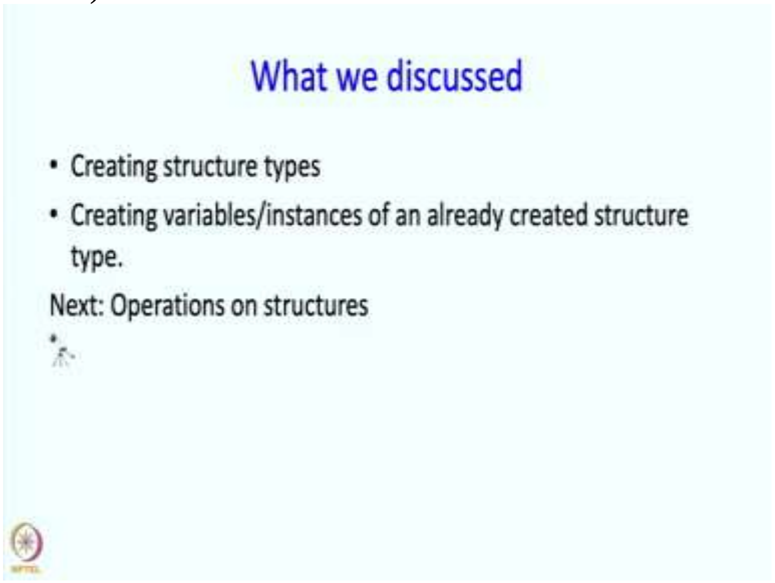
An Introduction to Programming through C++
Professor Abhiram G. Ranade
Department of Computer Science and Engineering
Indian Institute of Technology, Bombay
Lecture 19 Part 2 – Structures (Operations on Structures)

(Refer Slide Time: 0:19)



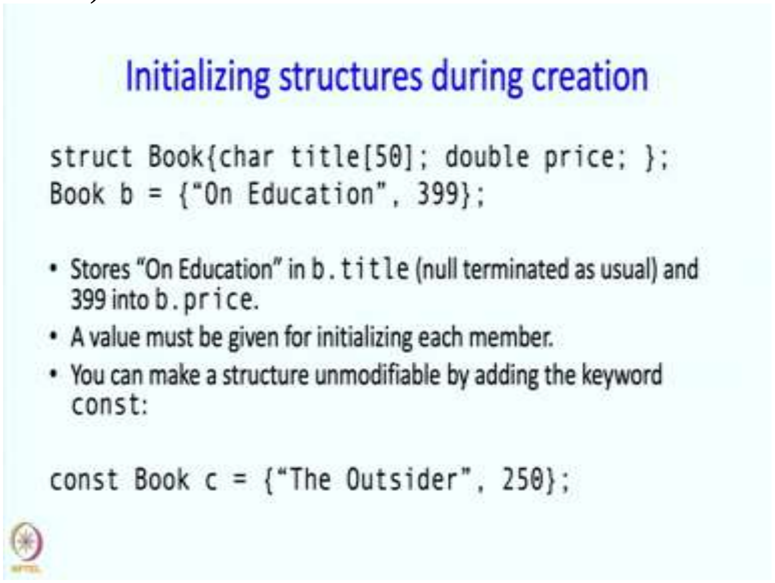
What we discussed

- Creating structure types
- Creating variables/instances of an already created structure type.

Next: Operations on structures

Welcome back, in the previous segment we discussed how to create structure types and also create variables or structures or instances of an already created structure type. In this segment we are going to see what kinds of operations we can perform on structures. So the first operation, the first thing we can do with structures is to initialize them during creation itself.

