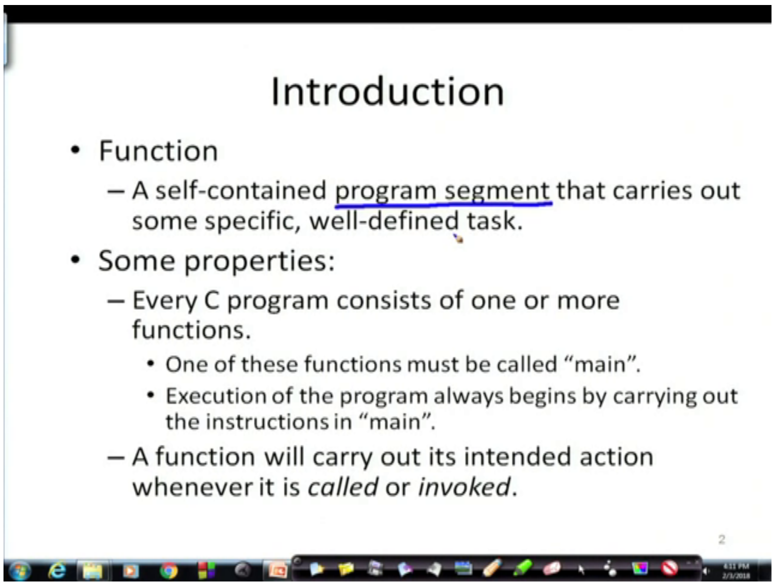
Problem Solving through Programming in C
Prof. Anupam Basu
Department of Computer Science & Engineering
Indian Institute of Technology, Kharagpur

Lecture – 36
More on Functions

We have seen what a function is, and why a function is required, what are the advantages of using a function? We have also seen, what is a parameter and what is an argument, and the types of the parameters in the argument must match. Otherwise, the system will of course, give you error and the reason is because there will be type mismatch and the data will not be passed properly ok. Because this is essentially a parameter is being assigned to an argument ok. Now, in this lecture we will look at some more detail nuisance, detailed points of function. So, to start with let us revise what we have seen.

(Refer Slide Time: 01:03)

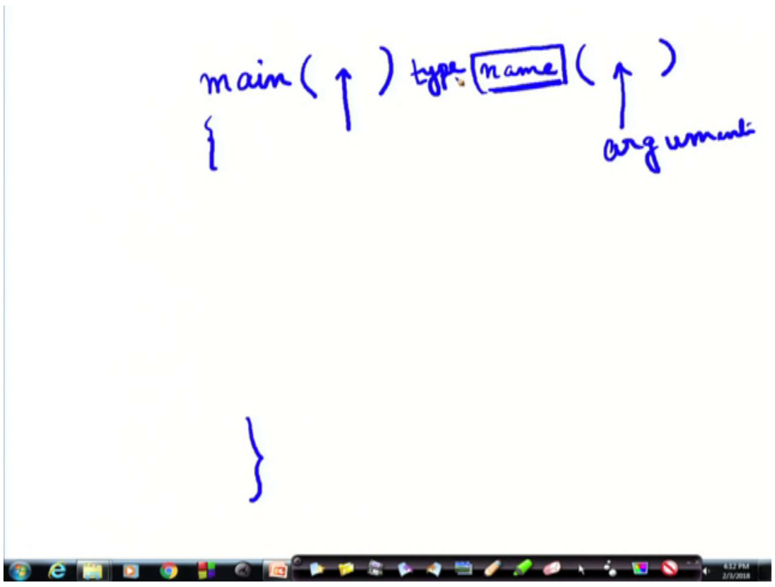


Introduction

- Function
  - A self-contained program segment that carries out some specific, well-defined task.
- Some properties:
  - Every C program consists of one or more functions.
    - One of these functions must be called "main".
    - Execution of the program always begins by carrying out the instructions in "main".
  - A function will carry out its intended action whenever it is *called* or *invoked*.

A function; what is a function? A function is a self-contained program segment. This very important; this not the whole program, but it is self-contained, and it is a program segment; part of a program. That is a main task and a part of that being executed is being implemented by this function, and this self-contained because this can be independently tested giving data and finding out, whether the task for which it has been designed is actually being fulfilled or not ok. It carries out some specific and well-defined task. Now that is the general concept of a function, now in C, any C program can consist of one or more functions. At least one should be there why, because we need always the main right we have seen that we need main.

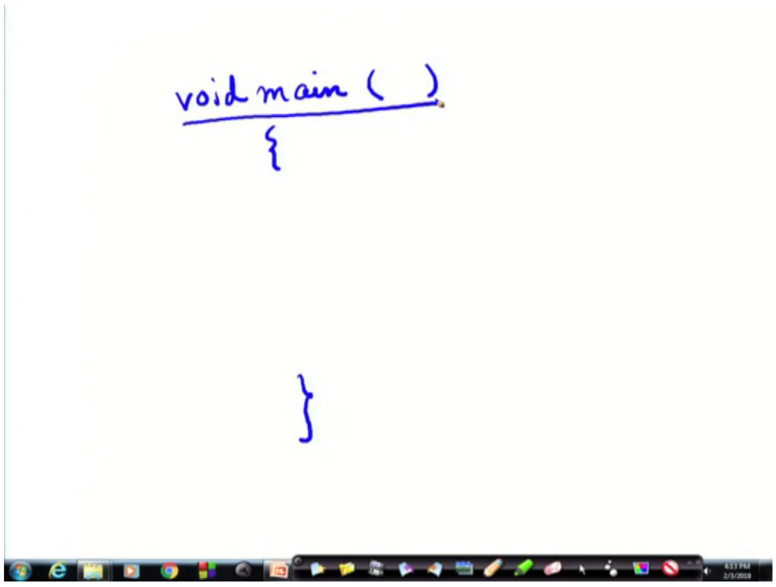
(Refer Slide Time: 02:12)



Any C function must have a main. And main is nothing but a function. And why did we put here this empty bracket; because the structure of a function is always a function name and a place for the arguments right.

