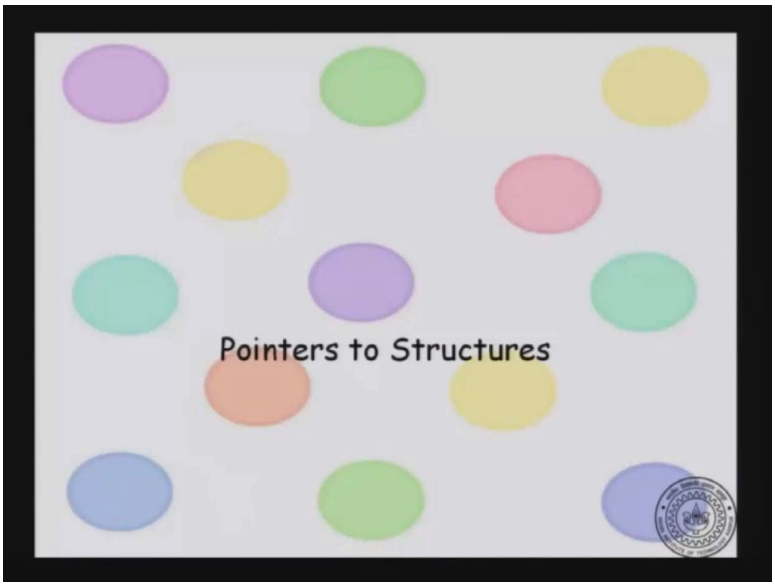
Introduction to Programming in C
Department of Computer Science and Engineering

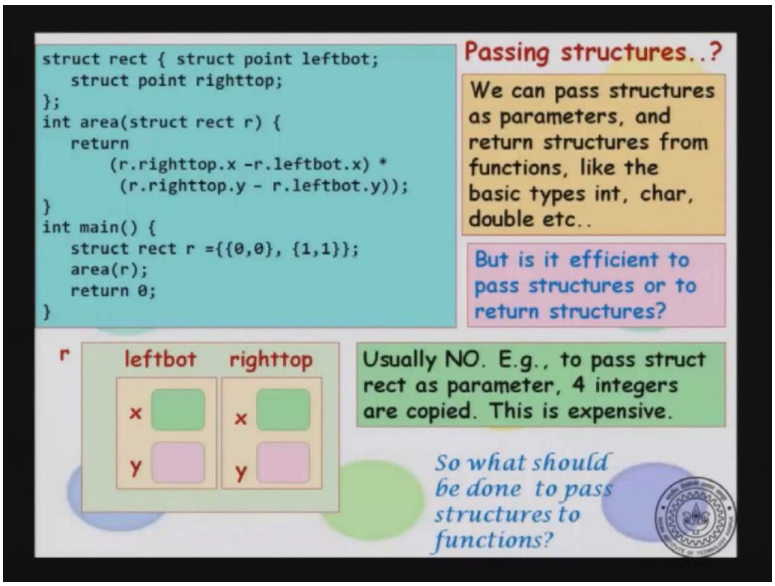
This lecture will continue our discussion of structures. So, if you remember are earlier lecture we were saying that user defined structures or user defined types which will be treated by C, in pretty much the same way as the basic data types.

(Refer Slide Time: 00:23)



So, will continue on that theme and look at topic on pointers to structures, we know that for a basic data type you can define a pointer to that type, I can declare int * or char * and so on. Similarly does it make sense to talk about struct points * for instance.

