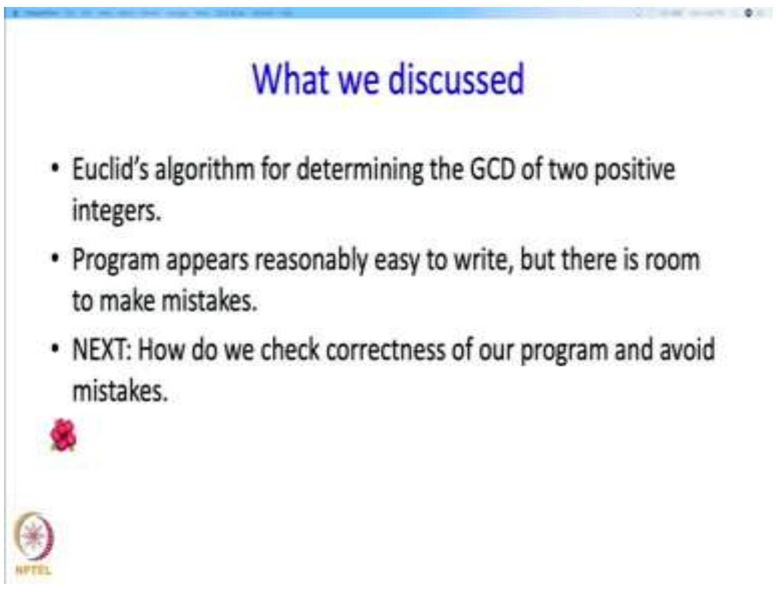
An Introduction to Programming through C++

Professor Abhiram G. Ranade
Department of Computer Science and Engineering
Indian Institute of Technology Bombay

Lecture No. 7 Part – 6
Looping Statements
Correctness proof for GCD

(Refer Time Slide: 0:25)

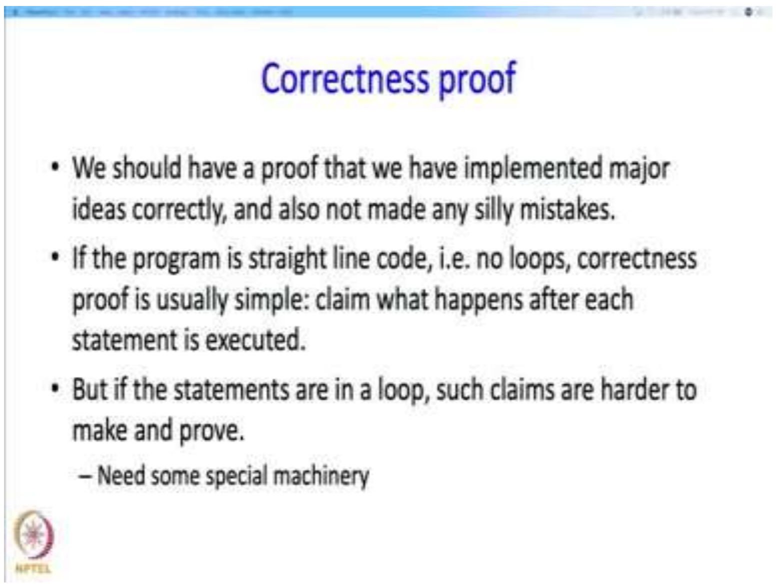


What we discussed

- Euclid's algorithm for determining the GCD of two positive integers.
- Program appears reasonably easy to write, but there is room to make mistakes.
- NEXT: How do we check correctness of our program and avoid mistakes.

In the previous segment we discussed Euclid's algorithm for determining the GCD of two positive integers. And we said that the program is simple, but we also said that it is possible to make mistakes and therefore, it is a good idea to do some cross checks later on to ensure that we did not make any mistakes. So that is what we are going to do in this segment.

