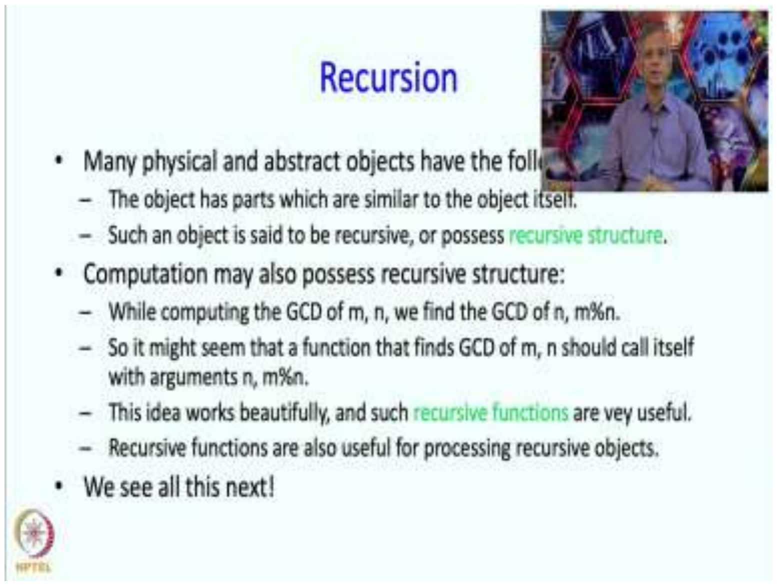
An Introduction to Programming through C++

Professor Abhiram G. Ranade
Department of Computer Science and Engineering
Indian Institute of Technology Bombay
Lecture No. 11 Part - 1
Recursion
Introduction

Hello, and welcome to the course on An Introduction to Programming through C++. The topic for today's lecture sequence is recursive functions and the reading for this is chapter 10 of the text.

(Refer Slide Time: 0:33)



Recursion

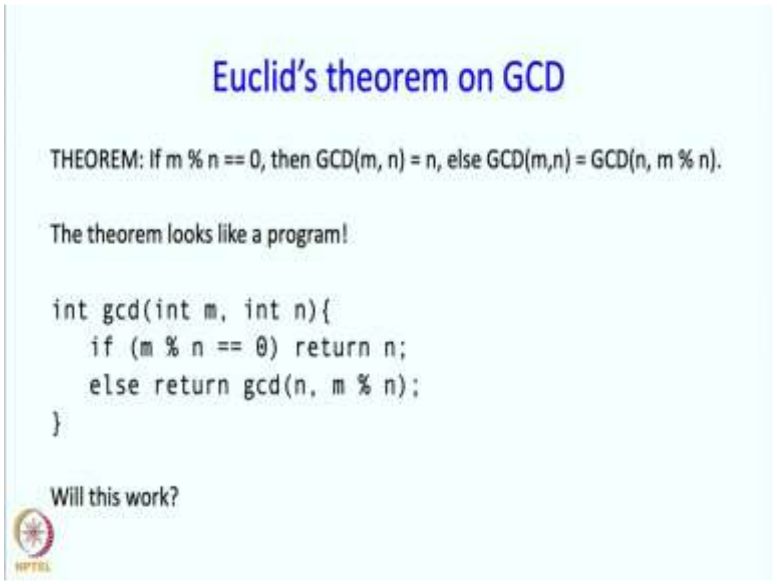
- Many physical and abstract objects have the foll
  - The object has parts which are similar to the object itself.
  - Such an object is said to be recursive, or possess recursive structure.
- Computation may also possess recursive structure:
  - While computing the GCD of m, n, we find the GCD of n, m%n.
  - So it might seem that a function that finds GCD of m, n should call itself with arguments n, m%n.
  - This idea works beautifully, and such recursive functions are vey useful.
  - Recursive functions are also useful for processing recursive objects.
- We see all this next!

So, to introduce the topic of recursion, let me observe that many physical and abstract objects have the following interesting property - the object has parts, which are similar to the object itself. So, the natural question is, what role is this similarity going to play in the computation and that is where recursion comes in. So, such objects said to be recursive or said to possess recursive structure.

Now, one such object could be computation itself. Computation may also possess recursive structure. And in fact, you have seen an example of this already. While computing the GCD of M and N, we said that we really should compute the GCD of N and M mod N. So, it might seem that a function that finds that you GCD of M and N should call itself with arguments, N and M mod N.

It turns out that this idea works beautifully and such recursive functions are extremely useful. Recursive functions are also useful for processing recursive objects. And in this lecture sequence, we are going to see all of these things.

(Refer Slide Time: 2:04)



Euclid's theorem on GCD

THEOREM: If m % n == 0, then GCD(m, n) = n, else GCD(m,n) = GCD(n, m % n).

