on.

So, what happens over here? So I equal to 1 is the initialization, so we start by saying int I equal to 1, so I becomes 1, then we are going to check the condition, so this step happens, if this condition is true which in the current case it is because 1 is less than or equal to 100, so then we are going to execute the body, so see that is what is happening over here. After we execute the body then we are going to execute the update, so at the end of the body, after the body really this update is executed, so after this we increment I, so now I becomes 2. So we printed for I equal to 1, we printed 1 and 1, now I has become 2, so I has become 2 and we go back and we start from the condition check again, so we again execute this is I less than 100, so 2 still less than 100, so we again print 2 and 8 and so on until we get to a value of 101 over here, so 100 will pass this and so we will print 100 and 100 cube, but 101, this condition will not be true and therefore, will exit this and we go on to the next statement following the for.

(Refer Slide Time: 7:43)

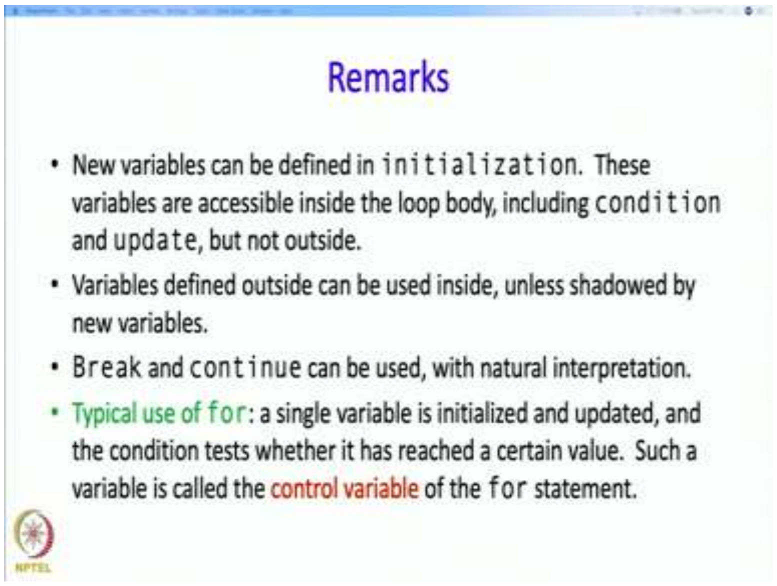


Flowchart for for

Previous statement in the program
Initialization
Condition
False
True
Body
Update
Next statement in the program

```
for(initialization;
    condition;
    update)
  body

for(int i=1;
    i <= 100;
    i++)
  cout <<i<<' '<<i*i*i<<endl;
```



Remarks

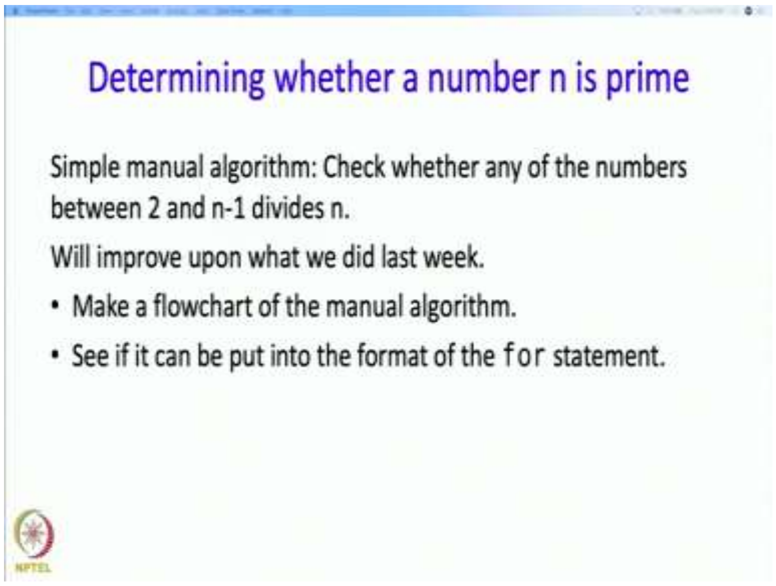
- New variables can be defined in `initialization`. These variables are accessible inside the loop body, including `condition` and `update`, but not outside.
- Variables defined outside can be used inside, unless shadowed by new variables.
- `Break` and `continue` can be used, with natural interpretation.
- Typical use of `for`: a single variable is initialized and updated, and the condition tests whether it has reached a certain value. Such a variable is called the control variable of the `for` statement.

Now some remarks are in order, so as we saw in our example new variables can be defined in initialization. Now these variables are accessible inside the loop body, including the condition and update but they are not accessible outside, so they are within the scope of that for statement. So that something to be kept in mind, the intention is that those are sort of internal to the for statement and therefore, we should not expose them outside, so sometimes, so by and large this is useful and sometimes if we need something different, we can of course do something different.

