An Introduction to Programming through C++

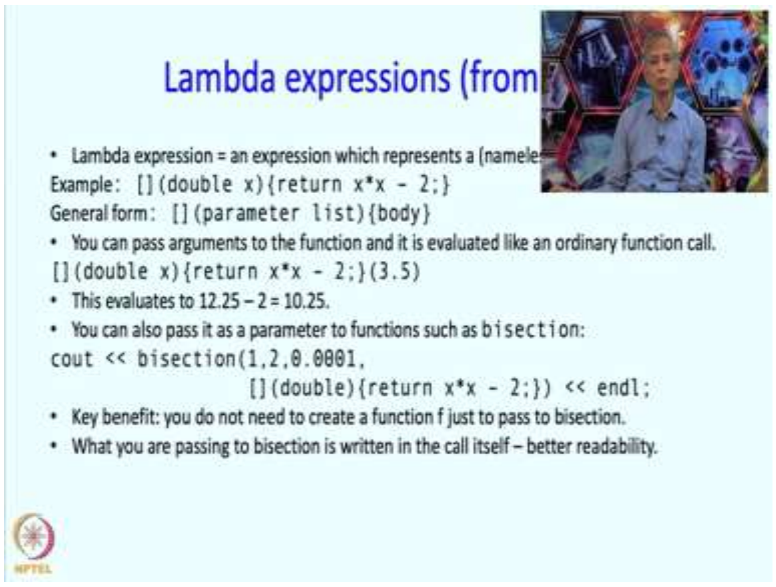
Professor. Abhiram G. Ranade
Department of Computer Science and Engineering,
Indian Institute of Technology Bombay India

Lecture 14

Advance Features of Functions Lambda expressions

Welcome back, in the last segment we discussed functions which operate on functions. We are going to continue the theme and now we are going to describe something called lambda expressions, which make it easier to pass functions to other functions. Lambda expressions originated from lisp, and in lisp they were present maybe in the 50's-60's something like that, 1950s-1960s and in C++ they have been present at least for 15-20 years now.

(Refer Slide Time: 00:57)



Lambda expressions (from

- Lambda expression = an expression which represents a (namele

Example: `[](double x){return x*x - 2;}`

General form: `[](parameter list){body}`

- You can pass arguments to the function and it is evaluated like an ordinary function call.

```
[](double x){return x*x - 2;}(3.5)
```

- This evaluates to 12.25 – 2 = 10.25.
- You can also pass it as a parameter to functions such as `bisection`:

```
cout << bisection(1,2,0.0001,
                  [](double){return x*x - 2;}) << endl;
```

- Key benefit: you do not need to create a function f just to pass to bisection.
- What you are passing to bisection is written in the call itself – better readability.

So, a lambda expression is an expression which represents a function and the function does not really have a name. So, it is just sitting there but it does not have a name. Here is an example: so, square bracket into parentheses double x into braces return x times x minus two, is an expression and it represents a function. And in this case, it represents a function, f(x)=x^2-2. The general form of this is the square brackets first, and the square brackets; we will tell you what they do in a little bit later, but first there is the parameter list. So, x is the parameter over here. If the function that you want to represent has more parameters, you just put it here just like a regular old parameter list and then this is like a regular old body, in this case the function is really simple, so you just have single return statement. But, in general, you can have a whole a complete function here over here, however complex you want to make it.

So that is what a lambda expression is. Now, what can you do with it, well the simplest thing that you would expect is can you apply it, or can you use it to operate upon some arguments. So, yes you can.

