An Introduction to Programming through C++

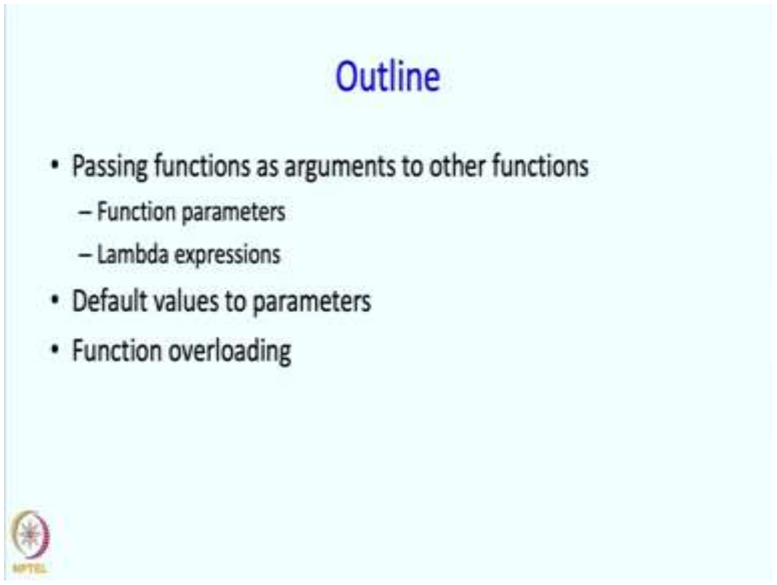
Professor. Abhiram G. Ranade
Department of Computer Science and engineering,
Indian Institute of Technology Bombay India.

Lecture 14

Advanced Features of Functions Introduction and passing one Function to another

Hello and welcome to the NPTEL course on an introduction to programming threw C++. I am Abhiram Ranade and the topic for today is some advance features of functions and the reading for this is from chapter 12 of the text. So, here is an outline for the, this lecture sequence. I am first going to talk about passing functions as arguments to other functions.

(Refer Slide Time: 00:43)

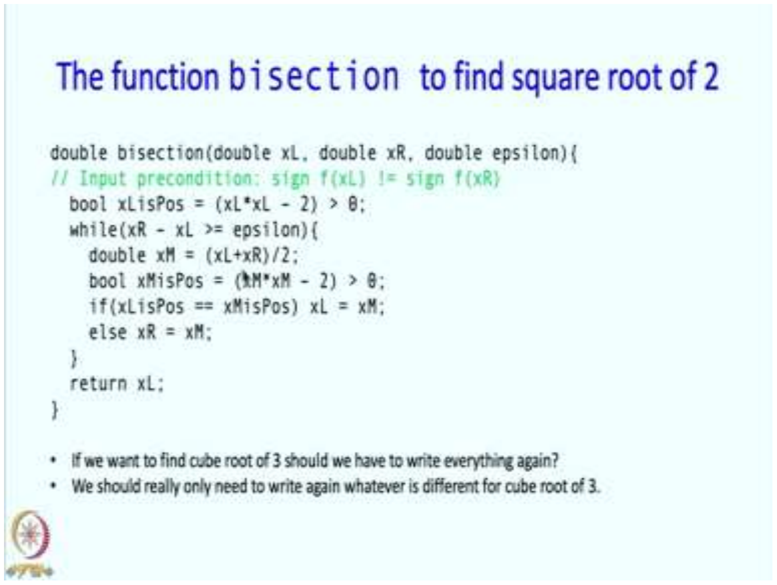


Outline

- Passing functions as arguments to other functions
  - Function parameters
  - Lambda expressions
- Default values to parameters
- Function overloading

And this will be in two ways - functions function parameters there will be two subtopics sort of and I will also talk about something called lambda expressions. Then I will talk about how to assign default values to parameters of a function. And I will also talk about something called function overloading. So, let me start of with the function bisection that we wrote some time ago to find the square root of 2.

