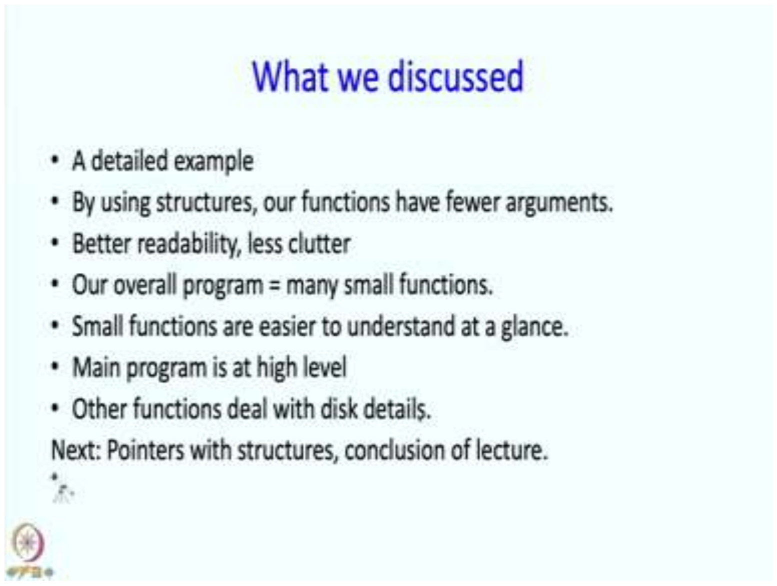
An Introduction to Programming through C++

Professor Abhiram G. Ranade
Department of computer science and Engineering,
Indian Information Technology Bombay
Lecture 19 Part 4
Structures
Pointers and lecture conclusion

Welcome back.

(Refer Slide Time: 0:29)



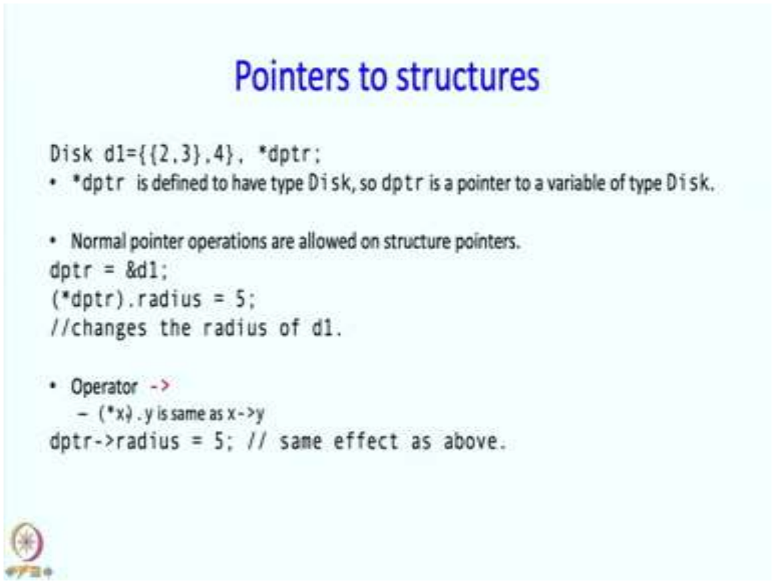
What we discussed

- A detailed example
- By using structures, our functions have fewer arguments.
- Better readability, less clutter
- Our overall program = many small functions.
- Small functions are easier to understand at a glance.
- Main program is at high level
- Other functions deal with disk details.

Next: Pointers with structures, conclusion of lecture.

In the last segment we did a somewhat detailed example of how structures could be used and how they might presumably increase the readability of the overall programs. Now, in this segment we are going to talk about how pointers can be used in conjunction with structures and we will also have a conclusion for this entire lecture sequence. So, we have seen pointers earlier and pointers to structures are created in a natural manner.

(Refer Slide Time: 0:59)



Pointers to structures

```
Disk d1={{2,3},4}, *dptr;
```

- `*dptr` is defined to have type `Disk`, so `dptr` is a pointer to a variable of type `Disk`.