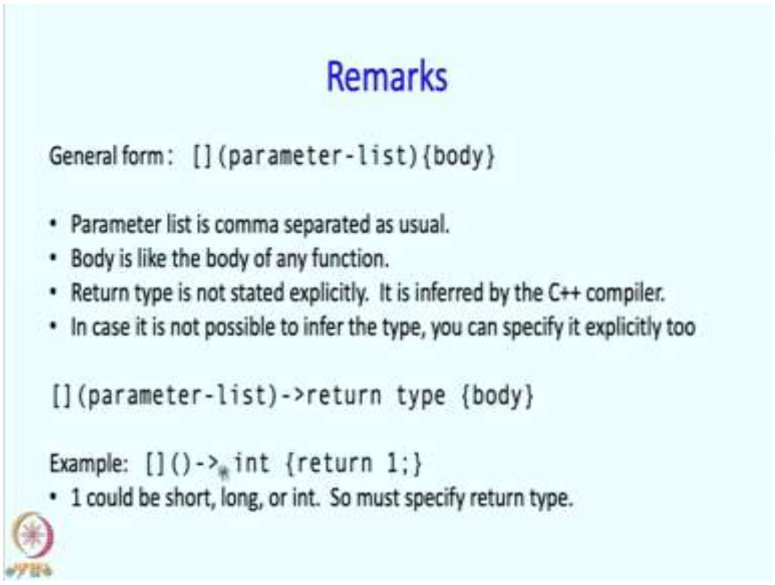
So, here I have shown this lambda expression; which is the function x^2-2 and that is operating on 3.5, so, it is computing, f of x f of 3.5 where f of x is x square minus two. So, this evaluates, x times x is 12.25 minus 2. So, this evaluates to ten point 12.25 as you might want and we can certainly write, z equals square bracket double x all of these 3.5 and then z would be said to 10.25.

The more interesting thing for us is that, you can pass it as an argument I guess I should say to function such as bisection. So, in the main program that you saw earlier (we had) we were printing out the value, square root of 2. So, there we wanted to pass a function, where x square minus 2. So, there we wrote a function and past its name, but here we can just send the expression for the function x square minus two instead.

So, this is all that is needed, I can write this in the body of my programme, and this will call bisection with this function. And this will work, so this will indeed produce a square root 2, the bisection will return square root of 2. So, this is as good as defining a full function outside your main programme and giving it a name (say whatever) f as we called it and sending f over here. But you do not really need to do that, you can just make this something local.

So, here as I read my code, I can see that am just trying to find the square root of 2. I do not need to go to some other place. So, basically this is going to make your programme less clutter and it will improve local readability.

(Refer Slide Time: 04:49)



Remarks

General form: `[](parameter-list){body}`

- Parameter list is comma separated as usual.
- Body is like the body of any function.
- Return type is not stated explicitly. It is inferred by the C++ compiler.
- In case it is not possible to infer the type, you can specify it explicitly too

```
[](parameter-list)->return type {body}
```

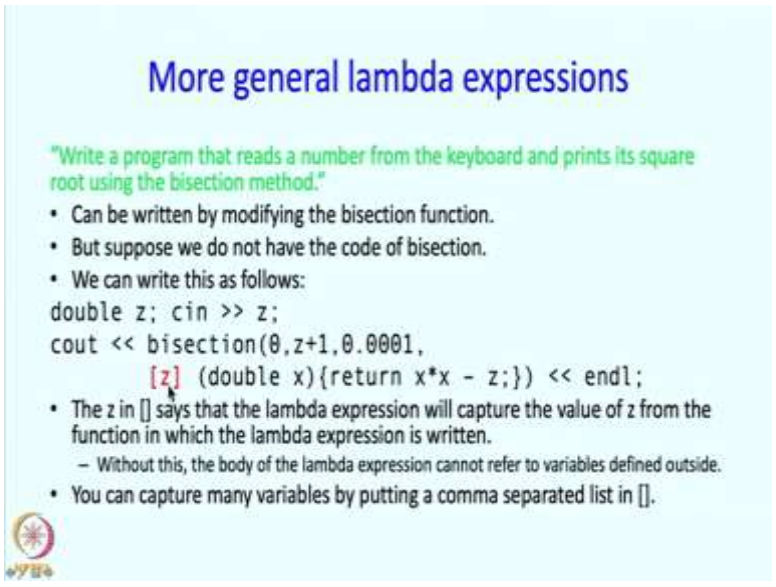
Example: `[]()->int {return 1;}`

- 1 could be short, long, or int. So must specify return type.

Now, am going to, some comments about this general form. So, the parameter list is comma separated as usual as I mentioned earlier, body is like the body of any function. Now, the return type is not stated explicitly. So, in a huge-well function the written type will be stated. And in this case the C++ compiler looks at the return statement and says, the return statement is inferring this type of value and therefore the return type of function must be this. Sometimes it is not possible to infer the type, and, in that case, you can specify it explicitly.

So, for example: here is another form, in which am saying that look, this function is supposed to return something of return type this. So, as an example: I might write all of this and inside I might have a return statement returning one. But now C++ complier cannot figure out what one is, one can have several types: one could be short, one could be long, one could be int and therefore it is desirable to specify the type and that is how you specify, over here.