(Refer Slide Time: 01:18)



The function `bisection` to find square root of 2

```
double bisection(double xL, double xR, double epsilon){
// Input precondition: sign f(xL) != sign f(xR)
  bool xLisPos = (xL*xL - 2) > 0;
  while(xR - xL >= epsilon){
    double xM = (xL+xR)/2;
    bool xMisPos = (xM*xM - 2) > 0;
    if(xLisPos == xMisPos) xL = xM;
    else xR = xM;
  }
  return xL;
}
```

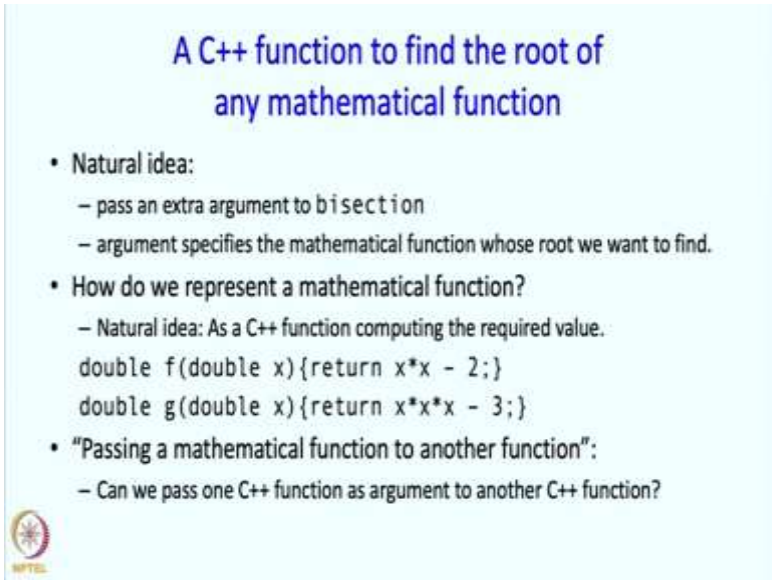
- If we want to find cube root of 3 should we have to write everything again?
- We should really only need to write again whatever is different for cube root of 3.

Let me begin with the function bisection to find square root of 2. So, here is the function. So this function meant to find root the root of any function and we are going to use it to find the square root of 2 by finding a root of the function x square minus 2. So, x square minus 2 would be 0 at square root 2. So, if we found root of this function x square minus 2 we would have found square root of 2.

So this function does do that and we could use this to find other things also. So say we want to find the cube root of 3, what should we do? Do we need to write everything again? Well not really. We should only right things which have to be different as compared to finding the square root of 2 and there really are two places. So this is the place where we were evaluating the polynomial x square minus 2 and here is a place again we were evaluating the polynomial.

So only in this two places the actual polynomial that we are evaluating is important and for this we would need to evaluate the polynomial x cube minus 3. So over here there should be x cube minus 3 xm cube minus 3 and there should be xl minus 3 again so that is the change that we want to make. So we could we could change the text but is there a better way of doing it? So the general question is can we right a bisection function which finds the root of any mathematical function what so ever provided satisfies the requirements of bisection and those requirements were that we need to supply to bisection two values xl and xr such that the signs of the function who’s root we want to find are not identical at xl and xr. So that is what bisection depended on and we should satisfy that but we would like it like a single function to be used to find the root of any mathematical function. So a natural idea for doing this is -

(Refer Slide Time 03:46)



A C++ function to find the root of any mathematical function

- Natural idea:
  - pass an extra argument to `bisection`
  - argument specifies the mathematical function whose root we want to find.
- How do we represent a mathematical function?
  - Natural idea: As a C++ function computing the required value.

```
double f(double x){return x*x - 2;}
double g(double x){return x*x*x - 3;}
```