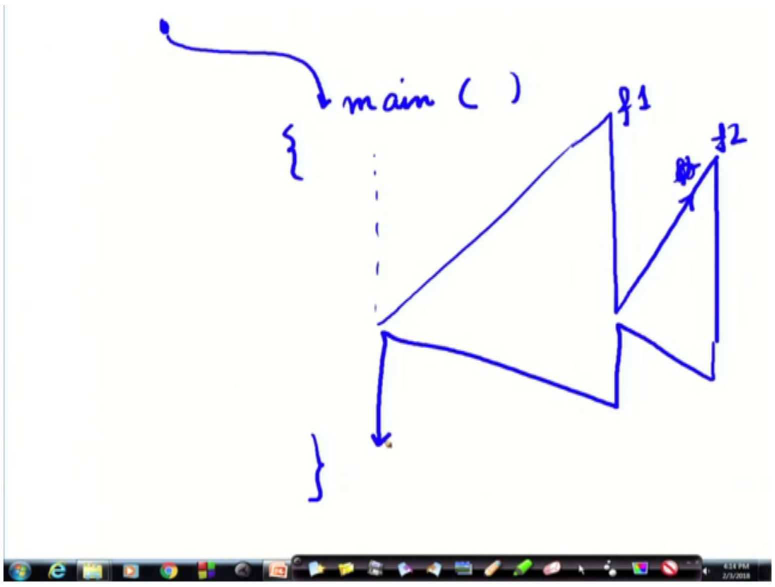
Now, you can say that well, when we are starting a main function we are writing a main function that we will do the task, it is not taking data from anywhere outside. Therefore, why there is no argument. Yes, you are right. There is no argument that is why we put 2 braces and there is an empty space here, this is empty, right? But still it looks like a function. Moreover, in the earlier lecture, we have seen that the result of the function is returned in it is name. And since this is a with like a variable, it will have a type. So, some type like for average we found float.

(Refer Slide Time: 03:37)



So, this main can also have a type. So, we will see later that this main can have a type. We will have a set of empty arguments and the body of the function, and there can be some there can be type. Say, sometimes a type can be int. If this main function; that means, this main function will return and integer and when we say that it does not return anything, we can say also that this function is of type void; that means, it does not return any value ok, void. So, we will see this later, but this is the reason why we say that every C program must consist of one or more than one function. One of these functions must be called the main; and the execution of the program always begins by carrying out the instructions in main. That is the entry points any program. I mean, I cannot say that main will be somewhere in the middle no.

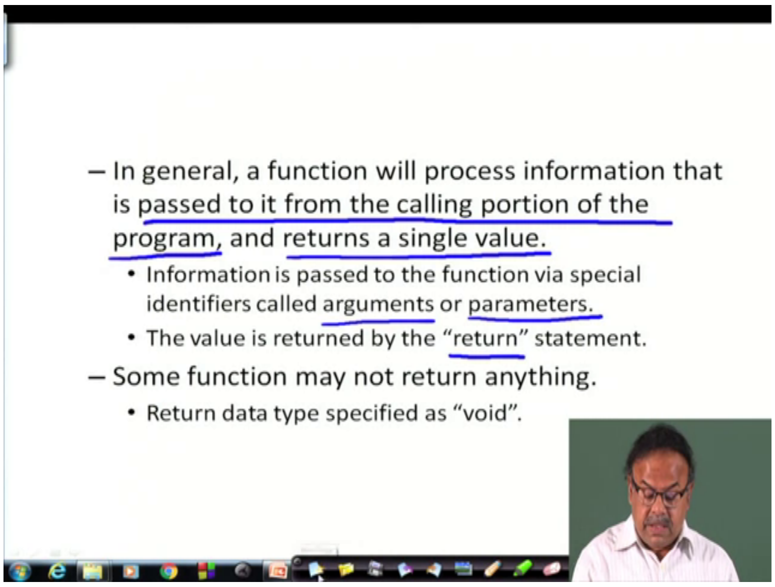
(Refer Slide Time: 03:51)



When I start, when I start, it must the gateway is the main. And then from within main, I can go out and use some function f 1, and from f 1 I can go out to another function f 2, another function sorry, another function f 2 like that.

But the gateway the entry should be main; and ultimately the exit say f 2 will return here, then f 1 will return here. Ultimately the exit will also be through main, alright? A function will carry out it is intended task, whenever it is called or invoked. So, we have encountered with this term called or invoked right.