Now variables which are defined outside can be used inside, unless they are shadowed by the newly defined variables. So in the previous case over here. I define that ‘I’ over here, but suppose an I was defined over here, then inside the body. I am going to get this I, not I defined outside. On the other hand if I just wrote I equal to 1 and I did not write this int I then that means this I will refer to the variable that was outside, so then no shadowing will happen, so I am not obliged to declare a new variable over here, but if I do, then that variable may shadow the variable which of the same name if any defined outside. Otherwise, I will just get a new name and that new name will vanish as soon as my for loop execution ends and break and continue can be used with the natural interpretation that is the moment you come to a break, the statement of the for will end, if you come to a continue then you do the update. So the rest of the body is going to be skipped.

The typical used of the for is that is a single variable is initialized and updated, and the condition test whether it has reached a certain value and the loop execution really depends very strongly on the value of this variable and therefore, this variable is called the control variable of the for statement. So in the previous table of cubes program ‘I’, whose cubes we were printing, the variable I was the control variable for this for statement.

(Refer Slide Time: 10:15)



Determining whether a number n is prime

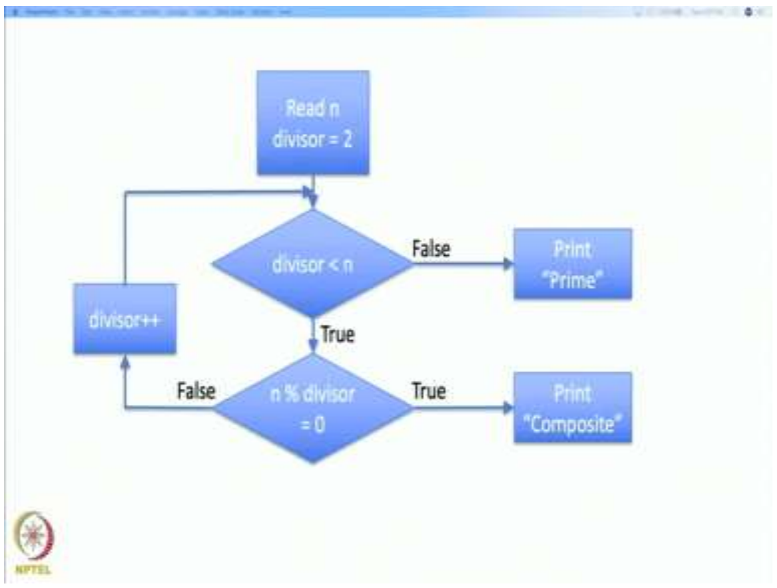
Simple manual algorithm: Check whether any of the numbers between 2 and n-1 divides n.

Will improve upon what we did last week.

- Make a flowchart of the manual algorithm.
- See if it can be put into the format of the `for` statement.

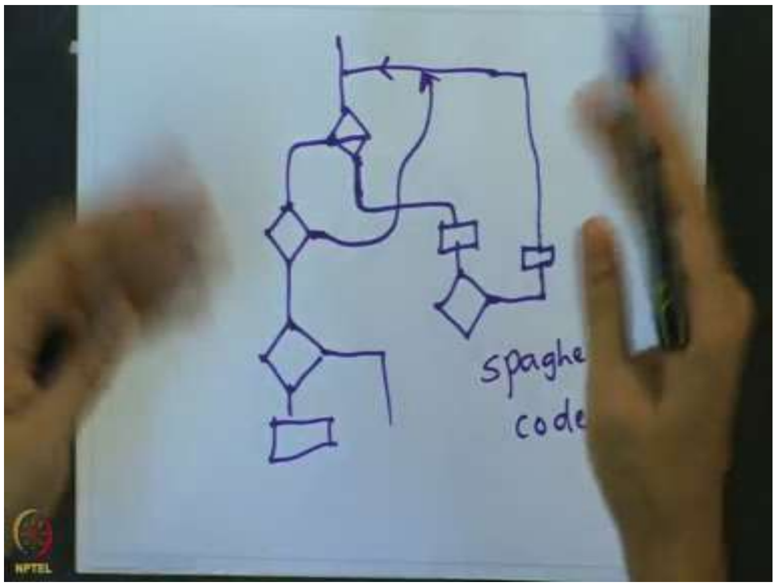
Okay, now I am going to take a somewhat elaborate example, a problem that you have seen before determining whether a number n is prime. So the manual algorithm is simple, and in fact we have seen this manual algorithm earlier, so the algorithm was check whether any of the numbers between 2 and n-1 divides n, so we are going to systematically go from 2 to n-1 and check whether those numbers, any of those numbers divides n. We looked at this problem last week, but there our program was a little bit inefficient, once we found that a number between 2 and n-1 divides n, we know already that n is composite and we do not need to do any additional divisions. Okay, now, our program will be efficient in that it will indeed not to any additional divisions, whereas our last week's program would be doing those additional divisions, even though the answer last time was also correct, last week's program was a little bit inefficient and this time we will be eliminating that inefficiency. So, let us begin by making a flowchart of the manual algorithm and then we will try to use the for statement because it seems that the for statement is rather nice over here, you could think that the divisors which we are going to try out should sort of be the control variables for a for statement that you might write. Okay, so yes, so that is our goal to see if this is a flowchart can be put into the format of the for statement.