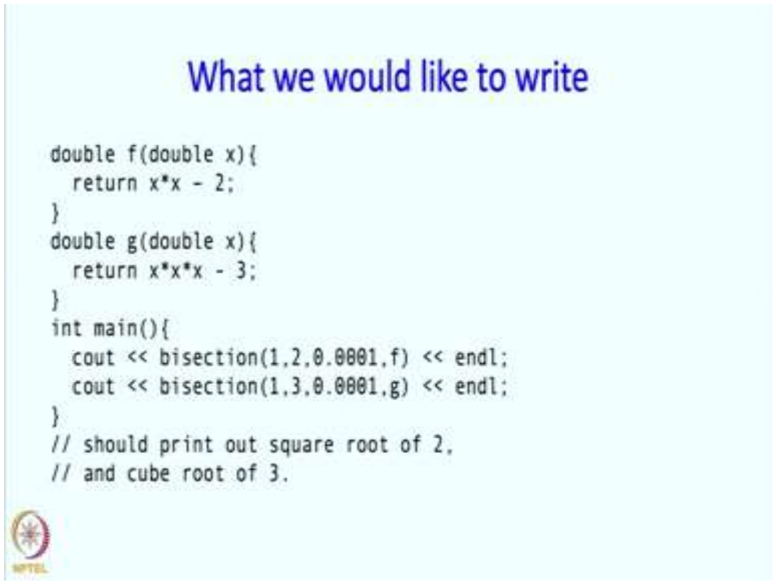
- "Passing a mathematical function to another function":
  - Can we pass one C++ function as argument to another C++ function?

Well to bisection we should pass somehow to pass the mathematical function whose root we want to find so there is an extra argument and that argument should specify the mathematical function of whose root we want to find. Now how do we represent mathematical function how do we tell bisection that look here is the function. A natural idea again is that after all we have C++ functions which compute the function with whose root we want to find.

So suppose we could somehow send that C++ function that should somehow be adequate. So in the case of our square root of 2 we were trying to find the root of this function f(x) equals x square minus 2 or the new thing we want to do use bisection to find a cube root of 3 we want to find the root of this function.

So, somehow we should supply this function f and g to the bisection function through that extra argument and maybe that should some we should somehow be able to make it work. So now the question is that we have a C++ function bisection and to it we want to pass another C++ function ok f or g whichever one is a be might be interested so how do we do that.

(Refer Slide Time: 05:13)

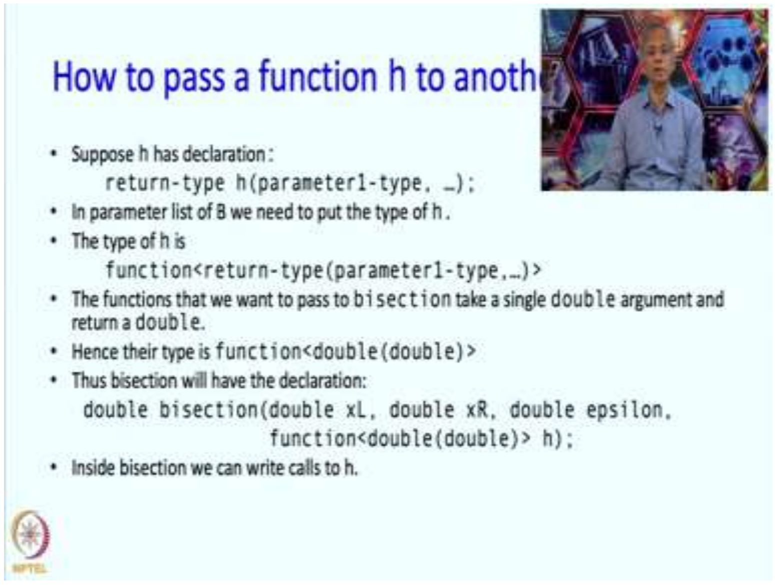


What we would like to write

```
double f(double x){
  return x*x - 2;
}
double g(double x){
  return x*x*x - 3;
}
int main(){
  cout << bisection(1,2,0.0001,f) << endl;
  cout << bisection(1,3,0.0001,g) << endl;
}
// should print out square root of 2,
// and cube root of 3.
```

So, in short this is what we would like to right as far as rest of the code goes. So here, we have define our f we have define our g and in the main program we have an extra argument so this time we have given the argument f presumably they should somehow send this function over to bisection to the function bisection and bisection should print the root of root corresponding to this that should find and print the root of square root of 2 and similarly in this case bisection should be sent this function this g over here and after it executes bisection should print the cube root of 3.