(Refer Time Slide: 1:05)

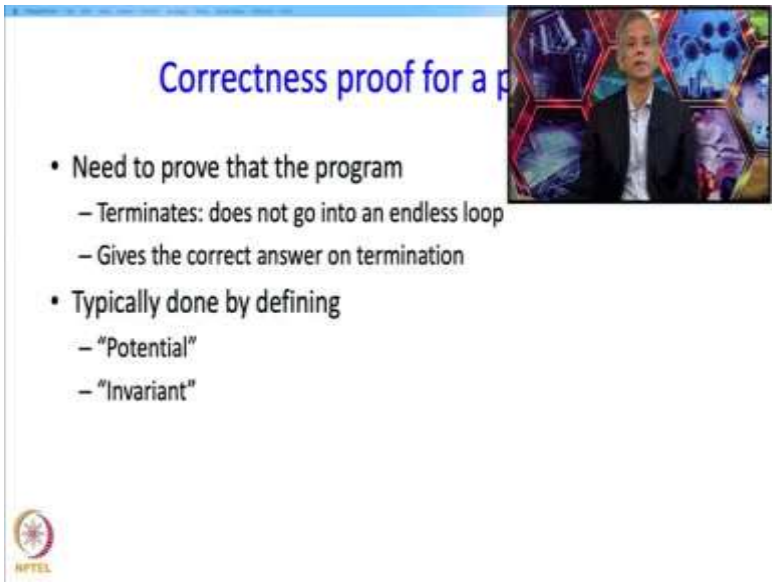


Correctness proof

- We should have a proof that we have implemented major ideas correctly, and also not made any silly mistakes.
- If the program is straight line code, i.e. no loops, correctness proof is usually simple: claim what happens after each statement is executed.
- But if the statements are in a loop, such claims are harder to make and prove.
  – Need some special machinery

So basically we are going to try and produce some kind of a correctness proof. So a correctness proof I guess has two things it does. It sort of says that we have implemented the major ideas correctly, so that Euclid's theorem correctly, but we have not make any silly mistakes. So if the program just happens to be one statement after another without any loops then we can visually check because the loops really sort of confuse the issues right. I mean the loop say oh in the previous iteration what happened in the previous iteration before that what happened and all that. So we have to make sort of precise statements about what happens in each iteration. So therefore, we need to be a little bit careful and we sort of need some special machinery for that.

(Refer Time Slide: 1:37)

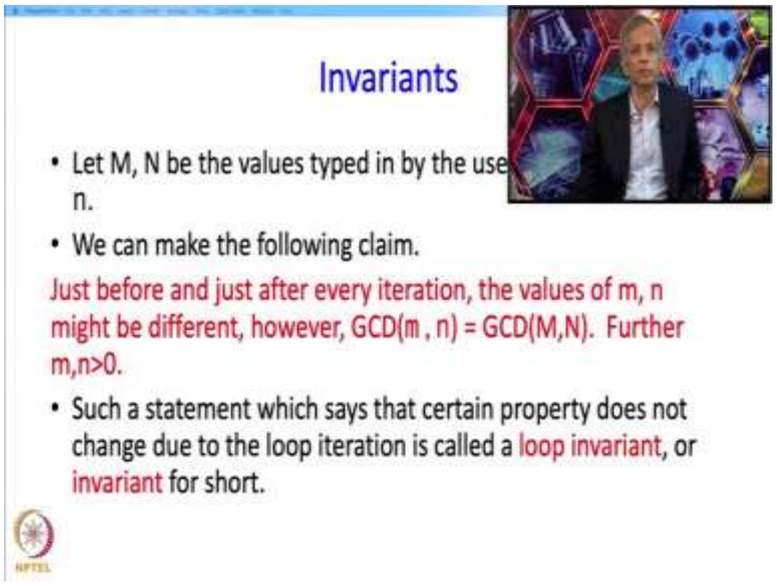


Correctness proof for a p

- Need to prove that the program
  - Terminates: does not go into an endless loop
  - Gives the correct answer on termination
- Typically done by defining
  - "Potential"
  - "Invariant"

Okay, so, how does a correctness proof of a program look like? So, we need to prove that the program must terminate because this is one way in which a program could do something wrong, so a program may go into an endless loop. So we do not want that. And we want that the program should give correct answer when it terminates. So typically this is done by defining something called a potential which tells us that the program terminates and then we also define something called the invariant which says that the program will sort of give the correct answer. So we will see these two things now.

