

Programming, Data Structures and Algorithms
Prof. Shankar Balachandran
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

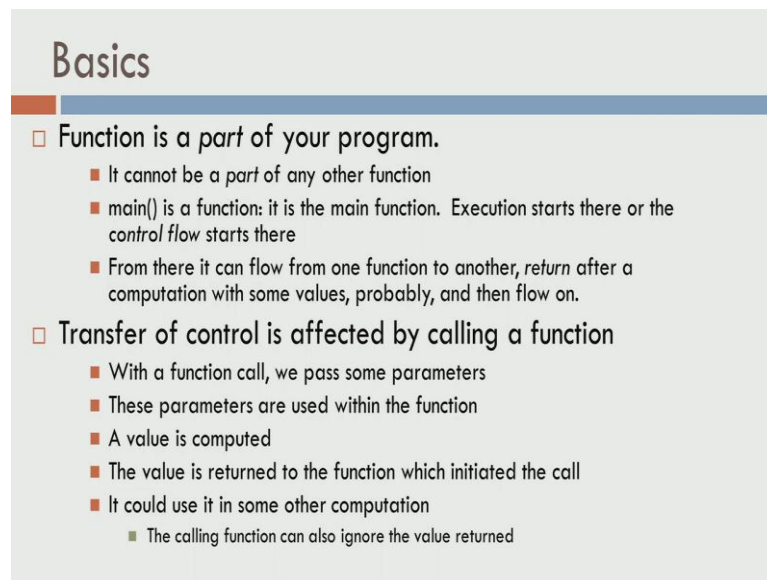
Module - 10B

Lecture - 21

Function definition, function prototype, function invocation or call
Control flow, calling inside a loop

In the previous module, we saw the basics of functions and how it is nice and how it makes it readable and so, on. In this module, we will go and look at this notion of functions and lots of gory detail. So, what is a function?

(Refer Slide Time: 00:26)



Basics

- Function is a part of your program.
 - It cannot be a part of any other function
 - `main()` is a function: it is the main function. Execution starts there or the control flow starts there
 - From there it can flow from one function to another, return after a computation with some values, probably, and then flow on.
- Transfer of control is affected by calling a function
 - With a function call, we pass some parameters
 - These parameters are used within the function
 - A value is computed
 - The value is returned to the function which initiated the call
 - It could use it in some other computation
 - The calling function can also ignore the value returned

Function is essentially a part of your program. So, one thing that you have to be careful about is, it cannot be part of any other function. A function cannot be part of any other function. So, in the previous module we saw that there was a function called main, which is the caller and there was a function called power which was the callee. We cannot go and take the program for power and completely put it within the caller.

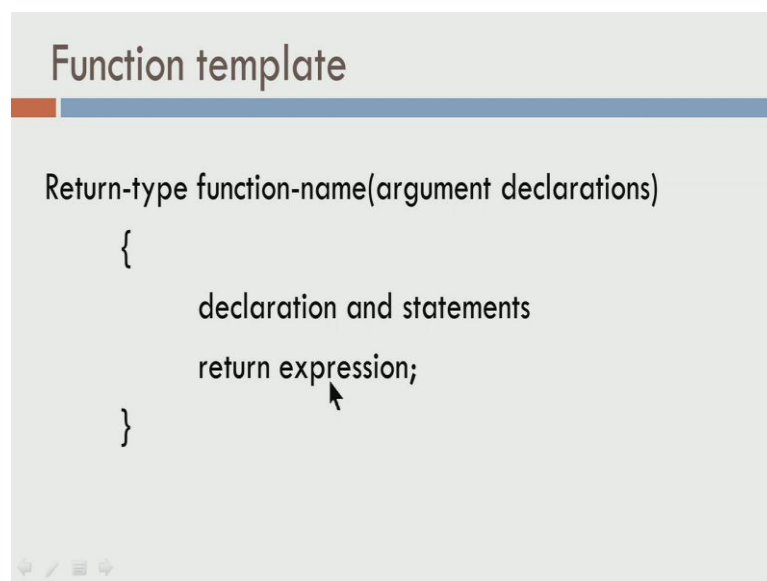
So, a function cannot be completely written up inside another function. So, there are some languages which allow you to do that. C does not allow you to do it, I mentioned this earlier that main itself it is the main function. So, whenever you have multiple sub tasks, somebody has to know, what is the very first task that you have to start with. So, usually it is the main. So, you start with main, main will have a sequence of steps, if there is any delegation of duty that you want from main, the main program should have

that, the main function should have the delegation of duty.