(Refer Slide Time: 06:10)



How to pass a function h to anoth

- Suppose h has declaration:
  `return-type h(parameter1-type, …);`
- In parameter list of B we need to put the type of h.
- The type of h is
  `function<return-type(parameter1-type,…)>`
- The functions that we want to pass to `bisection` take a single `double` argument and return a `double`.
- Hence their type is `function<double(double)>`
- Thus bisection will have the declaration:

```
double bisection(double xL, double xR, double epsilon,
                 function<double(double)> h);
```

- Inside bisection we can write calls to h.

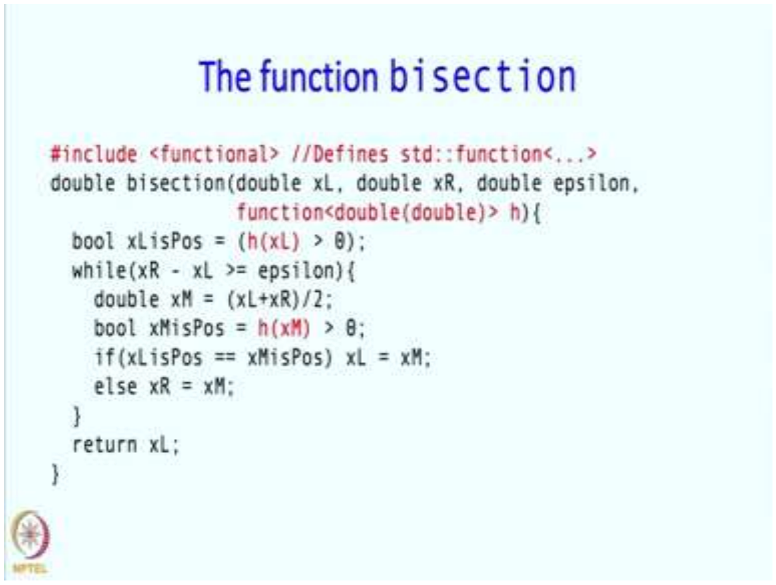
Now exactly how do we pass a function h to another function b well let us see - suppose h has this declaration So, written type h and the parameter test then in parameter list of b to which we want to pass s we need to put in the type of h. Now it turns out the type of h of a function defined in this manner is something like this so it’s this funny looking funny looking quantity, funny looking string. So, it begins with this word function and angle braces, return type all of these things.

Now this a general h but the function we want pass to bisection takes a single double argument and returns the double all such functions that we want to pass are going to take a single double argument and return a double. So, we can say the type of all such functions is function return type double and single parameter type also double.

So, we have discovered how we can declare what should be the parameter list of b or of bisection so we need to put this in the parameter list of bisection. So, bisection is going to have this declaration so these were the original parameters and this now is the new parameter.

So this way, the extra argument that we send f or g will be received by bisection as this argument h and in fact inside bisection we can write calls to h so bisection can be just directly used we do not need to modify the function. Modify function to include the code for h explicitly of course we need to modify the function because we have indent that parameter.

(Refer Slide Time: 08:11)

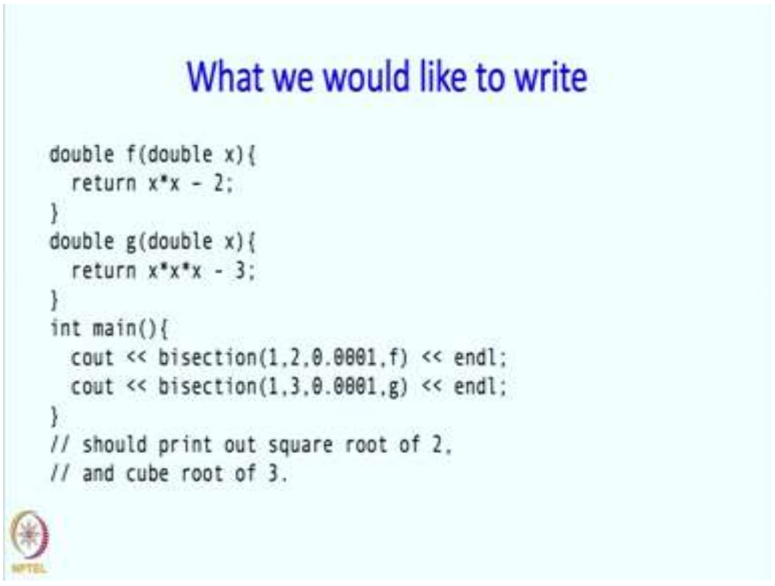


The function bisection

```
#include <functional> //Defines std::function<...>
double bisection(double xL, double xR, double epsilon,
                 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;
}
```