(Refer Time Slide: 2:26)



Invariants

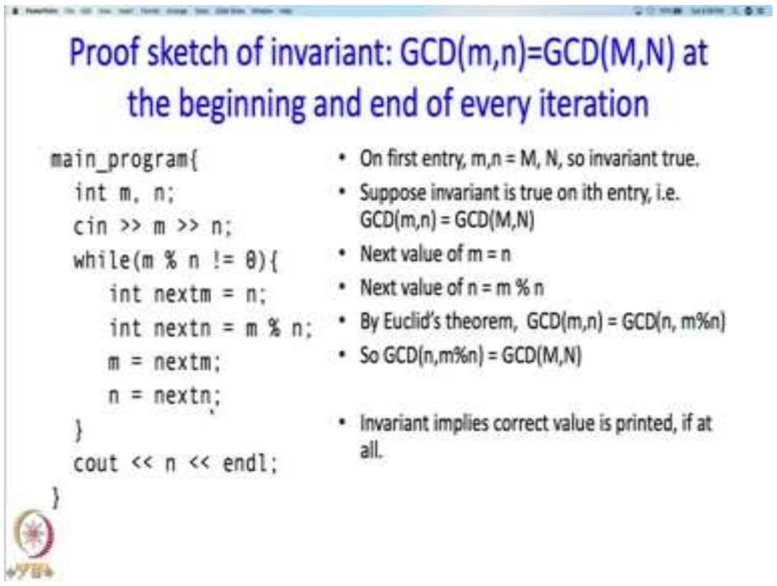
- Let M, N be the values typed in by the use
  n.
- We can make the following claim.

Just before and just after every iteration, the values of m, n might be different, however, GCD(m , n) = GCD(M,N). Further m,n>0.

- Such a statement which says that certain property does not change due to the loop iteration is called a loop invariant, or invariant for short.

So invariants, so let us say that capital M and capital N are the values typed in by the user into the variables little m and little n. Then we would like to make the following claim just before and just after every iteration the values of the variables m and n might be different from iteration to iteration. However, GCD of what is in m and n is exactly the GCD of capital M and capital N which is the GCD of the numbers that we wanted. So this basically says that yes you are allowed to do something to those values m and n, but we are not losing any information so hopefully the numbers are decreasing, but we are not losing any information. So that is what the statement says or this claim says. So such a claim which says that a certain property does not change due to a loop iteration is called a loop invariant or sometimes just invariant.

(Refer Time Slide: 3:55)



Proof sketch of invariant: GCD(m,n)=GCD(M,N) at the beginning and end of every iteration

```
main_program{
  int m, n;
  cin >> m >> n;
  while(m % n != 0){
     int nextm = n;
     int nextn = m % n;
     m = nextm;
     n = nextn;
  }
  cout << n << endl;
}
```

- On first entry, m,n = M, N, so invariant true.
- Suppose invariant is true on ith entry, i.e. GCD(m,n) = GCD(M,N)
- Next value of m = n
- Next value of n = m % n
- By Euclid's theorem, GCD(m,n) = GCD(n, m%n)
- So GCD(n,m%n) = GCD(M,N)

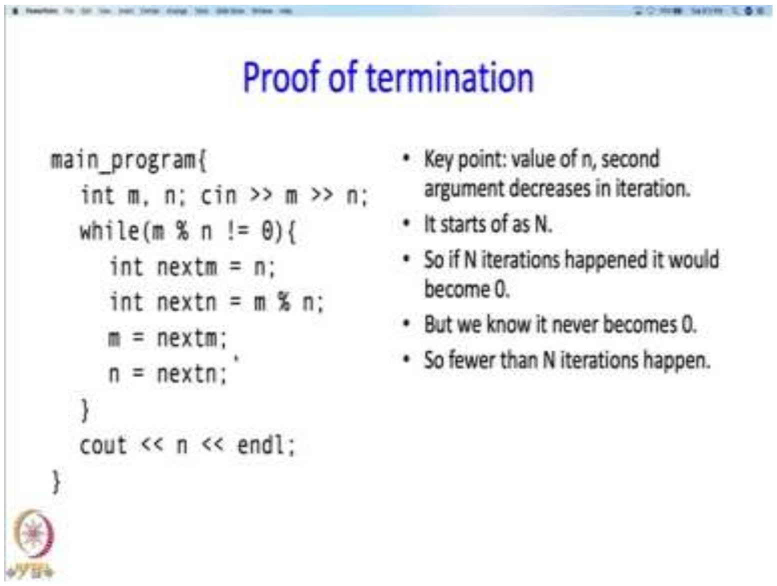
- Invariant implies correct value is printed, if at all.

