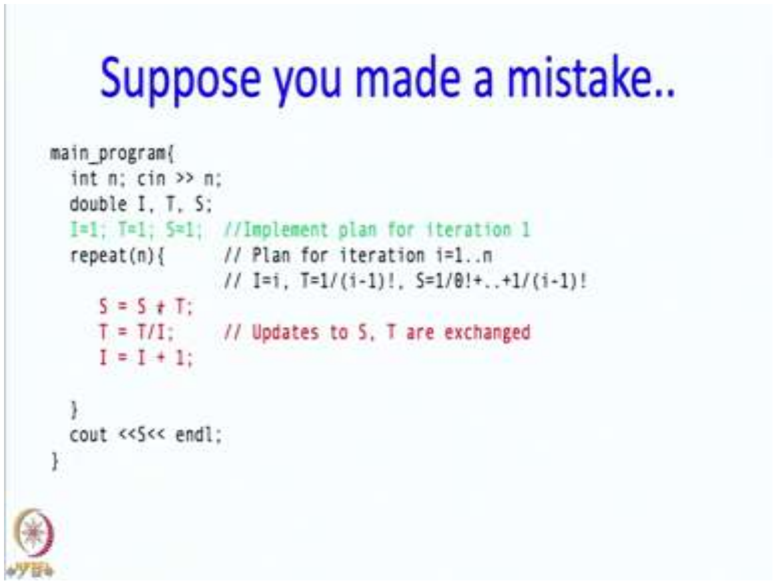
An Introduction to Programming through C++

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

Lecture No. 4 Part - 3
Program Design
Debugging

In the last segment, we looked at how to translate manual algorithms into programs. We ran one program and in this segment we are going to see how we might recover if we make some mistake in writing the program.

(Refer Slide Time: 00:38)



Suppose you made a mistake..

```
main_program{
  int n; cin >> n;
  double I, T, S;
  I=1; T=1; S=1;  //Implement plan for iteration 1
  repeat(n){      // Plan for iteration i=1..n
                  // I=i, T=1/(i-1)!, S=1/0!+..+1/(i-1)!
     S = S + T;
     T = T/I;     // Updates to S, T are exchanged
     I = I + 1;

  }
  cout <<S<< endl;
}
```

Okay, so here is an example. So this is the same program from last time except that these two statements, so this statement and this statement their order has been exchanged. So in the real program this statement came later and this statement came earlier, so we have exchanged it, exchange the order.

(Refer Slide Time: 01:06)

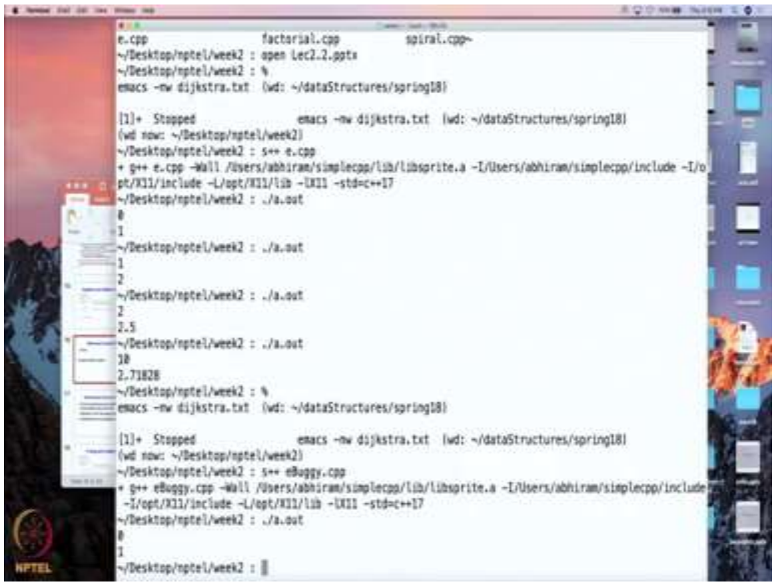
Now you run and find..

- Demo

Program made a mistake!

Okay, so first of all this is a bad program, it will not give the correct answer and let us see that.

