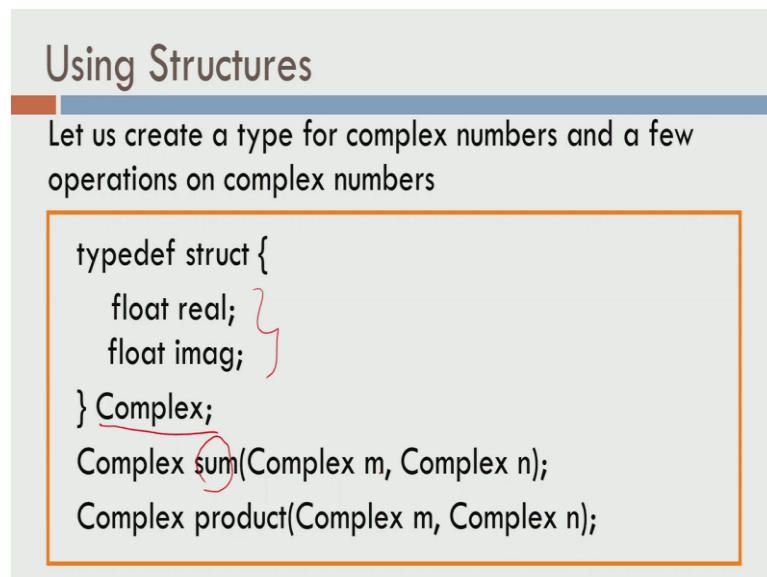


Programming, Data Structures and Algorithms
Prof. Shankar Balachandran
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Module - 11C
Lecture - 38
Using Structures and Pointers to Structures
Contents
Example: Complex numbers and Operators
Pointers to Structures
Precedence and association

Welcome to this module on Structures. In this module, we are going to look at how do we use structures and how do we use Pointers to Structures. Because this is something that comes in handy in many places, even in the course later you will be looking at the notion of linked list and so on. And for that you need not just structures, but also pointers to structures.

(Refer Slide Time: 00:35)



Using Structures

Let us create a type for complex numbers and a few operations on complex numbers

```
typedef struct {  
    float real; }  
    float imag;  
} Complex;  
Complex sum(Complex m, Complex n);  
Complex product(Complex m, Complex n);
```

So, in this lecture we will go and look at this basic data structure, basic structure called complex numbers. So, C does not give you a data type for complex numbers, there are a few languages which give you that, but C does not provide you that and one way to get complex numbers is by faking it. What we will do is, we will define a structure called

complex and the complex structure it has two fields or two members, namely real and imaginary.

So, we are going to keep this as our complex number. So, complex numbers is going to have a real number, and an imaginary number, and from now on complex is a data type that I can use. I can take this data type, pass it on to functions, pass them back from functions and so on. So, I am going to write a function call sum, which takes two complex numbers m and n and returns a complex number. And we are also going to write a function called product, which again takes two complex numbers and returns a complex number. So, this in some sense is similar to passing two basic data types and getting a value out of it. So, C does not prevent you from passing structures to or back from a function.

(Refer Slide Time: 01:50)

Using Complex Type

```
main( ){  
    Complex a,b,c,d;  
    scanf("%f %f", &a.real, &a.imag);  
    scanf("%f %f", &b.real, &b.imag);  
    c = sum(a,b); d = product(a,b);  
    printf("Sum of a and b is %f + i%f\n", c.real, c.imag);  
    printf("Product of a and b is %f + i%f\n", d.real,  
          d.imag);  
}
```

Dot Notation:
Accessing components of a structure

So, let us look at how this might be done. So, in the first line here, we do declaration for 4 complex numbers - a, b, c and d. And we are going to take the complex number c as the sum of the complex numbers a and b, and the complex number d is supposed to be the product of the complex numbers a and b. So, we first go ahead and scan the real and imaginary parts of a and b, and once that is done, I would like to do something like this. c equals sum of a comma b, it should add up the corresponding real values and put it as,

c is a real value, take the imaginary values of b, add it up and put it to the imaginary value of c and so on. And I may also want to do things like product of a comma b, print sum and product. So, this is something that I want to do, let us go and see how the sum function and product function are going to get return up.

(Refer Slide Time: 02:50)

```
Sum and Product

Complex Sum(Complex m, Complex n){
    Complex p;
    p.real = m.real + n.real; p.imag = m.imag + n.imag;
    return (p);
}

Complex Product(Complex m, Complex n){
    Complex p;
    p.real = (m.real * n.real) - (m.imag * n.imag);
    p.imag = (m.real * n.imag) + (m.imag * n.real);
    return (p);
}
```

So, the sum function is supposed to take two complex numbers m and n. So, inside we have a local variable called complex p, so p is a local variable. The p's real value gets m dot real and n dot real added up and p's imaginary value gets m dot imaginary and n dot imaginary added up, and you return that to the caller. So, if you notice p is return to the call are here, so you can see that here, p is return to the caller.