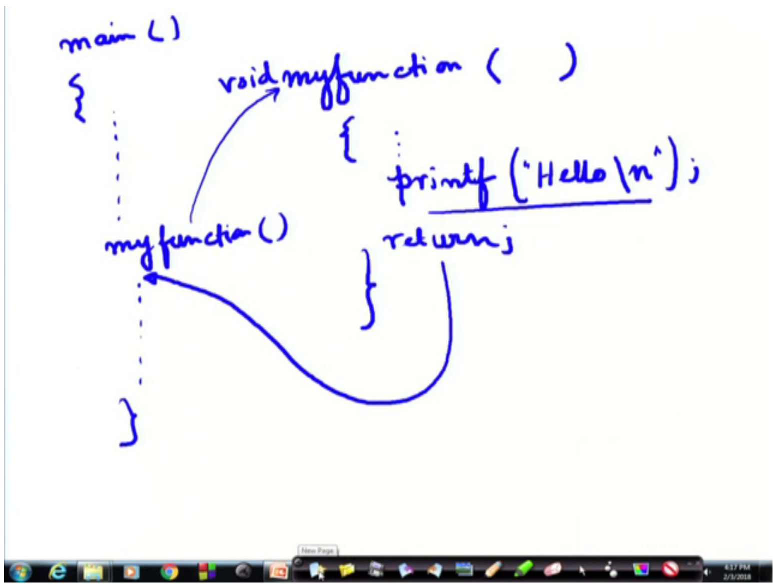
(Refer Slide Time: 05:51)



- In general, a function will process information that is passed to it from the calling portion of the program, and returns a single value.
  - Information is passed to the function via special identifiers called arguments or parameters.
  - The value is returned by the "return" statement.
- Some function may not return anything.
  - Return data type specified as "void".

So, in general, a function will process information, that is passed to it, from the calling portion of the program. From, where it has from where it has been called or invoked. And then, the function will take the data; that is, if some data is to be passed, then it will be passed to it from the calling part. And it will return a single value. So, what are these points? Information is passed via special identifiers, called arguments and parameters. And the value that is returned is returned by the return statement. Some functions do not return anything. For example, let us say this. Say, I am writing a function, say my function.

(Refer Slide Time: 07:01)



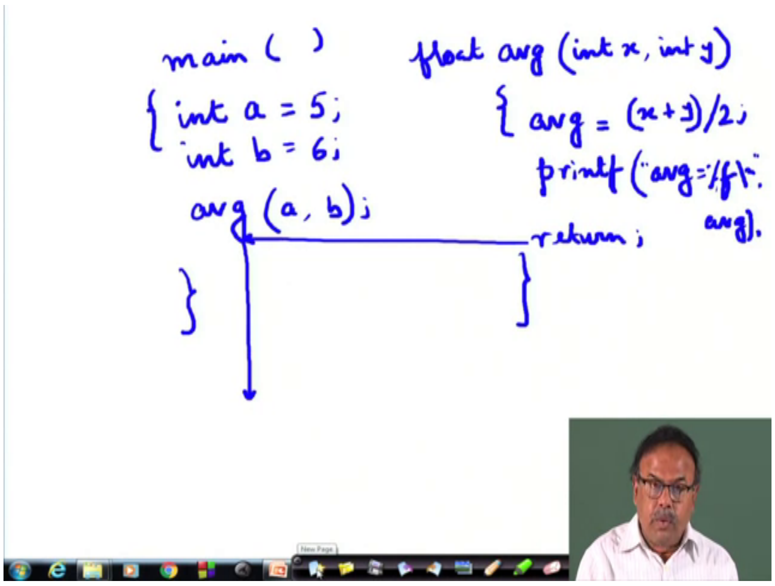
```
main ()
{
  .
  .
  myfunction()
  .
  .
}

void myfunction (   )
{
  .
  printf ("Hello\n");
  return;
}
```

It is maybe, it is not taking any parameters. Because in the body it is simply nothing doing nothing just printing printf, hello, let us also valid function. So, in this case the main function, where it was it was just here and it just called my function here with nothing this also possible. So, my function it goes there and does not. So, I do not need to give any return, but if I give a return that is also fine it does not return anything. So, this of type void it is not returning anything. So, but the control in any case it does not return any value, but the control will come back here and from here it will be executed.

So, there may be some functions, which do not return any value like it just printing some data ok. That is possible for example; I can make it even more realistic.

(Refer Slide Time: 08:35)



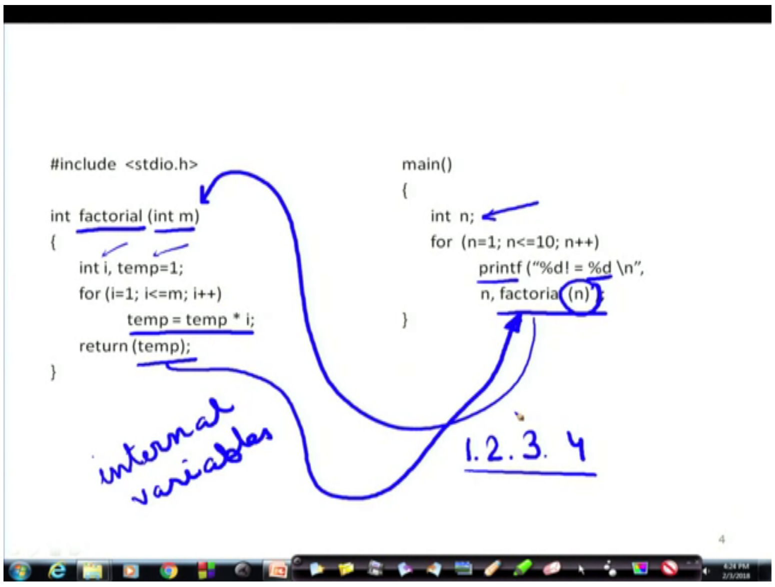
```
main ( )
{ int a = 5;
  int b = 6;
  avg (a, b);

}

float avg (int x, int y)
{ avg = (x+y)/2;
  printf ("avg=%f\n",
                 avg).
  return;
}
```