So now, I am going to give a proof sketch of our invariant. Our invariant was that the GCD of the values contained in the variables m and n is equal to the GCD of capital M and capital N the values typed by the user in the beginning and that this holds at the beginning and end of every iteration. So at end and beginning of every iteration the variables little m and little n contain values whose GCD is the answer that we want. So while the values in m and n changed their GCD remains what we want.

So here is our main program. Now, the way to prove this is to first see what happens on the very first iteration. So on the very first iteration you can see that we are reading in values capital M and capital N into these variables. So when you enter the iteration then clearly m and n will have exactly, the variables m and n will have exactly the values capital M and capital N, so this will be trivially true. So on the first iteration everything is easy. So, so, let us worry about the second iteration or in general the ith iteration because we are going to assume that it is true at the beginning of some ith iteration and we are going to prove that it will hold at the end of the ith iteration. So since we have proved that it is true at the beginning of the first iteration, then what if we prove it for general i the fact that it is true at the beginning of the first iteration will guarantee that it will true at the end of the first iteration, but being true at the end of the first iteration is the same thing as being true at the beginning of the second iteration. So the fact that it is true at the beginning of the second iteration will guarantee that it will be true at the end of the second iteration and so on. So we just have to make this assumption that it is true at the ith entry and from this we have to derive that it must be true on the end of the ith iteration. So let us see what happens in this iteration. So the next value of m this next m is going to be n, the next value of n is going to be m mod n. So note first that by Euclid’s theorem the GCD of the original values in m and n is equal to the next values of n and m mod n. Why? Euclid’s theorem and Euclid’s theorem in the case m mod n is not 0. So why is m mod n is not 0? Well the very fact that we entered over here says that m mod n must not be 0. So by Euclid’s theorem, the GCD of these values must be the GCD of the original contents of m and n. But at the end what are we doing we are placing these values inside m and n so therefore we have proved the invariant holds also at the end. So what we have done? We have proved that the final values of m and n which are these, where GCD is also the same as the original, the GCD of the original values. So we have proved that if the invariant is true at the beginning then the invariant must be true at the end. And therefore, we have proved that it must be true at the beginning and at the end of all iterations.

Now, I want to argue that this invariant actually implies that the correct value is printed in this last statement as well. The correct value is printed over here. Why is that? If the correct value is printed then control comes over here only from this point. So at this point m mod n must have been equal to 0. But if m mod n is equal to 0, then m must divide n not only that n must be greatest divisor of both little m and little n. But the loop invariant assures us that little m and little n have the same divisor as capital M and capital N. Therefore, the divisor the greatest divisor of little m and little n is exactly the answer that we are looking for and that is the answer that we have printed. Of course, so we have only proved right now that the correct answer is printed if control ever gets to this point. So next we have to argue that the control actually gets to this point, so that is the proof of termination.

(Refer Time Slide: 8:54)



Proof of termination

```
main_program{
   int m, n; cin >> m >> n;
   while(m % n != 0){
      int nextm = n;
      int nextn = m % n;
      m = nextm;
      n = nextn;
   }
   cout << n << endl;
}
```

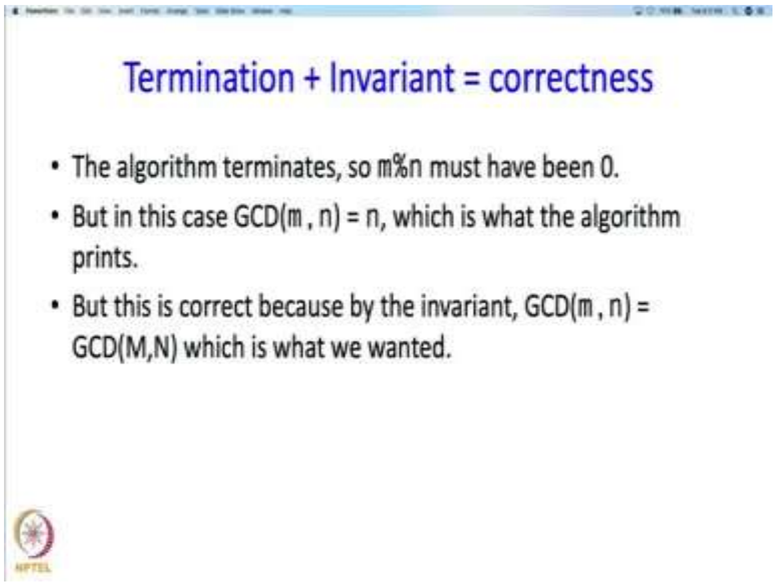
- Key point: value of n, second argument decreases in iteration.
- It starts of as N.
- So if N iterations happened it would become 0.
- But we know it never becomes 0.
- So fewer than N iterations happen.