So, here I am showing you the new version of bisection. So, what has happen over here is that the red portions are were some changes happened. So, first of all we need to have we need to include a new header file ok so this header file is called functional. So, this file contains code which allows you right something like this. So, function is the name which is defines in the standard name space ok. So, you want to use all this things you must include this header file functional. So, we have the bisection function and we have a extra parameter now and then if you remember here we were calculating that function who's root we wanted to find and now we wanted to find the root of h so we are going to calculate that h(xl) and here also we were calculating that function xm and again since we now want to find root of h we are just going to h(xm). What this going to do is this is going to call h is after all a function which is present in the memory of the computer. So, it will call h with argument xm and xl here and get that value and use that value that's it ok. So, now we can use this together with the same main program that we have written ok.

(Refer Slide Time: 09:41)

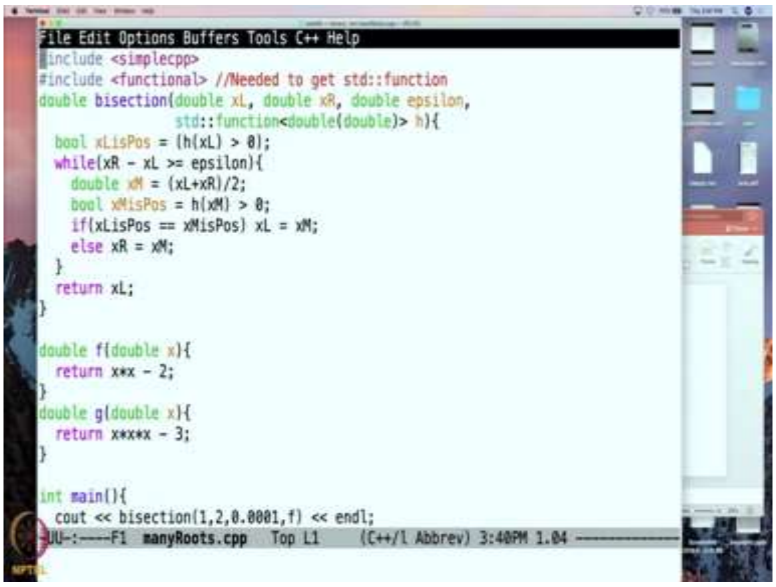


What we would like to write

```
double f(double x){
  return x*x - 2;
}
double g(double x){
  return x*x*x - 3;
}
int main(){
  cout << bisection(1,2,0.0001,f) << endl;
  cout << bisection(1,3,0.0001,g) << endl;
}
// should print out square root of 2,
// and cube root of 3.
```

This together with the bisection function that we just showed will allow us to do what we found desirable over here so this will indeed print the square root of 2 and the cube root of 3. So let us do a quick demo of this.