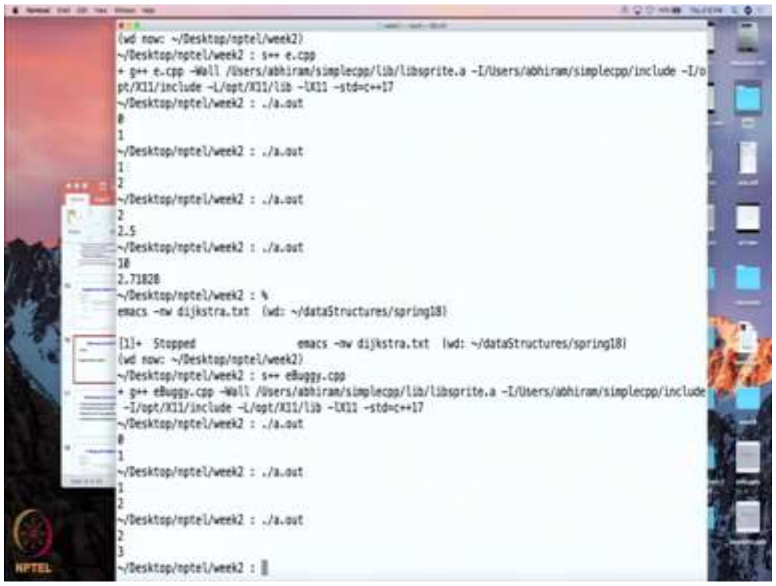
(Refer Slide Time: 01:15)



```
e.cpp                  factorial.cpp          spiral.cpp~
~/Desktop/nptel/week2 : open Lec2.2.pptx
~/Desktop/nptel/week2 : %
emacs -nw dijkstra.txt  (wd: ~/dataStructures/spring18)

[1]+  Stopped                 emacs -nw dijkstra.txt  (wd: ~/dataStructures/spring18)
(wd now: ~/Desktop/nptel/week2)
~/Desktop/nptel/week2 : s++ e.cpp
+ g++ e.cpp -Wall /Users/abhiram/simplecpp/lib/libsprite.a -I/Users/abhiram/simplecpp/include -I/o
pt/X11/include -L/opt/X11/lib -lX11 -std=c++17
~/Desktop/nptel/week2 : ./a.out
0
1
~/Desktop/nptel/week2 : ./a.out
1
2
~/Desktop/nptel/week2 : ./a.out
2
2.5
~/Desktop/nptel/week2 : ./a.out
10
2.71828
~/Desktop/nptel/week2 : %
emacs -nw dijkstra.txt  (wd: ~/dataStructures/spring18)

[1]+  Stopped                 emacs -nw dijkstra.txt  (wd: ~/dataStructures/spring18)
(wd now: ~/Desktop/nptel/week2)
~/Desktop/nptel/week2 : s++ eBuggy.cpp
+ g++ eBuggy.cpp -Wall /Users/abhiram/simplecpp/lib/libsprite.a -I/Users/abhiram/simplecpp/include
 -I/opt/X11/include -L/opt/X11/lib -lX11 -std=c++17
~/Desktop/nptel/week2 : ./a.out
0
1
~/Desktop/nptel/week2 : 
```

So here is the program in which the value of the statements has been exchanged. So I have called it e-buggy.CPP. So let us compile that program. So it compiles fine. So now, let us run it. So let us try our test case 0 so this seems to run fine.

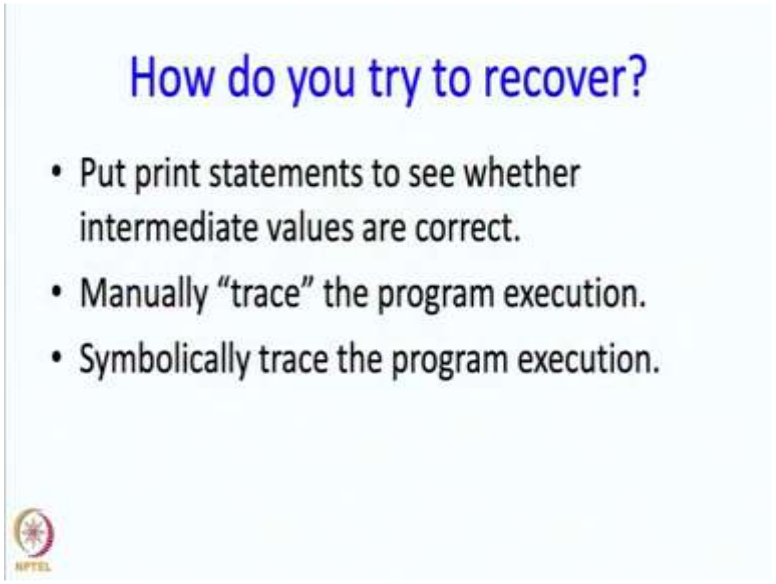
(Refer Slide Time: 01:55)



```
(wd now: ~/Desktop/nptel/week2)
~/Desktop/nptel/week2 : s++ e.cpp
+ g++ e.cpp -Wall /Users/abhiram/simplecpp/lib/libsprite.a -I/Users/abhiram/simplecpp/include -I/o
pt/X11/include -L/opt/X11/lib -lX11 -std=c++17
~/Desktop/nptel/week2 : ./a.out
0
1
~/Desktop/nptel/week2 : ./a.out
1
2
~/Desktop/nptel/week2 : ./a.out
2
2.5
~/Desktop/nptel/week2 : ./a.out
10
2.71828
~/Desktop/nptel/week2 : %
emacs -nw dijkstra.txt  (wd: ~/dataStructures/spring18)

[1]+  Stopped                 emacs -nw dijkstra.txt  (wd: ~/dataStructures/spring18)
(wd now: ~/Desktop/nptel/week2)
~/Desktop/nptel/week2 : s++ eBuggy.cpp
+ g++ eBuggy.cpp -Wall /Users/abhiram/simplecpp/lib/libsprite.a -I/Users/abhiram/simplecpp/include
 -I/opt/X11/include -L/opt/X11/lib -lX11 -std=c++17
~/Desktop/nptel/week2 : ./a.out
0
1
~/Desktop/nptel/week2 : ./a.out
1
2
~/Desktop/nptel/week2 : ./a.out
2
3
~/Desktop/nptel/week2 : 
```

Let us try our test case 1, oh this also seems to run fine. So maybe well, let us see. Let us try our test case 2, so here there is a mistake. We were expecting a value 2.5 but we have got the value 3 okay all right. So, how do you recover? Well, there are three possible courses of action that you can take and you really should learn all three because once in a while one will be more useful than the other.