So again this is our program. And the key point to be noted over here is that the value of n the second argument is always going to decrease. So, why is that? So the second, the next value of the second argument is going to be m mod n. Okay, but m mod n is always smaller than n, the remainder when something is divided by n is always smaller than that number. So therefore, this second argument value is always going to decrease.

So the second argument value starts off as capital N it is going to decrease and it is an integer so it decreases at least by 1. And so if n iterations happen, then what would happen? Then this n would become 0, but we know that it does not become 0. We know that but we should really prove it but that is an easy proof, I will let you figure it out that it never becomes 0. So that means, n iterations can never happen. So that means fewer than n iterations happen or in particular some finite number of iterations happen and this algorithm terminates, alright?

So we have argued that not only does the algorithm prove the correct answer if it terminates, but we have also argued that it terminates and therefore, we know that it terminates and produces the correct answer. Now, I would like you to go back and ask yourself, what happens if these two were exchanged? You would see that then this termination would not happen because the value of n would not decrease. So just a silly mistake would prevent the termination from happening. This is the answer to the problem that was asked on an earlier slide.

(Refer Time Slide: 10:40)

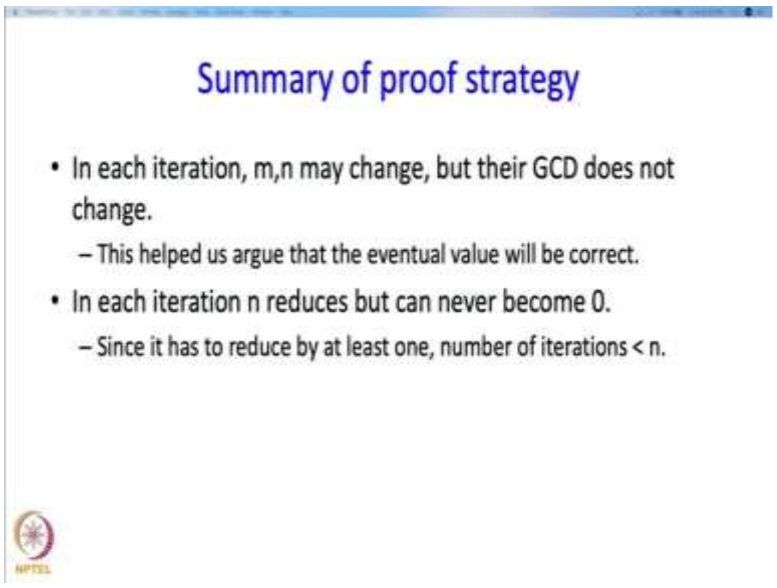


Termination + Invariant = correctness

- The algorithm terminates, so m%n must have been 0.
- But in this case GCD(m , n) = n, which is what the algorithm prints.
- But this is correct because by the invariant, GCD(m , n) = GCD(M,N) which is what we wanted.

So we have said basically that it terminates and it prints the correct answer and therefore, we have proved that it actually terminates and prints the correct answer. So we argued that the program terminates and we also argued that if it terminates it prints the correct answer. So therefore, we can say that it terminates and prints the correct answer. So this is just a writing down of that.

(Refer Time Slide: 10:53)