(Refer Slide Time: 6:12)



More general lambda expressions

"Write a program that reads a number from the keyboard and prints its square root using the bisection method."

- Can be written by modifying the bisection function.
- But suppose we do not have the code of bisection.
- We can write this as follows:

```
double z; cin >> z;
cout << bisection(0,z+1,0.0001,
          [z] (double x){return x*x - z;}) << endl;
```

- The z in [] says that the lambda expression will capture the value of z from the function in which the lambda expression is written.
  - Without this, the body of the lambda expression cannot refer to variables defined outside.
- You can capture many variables by putting a comma separated list in [].

Lambda general expressions can be more general. So, here is a problem that needs that higher generality, so the problem says, write a programme that reads a number from the keyboard and prints its square root using the bisection method. Now, of course you could again go back and ask, look do you need to modify the bisection method, but (now it) we said, look that does not seem very elegant and another thing is that you may not have the code, you may have the object module in which case there is no question of modifying it.

Now, this can also be written with this more general lambda expression. So, here is the code fragment, so, suppose we want the square root of z. So, what are we going to do, we are going to first read z. So, we are going to declare a variable to store z, and we are going to read a value into it.

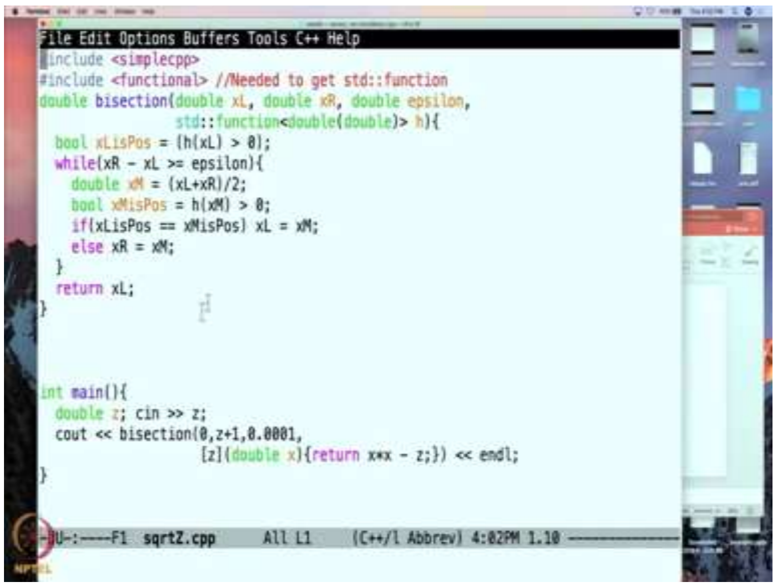
And after this we are going to have a call to bisection, almost like what we had earlier. So, double x the function is the same, except that here now we want to calculate the root of x square minus z and not x square minus 2. So, this root is being calculated, the root of x square minus z is calculated. And the z is taken or captured from outside of this body, so this is the body. So, normally you expect everything every variable to be used, that is used over here to be defined inside the body, just like a regular function.

But, you can do a little bit more, so I can put a variable over here which appears over here as well or I can put an expression as I have done, which uses variables which appear outside which have been defined outside. So, now we have to tell the C++ compiler that am doing this because the C++ compiler is always worried, that look, are you doing this deliberately or are you making mistake.

And therefore, just to indicate, that this variable is being captured from outside, inside the square brackets, we have to put in z. So, this says to the compiler, I am going to use the value of the variable z which has been defined outside. And now I can use that variable inside over here.

So, the z inside those square brackets says that, the lambda expression will capture, that is the technical term used, the value of z from the function in which the lambda expression is written. So, this lambda expression is written in this code fragment, which is a part of main function. And z is the variable defined in the main function and I can use the value of it, the value of it will be substituted over here, and then this function will be constructed. And then passed to bisection. And without this, the body of the lambda expression cannot refer to variables which are defined outside. And furthermore, I should say that if I want to capture more variables, I can just put them here, separated by comas.