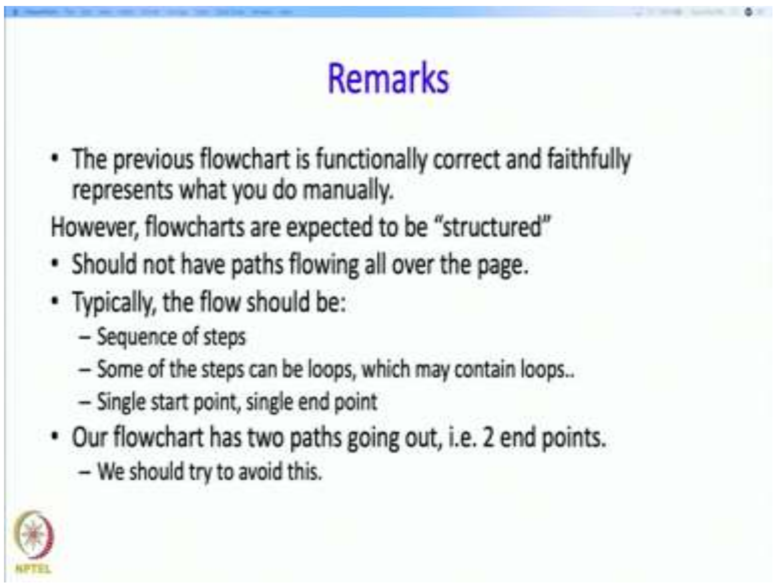
(Refer Slide Time: 12:02)



So here is the flowchart for the manual algorithm. Let me sort of tell you how this works, so first we are going to read n and then we are going to set a variable called divisor to 2, so this is the current divisor with which we are testing. Now, if this divisor is smaller than n, then we have to do something. But if it has already become larger than n okay, it has become n or larger than we take this branch okay, so when does it become n or larger? Well, it becomes n or larger if we have tried out all the numbers between 2 and n and they have not been able to divide okay, So there was always reminder when we tried to divide n by those numbers, so in that case, we know that we can get out of the loop and we can in fact print the message ‘prime’. If on the other hand, the current divisor that we are trying out is smaller than n, then we are not yet sure but the number is prime, so we are going to check is n mod divisor equal to 0 or does is n divided exactly by the divisor? If it is true, then we have found a divisor and therefore, we can immediately declare that the number is composite and we can stop, so that is what this branch is doing. But if it is false then we know now that this divisor did not work, but at this point we still cannot be sure, we still do not know whether the number is prime or composite, so, so far we have not found any divisors, but we may find some divisors going further. So what do we do? Well, we are going to increment the divisor by 1, so we are going to try the next divisor and then we are going to go and check again. So again, if the new divisor is smaller than n or let say it is bigger than n then we go out on this branch, but then that means we have tried out all divisors only then has divisor become equal to n okay, so in which case we have tried out all divisors and we can get out and we can stop after printing prime over here. So that is the overall logic.

(Refer Slide Time: 14:34)





Remarks

- The previous flowchart is functionally correct and faithfully represents what you do manually.

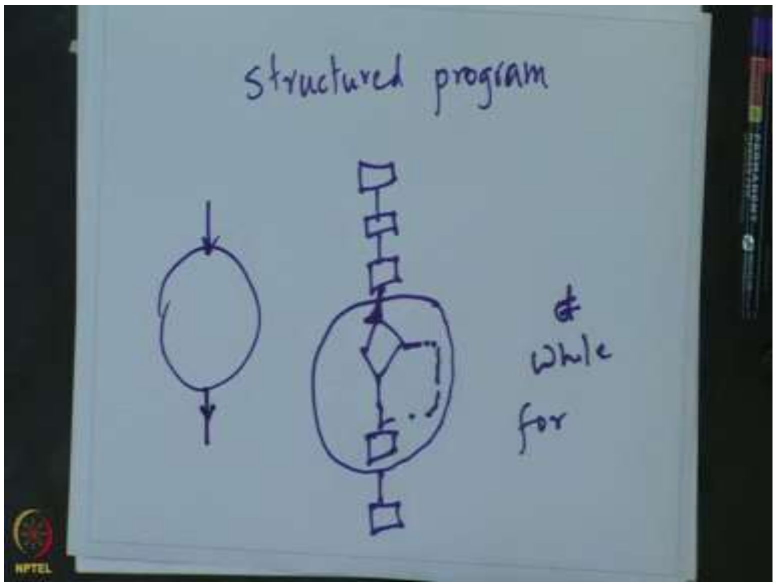
However, flowcharts are expected to be "structured"