If you go back to the caller, the caller is expecting something of the type complex numbers, the left side is the complex number, so everything is good there. Similarly, if I want the product, so we know that product of two complex numbers is given as follows. So, the real part comes from the product of the real part minus product of the imaginary part, and the imaginary part is the product of the real and imaginary combinations added up together.

So, this gives you the real part and this is the imaginary part. Again we have a local

variable of the type p, and you compute what should go into p dot real and p dot imaginary and you return p. So, you are returning a complex number and when you return on this side the products complex number. So, this the complex number that you got in return, the real part will get copy to d is real part, and the imaginary part that you return will get copy to the d is imaginary part. So, this is the way to pass on structure variables to functions and get back structures from functions. So, as of now we did not know have a mechanism I did not show you a mechanism to pass on pointers to it.

(Refer Slide Time: 04:41)

Pointers to structures

```

pointType point1, *ptr;

point1 = MakePoint(3,4);
ptr = &point1;
printf("(%d,%d)", (*ptr).x, (*ptr).y);
OR
printf("(%d,%d)", ptr->x, ptr->y);

```

the brackets are necessary

equivalent short form

point1

ptr

- The operator '->' (minus sign followed by greater than symbol) is used to access members of structures when pointers are used.

So, we will do that now, let us look at this basic thing. So, what we have is you are going to have pointType pt1, so pointType point1 and star ptr. So, let us take this a little slowly, so we already know pointType is defined to be struct point. So, point1 is a variable of the data type struct point, and star ptr on the other hand is a pointer to the structure of the type point. So, or it is a pointer to pointType data type, so here this is like a basic variable and this is only a pointer.

So, when we say point 1, we actually allocate two members x and y and give it to point 1, whereas when we say point type star ptr, we do not do an allocation for a structure. We just allocate space for a pointer that can point to a structure, we do not allocate a structure yet. So, point 1 is make point of 3 comma 4, this will give you a point whose x

coordinate is 3 and y coordinate is 4. Now, you can do things that you did on basic data types.

So, `ptr` is ampersand `point1`, takes the address of the structure `point1` and assigns it to `ptr`. Remember, it is not taking the address of the members of `point1`, it is taking the address of the whole structure `point1` and it is passing that as the value to `ptr`. So, at this point `ptr` is going to point to the structure. So, let us say `point1` is of this type, so this is `point1`, it has some x and y coordinate and this whole thing is going to sit inside some memory, it is going to sit somewhere. When we say `ptr`, `ptr` is also going to get one memory location and as of now, it is not pointing anywhere. The moment you say `ptr` is ampersand of `point1`, it would start pointing to the structure here. Actually, technically it points to the first byte of the first member of the structure, it goes to that location. Now, we want to know how to dereference this, like we did for pointers in basic data types and the way to do that is, use what is called as star operator.

So, in star operators, so let us see what this is doing. So, `printf` percentage d, percentage d star of `ptr` dot x, so `ptr` is of the data type pointer to a structure. If I do star `ptr`, I get structure, and therefore if I do structure dot x, I get to a member. So, this thing what you seen in the blue circle here is correct, because you started with `ptr` which is a pointer to a structure, when you do star `ptr` it actually dereferences the pointer and gets the structure and you are looking at structure dot x. So, it looking at the member x and member y here respectively.

So, the parenthesis here is actually crucial. So, thus star and parenthesis have different presidents. Therefore, you have to put this whole thing inside, the star and `ptr` should be within the parenthesis, but this is sometimes very laborious. So, C gives you another shortcut to look at the member that is pointed to by a pointer of a structure, shall I repeat that. I want to get to a member that is pointed to by pointer of a structure. So, the pointer of a structure is `ptr` here and I want to get this member called x.

So, one way to do that is use `ptr` arrow x. So, it is not an arrow symbol, you do not see a arrow symbol on the keyboard, it is actually a dash followed by the greater than symbol or a minus sign followed by the greater than symbol. So, it is two characters, so when

you say ptr arrow x, if you see something like that what it means is, ptr is expected to be of a pointer data type. It is not just any pointer, it is a pointer to a structure and when we use arrow, you actually get to the member.

So, ptr arrow x takes you to the x member of the structure which ptr is pointing to and ptr arrow y will take you to the y member of the structure that ptr is pointing to. So, this is one way to refer to the contents pointed by ptr, you can always do star ptr dot x, but be caution that you have to use parenthesis around star ptr.