(Refer Slide Time: 9:50)

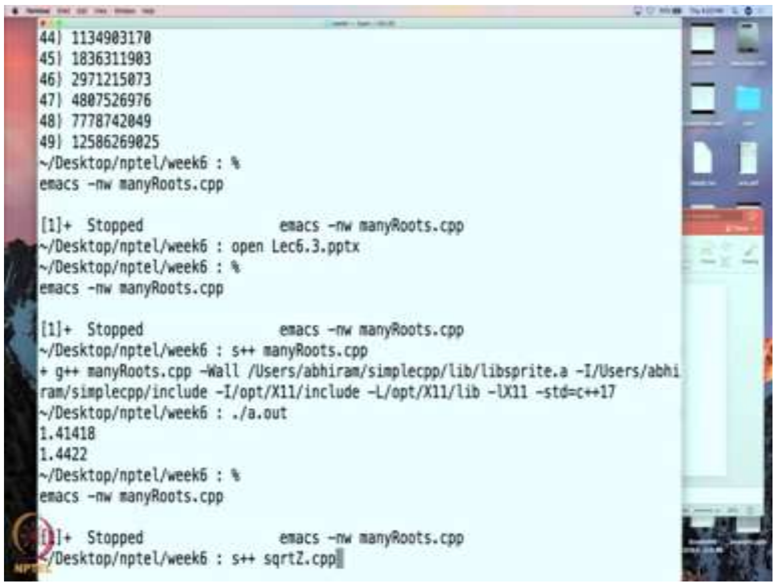


```
#include <simplecpp>
#include <functional> //Needed to get std::function
double bisection(double xL, double xR, double epsilon,
                 std::function<double(double)> h){
  bool xLisPos = (h(xL) > 0);
  while(xR - xL >= epsilon){
    double xM = (xL+xR)/2;
    bool xMisPos = h(xM) > 0;
    if(xLisPos == xMisPos) xL = xM;
    else xR = xM;
  }
  return xL;
}



int main(){
  double z; cin >> z;
  cout << bisection(0,z+1,0.0001,
                 [z](double x){return x*x - z;}) << endl;
}
```

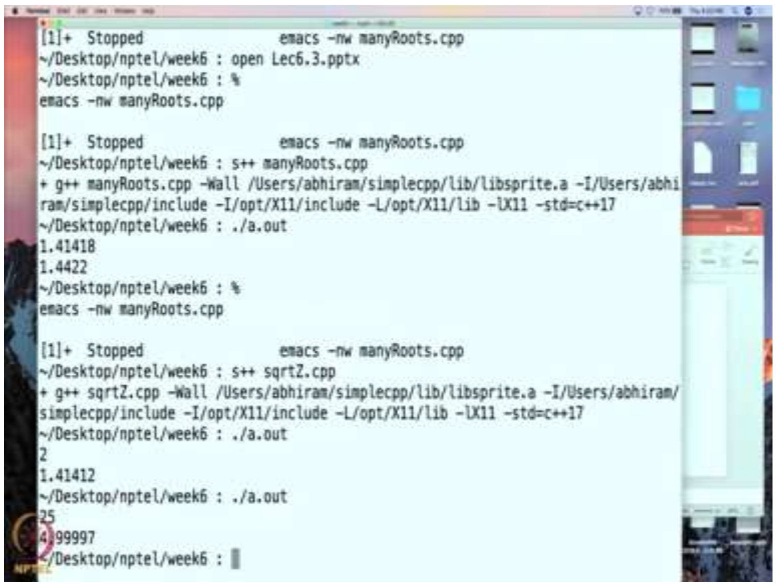
```
-UU-:----F1  sqrtZ.cpp      All L1     (C++/l Abbrev) 4:02PM 1.10
```

So, am going to give a demo of this programme, and let us look at it. So, this is the programme, as you see, this bisection function is the same as before, nothing changed. And over here and I am indicating this variable capture and finding the root of x square minus z and I am reading the value over here.