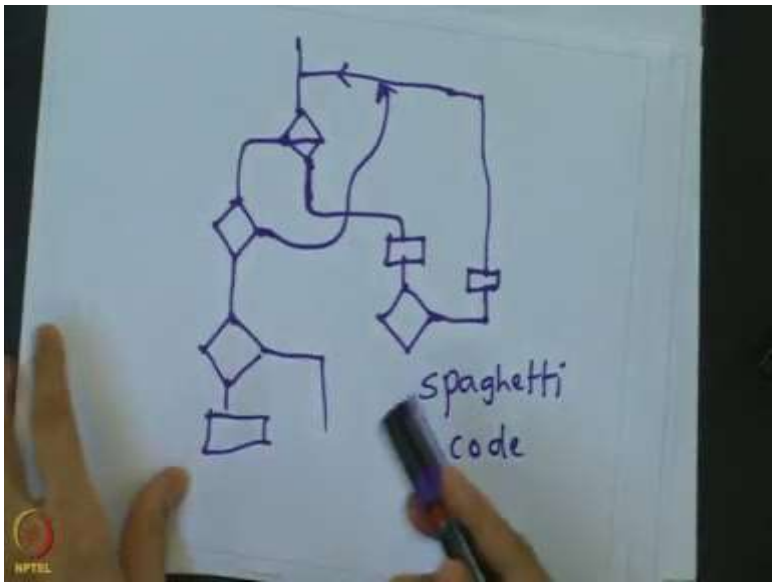
- Should not have paths flowing all over the page.
- Typically, the flow should be:
  - Sequence of steps
  - Some of the steps can be loops, which may contain loops..
  - Single start point, single end point
- Our flowchart has two paths going out, i.e. 2 end points.
  - We should try to avoid this.

Now, this logic is perfectly fine. Okay, however, flowcharts that you draw are expected to be structured and let me explain to you what that means, so basically when you draw flowchart you can have paths flowing all over the page. So let me show you one possible flowchart, so I am going to test something. Then from here, if it is then I am going to do and go and execute this block, then again I am going to test something, then if this is false, I am going to go and execute this block, and then from here maybe I come back and I merge with this point and if this was false, maybe I go off and check something. Okay and then may be from here also I merge at this point and otherwise I continue, maybe I do another check. Okay, maybe I execute something and so on.

So if your flowchart is really complicated like this, then it is rather hard to understand. Okay, so such flowchart or such flow of program is often called spaghetti code, why? Because you could imagine that these lines which are going in sort of random manners all over the place are like that pasta, noodles or whatever, which are just lying around and I mean it just look like a mess. Okay, so you really do not want your code to look like this.

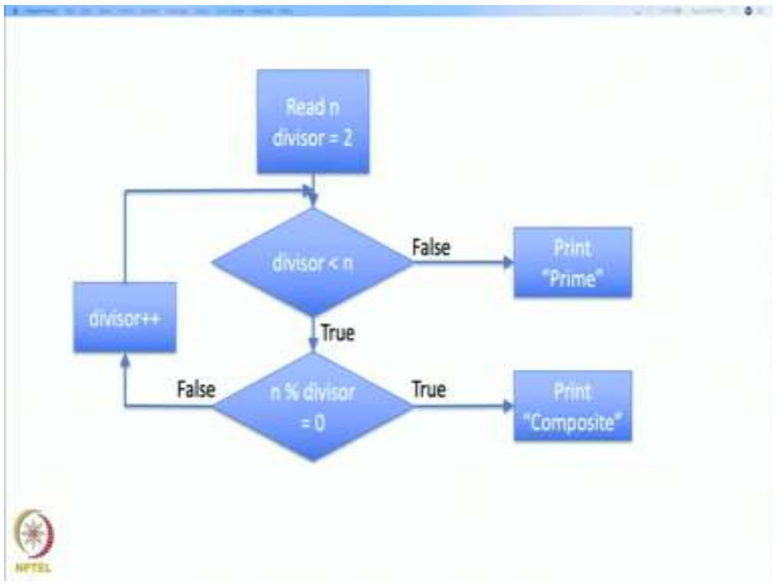
(Refer Slide Time: 16:22)