(Refer Slide Time: 0:50)



Initializing structures during creation

```
struct Book{char title[50]; double price; };
Book b = {"On Education", 399};
```

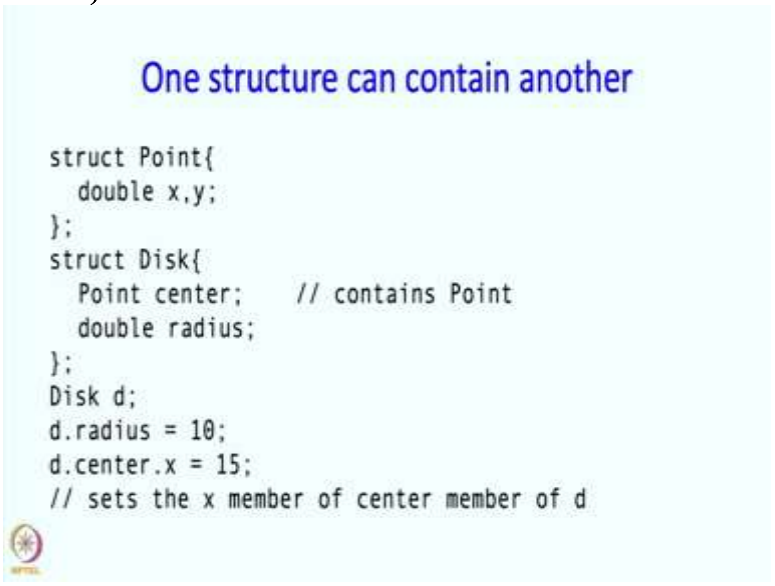
- Stores "On Education" in `b.title` (null terminated as usual) and 399 into `b.price`.
- A value must be given for initializing each member.
- You can make a structure unmodifiable by adding the keyword `const`:

```
const Book c = {"The Outsider", 250};
```

So let me just remind you this is our structure type Book, it contains numbers, title and price and I can create an instance or I can create a structure of this type and I can initialize it. So the initialization as usual happens through braces and the first element in the braces initializes the first member which is title and so b.title would be set to the string “On Education”. Then the second member in the braces initializes the second element in the braces I should perhaps say, initializes the second member of the structure type book which is price. So this initializes b.price to 399.

So again let me remind you that “On Education” is a character string and it is stored with a terminating null as usual. And you might have lots of members in a particular structure type, so you have to give as many as many values in braces and having the appropriate type. You can make structures unmodifiable by adding the keyword const. So for example you may define a structure variable c with title member being “The Outsider” and the price being 250 but here you are saying that c cannot be modified. Either of the two members of c cannot be changed as your program executes.

(Refer Slide Time: 2:37)



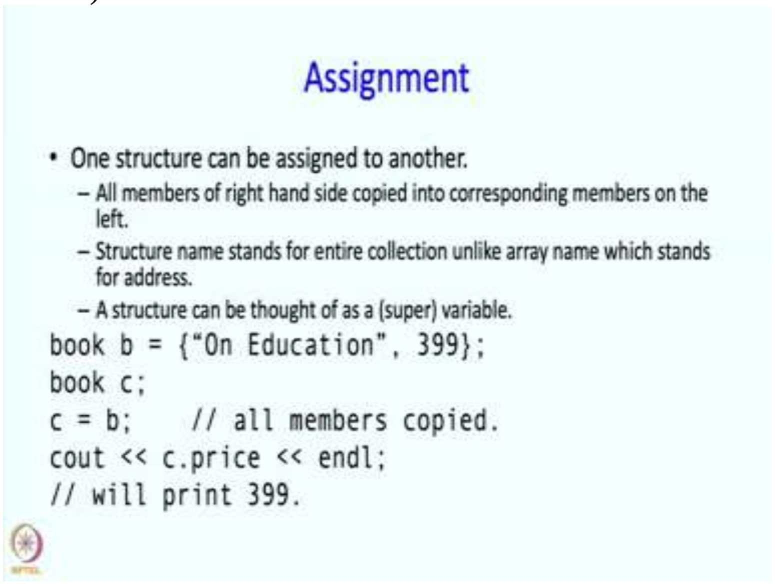
One structure can contain another

```
struct Point{
  double x,y;
};
struct Disk{
  Point center;    // contains Point
  double radius;
};
Disk d;
d.radius = 10;
d.center.x = 15;
// sets the x member of center member of d
```

Now one structure can contain another. So for example we might have a structure Point which contains say the coordinates of the point and now if I want to define the structure disk it would be natural to have the center being defined as a point. So the first member here is a point which is a type, so all that we really need is a proper type and so point center is perfectly fine. And then the second member is of type double and we are going to call it radius.