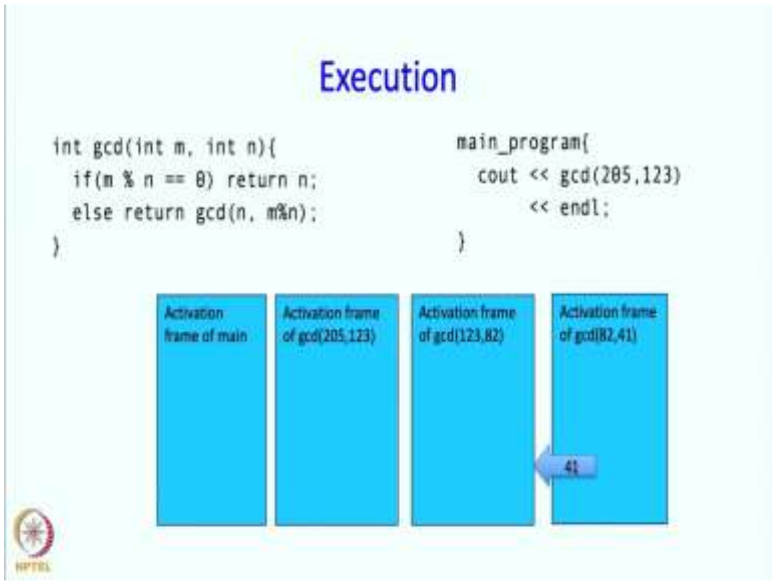
The theorem looks like a program!

```
int gcd(int m, int n){
   if (m % n == 0) return n;
   else return gcd(n, m % n);
}
```

Will this work?

So, let me remind you of Euclid's theorem on GCD, which we used earlier. So, the theorem says, If M mod N is equal to 0, then GCD of M and N should be N, as N. Else, the GCD of M and N is the GCD of N and M mod N. If you look at this theorem, it almost looks like a program, it looks like an if and else statement and nothing more. So, you might think could we perhaps write it like that? So, int GCD M N, and inside that the computation that you perform is if M mod N equal to 0, then return N because in this case the theorem says the GCD is N, otherwise, we return the GCD of N and M mod N. That is it. So, this seems like a, like a reflection of the theorem and a really compact program. The question is, will it work? It looks too good to be true perhaps.

(Refer Slide Time: 3:20)



Execution

```
int gcd(int m, int n){
  if(m % n == 0) return n;
  else return gcd(n, m%n);
}
```

```
main_program{
  cout << gcd(205,123)
       << endl;
}
```

Activation frame of main
Activation frame of gcd(205,123)
Activation frame of gcd(123,82)
Activation frame of gcd(82,41)
41

Ok, so, let us see, let us let us use the rules that we have mentioned for how functions execute and let us see what will happen if we try to execute this function. And so, I have the function on the left hand side and the main program on the right hand side. And the main program is doing nothing much, it is just invoking the function with arguments 205 and 123 and printing out the result.

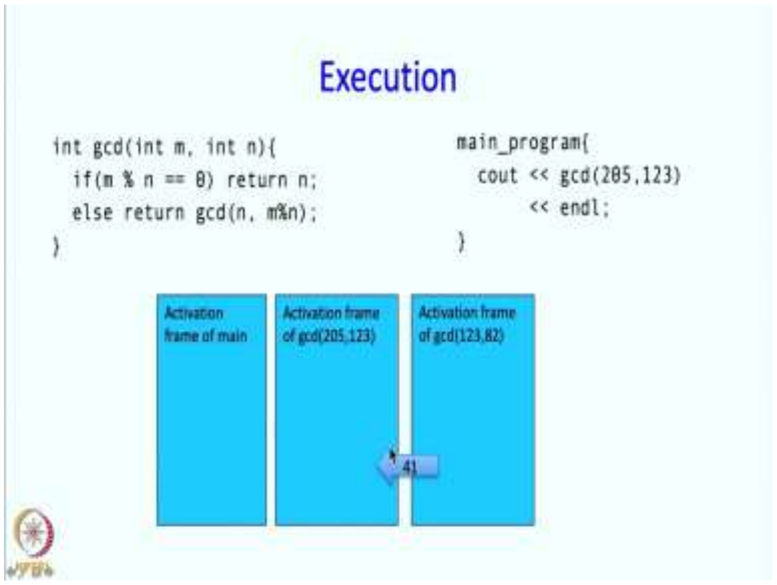
So, when this program executes, so, of course, first the main program starts executing and for that, the activation frame is created for main program. Now, when the main program encounters the GCD statement, the GCD call, then you know that the main program suspends and we create an activation frame for GCD. So, this activation frame will get created.

So, GCD will now start executing with M equals 205 and N equals 123. So, when it executes M mod N will not be equal to 0. So therefore, it will go to the else part and it will return the GCD of N which is now 123 and M mod N, which is 82. So, that will create another activation frame. So, another call, so another activation frame, and then the first call is going to suspend.