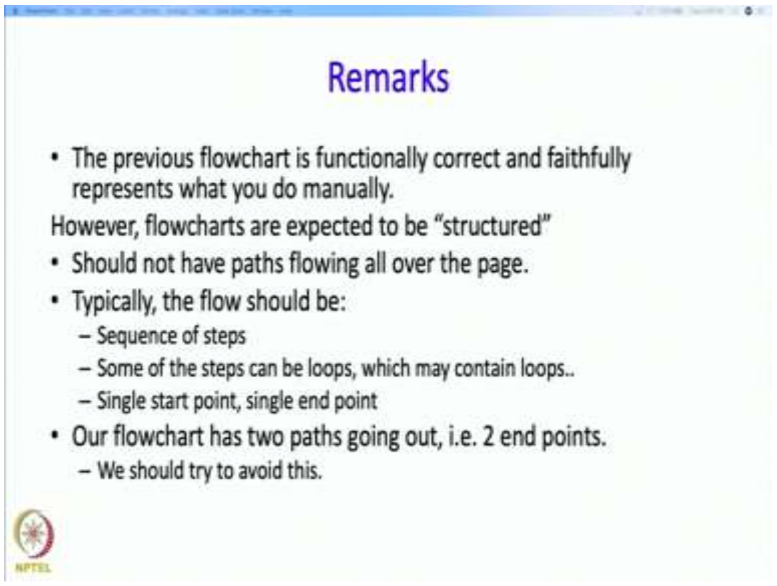




So, what kind of a code do you want? Well, the code that you want is called a structured code or structured programs sometimes. So what does that mean? So it means that I am going to do something, I am going to do something in the next. I am going to do something in the next. Okay, maybe I am going to have may be after at some point I am going to have a test, but the test I may do something over here and eventually I am going to come back to the same point over here, I may do something, I may go back but, so there might be a complicated blob over here, but again I am going to come back to the same point and this complicated blob should be again some kind of a while loop, it should be a while loop or it should be a for loop or something like that, so I might have nesting, so I might have straight-line code like this, one of which, one of the elements in it is a for loop and inside the body of the for loop maybe I can have a while loop okay, but the point is that this nesting should be sort of perfect in that manner. So it should not be like this where suddenly this line comes back and merges with here and this line comes back and merges with over here, because if it is complicated like this, then it is harder to understand, so basically we want that if we, even if we have multiple conditions, so if we have blob over here, then thing should enter over here and only one path of control should exit from this.

(Refer Slide Time: 18:14)





Remarks

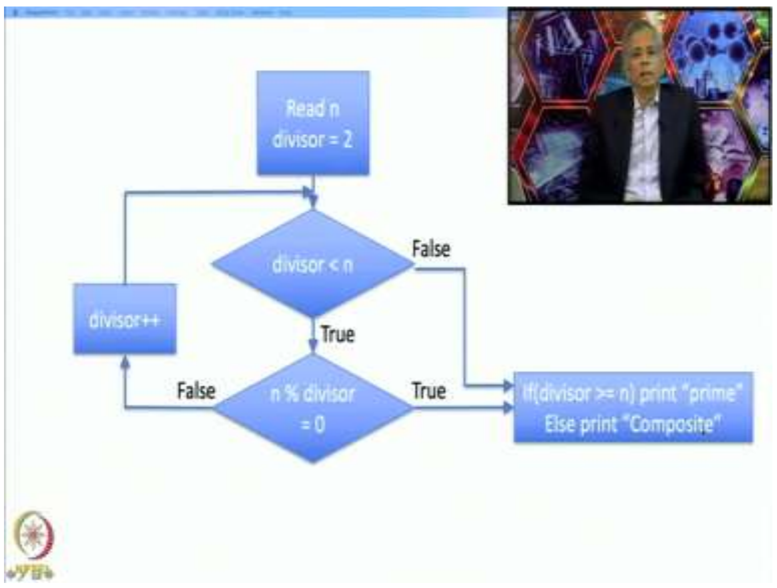
- The previous flowchart is functionally correct and faithfully represents what you do manually.

However, flowcharts are expected to be "structured"

- Should not have paths flowing all over the page.
- Typically, the flow should be:
  - Sequence of steps
  - Some of the steps can be loops, which may contain loops..
  - Single start point, single end point
- Our flowchart has two paths going out, i.e. 2 end points.
  - We should try to avoid this.