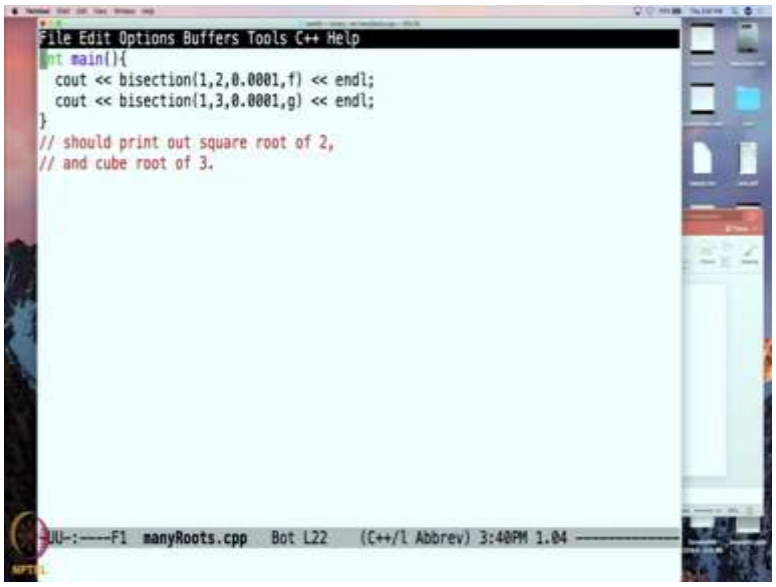
(Refer Slide Time: 09:57)



```
#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;
}

double f(double x){
  return x*x - 2;
}
double g(double x){
  return x*x*x - 3;
}

int main(){
  cout << bisection(1,2,0.0001,f) << endl;
UU-:----F1  manyRoots.cpp   Top L1     (C++/l Abbrev) 3:40PM 1.04 ------------
```

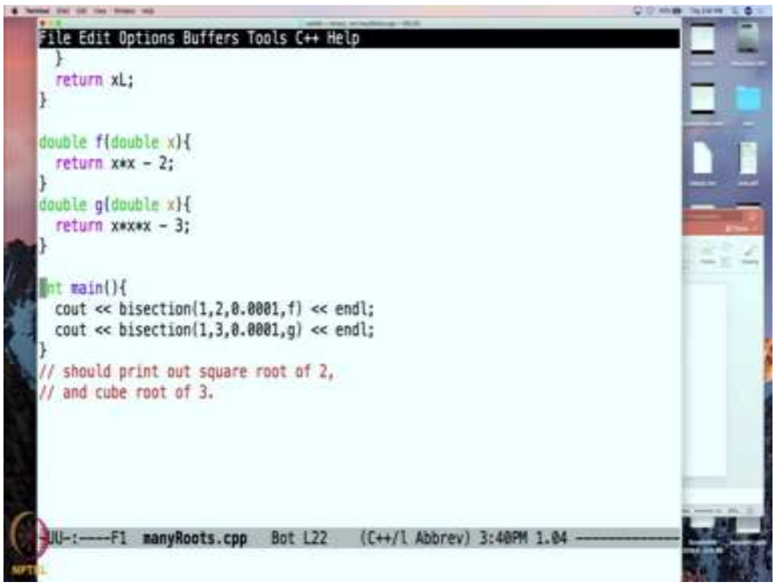
So, many roots is this program so at the top I have written down the bisection as we had in the slides then we have f and g also has in the slides and then over here we have our main program.

(Refer Slide Time: 10:08)



```
int main(){
  cout << bisection(1,2,0.0001,f) << endl;
  cout << bisection(1,3,0.0001,g) << endl;
}
// should print out square root of 2,
// and cube root of 3.
```

```
-UU-:----F1  manyRoots.cpp   Bot L22    (C++/l Abbrev) 3:40PM 1.04 ------------
```

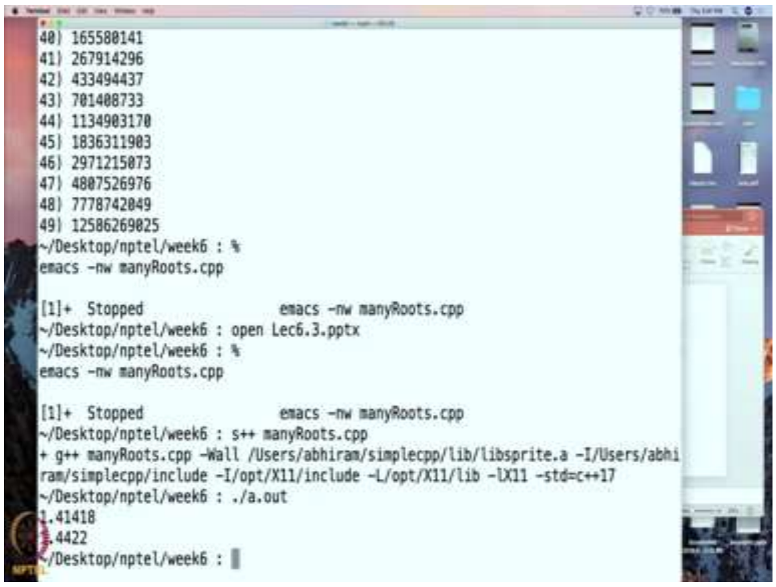


```
  }
  return xL;
}

double f(double x){
  return x*x - 2;
}
double g(double x){
  return x*x*x - 3;
}

int main(){
  cout << bisection(1,2,0.0001,f) << endl;
  cout << bisection(1,3,0.0001,g) << endl;
}
// should print out square root of 2,
// and cube root of 3.
```

```
-UU-:----F1  manyRoots.cpp   Bot L22    (C++/l Abbrev) 3:40PM 1.04 ------------
```