Say here main is here, and there is int A 5, int B 6, alright? And here, I call average a comma b, right. And here I write average int x, int y. And here I just compute average is equal to x plus y by 2. Printf a v g equals back slash, a v g equals, I am sorry, percentage f back slash n comma a v g. That you understand. And I can write return or I may not write return. So, what will happen? This will come here 5 and 6, 5 will be passed on here x will be 5 y will be 6. The value will be 5.5 that will be printed here. But I have to give a type to this average, because here the average type is not defined. So, I have to say this will be float average. Because there is some value, some type to this value some type to this variable. Which will hold the value? So, this also possible here you can see that, I do not need to write a return, but there is no harm, if I write just simply return, that is the good practice; that means, my return my control will come back here. And that is the last statement that will be the end ok. So, let us move forward. So, some function may not return anything the return type is specified as void.

(Refer Slide Time: 11:21)



```
#include <stdio.h>

int factorial (int m)
{
    int i, temp=1;
    for (i=1; i<=m; i++)
        temp = temp * i;
    return (temp);
}
```

```
main()
{
    int n;
    for (n=1; n<=10; n++)
        printf ("%d! = %d \n",
        n, factorial (n));
}
```

So, here is an example of factorial. We start with so, here first I have written the function here, and then I have written the main.

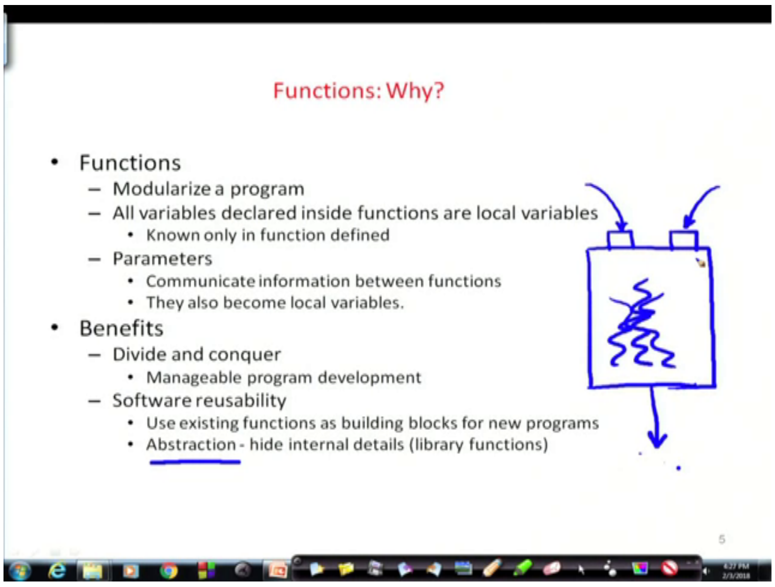
But however, in whatever way we write, the main the program while execution we will enter the main first, alright? So, so the execution will be something like this that will enter from here main. It takes the variable n, which is an integer; then n1 to less than 10 n plus, printf percentage d ok, and n and then factorial alright. So, factorial a now when this factorial n is encountered, then we the; this function is called. This function what is doing? It has got int m. So, m is the argument, and is being called with a parameter n; which has been which is being taken; first 1 then 2 then 3. Like that till it is 10.

So, I am taking it n, and then here let us see, what we are doing int i. Now, this i this variable i and this variable temp. These 2 variables are purely internal variables; that means, this variables are internal to the body of the function. Now each of the functions has got a life, each has a life right. So, factorial as long as factorial is running factorial is live, and when factorial is live, whatever variables are defined within factorial, they are live. As soon as we exit from factorial, that function is dead. And the variables associated solely I repeat. The variables associated solely with the function also dies with the death of the function. So, let us see here i and temp. Then for i equal to one to m temp equals temp times i. So, 1 times, 2 times, 3 times 4.

So, it is what is being computed, 1 times 2 times 3 times 4. So, like that whatever is the value of m that is being up to that we are computing a factorial, alright? So, if I call it, now here I have called it with n the value of n, alright? With a value of n, I am calling it, and I am getting the factorial. And the temp is being returned, where it is being returned, this temp is being returned to factorial temp is being returned to factorial, and that is being printed through this percentage d, alright?

So, this is being printed. So, return temp, because why it is different? Here I did not write return, because I have named it as factorial, and I have written temp then temp is going to this value factorial. If I had not done return then ultimately; because here I am keeping the result in temp; not in factorial has not been used as any variable inside the function. Now when I go there, and suppose if I had use some other temp here, this temp and this temp would have been different. We will come to this later, but this just an example of a function being invoked.

(Refer Slide Time: 15:58)



Functions: Why?

