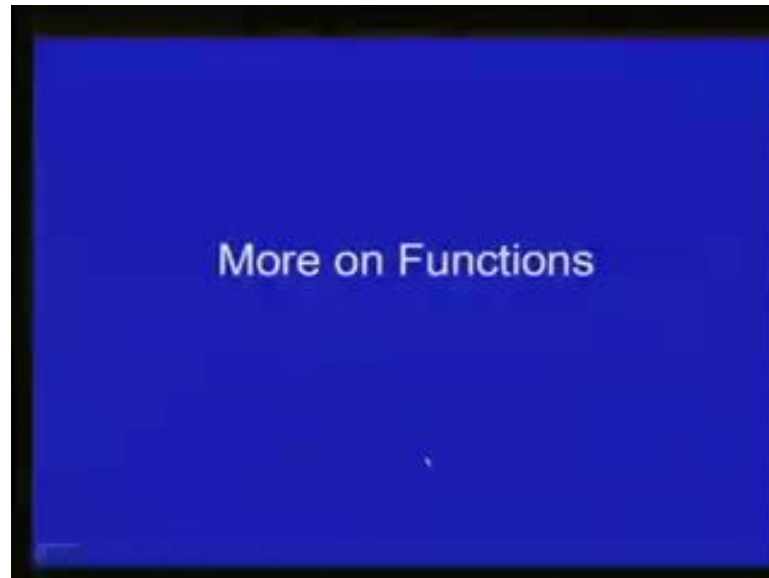


Introduction to Problem Solving and Programming
Prof. Deepak Gupta
Department of Computer Science Engineering
Indian Institute of Technology, Kanpur

Lecture – 15

(Refer Slide Time: 00:36)



In the last lecture, we have talked about functions and we saw how the use of functions can help us manage the task of solving complex problems, and developing large and complicate programs to solve these problem. Today we will discuss more about functions, but before that we will start with summary of what we had discussed last time.

(Refer Slide Time: 00:45)

Summary of Functions (1)

- Function definition:

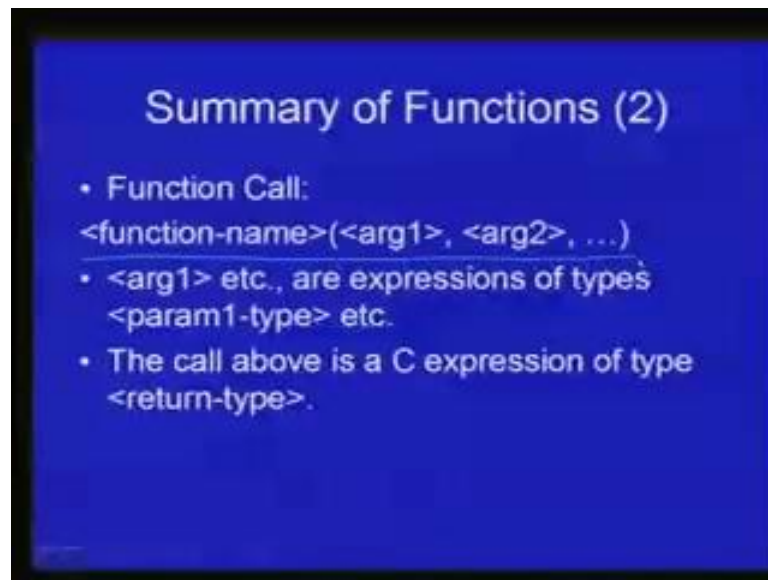
```
<return-type> function-name(  
    <param1-type> <param1-name>,  
    ...){  
    <function body>  
}
```

- The definition of a function should precede any call to the function in the program.
- Return statement is used to return a value from the function

So, here is how a function can be defined. We start with the return type of the function which could be any of the types that we have seen so far. This is followed by the function name and then within brackets we have to specify, what are the parameters for the function. For each parameter we have to specify, the name of the parameter and the type of the parameter. The parameter types can be again any of the types that we have seen so far and then the entire function body is enclosed within braces, and which follows just after the function header. Usually the definition of any function should proceed any call to a function in the program. So, this really means that if the function *f* calls the function *g* then usually it is a better to place the definition of the function *g* before the definition of the function *f*. This is not always necessary and we will see later on while we may need to make exceptions to this rule.

But at this point in time, where something we can always do. We also saw the return statement the return statement has the purpose of specifying the value of the function execution or as it is called the return value of the function. So, along with the return keyword, we also specify an expression and the value of the expression is what is returned to the calling function as the value return value from the function. And also recollect that when within a function, the return statement is executing, the execution of that function terminates immediately regardless of whether the function body has been finished or not, and control returns to the place where the function had been called from the calling function.