(Refer Slide Time: 02:29)



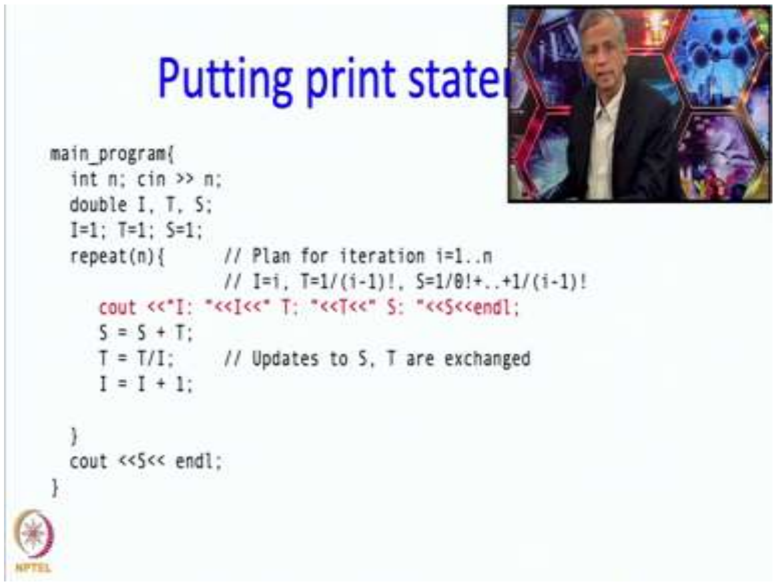
How do you try to recover?

- Put print statements to see whether intermediate values are correct.
- Manually "trace" the program execution.
- Symbolically trace the program execution.

Sort of the simplest thing to do is to put print statements to see whether not just the final value but whether the intermediate values are correct. What I mean by that is, you put print statements the program will print out the intermediate values and then you go over the intermediate values and you check are these the values that I was expecting or is something wrong?

Another possibility is not to put print statements, but do the same thing yourself. So you pretend that you are the computer you walk over the program and on the side keep writing what values different variables are and at the same time keep on reasoning - are these the values you expect to see? And if not, then you have found a mistake. And the last way is to do the same thing but in a symbolic manner. So rather than saying variable capital T has the value 1, we are going to say what if the variable capital T has the value 1/I! and we are going to symbolically try and reason out and see if everything is correct.

(Refer Slide Time: 03:43)

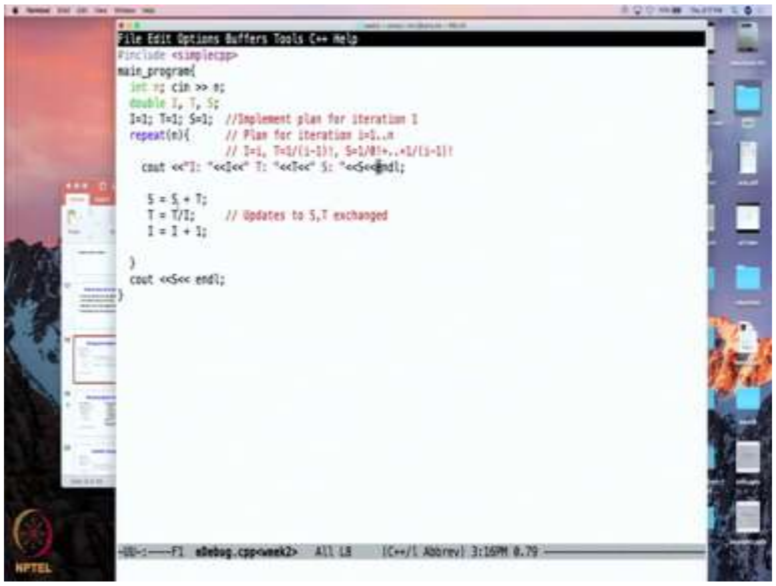


Putting print state

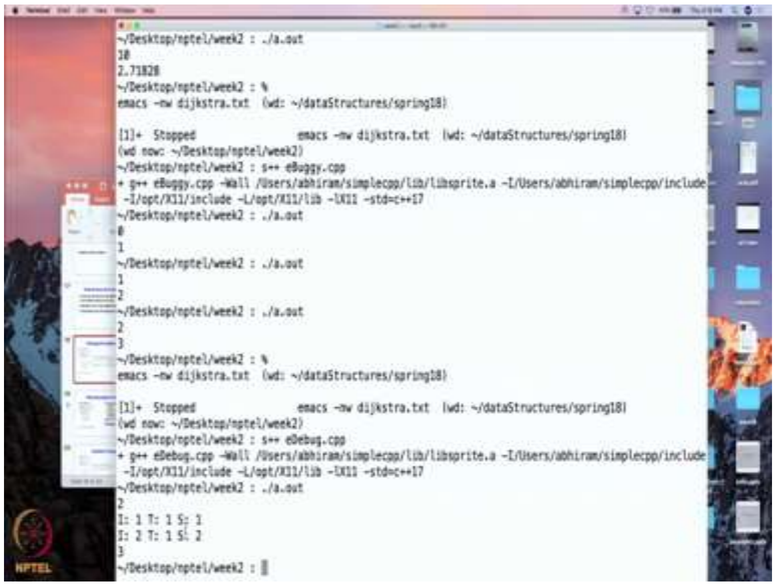
```
main_program{
  int n; cin >> n;
  double I, T, S;
  I=1; T=1; S=1;
  repeat(n){      // Plan for iteration i=1..n
                  // I=i, T=1/(i-1)!, S=1/0!+..+1/(i-1)!
     cout <<"I: "<<I<<" T: "<<T<<" S: "<<S<<endl;
     S = S + T;
     T = T/I;     // Updates to S, T are exchanged
     I = I + 1;

  }
  cout <<S<< endl;
}
```

Okay so how, what do I mean when I say I put print statements? Well this last this red line okay, so we should remember that as it is usually convenient to put print statements at the beginning of the loop. So that is because our plan has also been made for the beginning of the loop. So we can look at the values get that get printed and we can look at the plan and we can see whether everything is fine. So our plan contains variables capital I, T and S. So those are the variables that we are going to print okay so let us again see this in action.