So, after that, the second call is to start executing and this time M has the value 123 and N has the value 82. So, again in the first step of execution M mod N will not be equal to 0. So, the else statement will be executed. In this case, we are going to return the GCD of N which now is 82, and M mod N which is 123 mod 82 which is 41. So, we require that the GCD of 82 and be 41 returned.

So, this will naturally require this call to suspend and a new call and a new activation frame to be created. So, this is it. So, now we have an activation frame of the GCD called with arguments 82 and 41. So note that at this point, we have actually 4 activation frames in memory. So, the main program had suspended and therefore its activation frame is present the GCD the first call has suspended so its activation frame is present, the second GCD call has suspended so its activation frame is also present and the last GCD call that we are currently executing, is executing. So, it also needs an activation frame. So, this GCD calls has arguments at 82 and 41. So, when we now try to execute this call, so, M has value 82, N has value 41. So, this time M is a multiple of N, 82 is just twice a 41. So, this expression will turn out to be 0 and therefore, we will return N. So, this call is going to return 41 so, that 41 will be sent back. So, the 41 will be sent back and we come back to the activation frame of GCD 123, 82.

(Refer Slide Time: 7:10)



Execution

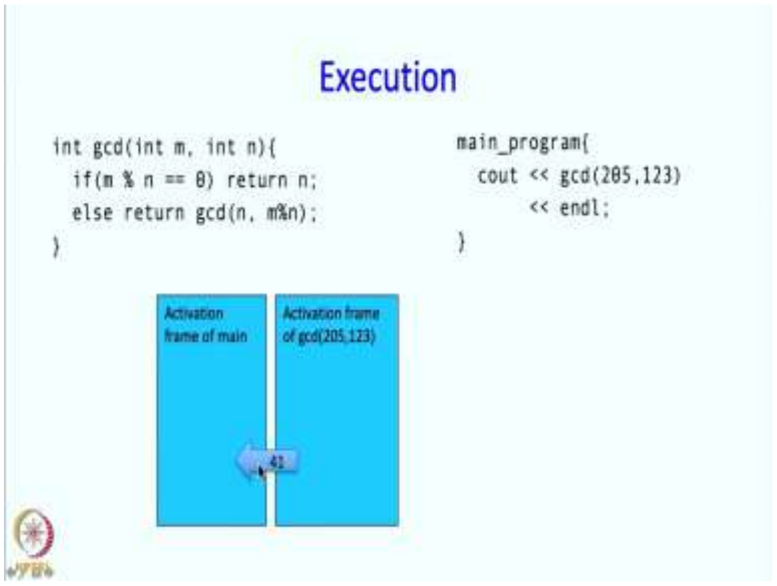
```
int gcd(int m, int n){
  if(m % n == 0) return n;
  else return gcd(n, m%n);
}
```

```
main_program{
  cout << gcd(205,123)
       << endl;
}
```

Activation frame of main
Activation frame of gcd(205,123)
Activation frame of gcd(123,82)
41

So remember, this call had been waiting to receive a value from so that it can return whatever value it receives back to its caller. So it has received the value 41 so it can return 41 back to its caller, but of course before that the activation frame of 82, 41 will disappear. And now this, this call will return when 123, 82 and that will activate the very first call we made. So, this call will get activated because it has received the value.

(Refer Slide Time: 7:51)



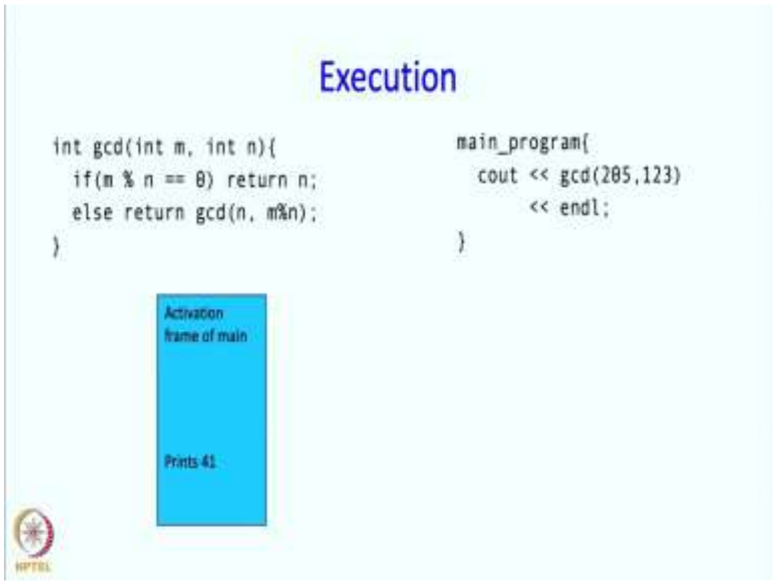
Execution

```
int gcd(int m, int n){
  if(m % n == 0) return n;
  else return gcd(n, m%n);
}
```

```
main_program{
  cout << gcd(205,123)
       << endl;
}
```

But and it will attempt to pass this value back. But of course before this, this activation frame will go away. And then M mod N, we will this last this very first call actually will pass the value it receives from the call that it made, it will be passed back to main. So main will have finally received the value that it wanted over here.