(Refer Slide Time: 02:33)



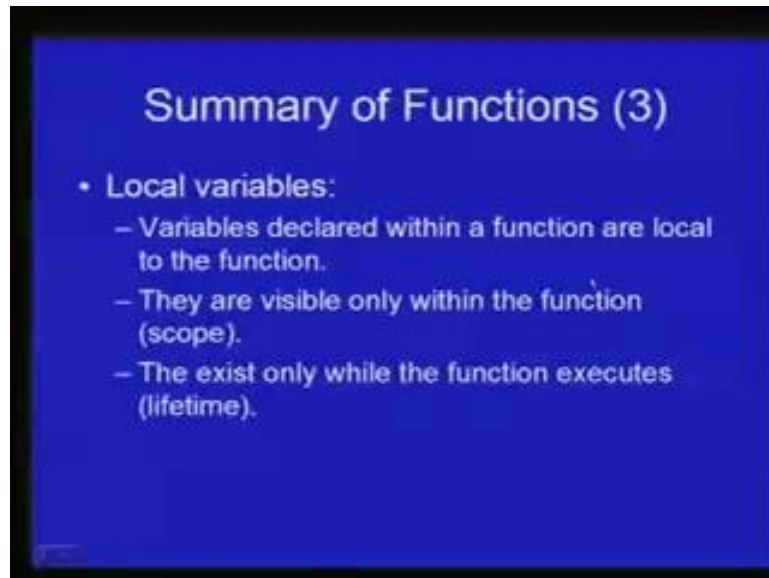
So, here is how we make a function call that is how we use a function. So, we specify a function name the function that we want to call and then for each of the parameters we have to supply a corresponding argument. The arguments must of course, be of the right types that is the argument one must be of the type that is expected for the parameter one that is declared for the parameter one. And similarly, argument two must be of the same type as parameter two and so and so forth. These arguments have to be expression, they can be arbitrary expression of the appropriate type, and this entire piece of code is treated by the C language as an expression. And this expression the type of this expression is the same as the return type of the function that we are calling.

This means that this entire expression can be used wherever the value or a variable wherever a value of type, which is same as the return type of the function is expected. For example, if the return type is an integer then this entire expression can be used wherever an integer is expected. Of course, as you know you can also convert expressions into statements by putting a semicolon after them, in which case the value of the expression is discarded the same is proof for these kinds of expression also.

If you put a semicolon after such an expression, when it becomes a statement, and essentially the return value of the function is being ignored, and the only effect that the function call has in that case is the side effects that the function can might have made.

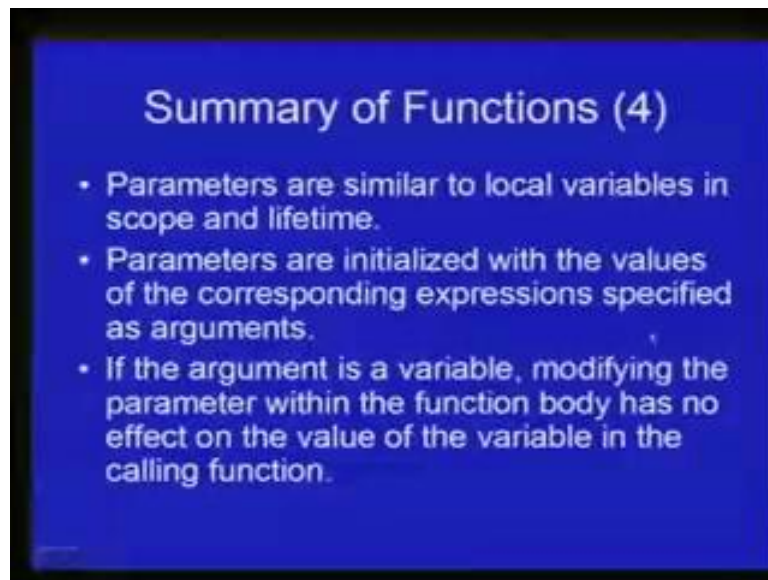
We have not really discussed the side effects in function we will look at that in today's lecture.

(Refer Slide Time: 04:26)



And then we talked about local variables in functions. A variable, which is defined within the body of the function is set to be local to that particular function. And we talked about the scope and life time in formally of variables, we will talk more about scope and life time. By the scope of the variable we may mean the portion of the program code where that variable is visible or where it can be used. So, local variables are visible only within the function in which they are defined. So, that is the scope of local variable. And by life time, we mean for how long does a variable exist, and the variable exist a local variable exist only while the function is executing. We will make these ideas more concrete with help of some examples later on in the lecture.