So, main program is doing just what we said. So, ideally if this runs correctly we should expect square root of 2 to be printed and cube root of 3 to be printed ok. So, lets run it so let's first compile it and then run it.

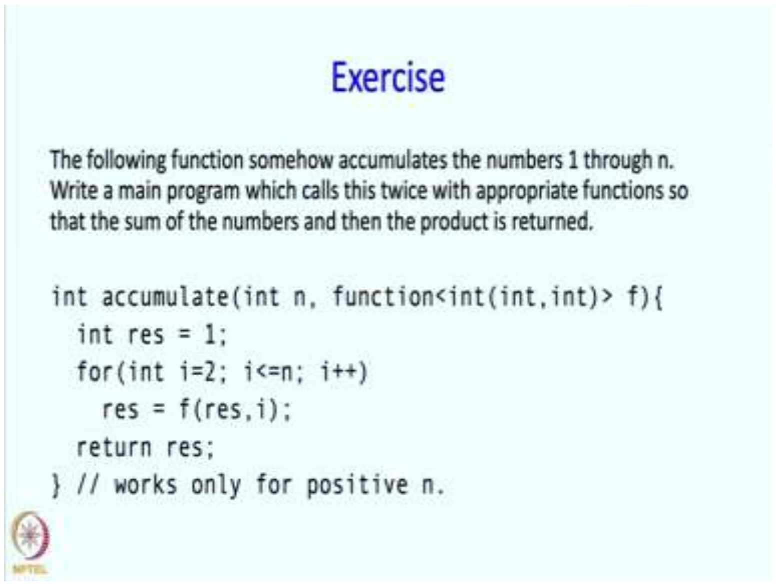
(Refer Slide time: 10:34)



```
40) 165580141
41) 267914296
42) 433494437
43) 701408733
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 : 
```

So, indeed it prints out 1.41418 which is a good approximation to square root 2 and 1.4422 which is a good approximation to cube root 3. So, here is a exercise for you.

(Refer Slide Time: 10:59)