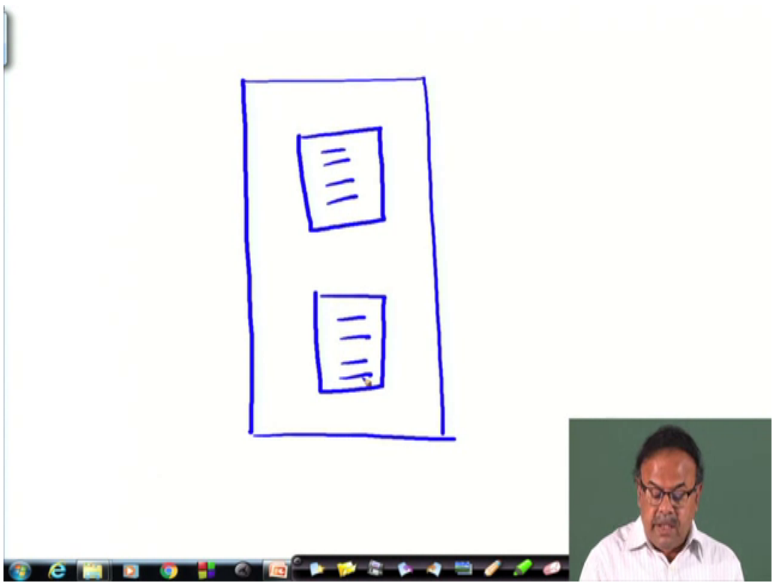
- Functions
  - Modularize a program
  - All variables declared inside functions are local variables
    - Known only in function defined
  - Parameters
    - Communicate information between functions
    - They also become local variables.
- Benefits
  - Divide and conquer
    - Manageable program development
  - Software reusability
    - Use existing functions as building blocks for new programs
    - Abstraction - hide internal details (library functions)

We just recapitulating, why we need functions? This term may be new to you. It is modularising a program, is breaking down a program into small parts. Each part independent part is called a module. That is why breaking down a big program into smaller parts is known as modularisation. So, it is modularising a program. All variables declared inside function are local variables. Very, very important. Which are declared inside functions are local variables; that means, they are known only as long as the function is remaining running. As soon as the function completes it is execution. They also seize to exist. Therefore, a value variable I for example, can be used as an internal variable of a function, and can also be used in a some other function or in the main function. They will actually physically be mapped to different memory locations.

So, we have also seen what parameters are. The parameters are communicate information between functions, parameter and argument ok. They also become local variables. So, parameter arguments are local variables the parameters means, the arguments are also local variables. The benefits are divide and conquer we know we, we has we have devoted one completely lecture on that; manageable program development. Software reusability; all existing functions can use can be used as building blocks for new programs, and this is another thing that you may realise later, that inside a function, how I am implementing or how somebody has implemented, it may be very complicated, and I need not bother about that.

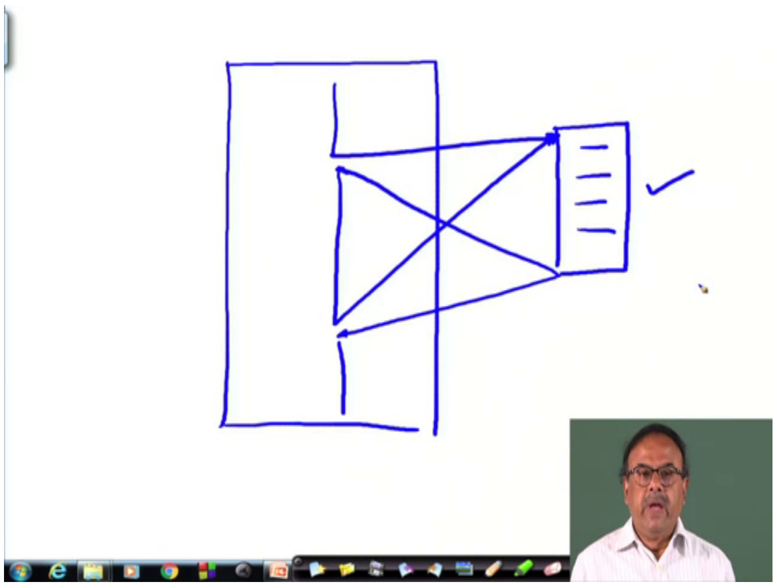
I know what it asks for, what variables have to be put in through this and what output I will get out of it? Therefore, whatever is there inside is abstracted out from it is hidden from it. The internal details are not needed; that is, another big advantage of using functions. And of course, avoid code repetition; something which somebody has written. Or say for example, in a main program. Let us take another example to understand.

(Refer Slide Time: 19:12)



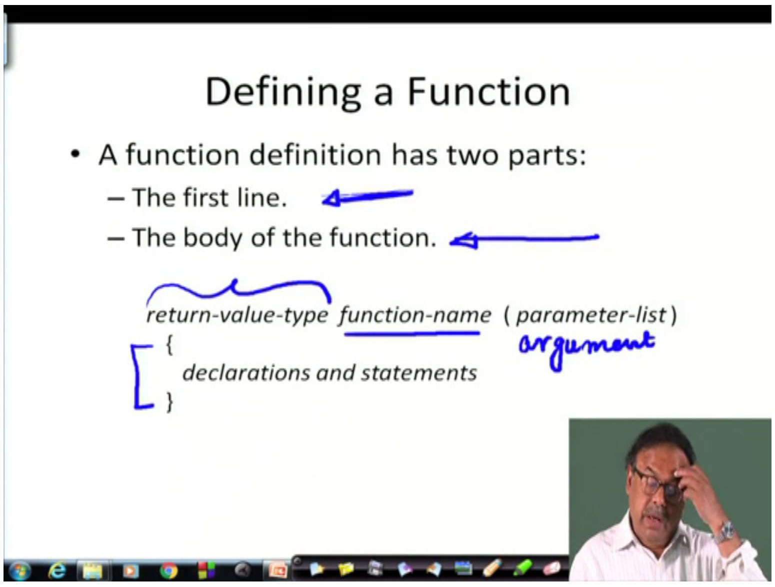
This better say, I have got a program, in which I am doing some computations. Say, computing the standard deviation here, I am computing the standard deviation here, and may be. So, at both these places I write this, write this, but that is not needed.