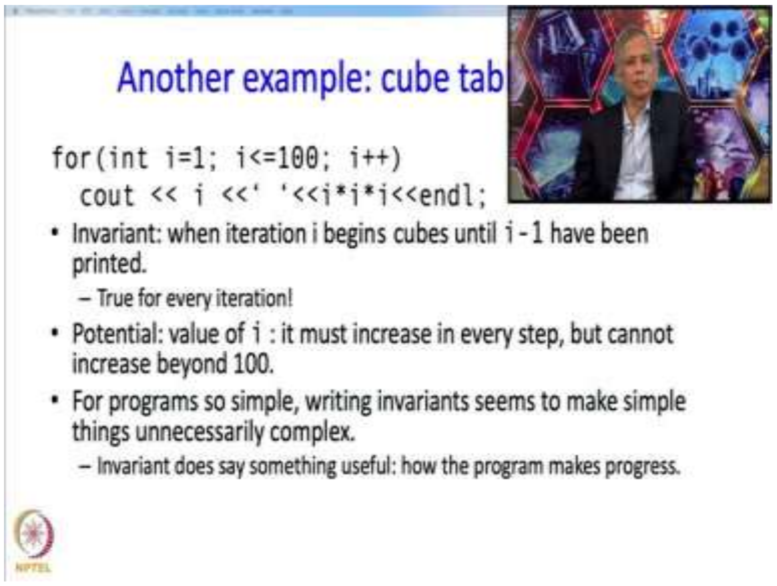


Summary of proof strategy

- In each iteration, m,n may change, but their GCD does not change.
  - This helped us argue that the eventual value will be correct.
- In each iteration n reduces but can never become 0.
  - Since it has to reduce by at least one, number of iterations < n.

So let us summarize, so we claimed that in each iteration, m and n may change but their GCD does not change. So essentially this is saying that means we are not losing any information although we are making the number smaller and smaller. So eventually if we get GCD at any point that will be the correct value and we said that in each iteration the second argument reduces but it can never become 0 and therefore, since it reduces by at least one, the number of iteration has to be smaller than the first value that this variable m takes.

(Refer Time Slide: 11:56)



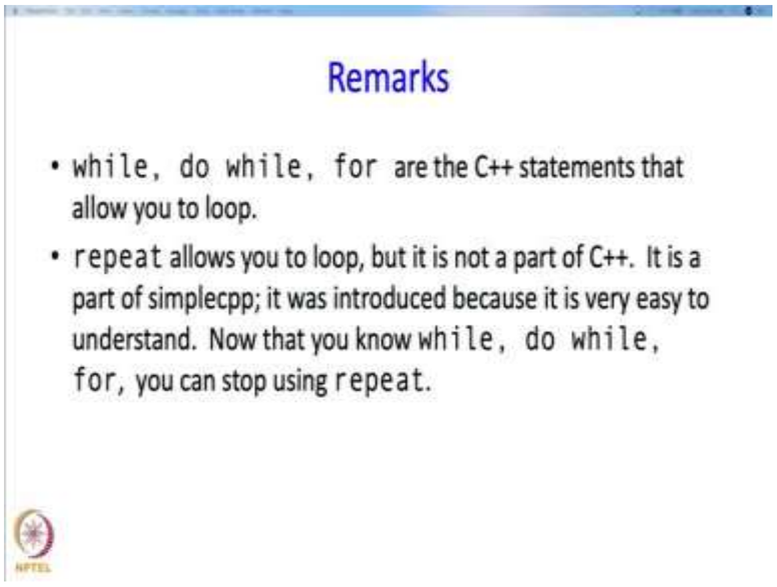
Another example: cube tab

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

- Invariant: when iteration i begins cubes until i - 1 have been printed.
  - True for every iteration!
- Potential: value of i : it must increase in every step, but cannot increase beyond 100.
- For programs so simple, writing invariants seems to make simple things unnecessarily complex.
  - Invariant does say something useful: how the program makes progress.

Now, we had another example of for loop this time which shows the cube table program. Now, you may ask does this idea of proof of correctness, invariant, does it apply to the cube table program as well? So this was our cube table program it is a rather trivial program, so in the first place its correctness it is pretty obvious just by looking at it. We do not really, there is no there is no real room for confusion over here. But we can write an invariant if we really want. And the invariant could be something like this. When iteration i begins, cubes until i minus 1 will have been printed. So that is true and this invariant is actually useful in the sense that it tells us very explicitly what progress we have made until that point. And the potential we could say is the value of I, and that it must increase in every step, but cannot increase beyond y. So the potential cannot beyond i it starts at 1 and therefore right away we know that there are 100 equation, so this part is rather trivial, but this part is still somewhat interesting because it is saying what progress we have made until step i. So as I said for such simple programs invariant seems to make the simple things more complex okay. So it is not recommended that you should write invariants. But the invariant does say something useful. So you should, you should have it in mind that it does make progress. So while formally an invariant is not needed, but it does say something interesting.