So, we had delegation of duty to power from the main function. So, execution starts from the main function, the control flows starts there, from there the control flow can move to one function or another and return back and so, on. So, every time you call a new function from a caller, you are essentially transferring the control to the callee. So, with the function call you usually pass some parameters. So, in our case we passed base and n and these parameters are used within the function.

You compute some value, the values return to the caller and the caller could use it in some other computation. The caller can also choose to ignore the value that is produced. In fact, printf to be technically correct, actually produces a return value. Only that, generally we choose to ignore the return value from printf. It does some work, it prints things on the screen, it actually returns a value also, only that we usually do not care about the return value. It is not useful for most programmers. Let us look at the template of a function.

(Refer Slide Time: 02:36)



We have a function name and we have all the argument declarations and we have a return type and you may want more declarations inside your program itself. You have further declarations in statements and finally, you have a return expression. So, the function name identifies the name by which you are going to have this sub task call. It may have a set of arguments, the arguments will have their names and the order in which the arguments have to be given.

So, for example, power of base comma n, if you pass 3 comma 5, we will do 3 power 5. Instead, if you say power of 5 comma 3, it will actually rise 5 to 3, you will get 125 as the result. So, you have to be careful in figuring out, what the arguments are and in what order the arguments are passed and you have various declarations that are within the functions. For example, we needed int i and int p which were useful in running the loop and so, on. And finally, we have a return expression, return p earlier and there is a return type in our power function, it was an integer.

(Refer Slide Time: 03:45)

Function Definition in C

```
return-type function-name (argument declarations)
{
    variable/constant declarations and
    statements
}
```

Arguments or parameters:
 the means of giving input to the function
 type and name of arguments are declared
 names are formal - local to the function

Return Value: for giving the output value
 return (expression); -- optional

Invoking a function: *func-name*(*exp₁, exp₂, ..., exp_n*)

No function declarations here!

Matching the number and type of arguments

So, one thing that you have to remember is, in the body of a function, it is a basic block of code and in the body of the function, you cannot have other function declarations. So, it is simply not allowed to have other functions which are completely declared and described within another function and the arguments essentially found the inputs, you need both the types and the order. Sometimes, the return expression is optional.

So, the keyword return can take an expression, it can also not take any expression and just say return or if you do not even have return, your program will come to the final closing brace of the sub task and automatically, it will go back to the caller. So, invoking of a function is usually done using this function name and you have the set of all the parameters that are passed.

(Refer Slide Time: 04:43)

Function Prototype

- defines
 - ▣ the number of parameters, type of each parameter,
 - ▣ type of the return value of a function
- used by the compiler to check the usage
 - ▣ prevents execution-time errors
- function prototype of **power** function
 - ▣ `int power (int, int);`
 - ▣ no need for naming the parameters
- function prototypes - given in the beginning

So, there is this notion of what is called a prototype of a function. So, we will see this in a little while. So, I had that in a program, but I did not go into the regions earlier, but we will see what a prototype is. A prototype is essentially defines the number of parameters and the type of each parameter, it also defines the return value of a function. So, a prototype essentially gives you a black box description. So, when every function has a prototype followed by the actual definition of the function.

So, the prototype just tells you what is the black box description, what are the inputs and what are the outputs and so, on and some where you have to write the program to do the function itself and that is the function implementation. So, the prototype is a necessary thing, the compilers usually check, whether a prototype is present or not and if we give a prototype, the compiler can actually indicate warnings, if you missed up something.

Let us say, you have return a function and it takes three parameters and let us say everywhere you called it, you have only two parameters that are passed. The compiler can spot it and tell you that this function expects three parameters and you have to passed only two. So, the prototype here is for power is as follows, `int power of int comma int`. So, in this place we are actually not named the parameters at all, power does take two parameters, but the names are not mentioned.

So, a compiler when it looks at this prototype knows that power is supposed to be a function which takes two integers, returns another integer. At this point, the compiler would not know, whether base is the first parameter and n is the second parameter or n is

the first parameter and base is the second parameter. Actually, he does not care at this point of time. However, later we actually put the variables base and n here and write a program appropriately. Typically, function prototypes are given in the beginning of the program.