(Refer Slide Time: 04:17)



```
File Edit Options Buffers Tools C++ Help
#include <simplecpp>
main_program{
  int n; cin >> n;
  double I, T, S;
  I=1; T=1; S=1;  //Implement plan for iteration 1
  repeat(n){      // Plan for iteration i=1..n
                  // I=i, T=1/(i-1)!, S=1/0!+..+1/(i-1)!
   cout <<"I: "<<I<<" T: "<<T<<" S: "<<S<<endl;

    S = S + T;
    T = T/I;      // Updates to S,T exchanged
    I = I + 1;

  }
  cout <<S<< endl;
}
-UU-:----F1  eDebug.cpp<week2>   All L8    (C++/l Abbrev) 3:16PM 0.79
```



```
~/Desktop/nptel/week2 : ./a.out
10
2.71828
~/Desktop/nptel/week2 : %
emacs -nw dijkstra.txt  (wd: ~/dataStructures/spring18)

[1]+  Stopped                 emacs -nw dijkstra.txt  (wd: ~/dataStructures/spring18)
(wd now: ~/Desktop/nptel/week2)
~/Desktop/nptel/week2 : s++ eBuggy.cpp
+ g++ eBuggy.cpp -Wall /Users/abhiram/simplecpp/lib/libsprite.a -I/Users/abhiram/simplecpp/include
 -I/opt/X11/include -L/opt/X11/lib -lX11 -std=c++17
~/Desktop/nptel/week2 : ./a.out
0
1
~/Desktop/nptel/week2 : ./a.out
1
2
~/Desktop/nptel/week2 : ./a.out
2
3
~/Desktop/nptel/week2 : %
emacs -nw dijkstra.txt  (wd: ~/dataStructures/spring18)

[1]+  Stopped                 emacs -nw dijkstra.txt  (wd: ~/dataStructures/spring18)
(wd now: ~/Desktop/nptel/week2)
~/Desktop/nptel/week2 : s++ eDebug.cpp
+ g++ eDebug.cpp -Wall /Users/abhiram/simplecpp/lib/libsprite.a -I/Users/abhiram/simplecpp/include
 -I/opt/X11/include -L/opt/X11/lib -lX11 -std=c++17
~/Desktop/nptel/week2 : ./a.out
2
I: 1 T: 1 S: 1
I: 2 T: 1 S: 2
3
~/Desktop/nptel/week2 :
```

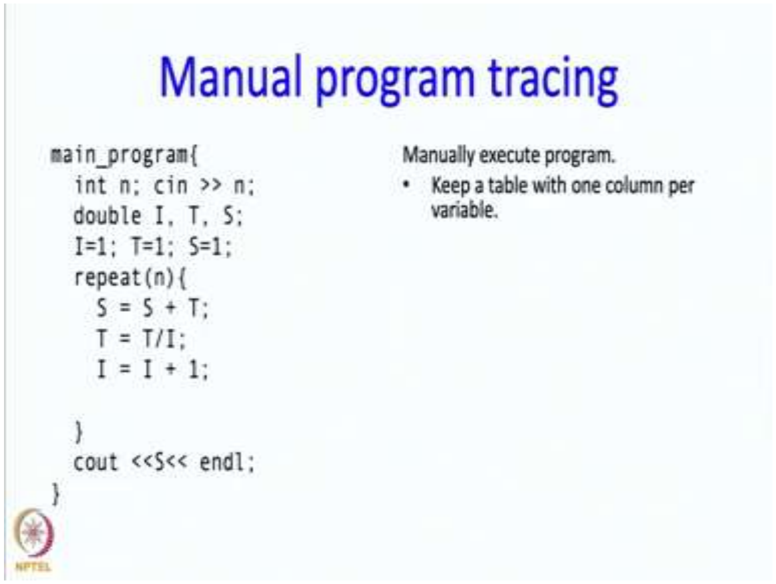
So I have put in the statement in a program called eDebug. So this here is here is the line where the print statements are there okay. So let us now compile and execute this. Okay, compiled and let us say run. We know that this program makes a mistake for the value 2, so let us see what happens over there. So if we print 2 we are going to see the intermediate values during the iterations. Okay so here are the intermediate values. So the first time around when I was 1, T was 1, S was 1 okay this is as we expect, okay this is the very first iteration and in fact at this point the variables just have the values that we gave to them.

In the next iteration, when capital I is 2, we expect the term to be the previous term that got added and that was 1/1!. So in fact it is 1 so this is perfectly good. Then we expect capital S the sum at the beginning of the second iteration to have the value 1/0! plus terms until we get the term 1/(2-1)! or 1/1!. So there are just two terms 1/0! and 1/1!. So the value is going to be 2 which is also what we have expected.

So this means that there really was no error as far as the values were concerned in our variables until the beginning of iteration 2. So let us just go back to the program. Okay so we have discovered that this value is printed correctly and the values printed over here are correct and therefore, something must be wrong over here in this statement because until now S was correct, at this point S was wrong, so something is wrong over here. So if this statement is wrong, then we are pretty close to the mistake and then all that we need to check is are we adding the right thing over here?