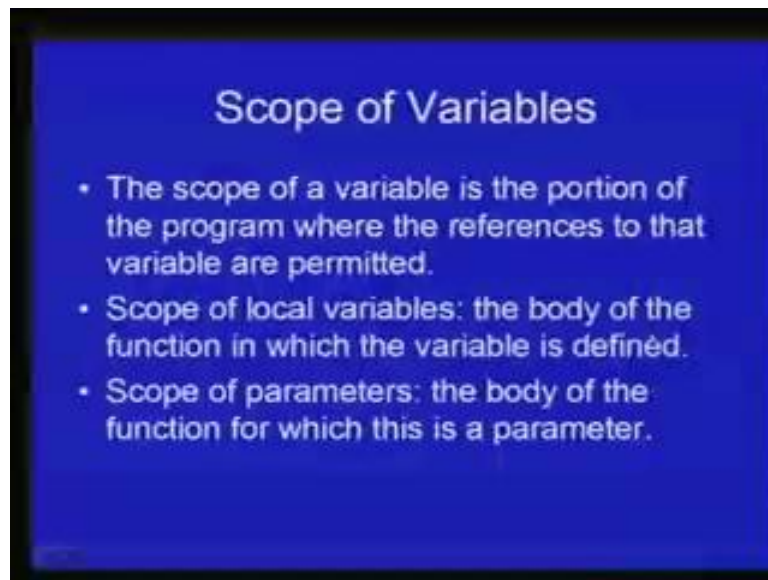
(Refer Slide Time: 05:17)



And parameters are similar to local variable in scope and life time which means that a parameter access like a local variable, it is visible only within the function body, it is created when the function is called and is destroyed when the function returns. So, what happens during the function call is that the argument for the functions are evaluated. Space is created for the parameter and local variables of the function, which is being called now as we saw in the last lecture. And then the parameters are initialized with the values of the corresponding expressions which are being passed as arguments.

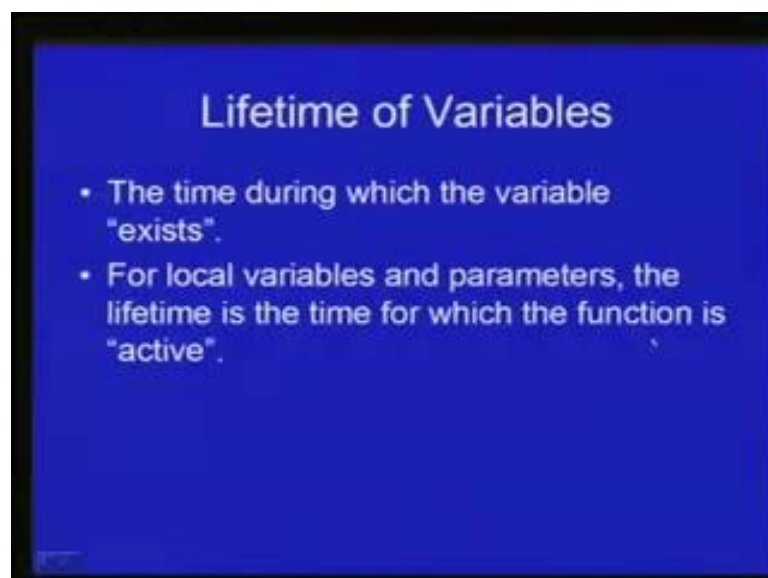
And then the function starts executing when the function return the control returns back to the place in the calling function where the function was called. Note that we saw last time that if in some case the argument whose function happens to be a variable, and in the function body, we happen to change the value of the corresponding parameter then this change has no effect on the value of original variables in the calling function when that function returns.

(Refer Slide Time: 06:26)