So we can create a structure d or an instance of type Disk by writing Disk d. So I can write d.radius because after all radius is a member of d and I can say d.radius to be 10. But d has a member center which in turn has a member x, so I can write d.center.x equals 15. Okay and of course I can write d.center.y as well to whatever y coordinate I think the center ought to have. I can assign one structure to another, okay.

(Refer Slide Time: 3:55)



Assignment

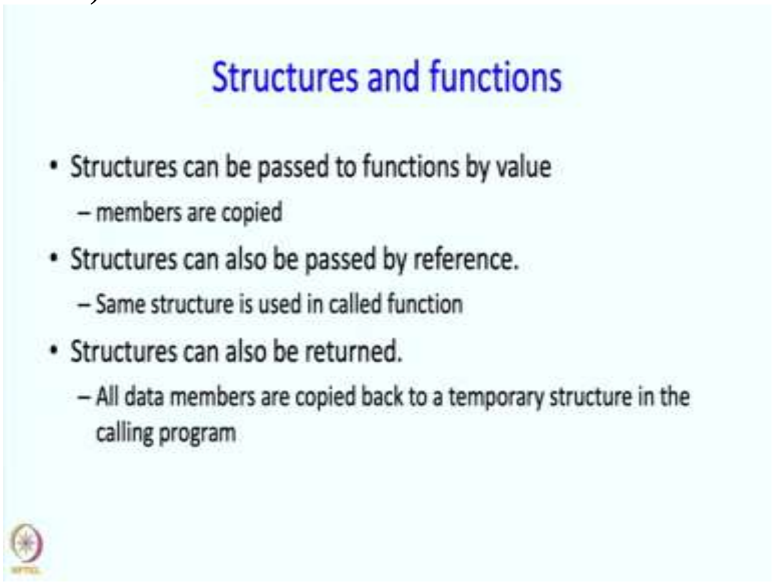
- One structure can be assigned to another.
  - All members of right hand side copied into corresponding members on the left.
  - Structure name stands for entire collection unlike array name which stands for address.
  - A structure can be thought of as a (super) variable.

```
book b = {"On Education", 399};
book c;
c = b;    // all members copied.
cout << c.price << endl;
// will print 399.
```

So basically all members of the right hand side get copied into the corresponding members of the left hand side, so the name of the structure stands for the entire collection unlike array names.

So array names stand for the address where the array has been allocated memory, structure are not like that, the structure name stands for the entire collection. And as I said earlier, a structure is a variable or you might actually think of it as a super variable because it contains you can also think of the members being the variables which are contained in this bigger variable. So as an example we have a book b with title “On Education” and price 399 and say we have a book c, so now we can write c=b. This will copy both the members and so if I print c.price the price member of c which is which has been copied over from b is 399 and so 399 will get printed.

(Refer Slide Time: 5:17)