So here we are adding 1 whereas we expect to add ½! to this, not 1/1! and therefore, we can see that this is definitely wrong and now we can realize that look, this statement should come before this and then everything will be fine. Okay, so we discovered what the error was and by the way errors are called bugs in computer parlance at times and so we have figured out what the bug is and this is also saying that we have debugged the program.

(Refer Slide Time: 07:29)



Manual program tracing

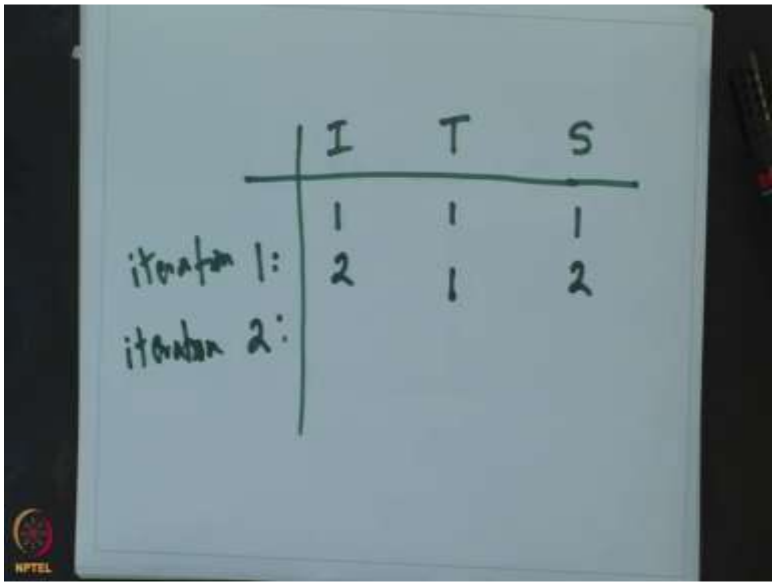
```
main_program{
  int n; cin >> n;
  double I, T, S;
  I=1; T=1; S=1;
  repeat(n){
    S = S + T;
    T = T/I;
    I = I + 1;

  }
  cout <<S<< endl;
}
```

Manually execute program.

- Keep a table with one column per variable.

Another way of debugging the program is through manual program tracing. So here is our same program and now we are going to pretend that we are a computer and we are going to execute it, execute the program and keep track of what values the different variables have.

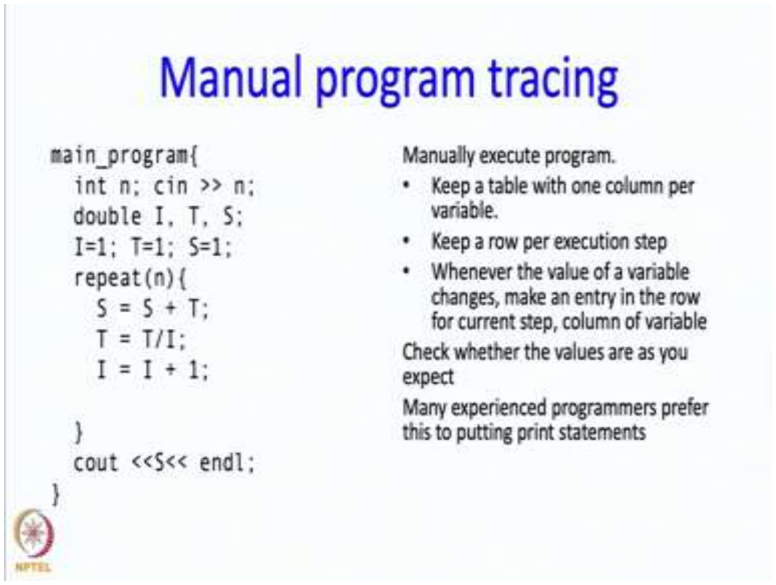
(Refer Slide Time: 07:40)



	I	T	S
	1	1	1
iteration 1:	2	1	2
iteration 2:			

So to facilitate this it is often recommended that you make a table. So you have a column for every variable. So let us not keep a column for n because n is going to be fixed once and for all and it is not going to change. So let us keep a column for I, let us keep a column for T, let us keep a column for S. So when we execute the first statement, which is this here, then these variables get the value 1, 1, 1 okay then when we go to repeat n so let us say iteration 1. What happens during that iteration 1? So first we are going to change S to S plus T, so as a result we are going to get the value 2 over here. After that we are going to do T equals T upon I, so T equals T upon I so we will get the value 1 over here and then we will have I=I+1, so we will get the value 2 over here. Then we are going to go to iteration 2. So in this way we continue and will realize that at some point the values that we have over here are different from the values that we expect. So again at that point we proceed in the way we did in the print statement case.

(Refer Slide Time: 09:09)



Manual program tracing

```
main_program{
  int n; cin >> n;
  double I, T, S;
  I=1; T=1; S=1;
  repeat(n){
    S = S + T;
    T = T/I;
    I = I + 1;

  }
  cout <<S<< endl;
}
```

Manually execute program.