- Normal pointer operations are allowed on structure pointers.

```
dptr = &d1;
(*dptr).radius = 5;
//changes the radius of d1.
```

- Operator `->`
  - `(*x).y` is same as `x->y`

```
dptr->radius = 5; // same effect as above.
```

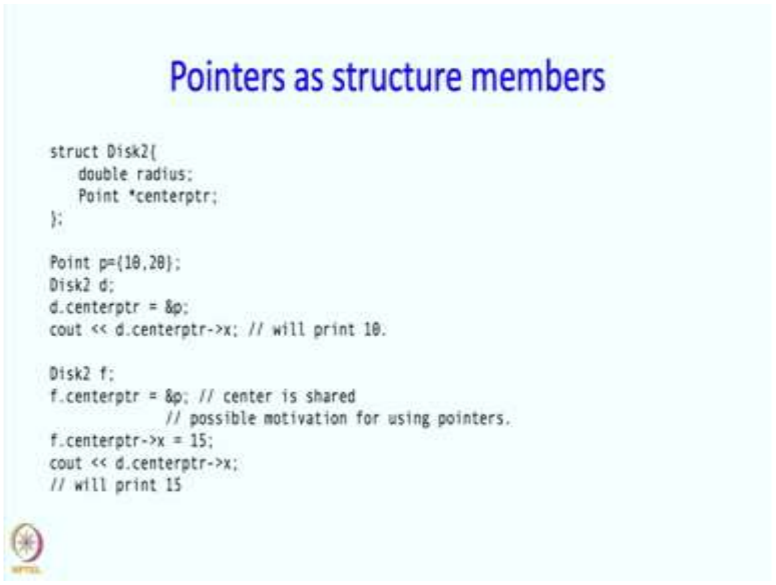
So, we have in this statement we are defining a disc. So, this is a disc, a variable of type disc and we are also defining a pointer to a structure or variable of type disc. So, this pointer is dptr. And I will just remind you what this is. So, this is a disc. So, I guess we have had two kinds of disc. Now, but this is a disc in which we have the first member was a point and here we are saying that the x coordinate is 2, the y coordinate of that center is 3 and then the radius is 4.

Now, we can do the normal pointer operations on structure pointers as well. So, for example, I can write dptr equal to the address of this disc type variable d1. So, that is what you normally do and then I can take, I can dereference this dptr as well. So, if I get dereferenced dptr, what do I get? I get d1 itself. But then I can take its radius member and I can say, "Look I want the radius to be 5 rather than the 4 that it currently is."

So, that is what this statement is going to do. Now, this idiom, *x. comes up quite frequently, and therefore, C++ gives a different way of writing it. So, here is another operator, the arrow operator, which is made up of the characters minus and greater than, you should read it as arrow. So, *x.y can be written as x->y. So, in x->y, this x should be a pointer and y should be a member.

So, I could have written this statement as this. So, I could have written dptr->radius equals 5, not dptr.radius. If I want to write dptr.radius, then dptr had better be a disc type variable. But dptr is not a disc type variable. It is a pointer double disc type variable, and therefore, I should use this arrow which dereferences first as well.

(Refer Slide Time: 3:46)



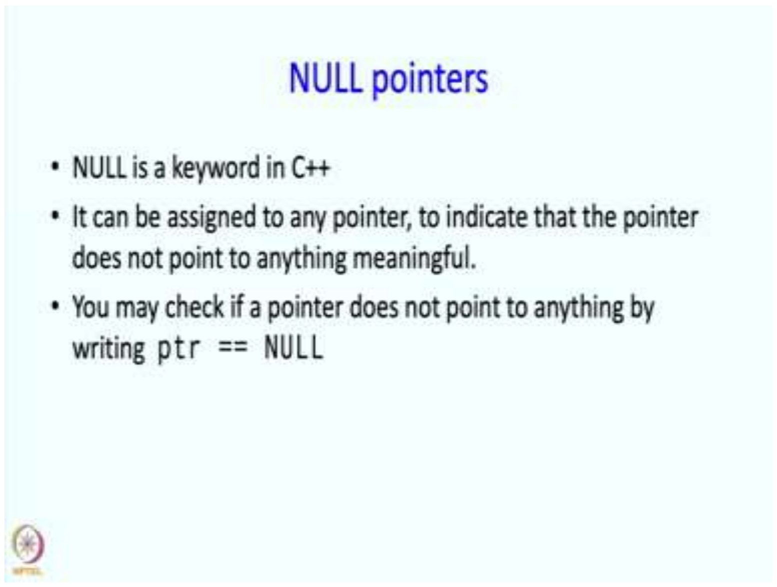
Pointers as structure members

```
struct Disk2{
   double radius;
   Point *centerptr;
};

Point p={10,20};
Disk2 d;
d.centerptr = &p;
cout << d.centerptr->x; // will print 10.

Disk2 f;
f.centerptr = &p; // center is shared
              // possible motivation for using pointers.
f.centerptr->x = 15;
cout << d.centerptr->x;
// will print 15
```