(Refer Slide Time: 10:09)



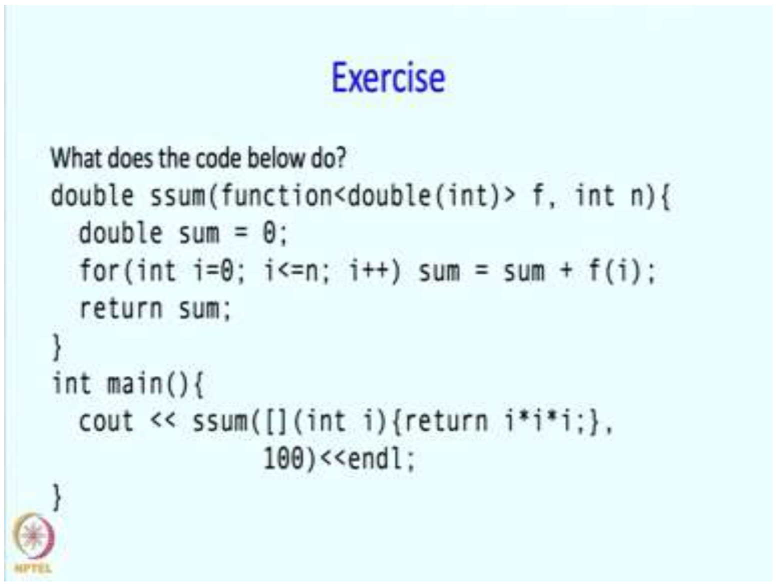
```
44) 1134903170
45) 1836311903
46) 2971215073
47) 4807526976
48) 7778742049
49) 12586269025
~/Desktop/nptel/week6 : %
emacs -nw manyRoots.cpp

[1]+  Stopped                 emacs -nw manyRoots.cpp
~/Desktop/nptel/week6 : open Lec6.3.pptx
~/Desktop/nptel/week6 : %
emacs -nw manyRoots.cpp

[1]+  Stopped                 emacs -nw manyRoots.cpp
~/Desktop/nptel/week6 : s++ manyRoots.cpp
+ g++ manyRoots.cpp -Wall /Users/abhiram/simplecpp/lib/libsprite.a -I/Users/abhi
ram/simplecpp/include -I/opt/X11/include -L/opt/X11/lib -lX11 -std=c++17
~/Desktop/nptel/week6 : ./a.out
1.41418
1.4422
~/Desktop/nptel/week6 : %
emacs -nw manyRoots.cpp

[1]+  Stopped                 emacs -nw manyRoots.cpp
~/Desktop/nptel/week6 : s++ sqrtZ.cpp
```



```
[1]+  Stopped                 emacs -nw manyRoots.cpp
~/Desktop/nptel/week6 : open Lec6.3.pptx
~/Desktop/nptel/week6 : %
emacs -nw manyRoots.cpp

[1]+  Stopped                 emacs -nw manyRoots.cpp
~/Desktop/nptel/week6 : s++ manyRoots.cpp
+ g++ manyRoots.cpp -Wall /Users/abhiram/simplecpp/lib/libsprite.a -I/Users/abhi
ram/simplecpp/include -I/opt/X11/include -L/opt/X11/lib -lX11 -std=c++17
~/Desktop/nptel/week6 : ./a.out
1.41418
1.4422
~/Desktop/nptel/week6 : %
emacs -nw manyRoots.cpp

[1]+  Stopped                 emacs -nw manyRoots.cpp
~/Desktop/nptel/week6 : s++ sqrtZ.cpp
+ g++ sqrtZ.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/week6 : ./a.out
2
1.41412
~/Desktop/nptel/week6 : ./a.out
25
4.99997
~/Desktop/nptel/week6 :
```

So, lets compile it. let’s run it. So, now I want to give it value, so let us try two, so it is printing the value that we have seen before, the correct value, nearly correct value, of course this is an approximation to certain number of decimals of bits. So, let us try one more, so maybe, let’s give it a value which is perfect square. So, it is printing out 4.9997. So, again as I said, this is going to print an approximate value and indeed it is a close enough value.

(Refer Slide Time: 10:53)