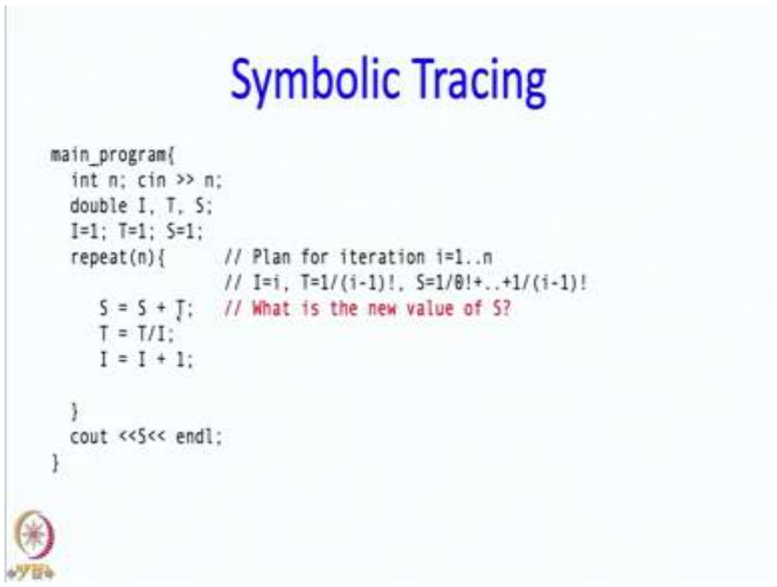
- Keep a table with one column per variable.
- Keep a row per execution step
- Whenever the value of a variable changes, make an entry in the row for current step, column of variable

Check whether the values are as you expect

Many experienced programmers prefer this to putting print statements

So once we once we do that, we will again figure out that oh here is the mistake and then we can remove it okay. Now, many experienced programmers prefer doing this manually rather than putting print statements because somehow it feels that as you do this, you are getting to know your program better and sometimes the print statements sort of are too confusing, there are too many things that happen at once and yeah so it really depends on your temperament, but you should definitely consider this.

(Refer Slide Time: 09:43)

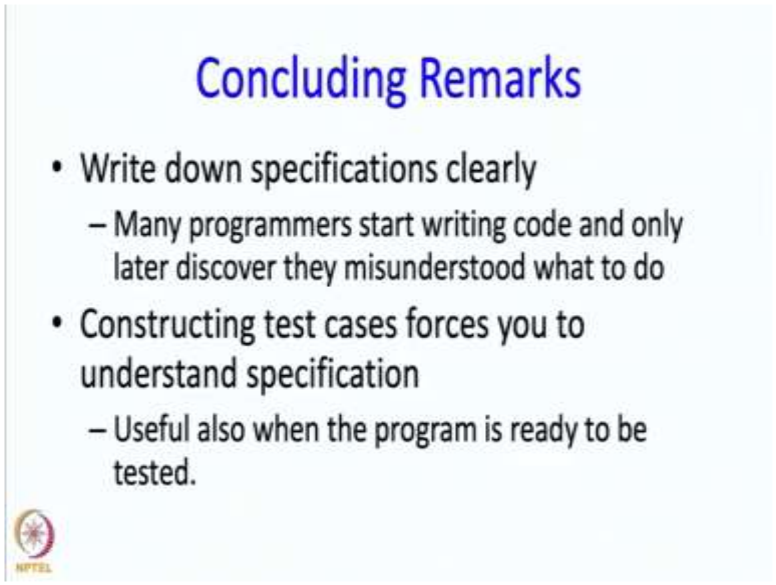


Symbolic Tracing

```
main_program{
  int n; cin >> n;
  double I, T, S;
  I=1; T=1; S=1;
  repeat(n){      // Plan for iteration i=1..n
                  // I=i, T=1/(i-1)!, S=1/0!+..+1/(i-1)!
     S = S + T;   // What is the new value of S?
     T = T/I;
     I = I + 1;

  }
  cout <<S<< endl;
}
```

Then the last and by no means the least important method is symbolic tracing. So again it is the same program and the question that we are going to ask is let us assume that things have worked out until this point. Well, we can check whether they have worked out at the first iteration. So, how do we check that? We look at I equals 1, T equals 1, S equals 1. So that in fact has worked out according to plan because I, T and S have exactly the values recommended over here. Now, we are going to check symbolically whether we did the right thing that is whether our updates are in fact doing what we want them to do. So, what is this update doing? It is taking the old value of S and adding it to the value T. So we are going to we are assuming that the T value is this, the S value is this. So now, if we add this value to this value what happens? Oh something wrong. The last time over here was 1/(I-1)! and we are adding 1/(I-1)! to it, whence we should have added 1/I!. So clearly we have made a mistake and again, if you think a little you will realize that oh we really should have divided first so this statement should come first all right.

(Refer Slide Time: 11:06)

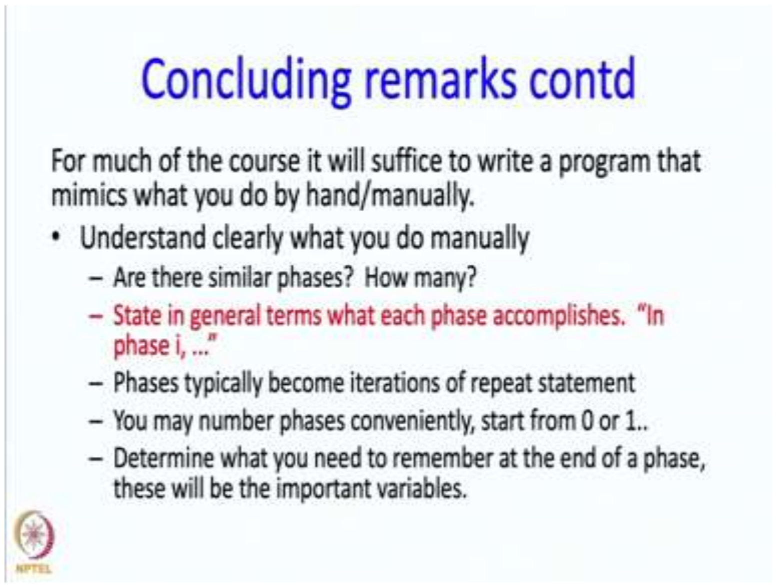


Concluding Remarks

- Write down specifications clearly
  - Many programmers start writing code and only later discover they misunderstood what to do
- Constructing test cases forces you to understand specification
  - Useful also when the program is ready to be tested.

