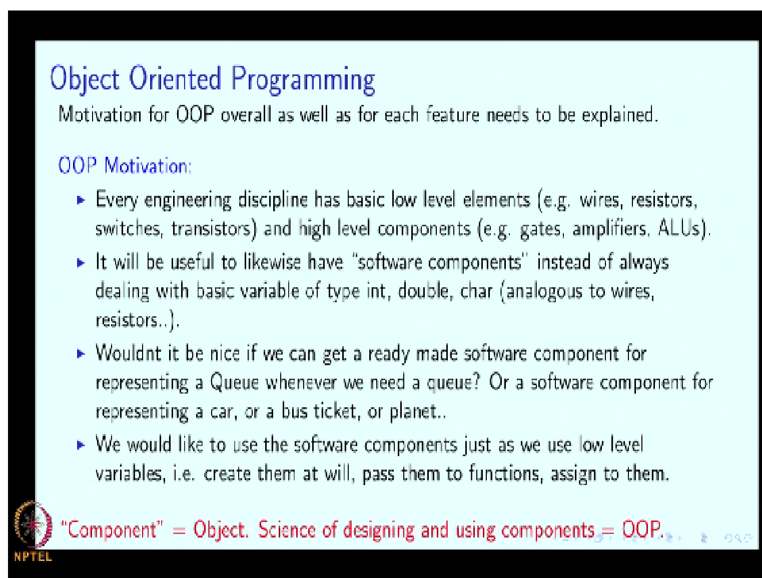
Design and Pedagogy of The Introductory Programming Course
Prof. Abhiram G. Ranade
Department of Compute Science and Engineering
Indian Institute of Technology - Bombay

Lecture – 19
Advanced Programming Topics. 2: Object Oriented Programming, Concluding Remarks

Welcome to the last lecture in our sequence on Advanced Programming topics. This sublecture is on Object Oriented Programming and in this we will also conclude the, the entire topic of advanced programming.

(Refer Slide Time: 00:37)



So Object Oriented Programming is one of the major ideas in programming but it is not readily understandable or at least the entire topic is not readily understandable and so motivation must be explained. Now the motivation must be explained for and then introduction should be given for Object Oriented Programming overall as well as we should give introduction for each feature. So let us talk about Object Oriented Programming motivation.

What can be the motivation for Object Oriented Programming? Well here is one way to go about it. We could say something like every engineering discipline has the notion of elements or components. So there might be basic low level components or basic low level elements. So for example, in electrical engineering, we might have wires, resistors, switches, transistors but when we work, when we actually do work, we are not always talking about these low level objects.

We may also, we will typically also talk about high level components. For example, we might talk about gates, we may talk about amplifiers, we may talk about arithmetic logic units and these will be big components which might internally contain low level elements but we may not want to worry about what they contain internally, okay. So we might, we might want to work with these high level objects rather than every time worry about what really is going on at the lowest level.

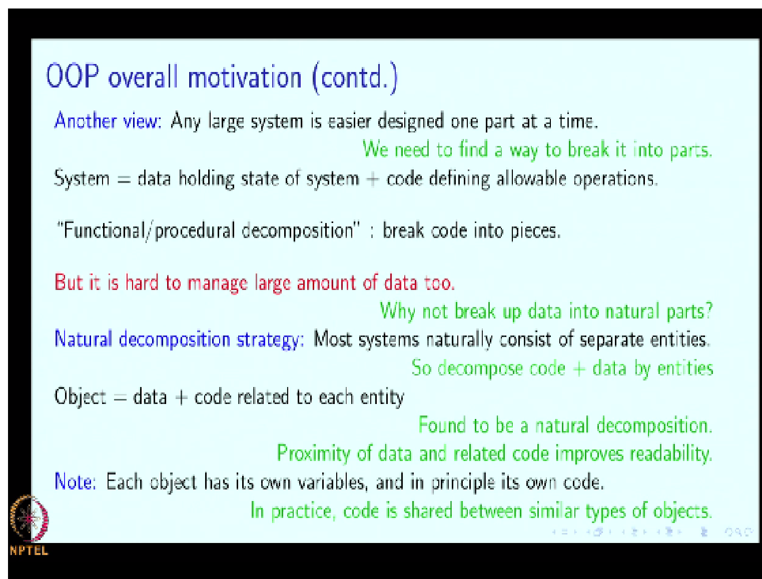
Now likewise, when we write programs, it seems like natural idea, reasonable idea that instead of always worrying about basic variables say of type, integer, double character which are sort of analogous to wires, maybe we should, we would like to program with some bigger objects, right. So would not it be nice if instead of just getting integers and doubles when as ask for them, why do not we, why cannot we get readymade software components for representing a queue.