(Refer Slide Time: 09:40)

Precedence and association

Both . and -> associate left to right. They are at top of precedence hierarchy. The following forms are equivalent:

`struct rect r, *rp = &r;`

- `r.pt1.x`
- `rp -> pt1.x` ✓
- `(r.pt1).x`
- `(rp -> pt1).x`

$vp \rightarrow (pt1.x)$ X
 $vp \rightarrow _$

So, now let us look at the notion of a precedence and association. If I have a combination of various operators, what gets precedence and how do I interpret, do I have left to right association or right to left association for the combination of operators, we will look at that. So, both dot and arrow are actually at the top of precedence hierarchy. So, if you have an expression which has a dot and an arrow along with plus and minus and so on, dot and arrow gets highest precedence over everything else.

So, let us see a small example here, we have a rectangle called r and we have a pointer to a rectangle which is called rp and rp is made to point to rectangular or itself. So, remember rectangle r is supposed to have, so let us say this is the space allocate for

rectangle r. Rectangle r itself had something called pt1 and something called pt2 and pt1 in turn had x and y, pt2 in turn had x and y. So, this should be the layout for r, r is a variable, struct rect is the data type.

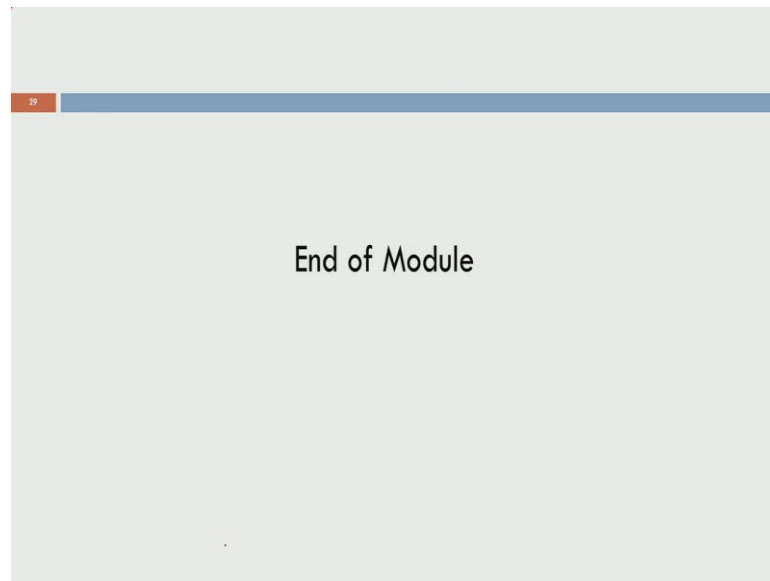
So, this is the layout for r not for rect and rp, the moment you say struct rect star rp, rp will get a memory location which is supposed to point somewhere and what is it pointing, it is pointing to ampersand r which means it is pointing to the first byte of the first member of r. So, it is actually going to point to the address where pt 1's x is present inside the rectangle r. So, if I say r dot pt1 dot x, I am referring to the pt 1's x member.

I can also say, rp arrow pt1 dot x, so in this case the way to interpret that is rp arrow pt1 is a structure and you want the member x of that structure. So, this is where the precedence and association comes in, both arrow and dot are of the same precedence and they go from left to right. So, therefore the way to read that is first do dereferencing with respect to rp, that gives us a structure and that structure dot x gives us the member.

So, what this does not do is, it is not interpreted as, I will first take pt1 dot x. So, pt1 dot x what is the data type for it? Pt 1 is a structure of the point data type and pt1 dot x is actually an integer and if I say rp arrow something, there is no rp arrow integer. Because, rp is a pointer to a rectangle and rectangle has only points inside, it does not have integers inside. So, the association went from left to right and the precedence also went from left to right. This is the correct interpretation, it is not interpreted as rp arrow pt1 dot x.

And you can also put arrows if you are in doubt, if you do not really know what it is doing and if you want to be sure, what it is supposed to do, you can always put arrows around them. So, in this case you can always put parenthesis around them, in this case you put parenthesis, so that r dot pt1 is supposed to give a structure of the type point. And when you do dot x, it gives you the x member of that point. And similarly rp arrow pt1 will give you a structure of the data type point and it is now to take the data type to do a dot x. Whereas, it should have been incorrect to do rp arrow pt1 dot x.

(Refer Slide Time: 13:39)



So, this brings us to the end of module on the structures and how to use pointers along with it. There is also lot of things that you have to be careful, when you use operators like plus and minus and so on when especially, when you have the dot operator and the precedence of and the arrow operators. So, remember that dot and arrow has higher precedence over various other things like plus and plus plus and minus and so on, just keep that in mind. And when you are in doubt, always put parenthesis around the structures that you want. So, this brings us to the end of module and thanks for watching.