(Refer Slide Time: 00:42)

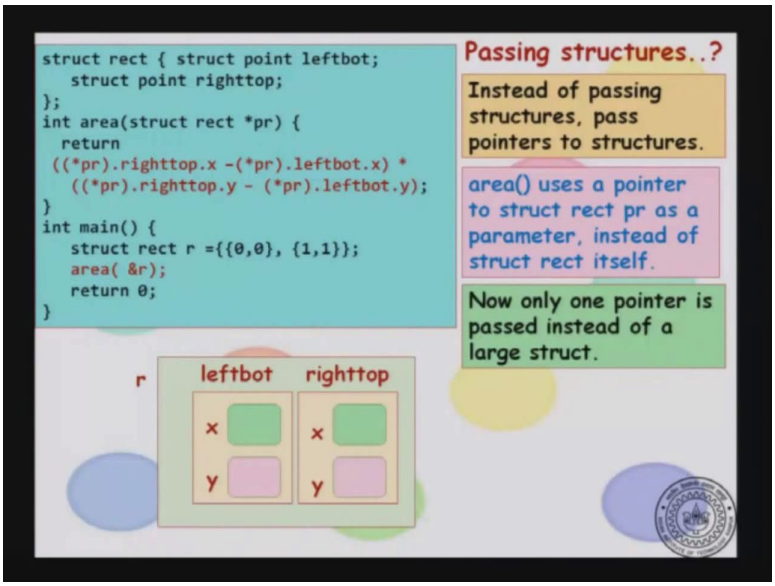


Let us look at an example were it makes sense. So, let us go back to the example of struct point and struct rectangle from the earlier lecture. So, let us say that struct rect has two points, which are leftbot and righttop and both of them are struct point. Now, we want to calculate the area of a rectangle. So, you have a rectangle r which is initialized to {0, 0}, {1, 1}. So, leftbottom will be {0, 0} and righttop will be {1, 1} and I want to compute its area.

Now, the area function is defined as follows, it is a function that returns in integer, it takes as parameter a struct rectangle and it does the following, it does (r.righttop.x – r.leftbot.x) * (r.righttop.y – r.leftbot.y). So, it does that particular function and it returns it. So, we know that we can pass structures as parameters and also return structures from functions, but is it efficient to pass structures or to return structures? And the answer is usually no, because copying a structure involves copying all it some feels.

So, generally when you caller function the value has to be copied onto the functions scope and we have seen this when discussing functions. So, when you pass a structure the entire structure has to be copied. Similarly, when you return a structure the entire structure that was created inside the function has to be copied back, this is usually and expensive operation. So, one way to get around it is to pass a pointer to the structure. So, what should be done to past structure has function in an efficient manner.

(Refer Slide Time: 02:44)



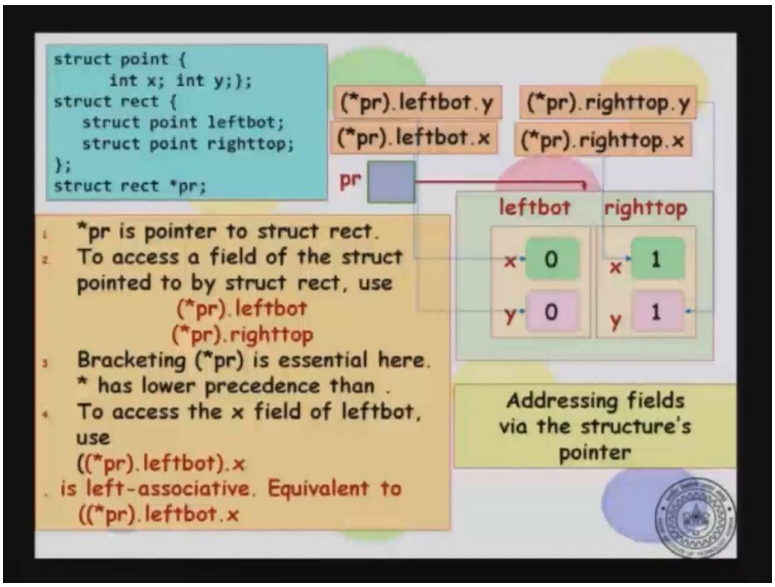
Now, one way to do it would be to define, what is known as a pointer to a structure, how do you define a pointer to a structure, you define it pretty much the same way as pointer to any other data type had this been an integer, you would declare int *pr. So, if you want to declare a variable which is a pointer to a structure, you would define struct rect *pr. So, pr is a pointer to struct rectangle.