(Refer Slide Time: 06:46)

Complete Program Which Uses Power Function

```
#include <stdio.h>
int power (int, int);
int main () {
    int num1, num2;
    num1 = power(3,5);
    num2 = power(-4,3);
    printf("%d", num1);
    printf("%d", num2);
    return 0;
}
```

```
int power (int base, int n)
{
    int i, p = 1;
    for ( i = 1; i <= n ; i ++ )
        p = p * base;
    return p;
}
```

The image shows a code editor with two panels. The left panel contains the main program code, and the right panel contains the definition of the power function. A pink box labeled "Function Prototype" points to the line `int power (int, int);` in the left panel. Another pink box labeled "Invocation with arguments" points to the lines `num1 = power(3,5);` and `num2 = power(-4,3);` in the left panel.

So, let us see this, I am showing the same piece of code that we had earlier. We have this power function. So, what you see on the right side is called the function definition or function description and what you see in this line here, on the left side is called the function prototype. So, we have function prototype and function definition. So, the function prototype by itself is not very interesting, but it is necessary to tell the compiler as well as tell the other programmers that power actually expects two variables and it will return only an integer, it is not going to return a character, it is not going to return a floating point value and so, on.

So, that anyone who uses the function, knows what they are getting into. So, this is called the function prototype and this is called the function invocation. So, whenever we call the function, we call that function invocation or calling the function. So, earlier I said, I want to touch up on what is called the control flow. I said, flows the program, flow starts with the main program and whenever it calls a function, it goes there and so, on. So, I want to make that slightly more clear now.

(Refer Slide Time: 08:00)

CONTROL FLOW

```
#include <stdio.h>
int power (int, int);
int main () {
    int num1, num2;
    num1 = power(3,5);
    num2 = power(-4,3);
    printf("%d", num1);
    printf("%d", num2);
    return 0;
}

int power (int base, int n)
{
    int i, p = 1;
    for ( i = 1; i <= n ; i ++ )
        p = p * base;
    return p;
}
```

So, let us look at this piece of program and let us see, how this function or how this program is going to get executed. So, as I said earlier, all executions start with main, you start with main. So, at this point you have two declarations, num1 and num2. So, as with any variables, the compiler would have allocated memory for num1 and num2. So, I am ready to execute this line, num1 is power of 3 comma 5. At that point what happens is, instead of getting to the next line here which is what happens usually.

So, when we looked at programs. So, for, for loops and other things, we always recent how this sequence of code that you see on the screen, how will it get executed internally. So, if you have no loops or branches you start from line 1, you go to line 2, line 3, line 4 up to line n and you return. If you have if then else branches, either if condition is true or else condition is true and only one of them works, if we have a for loop you have a repeated body and so, on, all those were examples of control flow. This is one another example of control flow.

So, at this point if you are in line number 2 within main, the control gets transferred to line number 1 of power and then you go and actually do this sequence of operations and once this sequence of operations is done, you return back to this line. When you return back what happens is, you have computed the value power of 3 comma 5, it should have computed number 243, as the result and that is assigned to num 1.

Now, you are again back in the caller, in the caller, you will by default go to the next line. So, the next line is also a function invocation or a function call, again the control is

transferred to the callee. In the callee, it goes through the sequence of code here, at the end of it you return back to the caller. In this case, minus 4 comma 3 would have computed minus 64, that value is assigned to num2. So, it returns to this location and then you are ready to do printing of both these numbers.

So, the control flow for this program is, you start with main, at this line you call this side. So, remember for every assignment operation, you evaluate the right side and you assigned it to the left side. So, there is evaluation on the right side. So, power of 3 comma 5 would transferred the control to the power function and you do the sequence of steps, the control comes back here, you do this assignment, the control falls to the next line by default. Again the next line on the right side it makes a function call, you do this you come back here, do the assignment and you have a printf followed by another printf and so, on.

In fact, technically printf is another function. So, when you come here, printf is a function which takes two parameters, a format and the value here. These two are the parameters that are passed to printf. In fact, what is hidden is what is happening inside printf.