(Refer Slide Time: 8:14)



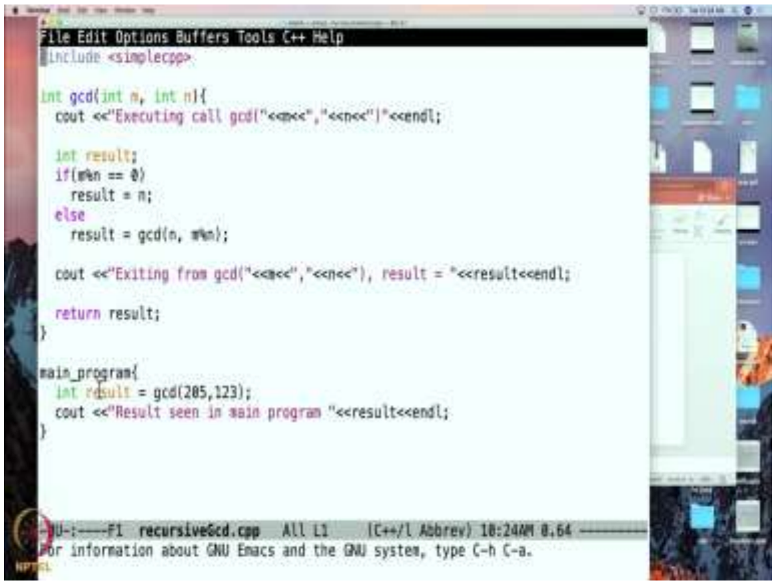
Execution

```
int gcd(int m, int n){
  if(m % n == 0) return n;
  else return gcd(n, m%n);
}
```

```
main_program{
  cout << gcd(205,123)
       << endl;
}
```

And then this activation frame will go away and main will actually do the printing. And so, it will turn 41. And as you can see 41 is indeed the greatest common divisor of 205 and 123. And so all this seems to have worked quite well. And indeed it does, we will see in little bit more detail why it does? And it is actually quite similar, the execution it actually turns out to be quite similar to the execution of the program, the non recursive program that you wrote earlier.

(Refer Slide Time: 9:01)



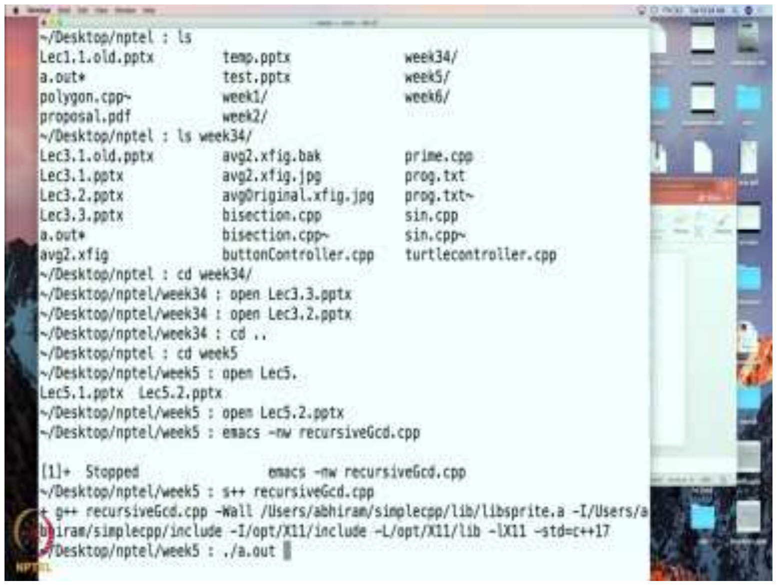
```
#include <simplecpp>

int gcd(int m, int n){
  cout <<"Executing call gcd("<<m<<","<<n<<")"<<endl;

  int result;
  if(m%n == 0)
    result = n;
  else
    result = gcd(n, m%n);

  cout <<"Exiting from gcd("<<m<<","<<n<<"), result = "<<result<<endl;

  return result;
}

main_program{
  int result = gcd(205,123);
  cout <<"Result seen in main program "<<result<<endl;
}
```

```
-UU-:----F1  recursiveGcd.cpp   All L1    (C++/l Abbrev) 10:24AM 0.64 ----------
For information about GNU Emacs and the GNU system, type C-h C-a.
```