(Refer Slide Time: 19:36)



If I have a function for computing standard deviation then, here is my main program and there is one function for standard deviation here. And from this point, I can simply call this get the value again continue, from wherever I need I can again call this, and I can get the value, and in that way, I save in the number of lines of code that I write. So, that is known as avoiding code repetition.

(Refer Slide Time: 20:16)



Defining a Function

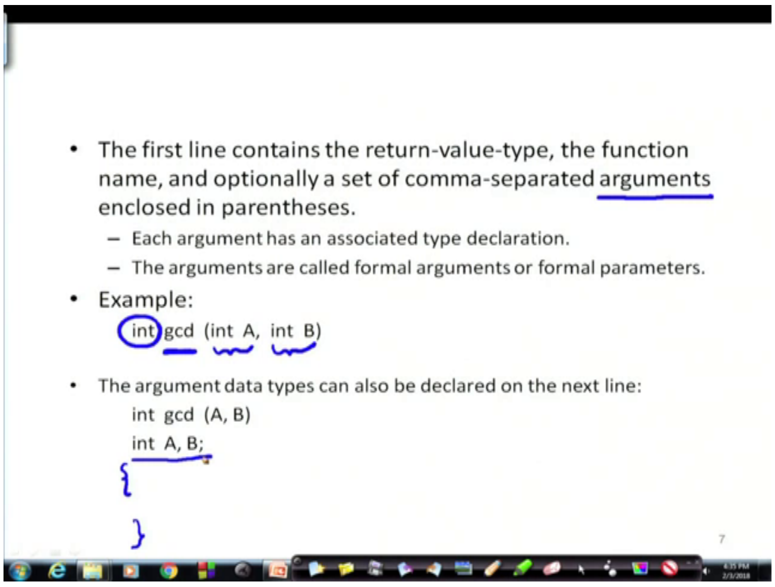
- A function definition has two parts:
  - The first line.
  - The body of the function.

```
return-value-type function-name ( parameter-list )
   {
     declarations and statements
   }
```

Now, defining a function; how do you? Now we are till now I was trying to explain you somethings, now we are looking at the syntactic details of a C function. Now that can vary from language to language, but more or less the ideas are same. So, a function definition has got 2 parts. The first line, first line and the sorry, the first line and the what is happening here? First line and the body of the function ok. Now, what is the first line typically? What can we have in the first line? First line we have got the function name. I do not know why it is not coming. In the first line we have got the function name, and the parameter list. I call it argument list, but whatever you.

So, a function has got the first line. And we can see; what the first line is? The first line will contain the function name, and a return value type, what type of value is being returned. Followed by this is the body of the function. The second part is the body of the function. Where there are declarations and statements. Some internal values can be declared internal variables can be declared here. And the other statements are also here. Now, so, that is; so, this one I actually I am calling them argument. Some people also call both of them to be parameters ok.

(Refer Slide Time: 22:22)



- The first line contains the return-value-type, the function name, and optionally a set of comma-separated arguments enclosed in parentheses.
  - Each argument has an associated type declaration.
  - The arguments are called formal arguments or formal parameters.
- Example:

```
int gcd (int A, int B)
```

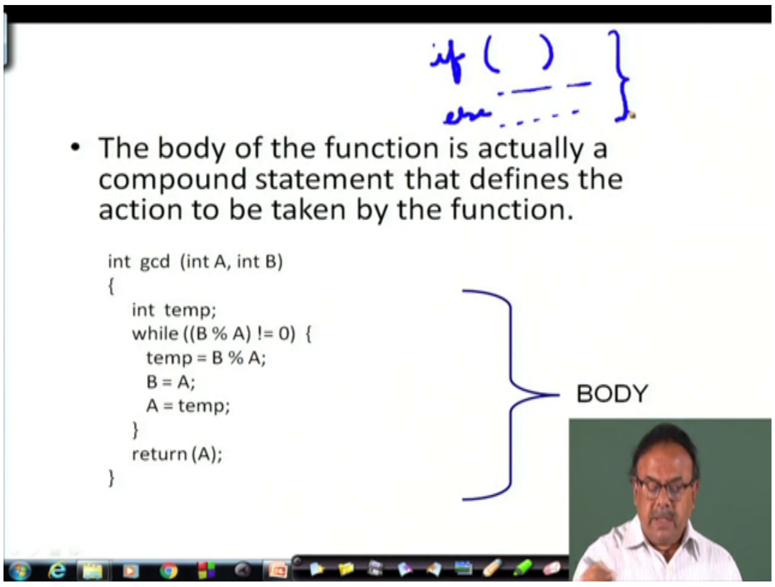
- The argument data types can also be declared on the next line:

```
int gcd (A, B)
int A, B;
{
}
```

Now so, the first line contains the return value type. The function name and optionally a set of comma separated arguments enclosed in parentheses. Like, say here I am defining a function called gcd, greatest common deviser. It has got a type int, and has got 2 arguments. One is A and one is B, I have written the types of them in this parentheses itself int A, comma int B they are comma separated.

The argument is possible that I can declare the arguments also on the next line. For example, I can say instead of writing it in this way, I can write in this way int gcd A comma B and in the next line before the body, before the before the bracket, probably in earlier example I did a mistake, I had put the bracket inside the bracket. It should not be inside the bracket. Immediately after that it you can write int A comma B; that means, you are declaring that these are A and B. Either inside or immediately after that, both of them are fine. Now these are called the formal arguments or some people also call it formal parameters.