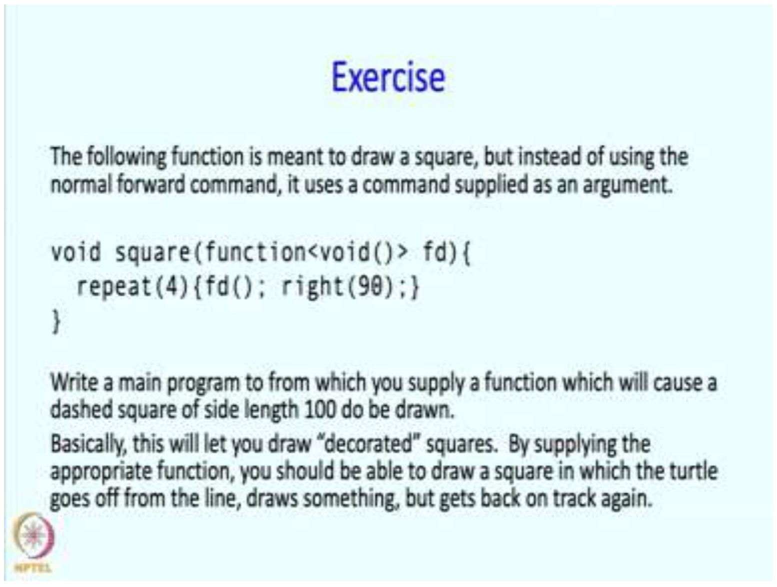
Exercise

What does the code below do?

```
double ssum(function<double(int)> f, int n){
  double sum = 0;
  for(int i=0; i<=n; i++) sum = sum + f(i);
  return sum;
}
int main(){
  cout << ssum([](int i){return i*i*i;},
              100)<<endl;
}
```

Here is an exercise for you. So, you are given a piece of code and you are supposed to tell what that function does. So, you should be reasonably you should be able to figure out reasonably quickly, from knowing, what lambda expressions are. And in any case, you can execute it to see what it does.

(Refer Slide Time: 11:14)



Exercise

The following function is meant to draw a square, but instead of using the normal forward command, it uses a command supplied as an argument.

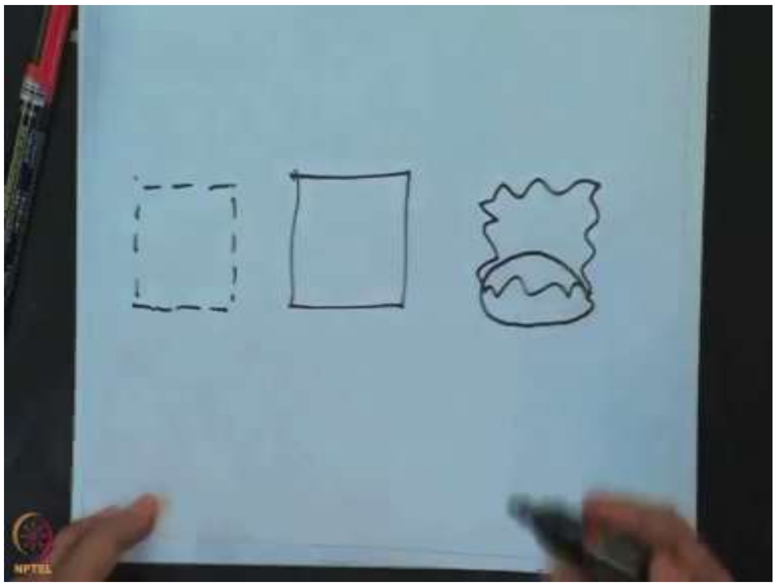
```
void square(function<void()> fd){
  repeat(4){fd(); right(90);}
}
```

Write a main program to from which you supply a function which will cause a dashed square of side length 100 do be drawn.

Basically, this will let you draw "decorated" squares. By supplying the appropriate function, you should be able to draw a square in which the turtle goes off from the line, draws something, but gets back on track again.

Here is another somewhat elaborate exercise. Now, in this case I have given a code, in which a function is being passed. The function that is being passed is not going to return anything nor is it going to take any arguments. And it is called FD, and that function FD is going to be called. So, this is going to be something which draws a square presumably and its going to repeat 4 and here you would have seen the line forward 100 or something like that and then it turns. Now, instead of doing forward 100, you can supply some different command through FD.