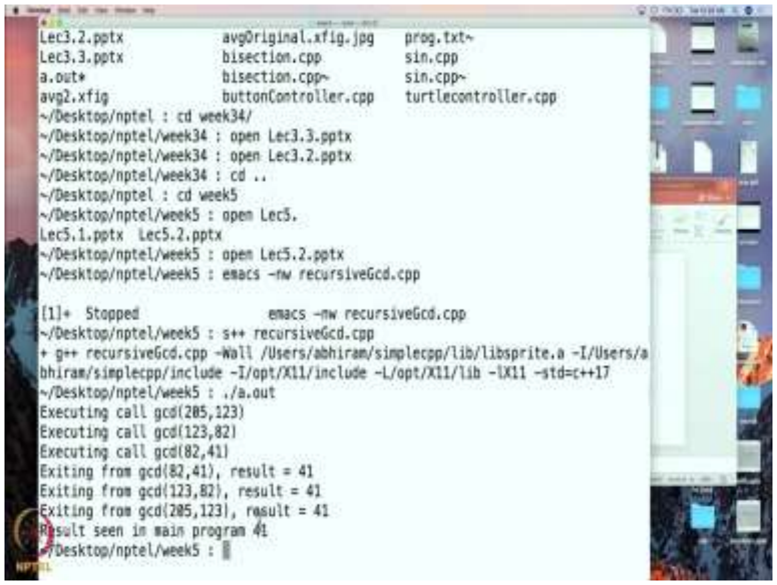
So, let us take a demo of this just to make just to see that this actually runs and get a little bit more comfort with this. So, this program is the same as the program that you saw except for a few minor changes. So, first of all, just to just to let us see what is going on I have put out this print statement at the very beginning of GCD. So, so, before the GCD executes, it will tell us what arguments it has received. So, that is what this print statement is doing. Next at the end, before GCD exits, I would like to print out what value it is going to return. So therefore, I have created a variable result into which the result to be return is first put in, ok. So, if an M mod N is equal to 0, then we put n into result, otherwise, we make the call GCD and the result of that is put into result. So this execution is really the same as the execution that we had on the slides, except that we are able to print some messages and to print those messages we are having one extra variable. So after the result is put over here we print out what is return and then we return the result.

Similarly, from the main program, we want we are printing a little bit more descriptive message so that we can get, get to know what result was actually returned. So, we print out this message and print out the result and even for this the GCD is not directly printed, but we first put it inside a result variable.

(Refer Slide Time: 10:42)



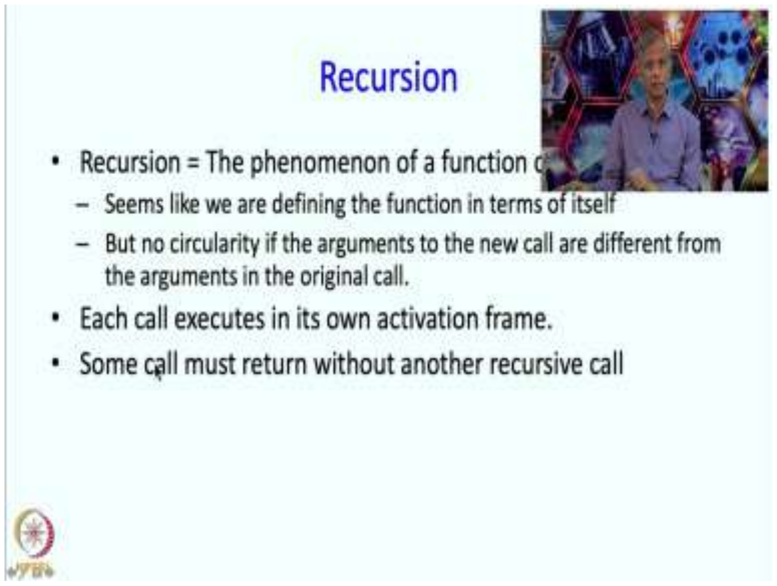
```
~/Desktop/nptel : ls
Lec1.1.old.pptx         temp.pptx               week34/
a.out*                  test.pptx               week5/
polygon.cpp~            week1/                  week6/
proposal.pdf            week2/
~/Desktop/nptel : ls week34/
Lec3.1.old.pptx         avg2.xfig.bak           prime.cpp
Lec3.1.pptx             avg2.xfig.jpg           prog.txt
Lec3.2.pptx             avgOriginal.xfig.jpg    prog.txt~
Lec3.3.pptx             bisection.cpp           sin.cpp
a.out*                  bisection.cpp~          sin.cpp~
avg2.xfig               buttonController.cpp    turtlecontroller.cpp
~/Desktop/nptel : cd week34/
~/Desktop/nptel/week34 : open Lec3.3.pptx
~/Desktop/nptel/week34 : open Lec3.2.pptx
~/Desktop/nptel/week34 : cd ..
~/Desktop/nptel : cd week5
~/Desktop/nptel/week5 : open Lec5.
Lec5.1.pptx  Lec5.2.pptx
~/Desktop/nptel/week5 : open Lec5.2.pptx
~/Desktop/nptel/week5 : emacs -nw recursiveGcd.cpp

[1]+  Stopped                 emacs -nw recursiveGcd.cpp
~/Desktop/nptel/week5 : s++ recursiveGcd.cpp
+ g++ recursiveGcd.cpp -Wall /Users/abhiram/simplecpp/lib/libsprite.a -I/Users/a
bhiram/simplecpp/include -I/opt/X11/include -L/opt/X11/lib -lX11 -std=c++17
~/Desktop/nptel/week5 : ./a.out
```



```
Lec3.2.pptx             avgOriginal.xfig.jpg    prog.txt~
Lec3.3.pptx             bisection.cpp           sin.cpp
a.out*                  bisection.cpp~          sin.cpp~
avg2.xfig               buttonController.cpp    turtlecontroller.cpp
~/Desktop/nptel : cd week34/
~/Desktop/nptel/week34 : open Lec3.3.pptx
~/Desktop/nptel/week34 : open Lec3.2.pptx
~/Desktop/nptel/week34 : cd ..
~/Desktop/nptel : cd week5
~/Desktop/nptel/week5 : open Lec5.
Lec5.1.pptx  Lec5.2.pptx
~/Desktop/nptel/week5 : open Lec5.2.pptx
~/Desktop/nptel/week5 : emacs -nw recursiveGcd.cpp

[1]+  Stopped                 emacs -nw recursiveGcd.cpp
~/Desktop/nptel/week5 : s++ recursiveGcd.cpp
+ g++ recursiveGcd.cpp -Wall /Users/abhiram/simplecpp/lib/libsprite.a -I/Users/a
bhiram/simplecpp/include -I/opt/X11/include -L/opt/X11/lib -lX11 -std=c++17
~/Desktop/nptel/week5 : ./a.out
Executing call gcd(205,123)
Executing call gcd(123,82)
Executing call gcd(82,41)
Exiting from gcd(82,41), result = 41
Exiting from gcd(123,82), result = 41
Exiting from gcd(205,123), result = 41
Result seen in main program 41
~/Desktop/nptel/week5 :
```