So, the control actually gets transferred to printf, printf does whatever it has do to print things on the screen, it comes back to this line and the next line, it moves to the caller, next line again it is a call to printf, you go to the internals of printf, whatever it does print on the screen it does it, comes back here and then it comes here and comes to this line with a return and the last line in the main function is return 0, it returns 0.

So, if you notice main is a function, this is the function name that is the return value and this is the return statement and main is a function which has not taken any parameters. So, there are ways in which you can take inputs to main also, we will probably cover that in a later class. So, main is a function and. So, is power and. So, is printf and scanf and so, on.

(Refer Slide Time: 12:17)

Calling Power Function In a Loop

```
#include <stdio.h>
int power (int, int);
main () {
for ( int i = 0; i < 20; i ++ )
    printf("%d %d %d\n", i, power(3,i), power(-4,i));
}
int power (int base, int n) {
    int i, p = 1;
    for ( i = 1; i <= n ; i ++ )
        p = p * base;
    return p;
}
```

This program calculates and prints 3^i and -4^i for $i = 0$ to 19

So, I want to show a slightly different example, where we use the same power function inside a loop. So, we have `int power (int, int)`. So, this says that this is the prototype, which takes two integers and returns an integer and what we have done is, inside `main` we have a loop for `int i = 0; i < 20; i ++`. You are printing `i`, `3 power i` and `minus 4 power i`. So, we have this function for computing power. So, this is the slightly more complicated control flow. So, we start with here. `main`, the first line after that is a for loop. The for loop will initialize `i` equal to 0, it will check if `i` is less than 20 or not. So, when it is 0, it is actually less than 20 and it comes to this line. So, when you go to `printf`, it actually requires `i`, `3 power i` and `minus 4 power i`. Only if you know the values, you can print them. So, when you see `power of 3 comma i`, it actually makes a function call to `3 power i`. It makes a function called `3 power i`, it computes the result, you get the result back and then you have `power of minus 4 power i`, this power function computes the result, it comes back.

So, you have `i` which comes from the loop, `3 power i` would have come from the power function, `minus 4 power i` would have come from the power function, `printf` has all the values that it requires, it will print things on the screen and then the control comes to this location. At this point, it says it is the time to check, go to the end of this for loop which is `i plus plus`. So, you are supposed to do the post loop function which is `i plus plus`. Check again, whether `i` is less than 20, so, `i` would have become 1.

It is still less than 20, again it comes to this line which has `printf`. At this point, it makes two function calls, get it is arguments, prints things on the screen, comes back to the end

of this for statement, at that point you do increment of i again and so, on. So, is the slightly more complicated control flow. So, you have loops and you have various function calls here.

(Refer Slide Time: 14:37)

Calling Power Function with $i=3$

```
printf("%d %d %d\n", i, power(3,i), power(-4,i));
```

```
int power (int base, int n) {  
    int i, p = 1;  
    for ( i = 1; i <= n ; i ++)  
        p = p * base;  
    return p;  
}
```

```
int power (int base, int n) {  
    int i, p = 1;  
    for ( i = 1; i <= n ; i ++)  
        p = p * base;  
    return p;  
}
```

So, let us take this example where i equals 3 is done. So, let us say at some point, i would have been 3 in the loop, let us see how that would have done, how that would have happen. So, when i equals 3, the printf would see that there is a call to power. So, power of 3 comma i, at that point that transfer goes to this segment and since you are returning 3 power i which is in this case 3 power 3, which is 27, the result comes back and that is a temporary place holder. So, 27 comes back, you have i which is already present and you have 27. So, you have two parameters ready, printf still needs one more parameter for it to print.

(Refer Slide Time: 15:23)

Calling Power Function with $i=3$

```
printf("%d %d %d\n", i, power(3,i), power(-4,i);}
```

```
int power (int base, int n) {  
    int i, p = 1;  
    for ( i = 1; i <= n ; i ++)  
        p = p * base;  
    return p;  
}
```

```
int power (int base, int n) {  
    int i, p = 1;  
    for ( i = 1; i <= n ; i ++)  
        p = p * base;  
    return p;}
```

So, at that point, you see that it is power of minus 4 comma i, that gets evaluated. The result is minus 64 that comes back as a value to the printf. So, at this point you have 3, 27 and minus 64, printf has everything that it needs to print. So, printf function will be called, at that point you will print 3 comma 27. So, 3 space 27 space minus 64 on the screen and then you are ready to go to the next iteration in i.