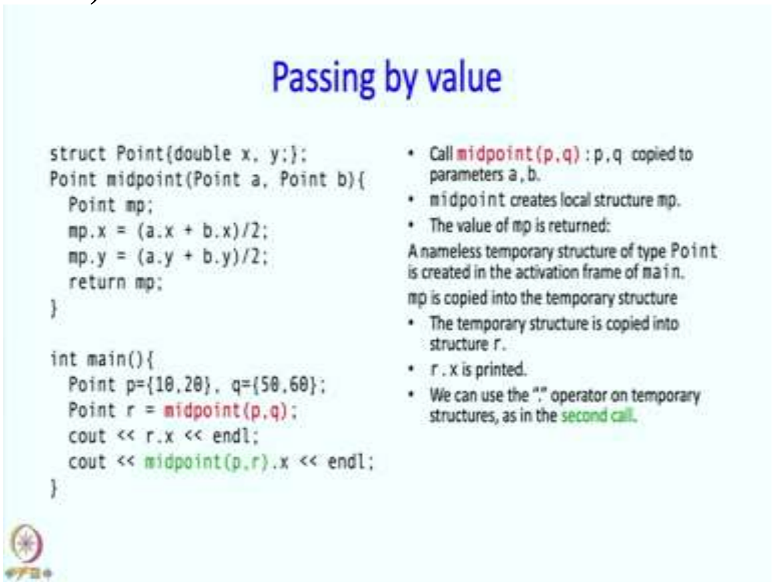


Structures and functions

- Structures can be passed to functions by value
  - members are copied
- Structures can also be passed by reference.
  - Same structure is used in called function
- Structures can also be returned.
  - All data members are copied back to a temporary structure in the calling program

Structures can be used with functions and they can be passed to functions by value. So when you pass it by value it sort of like assignment, all the members are copied to the parameter uh parameter structure and of course the usual thing is that the types must match. So, even in an assignment and even in passing the types must match. Structures can also be passed by reference and this means exactly the same thing as in case of variables. So in the calling, in the called program, the parameter name refers to the same variable which was passed from the calling program, the calling function. And you can return structures as well, so what does that mean? So all the data members of the structure that you want to return are copied back to a temporary structure in the call, in the calling program, and then you can do whatever you want with that temporary structure. Basically this temporary structure is going to be in place of the call, so call should be thought of as returning the result and that is where the temporary structure is going to be.

(Refer Slide Time: 6:41)



Passing by value

```
struct Point{double x, y;};
Point midpoint(Point a, Point b){
  Point mp;
  mp.x = (a.x + b.x)/2;
  mp.y = (a.y + b.y)/2;
  return mp;
}

int main(){
  Point p={10,20}, q={50,60};
  Point r = midpoint(p,q);
  cout << r.x << endl;
  cout << midpoint(p,r).x << endl;
}
```

- Call midpoint(p,q) : p,q copied to parameters a, b.
- midpoint creates local structure mp.
- The value of mp is returned:

A nameless temporary structure of type Point is created in the activation frame of main.

mp is copied into the temporary structure

- The temporary structure is copied into structure r.
- r.x is printed.
- We can use the "." operator on temporary structures, as in the second call.

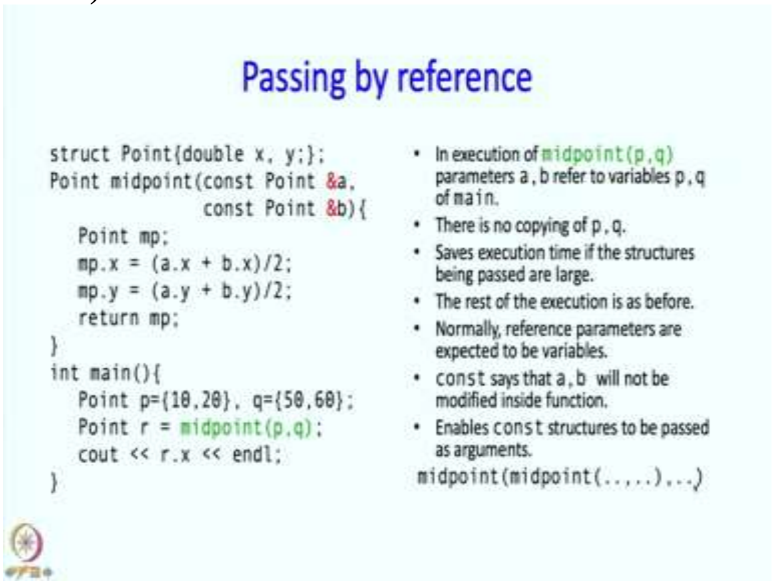
So we will see examples, so first an example of passing by value. So here is a structure Point the one we saw earlier, it has members x and y both double. Now here is a function called the midpoint, it takes as arguments a and b which are both points.