So now, if we go back in this code two paths of control are exiting, one path is exiting over here, another path is exiting over here, we do not want that, we have one entry into this, so coming from the top and we would also like the code to exit in exactly one place. Alright, so how do we do this? So here is what we should do? We should merge those two, merge these two paths okay, but now if we merge these two paths, then on one side I am printing prime, on the other side I am printing composite, so I cannot just merge them, I have two somehow print only one of the messages. So what we need to do is we could merge them, but we could decide based on some information that we might have, what message to print okay, so that is exactly what we are going to see in next.

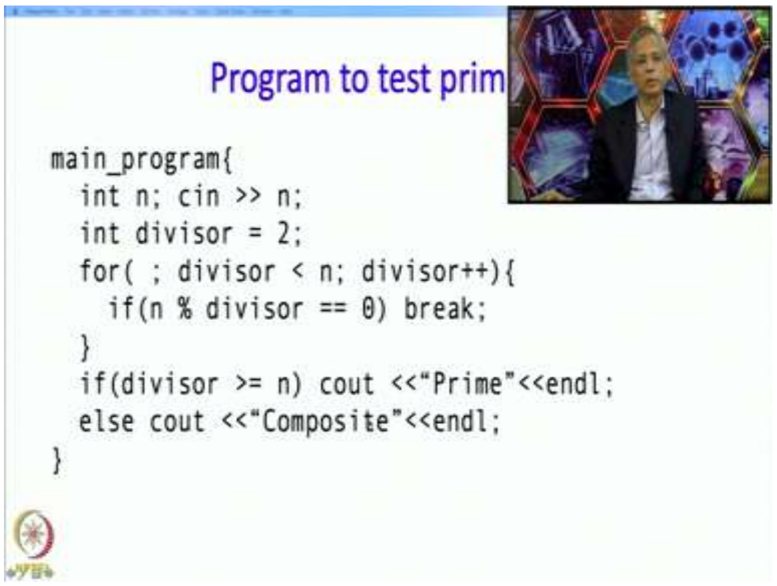
(Refer Slide Time: 19:07)



So here this earlier portion is the same. Okay, but instead of having two outgoing, two paths going out, we have that same path coming into this, two paths into this same block. Okay, and here we are again to check is the divisor greater than or equal to n, if the divisor is greater than or equal to n, then we know we came out on this path, but if we came out on this path we wanted to print prime, so we print prime. Otherwise, we must have come out on this path and therefore, we want to print composite, so we are printing composite. So with this simplification or rather, maybe this is not simplification, maybe this is complication, but let us say with this modification our new flow chart it matches the structure of a for statement. So let me tell you how. So here for example, I can think of as the condition in the for statement, this divisor equals to 2 is the initialization in the for statement, this divisor++ is the update part of the for statement and what is this? Well this is the body, but the body now contains a break and this break is going to take me outside, but you know that the condition also takes me outside, so this is really the outside part, I should draw it over here saying that it is coming next after the for statement, but it would be really congested over here to draw and therefore, I have drawn over here okay.

But that is really the idea that this is the condition check, this divisor equals to 2 is the initialization part of the for, this is the body of the for and this is the update part of the for, so now we are ready to write a program.

(Refer Slide Time: 21:06)



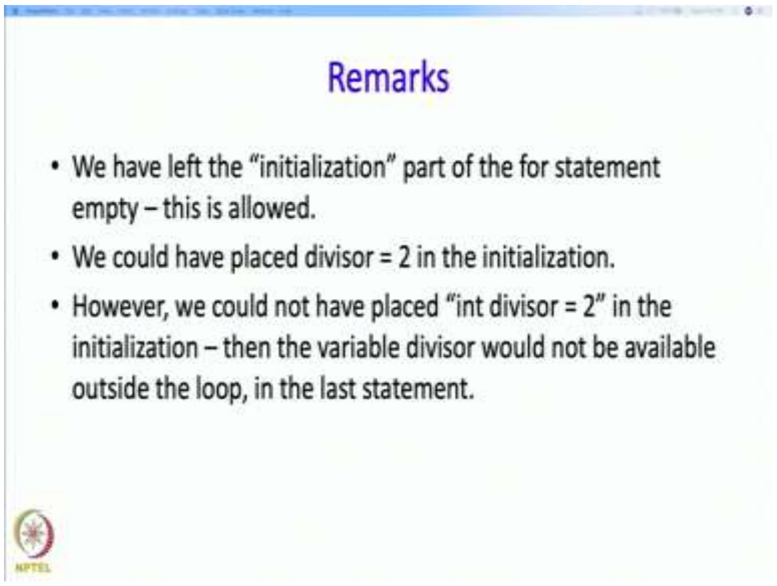
Program to test prim

```
main_program{
  int n; cin >> n;
  int divisor = 2;
  for( ; divisor < n; divisor++){
    if(n % divisor == 0) break;
  }
  if(divisor >= n) cout <<"Prime"<<endl;
  else cout <<"Composite"<<endl;
}
```

So here is the program, so we are going to read in n, then we are going to initialise the divisor. Well, here we have pulled out the initialization and what we have done is, we have kept the initialization empty, but I could have declared the divisor over here and I could have initialize it over here it does not really matter. So both are equivalent okay, so long as I put I do not redefine divisor over here, then things are fine, divisor is a control variable and I will set its value over here.