Okay, so let me conclude, so first I want to say that you must write down specifications clearly. In this particular program things were quite simple, but if you take a real life problem, the experience is that in many programming projects there are delays and people write wrong code because they did not understand what was required of them. They did not understand what the specification was. So please take the time to make sure that you are understanding clearly what is expected of you. Then you should construct test cases because first of all they are useful when the program is ready, but because they also are an indication that you are actually understanding what is expected of you.

(Refer Slide Time: 12:00)



Concluding remarks contd

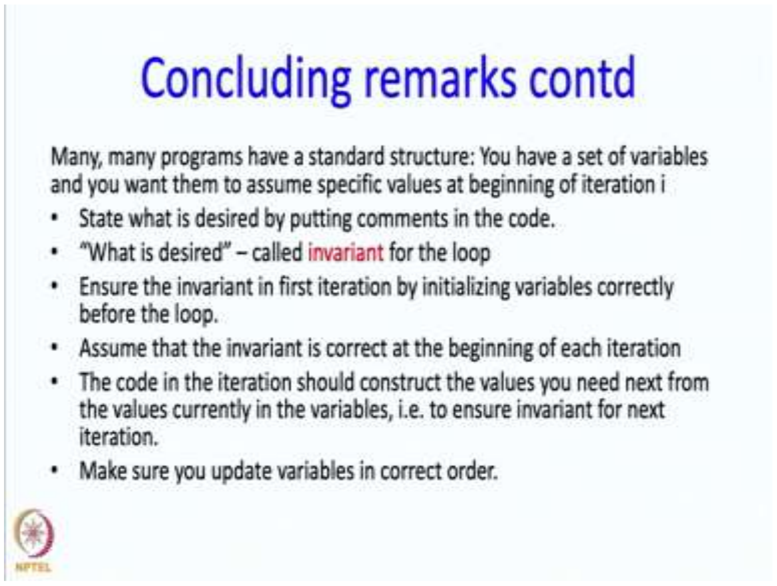
For much of the course it will suffice to write a program that mimics what you do by hand/manually.

- Understand clearly what you do manually
  - Are there similar phases? How many?
  - State in general terms what each phase accomplishes. "In phase i, ..."
  - Phases typically become iterations of repeat statement
  - You may number phases conveniently, start from 0 or 1..
  - Determine what you need to remember at the end of a phase, these will be the important variables.

For much of this course, it will be sufficient for you to write a program that mimics what you do by hand or what you do manually. For this you have to be quite conscious, you have to be very-very intimately knowing what exactly you do manually. So think about what you do and think about what structure it has. So for example you may ask, when I am doing solving this problem I am doing these phases which look very similar. So you might ask how many phases? Because that is going to tell you that oh I should use a repeat Loop and it should have so many iterations.

You should also state in general terms what each phrase accomplishes. So in terms of that phrase number, so you should say something like phase number, in phase I, I am going to take the term calculated in phase I minus 1 and whatever okay so it had better be general because generality is what helps you write programs and as I said phases become iterations of the repeat statement and you should note that you may want to number the phases, so you can number them conveniently. In our case we started with fun but there may be other situations where may, it may want to start with 0 and you should determine what you need at the end of the phase or at the beginning of the phase. So the end and beginning are really very similar because the end of phase I is the beginning of a phase I plus 1. So in any case you need to know what it is that you need to remember at that point and those - the things that you need to remember will need variables.

(Refer Slide Time: 13:58)



Concluding remarks contd

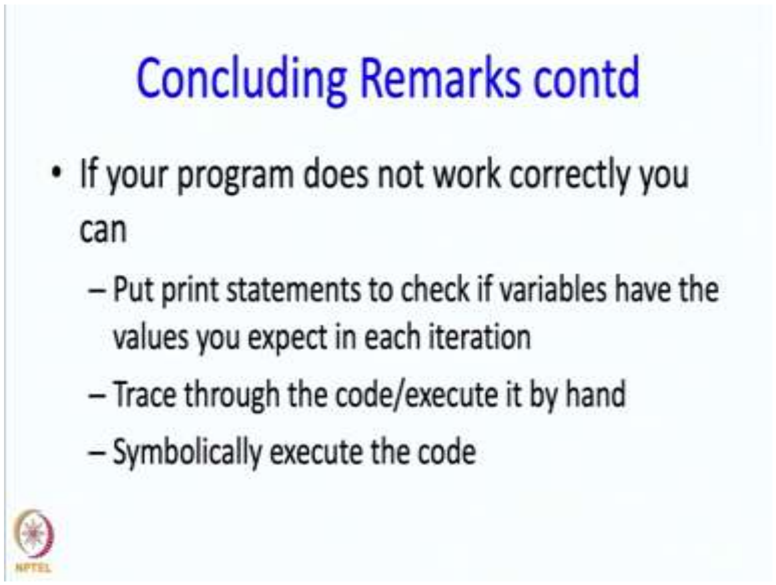
Many, many programs have a standard structure: You have a set of variables and you want them to assume specific values at beginning of iteration i

- State what is desired by putting comments in the code.
- "What is desired" – called invariant for the loop
- Ensure the invariant in first iteration by initializing variables correctly before the loop.
- Assume that the invariant is correct at the beginning of each iteration
- The code in the iteration should construct the values you need next from the values currently in the variables, i.e. to ensure invariant for next iteration.
- Make sure you update variables in correct order.