So, let us compile this and execute this. So, I hit ./a.out and this is what happens. So, what has happened? The first call was indeed as we expected, it was executing GCD with arguments 205, 123. After that, the second call was made when 123, 82, after that the 82, 41 call was made, this is exactly what happened on the slide as we saw, then the last call that was made, returned and it returned the result 41 then the call before that returned and that returned the result 41 and then the call before that returned and again the result that was returned was the same and this is the print statement that we put in, in the main program. So, the main program also receives the value and that value is indeed 41 and so, everything is fine, ok.

(Refer Slide Time: 11:53)



Recursion

- Recursion = The phenomenon of a function c
  - Seems like we are defining the function in terms of itself
  - But no circularity if the arguments to the new call are different from the arguments in the original call.
- Each call executes in its own activation frame.
- Some call must return without another recursive call

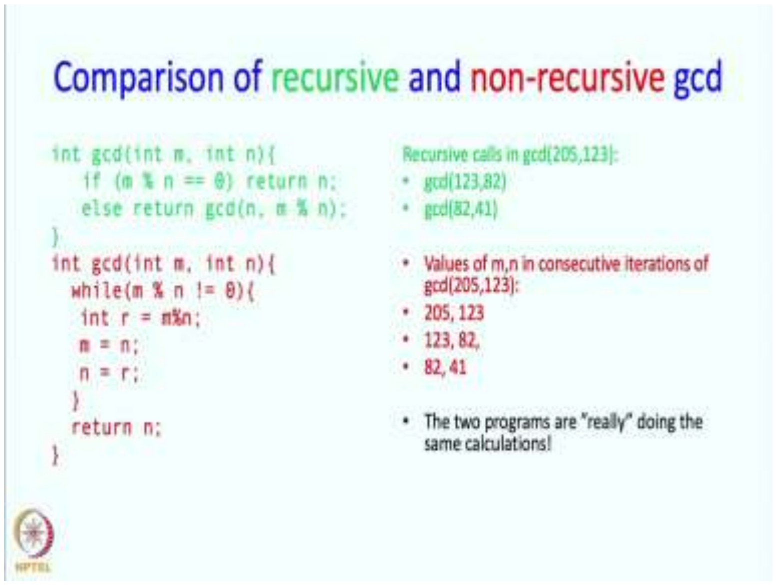
So, what have we seen here? Well, we have seen the recursion and let me just define it so to say. So, recursion is simply the phenomena of a function calling itself. Now, it might seem like we are defining the function in terms of itself and that is not a good idea. So, if we are defining a term then we should define the term using some other terms.

But here, that is not a problem actually. So, there is no circularity in this definition. Because when we make the new call, the arguments to it are different from the arguments in the original call, ok. Now, further, yeah, so, so, (that is not) there is no circularity over there. And a new call is going to execute in its own activation frame. So, we have also seen that, and we have also seen, and, and I guess, I should point out over here, that this process can go on indefinitely if you are not careful and that will, that is what might happen. If the arguments to the new call, for example, are the same as the arguments to an old call, because then they exactly the old execution will happen. So, we do not want that to happen.