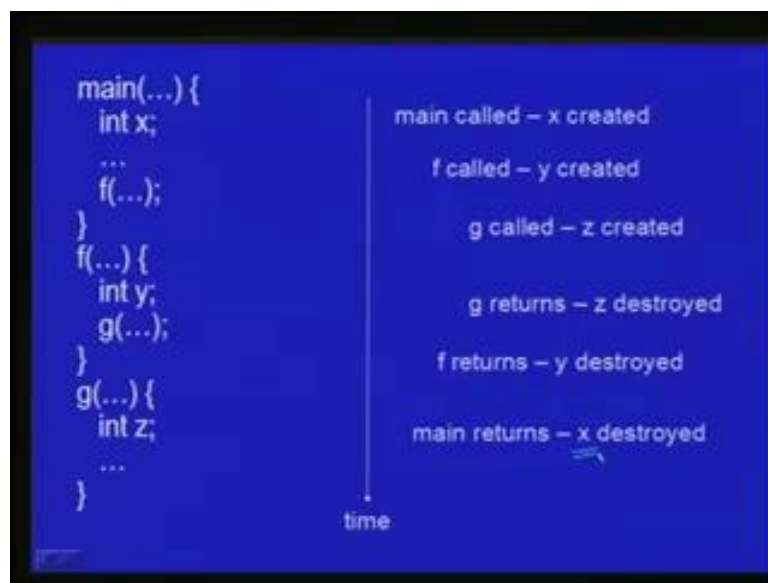
Let us now discuss the scope of variables little more formally. So, the scope of variable is defined as the portion of the program where the references to that variable are permitted. So, as you have already seen, the scope of local variables extend to the body of the function within which the variable is defined and the same is true for scope of parameter. Scope of a parameter to a function f is also within the body of the function f . So, scope is a property of the program itself that is you can examine a program and demark a portion of the code where a particular variable is in scope for not is not scope and so on. So, we say that it is a static property of the program.

(Refer Slide Time: 07:32)



The lifetime of a variable on the other hand is a dynamic property because it refers to the execution time of the program. So, by lifetime, what we really mean is the time during which a particular variable exist through the stage. So, variables are created and they get destroyed the time from the time when the variable was created to the time, when it was destroyed is called the lifetime of the particular variable. For local variables and parameters that we discussed life time is the time for which the function is active. Now, what do we really mean by a function being active, we have not really discussed this in detail in the last lecture. Let us do that now with the help of a simple example.

(Refer Slide Time: 07:57)



Consider this program. So, we have a function main with the local variable x. The function main calls function f with some arguments, which are not important for this example. Here is the definition of the function f, it has a local variable y, and it calls another function g, and within g we have declared a local variable z. Now, let us see the execution of this program on a timeline and see what is happening and when are variables getting created and destroyed. So, this is our time line.

So, when the function starts executing as you know the main function gets automatically called. So, when the main function gets called the variable x gets created. So, when at this point in time, the function main has been called and the variable x has been created. Now, the function f is called from main, so f is called and y is created. Note that when f is called the body of the function f starts executing and f is the currently executing

function, but that does not mean that the main function is not active, because the function f has been called from main and when main returns execution will go back to main. So, main is not finished yet, it is still active.

So, at this point in time the variable x still exist it has it has not been destroyed, but because of the limitations of this scope, it cannot be accessed right now, because from within f, the variable x cannot be accessed. The scope of the variable x is only the body of function main. So, within the function f, the variable x cannot be accessed, but it still exist. So, it has not yet been destroyed. So, when the function f calls g, similarly the variable z gets created, and again the function f, and function main are still active because when g returns the execution will go back to f and when f returns the execution will go back to main. So, the variable x and y they still exist though of course, again they cannot be accessed from within the function g, because the scope of this variable does not include the body of the function g.

So, when finally, g returns at that point in time, the variable z gets destroyed. So, this is because the function g is finished, it is no longer active. So, the variable z is destroyed, it does not exist. So, the life time for the variable z is this period, during which the variable z existed. And similarly now the control has gone back to the function f and finally, the function f also returns and at that point in time y also gets destroyed. So, the lifetime of the variable y is the time from which f was called to the time when f returns. When main returns and the program actually terminate and that time the variable x gets destroyed. So, this lifetime of the variable x is the entire time for which the program was executing because as long as the program is executing a function main is always active.

(Refer Slide Time: 11:17)

Implication of Lifetime of Local Variables

- Consider the program:

```
main(void) {  
    f(1);  
    f(2);  
}  
  
int f(int x) {  
    int y;  
    if (x == 1)  
        y = 10;  
    printf("%d\n", y);  
}
```

Handwritten notes on the right side of the slide:

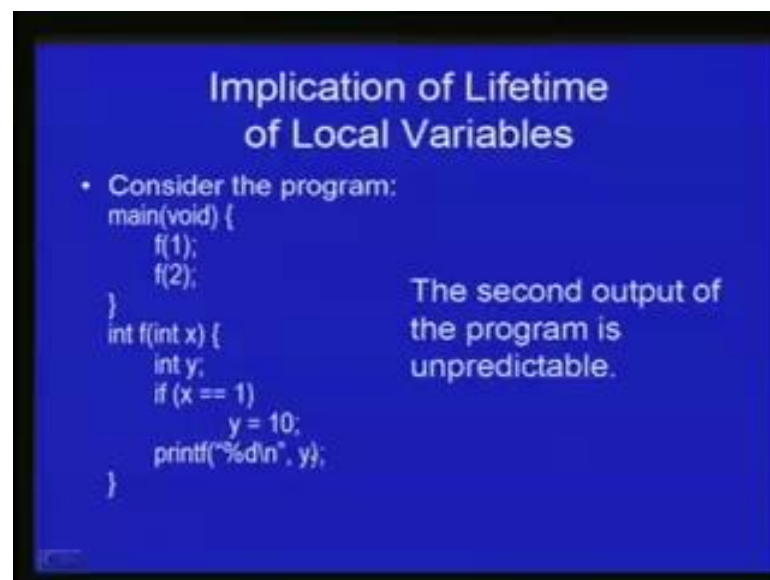
- main called
- f called
- x, y created
- y = 10
- 10 is printed
- f returns
- x, y are destroyed
- f called again
- x, y created

Now what are the implications of this kind of a lifetime. Let us try to understand that using another simple example. So, I have a program here in which the function main calls the function f twice ones with the argument one, and ones with argument two. Here is the body of the function f the parameter is called x and there is a local variable called y. So, what happens let in this function f, what happens is that if that if the parameter has the value one, then y is assigned ten otherwise it is not assigned anything and then finally, the value of y is printed. So, now, when f is called the first time, it is called with parameter with argument one, so the value x is one. So, y gets assigned ten and the value ten gets printed. The question is what happens when f is called again with argument two.

So, in the second execution of f, if the variable y which is not assigned any value in the second execution, because x would have the value two what value would it print. So, the answer really is that it is unpredictable. Let us see why its unpredictable. So, let us again draw the timeline and see how this program executes. So, when main is called and then f is called and at that point in time, x, y are created. The first call to f then y is assigned the value ten, ten is printed and then f returns x and y are destroyed. Now when f returns and control goes back to main f is called again. So, x and y are created again, but because they have been created again which means that this x and this y are not necessarily the same as this x and this y, even though they are the same variable of the same function.

On a time line this x is not the same as this x. And similarly this y is not the same as this y. So, now, the y assigned ten had assigned a value ten to this y, but this variable has been destroyed and another variable which also happens to be called y has been created. There is no guarantee that new variable y, which has been created would be created at the same place, where the old y also existed, and therefore, there is no guarantee that this time the y variable y will retain its old value of ten. So, what that implies is that the second output of this program is unpredictable. The first output will be always be ten of course, because x is given the value one and y is assigned ten and then it is immediately printed. So, the first output will always be ten, but the second output is unpredictable.

(Refer Slide Time: 14:32)



Implication of Lifetime of Local Variables

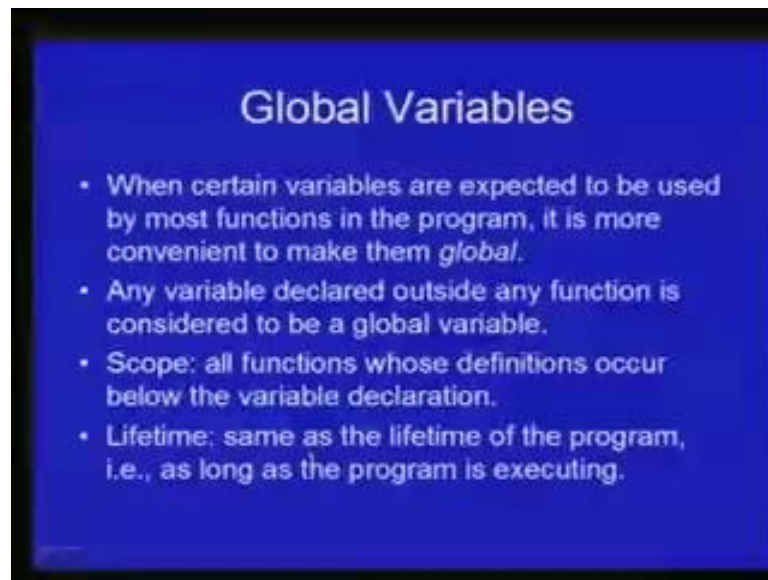
- Consider the program:

```
main(void) {  
    f(1);  
    f(2);  
}  
int f(int x) {  
    int y;  
    if (x == 1)  
        y = 10;  
    printf("%d\n", y);  
}
```

The second output of the program is unpredictable.

So, essentially the idea is that a local variable of a function does not retain its value across multiple calls to the same function, that is a point that we have to remember. We cannot assume that the value that we assign to a particular local variable of a function, the variable will have the same value when the function is called again at some (()).