Okay, so the structure that we had in this program is actually very common. So what is that structure? You have a set of variables and you want them to assume specific values at the beginning of iteration I, where I is anything, I could be anything for all possible values of I you know what values they need to take. So this knowledge should be put in the code in form of comments, and this knowledge, what values are desired is often called the invariant for the loop, okay. So for all loop iterations this statement has to be true and therefore it is called an invariant.

You can ensure that the invariant is true for the first iteration by initializing variables correctly before you enter the loop. Now, for the subsequent iterations, you can assume that the invariant is correct at the beginning and then you should change the values of the variables so that it becomes correct at the end, okay, so that is what the code in each iteration is supposed to be doing and to get this right it is good to be careful, it is good to ask-look I want the value of this variable to be divided by this variable. Now, it could be the value at the beginning, the value at the end which value am I supposed to use? So be very careful about that and then it will make sure that your statements, your updates will happen in the right order.

(Refer Slide Time: 15:45)

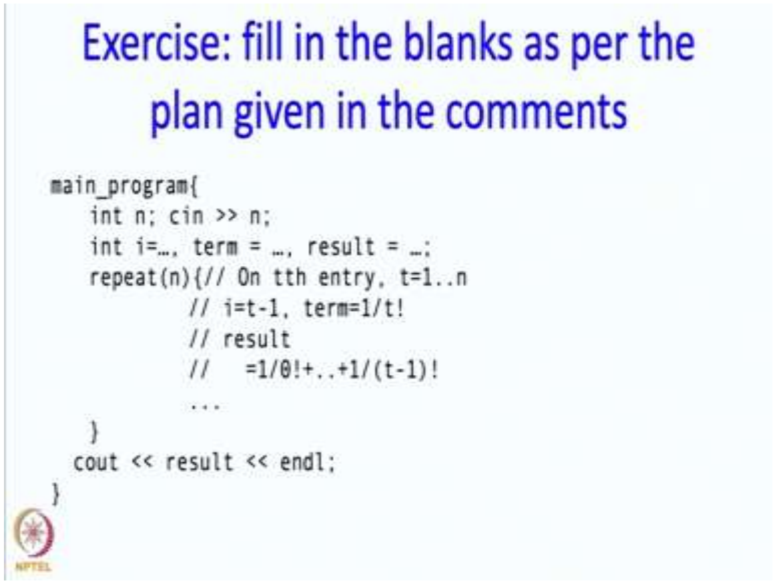


Concluding Remarks contd

- If your program does not work correctly you can
  - Put print statements to check if variables have the values you expect in each iteration
  - Trace through the code/execute it by hand
  - Symbolically execute the code

If you write a program and it does not work correctly, you should not give up hope, you can make sure that it runs, you can attempt to remove the bug, the error so for this you can put print statements to check with the variables have the values you expect in each iteration. You can trace or manually execute your code or you can symbolically execute your code.

(Refer Slide Time: 16:14)

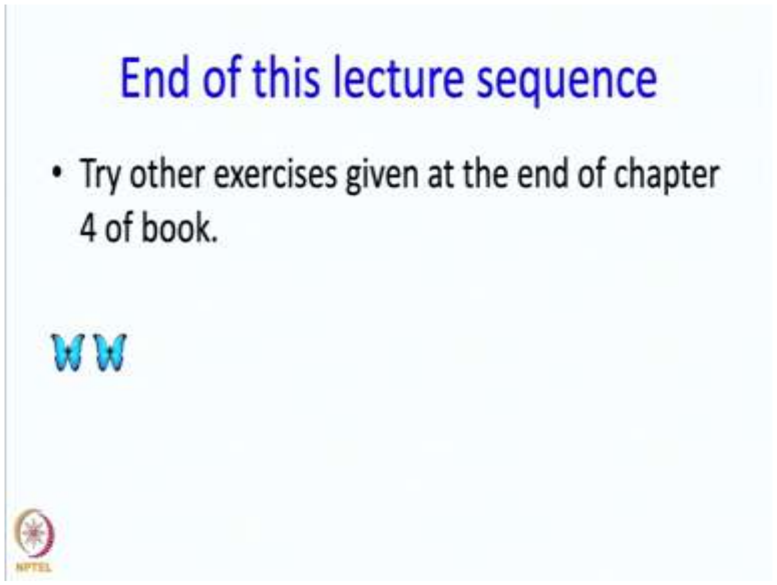


Exercise: fill in the blanks as per the plan given in the comments

```
main_program{
   int n; cin >> n;
   int i=…, term = …, result = …;
   repeat(n){// On tth entry, t=1..n
          // i=t-1, term=1/t!
          // result
          //   =1/0!+..+1/(t-1)!
          ...
   }
  cout << result << endl;
}
```

Now, I am going to leave you with an exercise and this is again the same series that you are supposed to calculate, except the plan or the invariant given to you is slightly different over here okay. So the invariant is stated in those comments and you are required to complete the initialization and the loop steps and make the program work correctly.

(Refer Slide Time: 16:41)



End of this lecture sequence

- Try other exercises given at the end of chapter 4 of book.

Okay, so this is the end of this lecture sequence and I will strongly urge you to try the other exercises given at the end of chapter 4, which is the reading for this lecture sequence. Thank you.