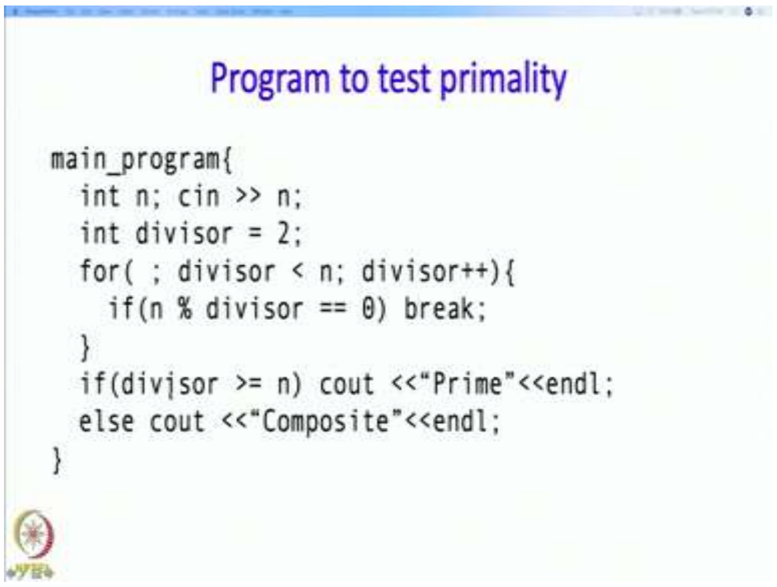
So, what does this loop say? It says that starts with value 2 for divisor and if that value, so long as that value is smaller than n execute this body, after this body is executed, execute this update statement and then go back and repeat. And, at the end of the for divisor now has certain value, so this divisor have already become n, if it is no longer smaller than n, then we know that it must, that we must have tried out all the values between 2 and n minus 1 and so we print out prime okay. Otherwise, we are going to print out composite and we are going to stop.

(Refer Slide Time: 22:31)



Remarks

- We have left the "initialization" part of the for statement empty – this is allowed.
- We could have placed divisor = 2 in the initialization.
- However, we could not have placed "int divisor = 2" in the initialization – then the variable divisor would not be available outside the loop, in the last statement.

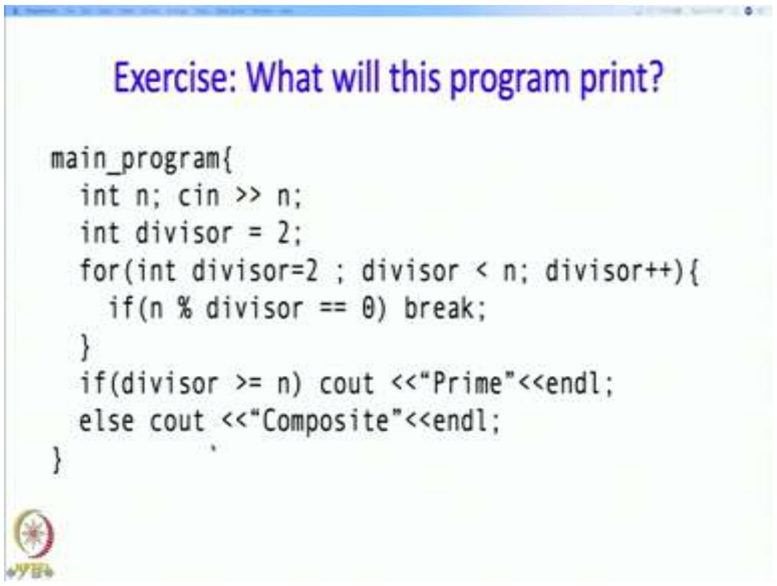


Program to test primality

```
main_program{
  int n; cin >> n;
  int divisor = 2;
  for( ; divisor < n; divisor++){
    if(n % divisor == 0) break;
  }
  if(divisor >= n) cout <<"Prime"<<endl;
  else cout <<"Composite"<<endl;
}
```

So we have left the initialization part of the statement empty and this is allowed as we said earlier and we could have placed divisor equal to 2 in the initialization, but not the n. Okay, and why do we not want n to be placed because we want divisor to be used over here, so that divisor had better be defined earlier, if it is defined here, then we would not be able to use it here.