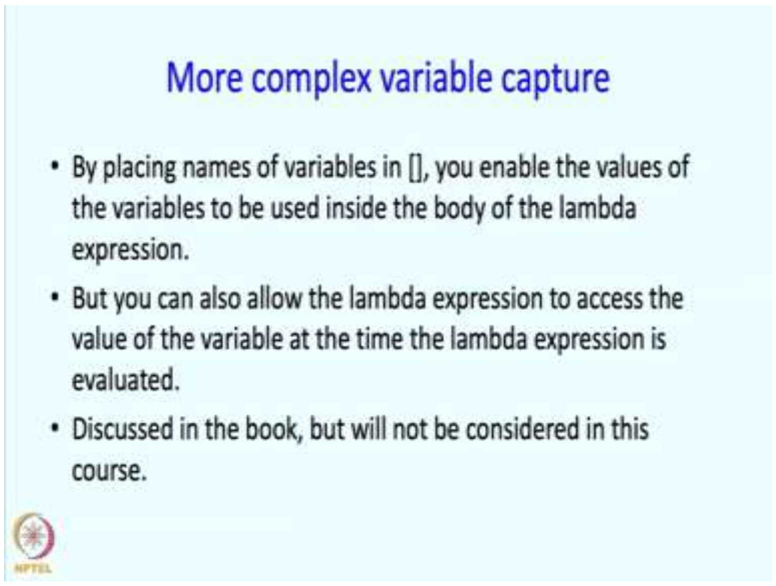
(Refer Slide Time: 12:04)



So, for example: you might say, look instead of just going straight and drawing the square, maybe I want turtle to do funny things like this. So, what you can do now is, I can write a single function square and I can pass to it, the function which does this, and that function will be used to draw the four sides.

So, essentially by supplying the appropriate functions you can do other things. So, maybe the other thing you could do is supply a function which draws a dotted line. Or you can just supply FD, forward 100 and it will draw the square. So, whatever it is, but you can have the same square function draw squares which are sort of decorated in a different manner. Now, it turns out that you can do more complex variable capture.

(Refer Slide Time: 13:08)



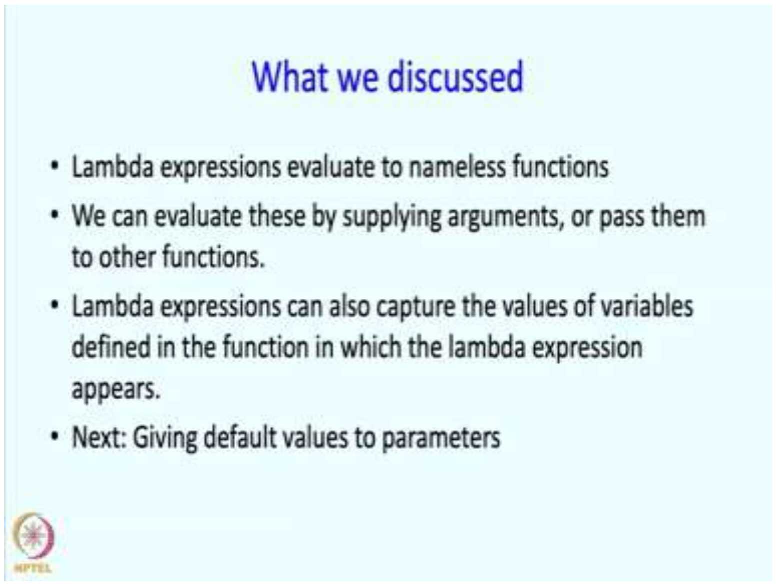
More complex variable capture

- By placing names of variables in [], you enable the values of the variables to be used inside the body of the lambda expression.
- But you can also allow the lambda expression to access the value of the variable at the time the lambda expression is evaluated.
- Discussed in the book, but will not be considered in this course.

So, what you have seen so far is that, if you place names of variables inside this you enable the values of the variables to be used inside the body of the lambda expression. But you can also cause the lambda expression to not take the current value but take the value of the variable at the time the lambda expression is actually evaluated.

So, during the time when the variable when the expression is evaluated and then the value changes, you can figure out what the value is. And you can do that, and it is discussed in the book, but it will not be considered this course, because this is just a little bit too elaborate and we are not going to really make interesting use of it.

(Refer Slide Time: 14:06)



What we discussed

- Lambda expressions evaluate to nameless functions
- We can evaluate these by supplying arguments, or pass them to other functions.
- Lambda expressions can also capture the values of variables defined in the function in which the lambda expression appears.
- Next: Giving default values to parameters

Alright, so, what have we discussed, we have said that lambda expression evaluates to nameless functions. We can evaluate these by suppling arguments or pass them to functions. We also said that, lambda expression can also capture the values of variables defined in the function in which the lambda expression appears.

So, this concludes this segment. In the next segment we will discuss, functions, a feature of functions, were you can write functions so that you specify default values to some parameters. So, we will take a break here.