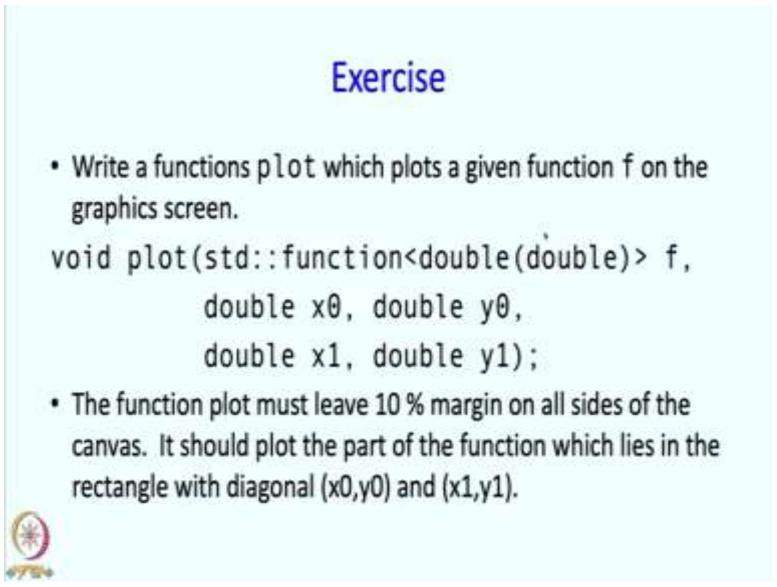
Exercise

The following function somehow accumulates the numbers 1 through n. Write a main program which calls this twice with appropriate functions so that the sum of the numbers and then the product is returned.

```
int accumulate(int n, function<int(int,int)> f){
  int res = 1;
  for(int i=2; i<=n; i++)
    res = f(res,i);
  return res;
} // works only for positive n.
```

Now what is given here is a function which does take an argument a function f which takes two integer arguments and returns a integer and it is somehow using this inside and it is going to return this result value. What you are suppose to do is write a main program which calls this twice with appropriate functions past for f. So, that the first time around it should return the sum of the numbers between 1 and n. The first call should return the sum of numbers and in the second call should return the product of the numbers one through n. So, this should be n factorial.

(Refer Slide Time: 11:49)



Exercise

- Write a functions plot which plots a given function f on the graphics screen.

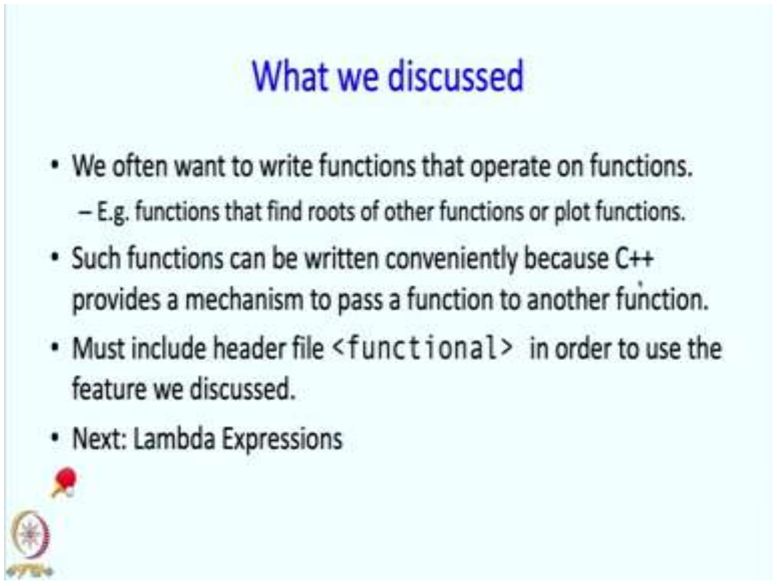
```
void plot(std::function<double(double)> f,
          double x0, double y0,
          double x1, double y1);
```

- The function plot must leave 10 % margin on all sides of the canvas. It should plot the part of the function which lies in the rectangle with diagonal (x0,y0) and (x1,y1).

And here is a little bit more detail exercise was you supposed to take a function and range of x values and y values. So, you should plot the values of the function which start at x0 and go to x1 and which lie in the range y0 and y1. And to look nice you should do things like maybe plot that rectangle or show that rectangle maybe leave some margin just try to make it nice.

But here is a really good use of how you might pass a function. If you want to find or show the plot of the function then you might want to write a single function which does the plotting and to which you can just pass functions whatever functions we want.

(Refer Slide Time: 12:36)



What we discussed

- We often want to write functions that operate on functions.
  - E.g. functions that find roots of other functions or plot functions.
- Such functions can be written conveniently because C++ provides a mechanism to pass a function to another function.
- Must include header file `<functional>` in order to use the feature we discussed.
- Next: Lambda Expressions

So what we have discuss. So, we say that we often want to right function that operate on functions. So, say for example functions which find roots of their function or plot function. Such functions can be written conveniently because C++ provides a mechanism to pass a function to another function and for this you need to include the header file functional which tells you which allows you to define the type of the functions you are passing. This concludes this segment in the next segment. I am going to talk about lambda expression which are which is one more way of doing all this and a little bit more convenient way at time. So, we will take short break.