Now, pointers can be structured members as well. So, I can have struct disc2 which has double radius and which has point, but this time my member is not a point, but a point* or it is a pointer to a point. So, f.centerptr equals to address of P. So, this is a pointer and into it I am storing the address of this point. Now, I can, I want to print the x coordinate of the center of this disc d. So how do I get that?

So, I get to d.centerptr which gives me a pointer to the point which is the center and since, this is a pointer, and I want the x variable, or the x member of it I write arrow x. Then I can have disc2 f. So, F is another disc 2 and maybe I can set it center ptr also to the same point. So, essentially, these, I am making these to be concentric disc, but there is a subtle point over here. If I change the center of this then the center of this also changes. So, you should do this only if you intend that. And yes, this might be a possible motivation for using pointers, so that we can have discs that are sharing their centers. And we can do similar things. We can set f.centerptr->x equals 15, that where set the coordinate for the disc d as well. This will also print out 15.

(Refer Slide Time: 6:09)

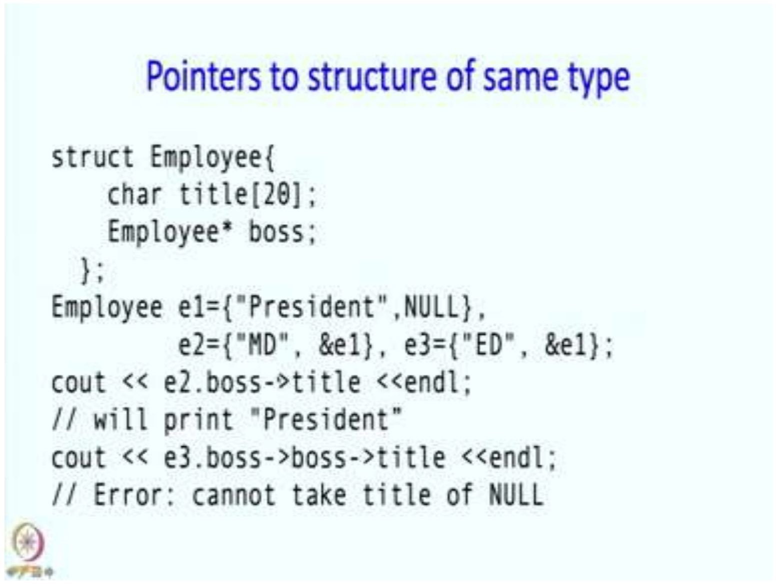


NULL pointers

- NULL is a keyword in C++
- It can be assigned to any pointer, to indicate that the pointer does not point to anything meaningful.
- You may check if a pointer does not point to anything by writing `ptr == NULL`

Now, with pointers especially in this current context, we might want to say that a particular pointer is not pointing to anything at all. For that there is a keyword ‘NULL’ which is reserved. So, it can be assigned to any pointer to indicate the pointer does not contain does not point to anything meaningful. We have remarked long ago that a pointer to discs cannot be assigned to a pointer to points, but null is kind of universal pointer. So you can assign, you can store null into any kind of pointer. Essentially, you can, say that this particular pointer does not point to anything at all and you can check whether the pointer points to thing by writing ptr==NULL.