(Refer Time Slide: 13:44)

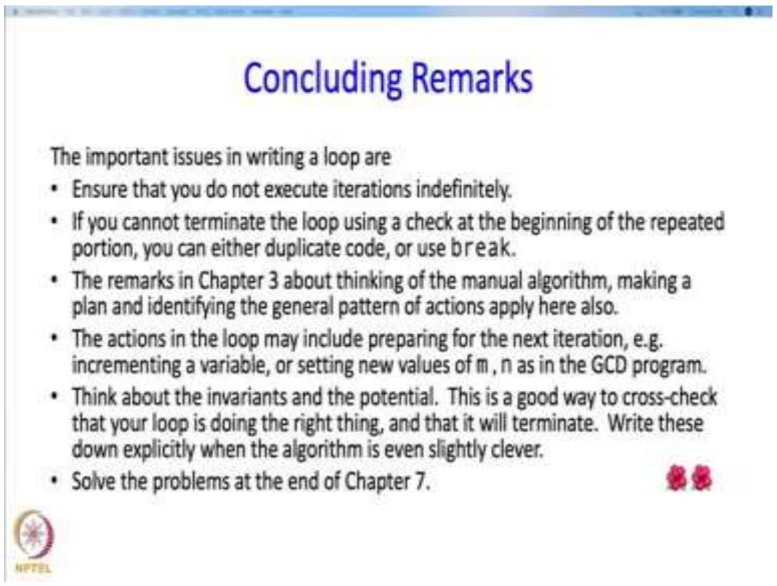


Remarks

- `while, do while, for` are the C++ statements that allow you to loop.
- `repeat` allows you to loop, but it is not a part of C++. It is a part of simplecpp; it was introduced because it is very easy to understand. Now that you know `while, do while, for,` you can stop using `repeat`.

Okay, so some concluding remarks. So we have looked at while, do while and for, which are the C++ statements that allow you to loop. Repeat allows you to loop, but it is not a part of C++, it is a part of simplecpp. And we introduced it because it is very easy to understand. But now that you know while, do while, for, you should stop using repeat, for one thing because you want to learn the C++ statements and another because the C++ statements are more general and more powerful and they can do things which the repeat cannot do.

(Refer Time Slide: 14:30)



Concluding Remarks

The important issues in writing a loop are
- Ensure that you do not execute iterations indefinitely.
- If you cannot terminate the loop using a check at the beginning of the repeated portion, you can either duplicate code, or use `break`.
- The remarks in Chapter 3 about thinking of the manual algorithm, making a plan and identifying the general pattern of actions apply here also.
- The actions in the loop may include preparing for the next iteration, e.g. incrementing a variable, or setting new values of `m` , `n` as in the GCD program.
- Think about the invariants and the potential. This is a good way to cross-check that your loop is doing the right thing, and that it will terminate. Write these down explicitly when the algorithm is even slightly clever.
- Solve the problems at the end of Chapter 7.

So, some more concluding remarks. The important issues in writing a loop are, ensure that you do not execute iterations indefinitely. If you cannot terminate the loop using a check at the beginning of the repeated portion, you can either duplicate the code or you can use a break. So some work is needed in order to match the structure of your manual algorithm which might have that condition checking step at different places so that it matches with whatever statements you write.

The remarks that we made in chapter 3 about thinking about the manual algorithm. Making a plan and identifying the general pattern of action apply here also. And in some sense what is happening over here, the invariants and the termination are sort of the part of the plan. So when you write a loop you should have something like the invariant in your mind which is kind of a plan. Then the actions in the loop may include preparing for the next iteration. So incrementing the variable or setting new values of m and n as in the GCD program. So typically, the loop may have some initial values of the variables, then maybe you do something with the variables like may be print them and then you change the value so that they are ready for the next iteration that is they satisfy whatever, if invariant is needed for the next iteration.

Thinking about invariants and potential is a good thing in the sense that you are checking that your loop is doing the right thing and that it will terminate. I would suggest that you should write these down when the algorithm is even slightly clever. You should think to yourself look am I completely sure that this loop is going to terminate? So for that, defining something like a potential is a really good idea. And it is really good idea also to write invariants because they tell you what your program is doing. So if you make a mistake, if you have a bug, then the invariant and the potential will help you find that bug. So the potential will tell you why is my program not terminating, the invariant might tell you why is my program not giving the correct answer, so at some point quite presumably quite possibly the invariant may be violated by your program so you want some property to hold, but while writing the program you made a silly mistake and that property is not holding any longer.

Now, the proof of learning to program is in writing lots of program so at the end of chapter 7 many problems are given. So I will strongly encourage you to write programs corresponding to these, to those problems and with that I will conclude this lecture segment, thank you.