And it thus it creates a new point mp and returns that, so we will see exactly what happens and then there is a main program and the main (main) program is going to call this function midpoint. So the first call the red call okay, suppose we start executing main and come to the red call, so what happens? So the arguments p and q are copied to the parameters a and b. So this is the call and these arguments p, q are copied to these parameters a and b and now the code of midpoint starts executing.

So the first step is that a local structure mp of type Point is created and this is created in the activation frame of this function midpoint. Its x and y variables are set suitably and if they are set to the mean of the x and y variables of x and y coordinates of points a and b and then this mp is returned. So this happens as follows: so a temporary structure of type point is created and that structure sort of stands in for this call, okay? And the elements of or the members of mp are copied into that temporary structure, so you can think of this entire structure is copied into that entire structure, okay?

And subsequently what will happen? Main program will start executing again and this will be assigned to r. So mp is copied into the temporary structure and the temporary structure is copied into the structure r and then in this we can print r.x, that is perfectly acceptable and here is another call, okay, so here we are calling midpoint but we are directly taking its x coordinate. We are not putting it into any local variable and taking, taking the coordinate, taking the member, we can do this, we can just we can just put .x and this will (this will) just mean whatever this call returns take its x member, okay?

(Refer Slide Time: 9:54)



Passing by reference

```
struct Point{double x, y;};
Point midpoint(const Point &a,
                    const Point &b){
   Point mp;
   mp.x = (a.x + b.x)/2;
   mp.y = (a.y + b.y)/2;
   return mp;
}
int main(){
   Point p={10,20}, q={50,60};
   Point r = midpoint(p,q);
   cout << r.x << endl;
}
```

- In execution of `midpoint(p,q)` parameters `a`, `b` refer to variables `p`, `q` of `main`.
- There is no copying of `p`, `q`.
- Saves execution time if the structures being passed are large.
- The rest of the execution is as before.
- Normally, reference parameters are expected to be variables.
- `const` says that `a`, `b` will not be modified inside function.
- Enables `const` structures to be passed as arguments.

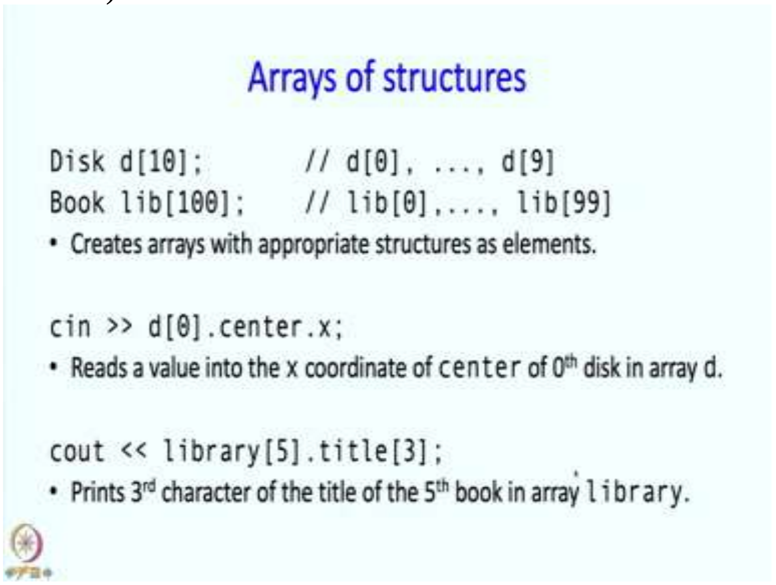
`midpoint(midpoint(..,..),..)`

We can pass by reference as well and in this case the code is really the same except that we have these two &s which indicate that is the a and b are passed by reference, okay? So in the execution of midpoint(p, q) the parameters a, b will refer to variables p and q okay, nothing will be copied over. Now there is no coping of p, q and this will save execution time if the structures being passed are large and indeed if you are passing large structures it is good idea to pass them by reference. The rest of the execution is as before. And normally if I have a reference parameter as I have that because I want to modify it and therefore it is expected that reference parameters should be variables.