(Refer Slide Time: 24:04)

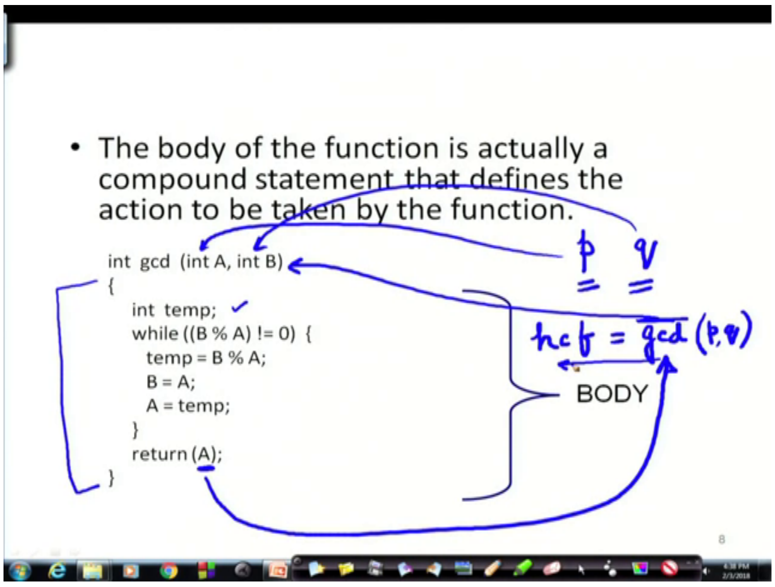


- The body of the function is actually a compound statement that defines the action to be taken by the function.

```
int gcd (int A, int B)
{
   int temp;
   while ((B % A) != 0) {
     temp = B % A;
     B = A;
     A = temp;
   }
   return (A);
}
```

Now what is the body of a function? The body of a function is actually a compound statement. I will just take just as I had statements like for loop ok, or. So, let us take or say while loop or if conditions, where I write if some condition then some statement, else some statement like that. So now, this whole thing is a compound statement. Similarly, here the entire body of the function can be assumed to be a complete compound say statement; where we are passing on to arguments and is giving us some value. Let us look at what is being done here. In gcd int A, int B. So, from the calling function must be I am supplying some values p and q, may be 2 integers of type integers, which are mapping to this p and q A and B.

(Refer Slide Time: 25:02)



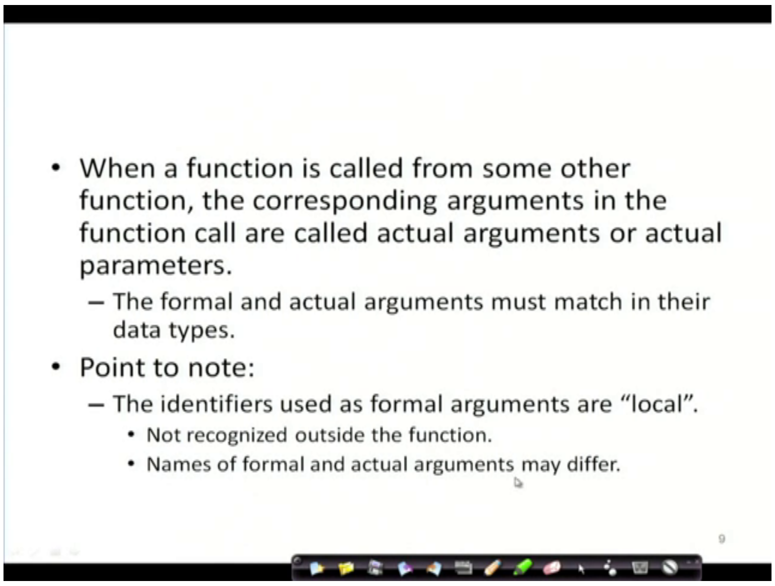
- The body of the function is actually a compound statement that defines the action to be taken by the function.

```
int gcd (int A, int B)
{
   int temp;
   while ((B % A) != 0) {
     temp = B % A;
     B = A;
     A = temp;
   }
   return (A);
}
```

Now here you know we have seen this algorithm earlier. We take a variable temp. Now that is now the body of the function is defined by these braces. Temp, while B is not divided by A, as long as that does not happen. We take we divide B by A and take the remainder, and then go on that classical way of finding the gcd, you find the reminder and make the remainder the deviser, right? So, that is exactly what we are doing in a loop.

Ultimately the final deviser which divides and gives us A 0, that will be the when it that will be the gcd. So, that gcd now; that gcd is coming in this A. Therefore, I cannot here just write simply return. I have to say that return A. And may be from the main function I had something like say hcf is equal to gcd p comma q. Now, this a is being returned to this gcd. And because this g c because it was a gcd function, this was called and that is why it is going to a is coming to the here, and then by this assignment statement that is going to hcf. That is how the function is being connected ok. We also see what is the body of the function?