So, we wanted, we want to make sure that some execution, some call must return without any recursive call. And if you remember, that did happen in our execution. And if that if that does not happen, then we have a problem because there is infinite recursion.

Now, in this example, in this GCD example, in the body of GCD, there was just one recursive call. We can have several recursive calls if we wish, and in fact, we will see an example very soon.

(Refer Slide Time: 14:00)



Comparison of recursive and non-recursive gcd

```
int gcd(int m, int n){
   if (m % n == 0) return n;
   else return gcd(n, m % n);
}
int gcd(int m, int n){
  while(m % n != 0){
   int r = m%n;
   m = n;
   n = r;
  }
  return n;
}
```

Recursive calls in gcd(205,123):
- gcd(123,82)
- gcd(82,41)

- Values of m,n in consecutive iterations of gcd(205,123):
- 205, 123
- 123, 82,
- 82, 41

- The two programs are "really" doing the same calculations!

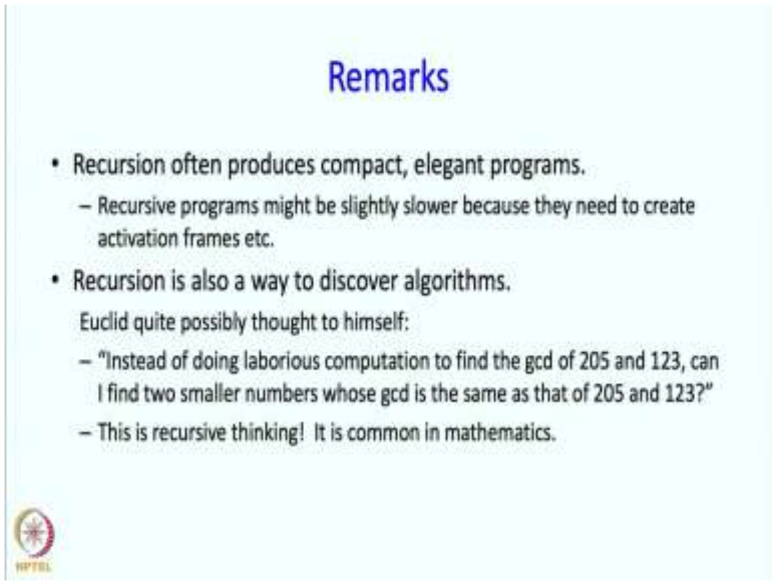
Before we go to more examples, I just want to observe I just want to compare how the recursive and non recursive versions of GCD work. So, first, this green portion is the recursive GCD and red portion is the non-recursive GCD that you saw some time ago in the in the last lecture. So now, let me first mention that if we look at the recursive calls in GCD the so we were calling this with GCD of 205, 123, then that call made a recursive call GCD 123, 82 that made a recursive call at 82, 41 and then that returned. Now, let us see what are the computations that happen in the non-recursive function. So, here we are going to look at what values M and N take in consecutive iterations of the call to GCD of 205, 123. So the first time around, when we call GCD with 205, 123 M and N will take values 205 and 123. Now, some iteration will this iteration will take place and as a result of it M and N will change. And you will see that in the next iteration the values are 123, 82. And in the iteration after that the values are 82 and 41. And after that, the answer 41 will be returned.

So, if you notice the M and N, which were the arguments over here, really are taking the same values as the variable variables M and N over here. So, in a sense, the recursive and non-recursive GCDs are really doing the same computation. So, so, in a sense we really do not need to worry about is this going to be correct. So, so, long as we can actually establish correspondence and we can establish that correspondence, then we will know that the recursive GCD should also work. So, so, that is certainly that is certainly a reassuring fact over here.

However, I should point out that for every recursive function there need not be a natural non recursive function that does the same thing. So, the question of how, how to think about recursion and how to argue that recursive functions are correct, it is still there, but here for this GCD you can sort of see that it really the execution of the recursive function really mirrors the execution of the non recursive function.

And but, the point to note is that on the surface they look very different, the non recursive function is a bit longer and somehow it seems to be doing a lot more stuff.

(Refer Slide Time: 17:09)



Remarks

- Recursion often produces compact, elegant programs.
  - Recursive programs might be slightly slower because they need to create activation frames etc.
- Recursion is also a way to discover algorithms.