So these so whatever is being called over here should be variables because only then can I modify them, modify these reference parameters in the code. However this const says that this code promises not to modify these things, these parameters.

And therefore with this const. you can pass constant points as arguments to midpoint as well. So constant structures can be passed as arguments. So in fact, I can write midpoint of the midpoint of p and q and q. So this will sort of get me a point which is a quarter of the distance away towards q between p and q, okay. So I can do that as well, I can nest these calls also.

(Refer Slide Time: 11:55)



Arrays of structures

```
Disk d[10];        // d[0], ..., d[9]
Book lib[100];     // lib[0],..., lib[99]
```

- Creates arrays with appropriate structures as elements.

```
cin >> d[0].center.x;
```

- Reads a value into the x coordinate of `center` of 0th disk in array d.

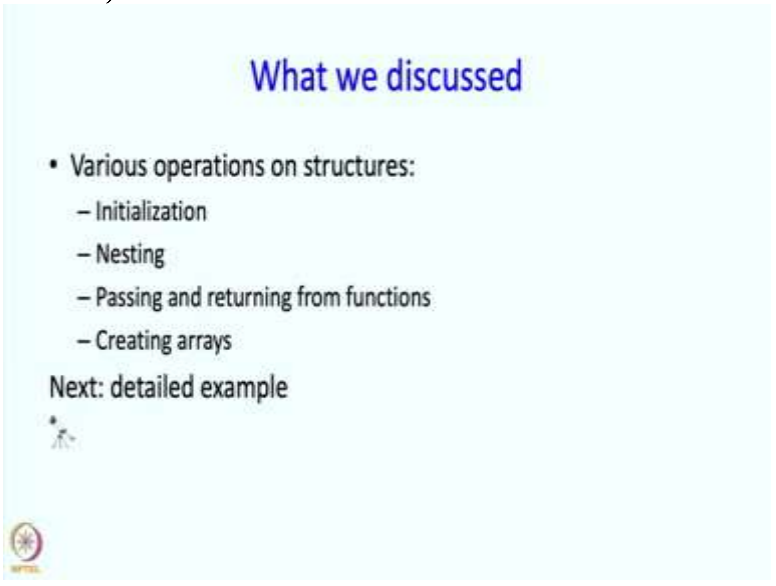
```
cout << library[5].title[3];
```

- Prints 3rd character of the title of the 5th book in array `library`.

Next I can have arrays of structures if I wish, so for example I can write Disk d and this just defines variables d[0] to d[9] each of which is a disk. Similarly I can have a variable lib maybe short for library which is an array of 100 books, so again this defines variables lib[0] through lib[99] each of which is a structure of type Book, okay?

So now these (these) variables are just like ordinary disk variables and so I can take the center x and all the usual stuff. And this also is sort of usual variable, so I can take its member but the member happens to be an array, so I can write I can index into it and so this is going to print the third character of the fifth book in the array library or I guess I should have called it lib because this is how we created it. So it is not really, read this as lib in both the cases.

(Refer Slide Time: 13:01)



What we discussed

- Various operations on structures:
  - Initialization
  - Nesting
  - Passing and returning from functions
  - Creating arrays

Next: detailed example

Alright, so what have we discussed? We have discussed number of operations on structures, initialization, nesting and by that I mean have a member b another structure and we have discussed passing and returning structures from functions, we have also discussed creating arrays for functions. In the next segment we will have a detailed example involving structures but let us take a quick break.