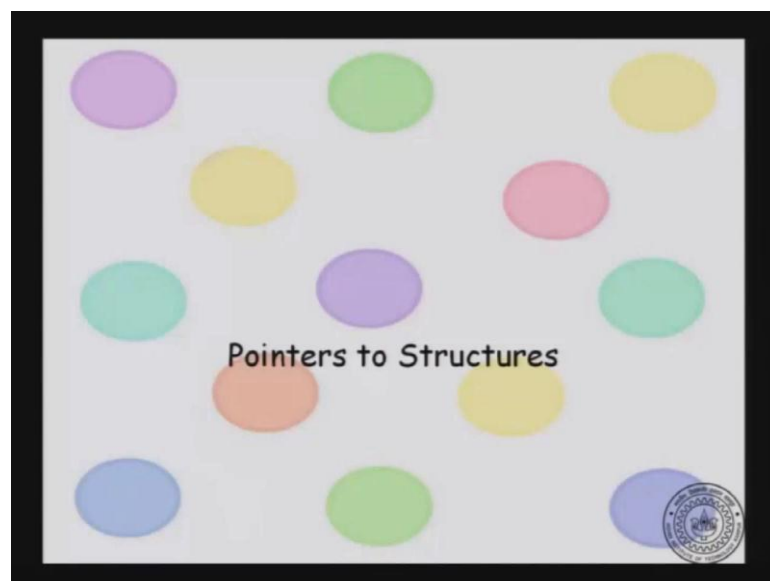


**Introduction to Programming in C**  
**Prof. Satyadev Nandakumar**  
**Department of Computer Science and Engineering**  
**Indian Institute of Technology, Kanpur**

**Lecture - 50**

This lecture will continue our discussion of structures. So, if you remember in 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 star` or `character star` and so on. Similarly does it make sense to talk about `struct points star` for instance.

(Refer Slide Time: 00:42)

**Passing structures..?**

```
struct rect { struct point leftbot;
             struct point righthtop;
};
int area(struct rect r) {
    return
        (r.righthtop.x - r.leftbot.x) *
        (r.righthtop.y - r.leftbot.y);
}
int main() {
    struct rect r = {{0,0}, {1,1}};
    area(r);
    return 0;
}
```

We can pass structures as parameters, and return structures from functions, like the basic types int, char, double etc..

But is it efficient to pass structures or to return structures?

Usually NO. E.g., to pass struct rect as parameter, 4 integers are copied. This is expensive.

*So what should be done to pass structures to functions?*

The diagram shows a variable 'r' pointing to a structure containing two 'point' structures: 'leftbot' and 'righthtop'. Each 'point' structure has 'x' and 'y' fields, represented by colored boxes (green for x, pink for y).

Let us look at an example where 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 rectangle has two points, which are left bottom and right top 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, left bottom will be 0 0 and right top 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  $r.\text{righthtop}.x - r.\text{leftbot}.x$  star  $r.\text{righthtop}.y - r.\text{leftbot}.y$ . 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 its members.

So, generally when you call a function the value has to be copied onto the function's 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 an expensive operation. So, one way to get around it is to pass a pointer to the structure. So, what should be done to pass a structure to a function in an efficient manner.

(Refer Slide Time: 02:44)

```
struct rect { struct point leftbot;
             struct point righttop;
};
int area(struct rect *pr) {
    return
    ((*pr).righttop.x - (*pr).leftbot.x) *
    ((*pr).righttop.y - (*pr).leftbot.y);
}
int main() {
    struct rect r = {{0,0}, {1,1}};
    area( &r);
    return 0;
}
```

**Passing structures..?**

Instead of passing structures, pass pointers to structures.

area() uses a pointer to struct rect pr as a parameter, instead of struct rect itself.

Now only one pointer is passed instead of a large struct.

r

	leftbot	righttop
x		
y		

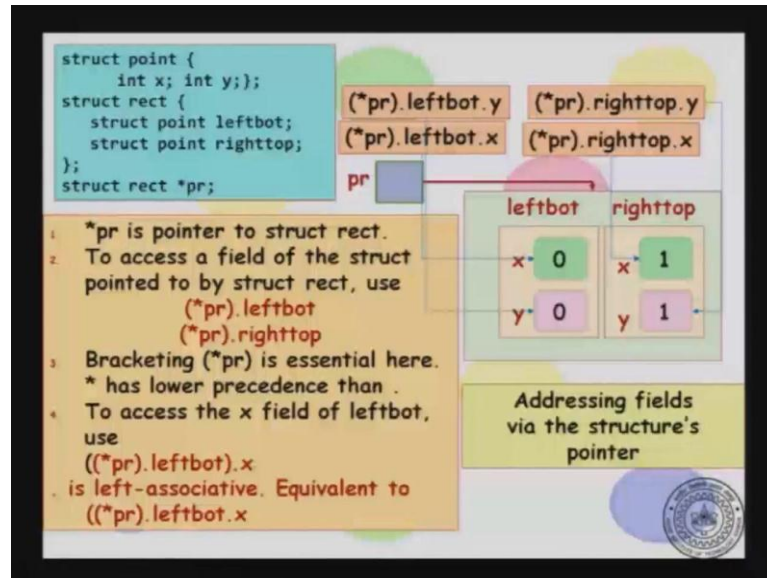
Now, one way to do it would be to define, what is known as a pointer to a structure, how do you define a pointer to a structure, you define it pretty much the same way as pointer to any other data type had this been an integer, you would declare in star pr. So, if you want to declare a variable which is a pointer to a structure, you would define struct rects star 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 dot right top dot 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 star operator, this is the same as addressing any basic data type, you would say that star pr would be the variable in that location.

So, in this case it would be star pr, star pr would be a struct rect and that struct rects dot right top dot x minus star pr dot left bot dot 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 rects star pr, pr is a pointed 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 star pr dot left bot dot y. So, it will come to the left bot field of rectangle and pick its some field y. So, star pr dot left bot dot y would refer to this location in the memory. Similarly, star pr dot right top dot 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)

**Structure Pointers**

```
struct rect {
    struct point leftbot;
    struct point righttop;
};
struct rect *pr;
```

1. Pointers to structures are used so frequently that a shorthand notation is provided.
2. To access a field of the struct pointed to by struct rect, use `(*pr).leftbot` or `pr->leftbot`
3. `->` is one operator. To access the `x` field of `leftbot`, use `(pr->leftbot).x`
3. `->` and `.` have same precedence and are left-associative. Equivalent to `pr->leftbot.x`

Highest precedence:  
[] (array subscript)  
function call ()  
`->` and `.` Left-right

Operator Precedence

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 `star pr` without parenthesis, because it means `pr dot left bottom` and `star` of that. So, that is not what you want, you want to say that take the structure in the location `star pr` and take it is `left bot`. So, in this case by its 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 dash and a left angle bracket. So, `pr arrow left bot` is the same as within bracket `star pr dot 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 `left bot` by saying `star pr dot left bot` or `pr arrow dot left bot`. Notice that, these two characters dash and left angle is just a single operator and they have... So, dash left angle, that is the arrow operator and dot have the same precedence and both are left associated.