(Refer Slide Time: 22:59)

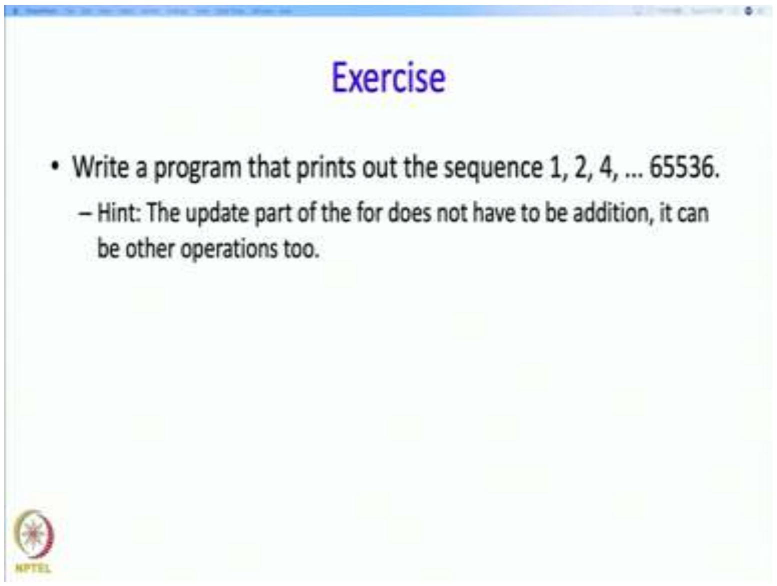


Exercise: What will this program print?

```
main_program{
  int n; cin >> n;
  int divisor = 2;
  for(int divisor=2 ; divisor < n; divisor++){
    if(n % divisor == 0) break;
  }
  if(divisor >= n) cout <<"Prime"<<endl;
  else cout <<"Composite"<<endl;
}
```

Okay, so an exercise for you, suppose we indeed wrote int divisor equal to 2, what would this program do? So this is something for you to think about.

(Refer Slide Time: 23:11)

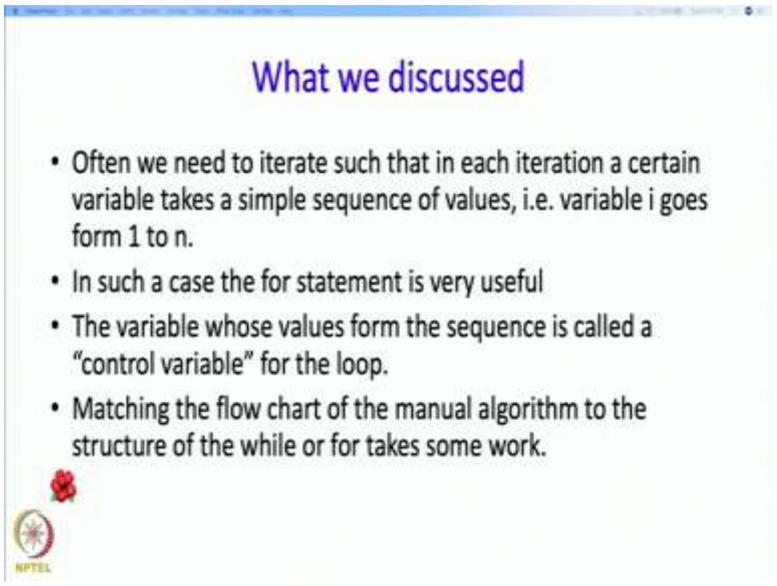


Exercise

- Write a program that prints out the sequence 1, 2, 4, ... 65536.
  - Hint: The update part of the for does not have to be addition, it can be other operations too.

Then here is one more exercise write a for loop which prints out the sequence 1, 2, 4 all the way till 65536, which is simply 2 raise to 16, and the hint given to you is that the update part of the for does not have to be addition, it can be other operations like multiplication for example.

(Refer Slide Time: 23:30)



What we discussed

- Often we need to iterate such that in each iteration a certain variable takes a simple sequence of values, i.e. variable i goes form 1 to n.
- In such a case the for statement is very useful
- The variable whose values form the sequence is called a "control variable" for the loop.
- Matching the flow chart of the manual algorithm to the structure of the while or for takes some work.

Alright, so what did we discussed in this segment? We said that many times we need to iterate such that in each iteration a certain variable takes a simple sequence of values. Say the variable values, variable I goes from 1 to n, in such cases the for statement is very useful. The variable whose values form the sequence is called the control variable, essentially that sequence is driving what is happening in each iteration.

And we also noticed here as well as in while that matching the flowchart of the manual algorithm to the structure of while or for is going to take some amount of work. So we'll stop over here for the segment.