(Refer Slide Time: 26:59)

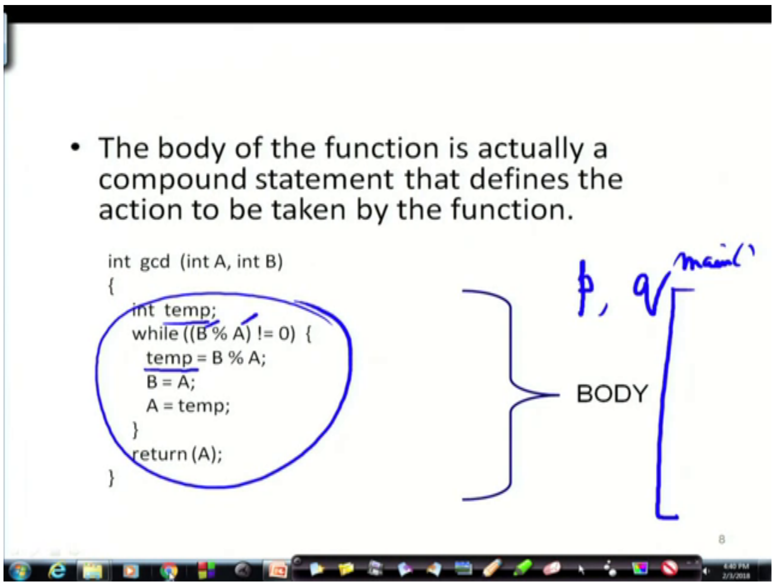


- When a function is called from some other function, the corresponding arguments in the function call are called actual arguments or actual parameters.
  - The formal and actual arguments must match in their data types.
- Point to note:
  - The identifiers used as formal arguments are "local".
    - Not recognized outside the function.
    - Names of formal and actual arguments may differ.

9

So, when a function is called from some other function. The corresponding arguments in the function are called actual arguments or actual parameters; that means what I am calling parameter, some people call them actual parameters. So, just in the earlier example p q were actual parameters. And what was there a b or x y, whatever was there is a b are the are the formal parameters or formal arguments. Now as we had said, the formal and the actual arguments must match in their data types. The identifiers used as formal arguments are local. Not recognised outside the function. The names of formal and actual arguments may differ as we have seen here, they are differing my actual arguments were p q.

(Refer Slide Time: 28:01)



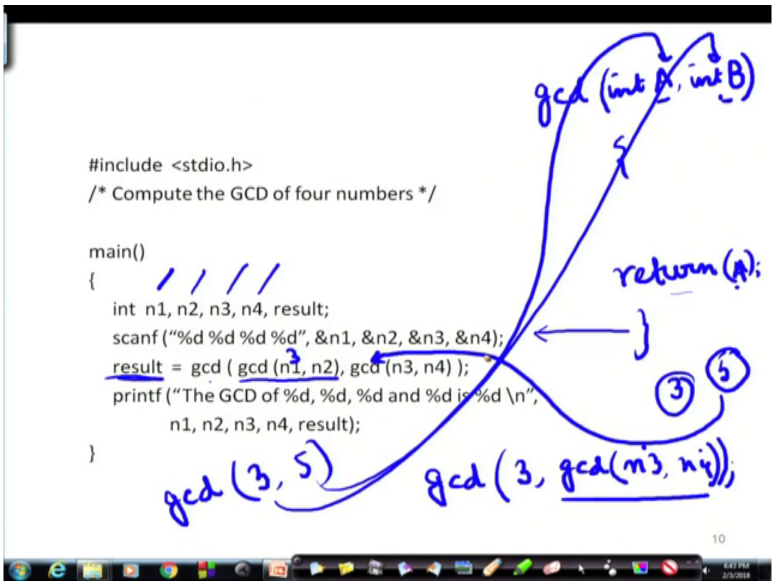
- The body of the function is actually a compound statement that defines the action to be taken by the function.

```
int gcd (int A, int B)
{
  int temp;
  while ((B % A) != 0) {
    temp = B % A;
    B = A;
    A = temp;
  }
  return (A);
}
```

And here it is A and B of course, they are differing. Now this A and B, whatever it is here, this A and B will not be reflected anywhere outside this function. Suppose, here there is a main, the main will not get the value of A and B, this completely local here and as soon as it is ends the A and B vanishes, A and B will no longer B available to you. Same is true for temp because temp has been defined internally within this function alright. So, this is a very key thing that you must understand.

So, so, when a function is called from some other function, the arguments are passed points to note that the names and of the formal and actual arguments may differ right.

(Refer Slide Time: 29:01)



```
#include <stdio.h>
/* Compute the GCD of four numbers */

main()
{
    int n1, n2, n3, n4, result;
    scanf ("%d %d %d %d", &n1, &n2, &n3, &n4);
    result = gcd ( gcd (n1, n2), gcd (n3, n4) );
    printf ("The GCD of %d, %d, %d and %d is %d \n",
            n1, n2, n3, n4, result);
}
```

So, here I will conclude with this lecture I will conclude with this example. So, here we are computing the gcd of 4 numbers. In the main function, we have got n1, n 2, n 3, n 4, 4 values have been read here, you can see that they are being read as and n1 and n 2 and n 3 and n 4. They are been read. Now the result is; now gcd of n1, n 2 gcd of n 3, n 4. So, there is one function gcd; which takes int A int B, right that is what we saw. Now first, when within this parentheses, we will first compute this gcd n1 and n 2. So, a will have n1 the value of n1, b will have the value of n 2, and the function will return something, return A as we saw in the last example, return A. And so, the body of the function ultimately ends with return a. And that a will come here suppose that value was 3 say. So, 3 comes here. So, what do I have now? I have got gcd of 3 comma gcd of n 3, n 4. So now, this same function will be called with a having the value of n 3 and b having the value of n 4. And suppose I compute them, and I find ultimately that the gcd is 5. So, the 5 will come here.

Then what happens to this? Then we will compute again called gcd with 3 and 5. So, again here now a will have 3, and b will have 5 and the return of course, a 3 and 5 will be a the result the hcf of 3 and 5 is 1. So, return a that will come here, and the result will be one. That is how the gcd is computed in cascading of the function calls. The same function you see is being called time and again repeatedly from this single statement in the main function. We will continue looking at functions more.