So now, how would you pass the argument, you would say address of the rectangle r. So, you would say area and the parameter is address of r. Now, inside the function earlier you remember it was *pr.righttop.x, now pr in this case is just a pointer to a rectangle. So, we have to access the variable in that address, how do you do it using the * operator, this is the same as addressing any basic data type, you would say that *pr would be the variable in that location.

So, in this case it would be *pr, *pr would be a struct rect and that struct rect .righttop.x – (*pr).leftbot.x and so on. So, the lesson here is that instead of passing structures, you pass pointers to structures and now whatever be the size of the structure. So, you have a struct rectangle which inside has two points and so on. So, you may want to pass a very large structure and copying that will take a long time. But, instead what you do is, you pass just a pointer, now regardless of the size of the structure only one pointer is copied.

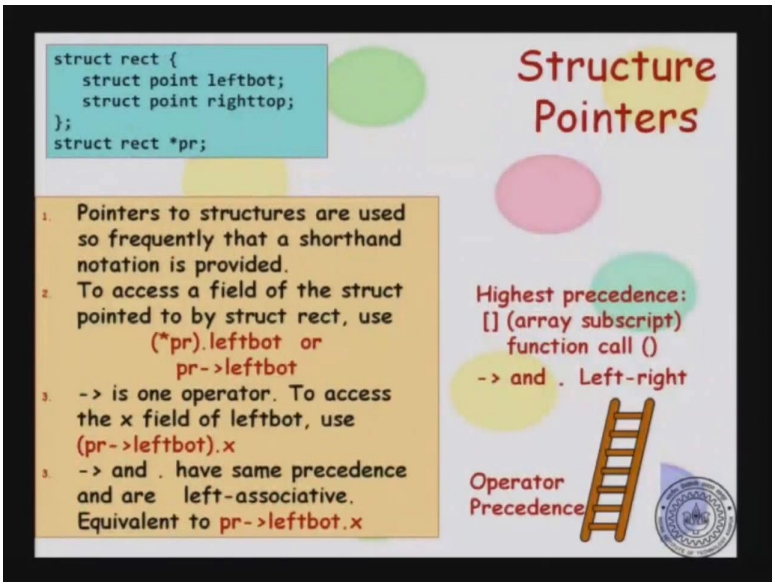
So, this same principle goes for returning structures as well, when on a structure from the function what you would do is to return a pointer to that structure. Of course, now the structure has to be allocated on the heap rather than the stack.

(Refer Slide Time: 04:55)



Now, let see how the memory depiction of this looks like. So, pr when you define struct rect *pr, pr is a pointer to a structure of type rectangle and then what is happening here is there, if you want to access the y coordinate of the left bottom, you would say ((*pr).leftbot).y. So, it will come to the leftbot field of rectangle and pick its some field y. So, ((*pr).leftbot).y would refer to this location in the memory. Similarly, ((*pr).righttop).y would be this location in the memory and so on. So, you can address the sub fields of address the fields of a structure using pointer.

(Refer Slide Time: 05:54)



There is one syntactic convenience that C provides you, because addressing structures is a fairly common occurrence. And because in this case by associativity, you cannot omit the parenthesis, you cannot say *pr without parenthesis, because it means pr.leftbot and * of that. So, that is not what you want, you want to say that take the structure in the location *pr and take it is leftbot. So, in this case by it is associativity and precedence rules, you have to include these parenthesis, you cannot omit them and this is inconvenient.

Therefore, C provides a syntactic convenience, which is pr arrow, arrow is actually two characters it is a - and a >. So, pr - > leftbot is the same as within bracket (*pr).left bot. So, there are two ways to address the fields of the location pointed to by pr. So, pr is a pointer to a struct rect you can access its leftbot by saying (*pr).leftbot or pr - > leftbot. Notice that, these two characters - and > is just a single operator and they have... So, - >, that is the - > operator and . have the same precedence and both are left associated.