(Refer Slide Time: 6:46)



Pointers to structure of same type

```
struct Employee{
    char title[20];
    Employee* boss;
  };
Employee e1={"President",NULL},
         e2={"MD", &e1}, e3={"ED", &e1};
cout << e2.boss->title <<endl;
// will print "President"
cout << e3.boss->boss->title <<endl;
// Error: cannot take title of NULL
```

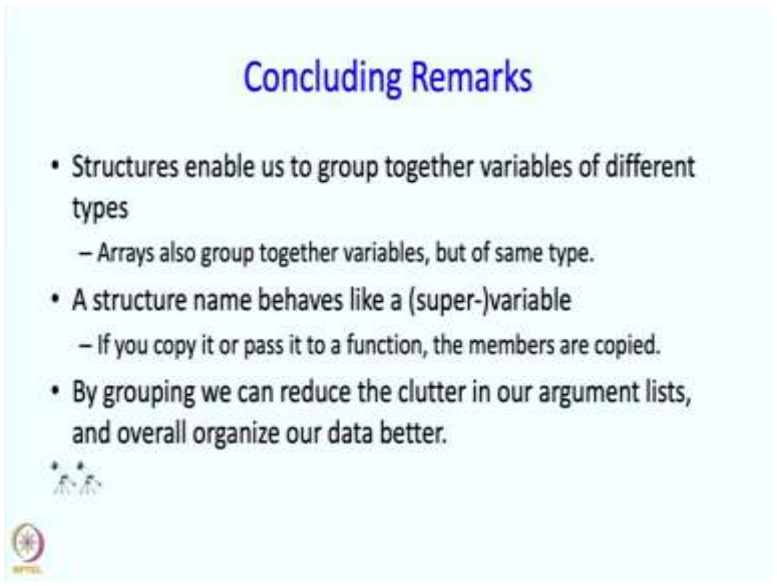
Here is an example where we have pointers to the structure of the same type. So we have a struct employee but inside it is an employee pointer. So, boss is a pointer to another employee. This is allowed. This looks like it is referring to itself. In some sense it is, but what the purpose of writing this at least in this limited context is we need to know how much how much space to reserve for this point ptr or this member boss.

But all ptrs essentially have the same amount of space and therefore, this is okay. This, of course, says that subsequently if you use boss it is expected to point variable of type employee. Here is how you might create employees. So, this is says e1 is president, so the title is President and there is no boss. So, presumably we might be writing this for somebody who was the head of the institution. So that person does not have a boss. So its boss is pointing to the null.

There might be an MD in that institution, a Managing Director and the managing director's boss is the president. So, here we can supply the address of the president and so address of e1 is supplied over here. Similarly, e3, maybe the executive director, has as boss the president, so here we are supplying the address of that president. Now, we can do this. So, we can say e2.boss, so we get to this address of e1 over here and if you want to print out the title of the person.

So, to get to the title, the title is a member of e1, but we have an address of e1, and therefore, we should put an arrow here. So, this will print president. Now, if we write e3.boss we will get to e1. We will get a pointer to e1. If we take its boss member, an arrow to boss, then we will get null. So, if we then print it out, then take the title, then we are making a mistake because null is not pointing to anything, so we cannot take a title of it, and therefore, this is going to cause an error. So at execution time, the program will say that there is something wrong.

(Refer Slide Time: 9:47)



Concluding Remarks

- Structures enable us to group together variables of different types
  - Arrays also group together variables, but of same type.
- A structure name behaves like a (super-)variable
  - If you copy it or pass it to a function, the members are copied.
- By grouping we can reduce the clutter in our argument lists, and overall organize our data better.

So what did we discuss in this lecture? So we talked about structures and we said that structures enable us to group together variables of different types. Now, I should note that arrays also group together variables, but in arrays the variables are of the same type, whereas in structures they can be of different types. They may be of the same type but they can be of as many different types as you want.

Structure name behaves like a variable. So, the structured name does not denote the address of that structure, but it denotes the content of the structure. So, if you pass it or if you copy it the content gets copied. Again this is something different from the way array names behave. And then, why we got onto this, we did all this because by grouping things together, we can reduce the clutter in our argument lists and we can organize our data better.

And because our clutter is reduced we are more inclined possibly to write smaller functions and these functions can be written as different level. So, as we saw in the example, the main function was written at a very high level. So, whereas the called functions, the other function sort of dealt with the internals of the disc, whereas the main function just said, “Oh, here are do these things with a disc without really worrying about what is going on inside.”

So, let me conclude this lecture then by saying that, structures are a way by which you can improve the readability of your program, because they help you to group all the attributes associated with an entity into a single structure or a single super variable. And as always I will suggest that, please, look at look at the problems at the end of this chapter. Of course, the next chapter, the next lecture will also continue with chapter 17. So, maybe you can wait until the next lecture to look at the problems at the end of chapter 17. Thank you.