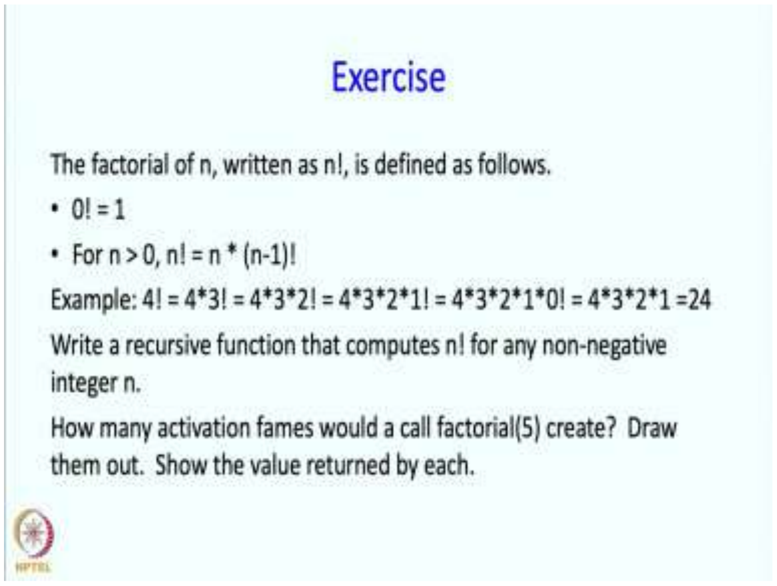
  Euclid quite possibly thought to himself:
  - "Instead of doing laborious computation to find the gcd of 205 and 123, can I find two smaller numbers whose gcd is the same as that of 205 and 123?"
  - This is recursive thinking! It is common in mathematics.

So it should, it should be observed that recursion often produces compact and rather elegant programs, elegant because the program is sort of following the main theorem that is the basis of that function. Now, I should also point out that recursive programs might be slightly slower, because they need to create the activation frame and destroy the activation frame, which the iterative or the non recursive programs do not have to do.

The exciting point about recursion and we will see this in a later lecture is that recursion is also a way to discover algorithms. And you may say that Euclid quite possibly did this, possibly he thought something like the following. "Instead of doing laborious computation to find the GCD of 205 and 123, can I find two smaller numbers whose GCD is the same as that of 205 and 123?" This is exactly recursion, this is recursive thinking and it is quite common in mathematics and therefore, and it is certainly a powerful tool in designing algorithms.

And we are going to see more examples of recursion in this lecture sequence as well as in subsequently lecture sequences.

(Refer Slide Time: 18:47)



Exercise

The factorial of n, written as n!, is defined as follows.

- 0! = 1
- For n > 0, n! = n * (n-1)!

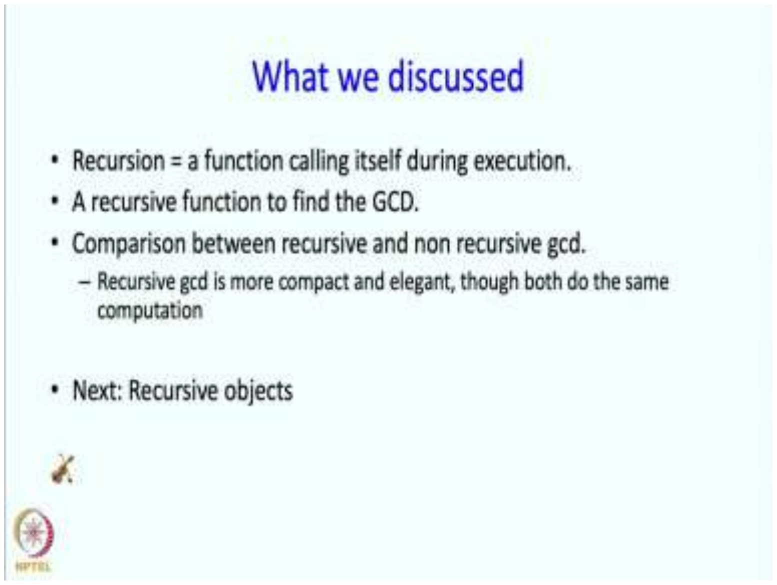
Example: 4! = 4*3! = 4*3*2! = 4*3*2*1! = 4*3*2*1*0! = 4*3*2*1 =24

Write a recursive function that computes n! for any non-negative integer n.

How many activation fames would a call factorial(5) create? Draw them out. Show the value returned by each.

So, here is an exercise it asks you to write a recursive program and you are supposed to consider our GCD program and sort of follow it but for this new function that has to be for which the program has to be written.

(Refer Slide Time: 19:05)



What we discussed

- Recursion = a function calling itself during execution.
- A recursive function to find the GCD.
- Comparison between recursive and non recursive gcd.
  - Recursive gcd is more compact and elegant, though both do the same computation

- Next: Recursive objects

So, what did we discuss? So, he said that recursion is a function calling itself during execution. When we said that recursive function we found we saw a recursive function to find the GCD.

Then we made a comparison between the recursive and non-recursive GCD functions and we saw that recursive GCD is more compact and elegant even though both of them are doing the same computations. Next, we are going to talk about recursive objects, but let us take a break.