(Refer Slide Time: 15:01)



Let us now talk about variables, which are known as global variables and these variables differ in scope as well as in lifetime from the kind of variables that we have seen so far. The situation where you want to use global variables is one where we have a large number of function which all need to access the same variable. Now, of course, one way of doing that is that you could pass the same variable as argument to all function. Now that of course, might make the program cumbersome because for these functions, the number of arguments may be very large, because number of the variables need to be accessed by a number of functions.

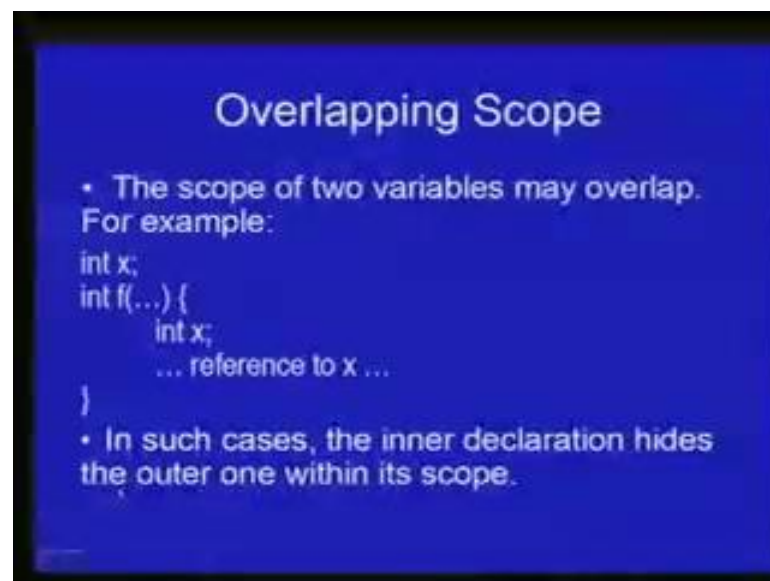
So, in those cases what you might prefer to do is to make these variables global and making a variable global implies that the variable is accessible from all functions in the program. And to make a function global what we need to do is to declare it outside any function that is the declaration of the variable is not within the body of the function of any function it is outside any function. And if we do that then the variable become automatically becomes global.

Now, in terms of scope, the scope of a global variable is all functions or all parts of the program which appear after the declaration that is in the program if we declare a global variable at a certain point then below that point in entire program that we are able to accessible everywhere. So, it differs in scope from local variable, because as you already saw the scope of a local variable extends only to the body of the function within which

the local variable is defined. And a global variable also differs from a local variable in the lifetime, the lifetime of a global variable is the same as the lifetime of a program that is it is it gets created when the program begins execution and is destroyed only when the program gets really terminated.

So, the essential idea of the global variable is that it is the variable which can be accessed anywhere in the program, because if you declare it right in the beginning of the program then it accessible everywhere below it, which means in the rest of the program. And remains in existence throughout the execution of program that is regardless of which functions are called and which functions returned and so on, this variable always access and has same value as what was assigned to it most recently. So, therefore, this variable can be used pretty much anywhere in the program when when we take up a large example of using function to solve a complicated problem. We will see examples of global variables as well.

(Refer Slide Time: 17:50)

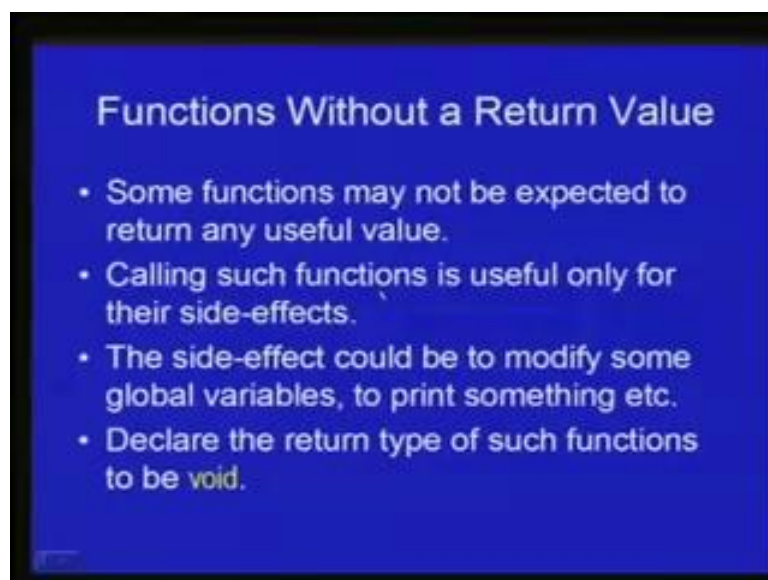


But now with the global variables, there can be a problem of overlapping scope and what that means is the following that there may be two variables with the same name, whose scope may be overlapping. So, let us consider this example. So, here we have declared a global variable x, note that this variable is declared outside the body of any function, and therefore, it is automatically a global variable. And in this function f, we have declared also a local variable whose name is also x.

Now, the scope of this function of this variable x is the entire program starting from this declaration onwards, and the scope of this variable x is the function body, which mean this particular piece of the program code. So, now if you look at this portion of the program, this x as well as this x are in the scope. So, the question is, if we use the variable x within this part of the program that is within the function body of the function f, which x are we referring to is it this x that we are referring to or is it this x that we are referring to.

So, the rules for that is that if there are two variables both of which are in scope at a given point in time, at a given point in the program then at that point in program, if a reference to that variable is made, these two variables have the same name, and if at that point in the program is very reference that variable is made, then the reference will always refer to the variable which has been declared inner most. Or in other words whose declaration is closest to the point of use. So, in this particular example, here is the reference to x this is one candidate declaration and this is the other candidate declaration. This declaration is closer to this use of x. So, therefore, this reference to x really means this x and not this x. So, essentially the inner declaration hide the outer declaration within the scope of the inner declaration.

(Refer Slide Time: 20:00)

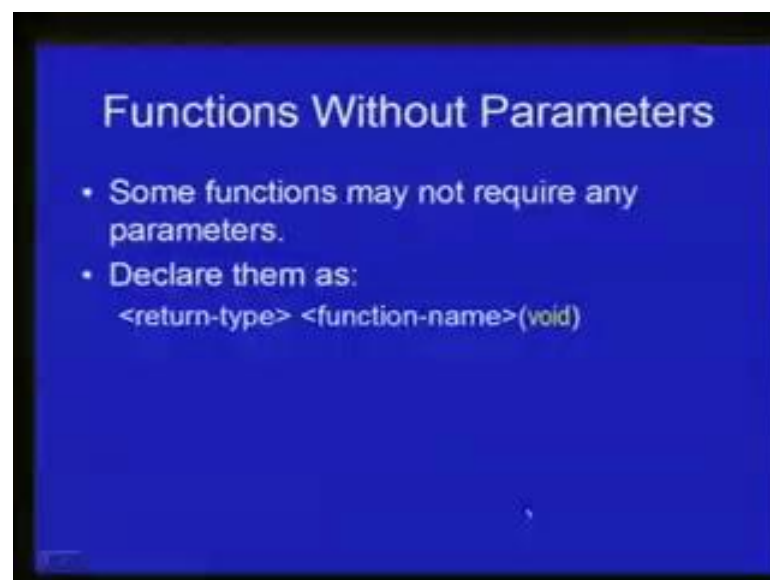


Let us now discuss functions in which we do not necessarily wish to return any value. So, in some cases, we may not really bother about returning any value from the function.

For example, you may only want to call a function for its side effects, the side effects might be to modify some global variables or it might be for example, to print something on the screen. For example, suppose you want to write a function which prints a given character, a given number of times using a simple for loop or something of that kind. Now, this way this function does not really have anything useful to return. So, this function would not have any return type and no return value would be returned from this function.

So, in such a case, what we could do is that we should declare the return type of such a function to be void, and again we will see examples later on. Also note that within such a function the return statement is not really necessary, because no return value has to be returned from the function, and the function will automatically return when its body is finished that is the execution of entire body is finished. Of course, we can still have a return statement within such a function also without a return value specified. So, the use of that would be that you want to return from the function prematurely or before the execution of its entire body is finished. We could do by putting that by executing a return statement and in the function of this kind the return statement would simply be the return key word followed by a semicolon no return value would be specified.

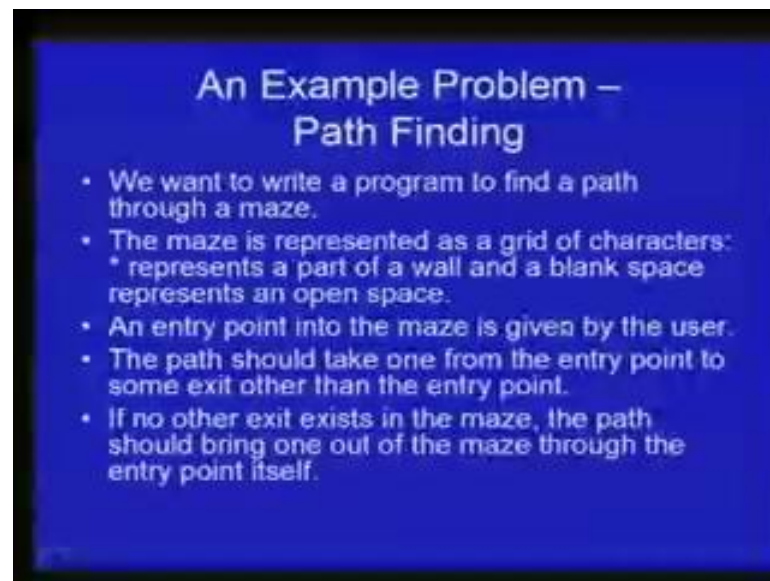
(Refer Slide Time: 21:50)



Similarly, we might want to define sometimes functions, which do not have any parameters and this might be required for example, if all the information that the function

use to access is available as a global variable. So, in that case again the function would be declared as shown here, all you have to say is that there are no parameters and so just put a void the key word void where the parameters are expected. So, let us end this lecture with a problem for you to think about, we will not solve this problem in this lecture, we will do that in the next lecture.

(Refer Slide Time: 22:41)

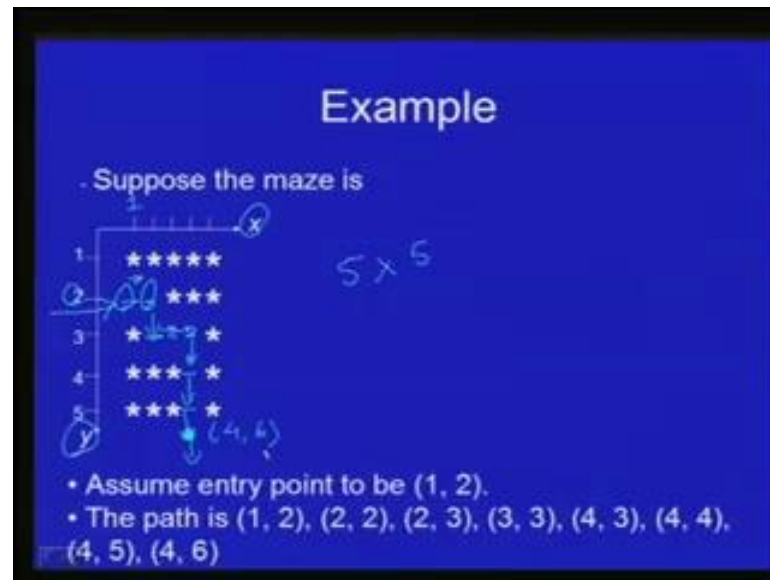


So, here is the problem, we want to write a program to find a path through a maze. So, where given a maze and you must have seen puzzles of these kind in very in popular magazines. So, you are given a maze and you have given an entry point to the maze and there is possibly another exit point in the maze, and you want this program to find a path from the entry point to the exit point. And in some cases, it is possible that there is no other exit point. So, in that case the path should bring us back to the entry point itself and we should and we should get out from the entry point itself, but if there is another exit in the maze then program should take us through that particular exit. So, the maze is given as an input by the user and the entry point is also given as an input by the user.

So, let us assume that the maze is represented as a two dimensional grid of characters, where a star represent a part of the wall, and a blank space represent an open space through which movement is possible. The maze is rectangular in shape and at any from any position in the maze one can go in any of the four directions provided that one does not run into a wall. So, the entry point into the maze is also given by the user and the

path should take one from the entry point to some exit other than the entry point. And of course, if no other exit exist in the maze the path should bring one out of the maze through the entry point itself.

(Refer Slide Time: 24:03)



So, here is an example maze for you to think about. So, this is a five cross five maze. So, there are five characters on the x axis, and five on the y axis; for ease of convenience, they have labeled the two directions as x and y axis. So, these stars represent walls in the maze, and these blank spaces represent open path in the maze. So, let us assume that the entry point is one comma two in the familiar Cartesian coordinate system. So, which will mean that the entry point is there x is one and y is two. So, and here is another exit from the maze. So, the path from the entry point should take us out of the maze through this particular exit point. So, the path will look like you we start from one two is this and we go to two two which is this.

So, we move in this way then we go to (2, 3) this way, then (3, 3), (4, 3), (4, 4), (4, 5) and then finally, we come out of the maze here which is four comma six. So, the algorithm for this problem is reasonably simple, but just think about this and you will find that even though the algorithm is simple when you actually try to implement this it becomes a quite a bit complicated and one has to use several functions and so on. So, we will take up this example in detail and develop a program for this problem in the next lecture, in the mean while please think about this problem.