So we can, we should be just able to say, oh, give me a queue, just as we say give me an integer and I am going to call it x y z, we could, why should not we be able to say give me a queue and I want to call it p q r, okay or why just queues, why not component for representing a car or may be a something like a bus ticket or maybe something abstract like an agreement or maybe something which represents an entire planet, okay.

Now we would like to use the software components just as we use low level variables. So we should want, we would like to create them at will whenever we want them, we should be able to pass them to functions, we should be able to make assignments, okay. So essentially, I mean, if we are putting up a wish list, these things like, these things seem like the natural things to ask for.

Now a component we are talking about is called an object in Object Oriented Programming and the science of designing and using components is called Object Oriented Programming, okay. So this is sort of a broad motivation that you can give for Object Oriented Programming. Here is a different way of saying the same thing perhaps, okay.

(Refer Slide Time: 04:20)



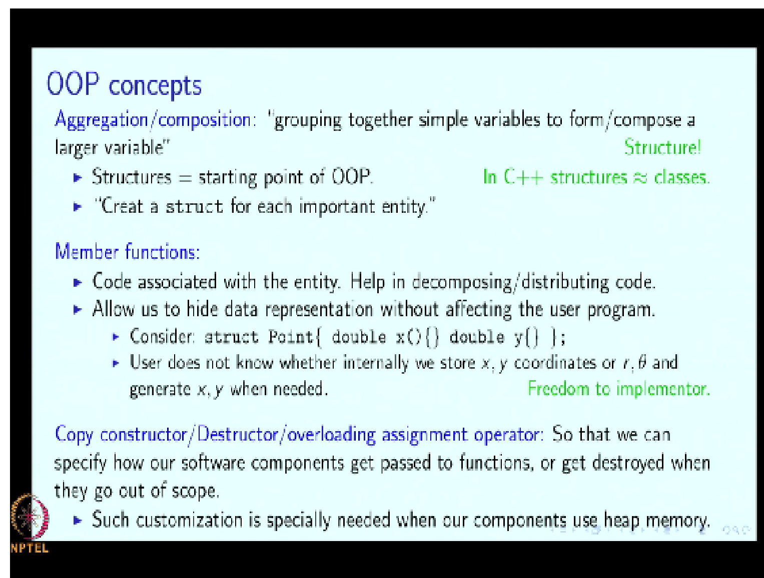
So you could say that any large system is easier designed one part at a time but for that we need to find a way to break it into parts. So what is a system? A system consists of data holding the state of the system as well as code defining allowable operations. So when we talk about functional or procedure, procedural decomposition or we talk about top down design, we typically mean that we are going to break the code into smaller pieces or smaller functions.

But managing data, large amounts of data is also difficult, okay. So why not break up data into natural parts, into small parts in some natural manner. So the natural decomposition strategy is that most systems naturally consist of separate entities. So we are going to decompose the code and the data by the entities, okay. So basically we are going to create at the level of a program something called an object which will consist of the data associated with an entity as well as the code related to that entity, okay.

So this is a natural way of decomposing things and that is what is Object Oriented Programming. Now we are going to put the data and the relating code together, okay. So that is also another, another structural principle and because the data and the code are together, presumably it is going to improve the readability of the whole thing. So essentially, we are going to say that each object contains all the data or all the variables required to store the data for that object and in principle, it contains code as well.

But of course, in practice, if you have several objects of the same type, then the code for them is the same and without really you worrying about them, the code will somehow be shared. So there will be a single copy of, of the code but you can, if you want, you can think of it as each object has its own copy. It does not really matter. Now individual OOP concepts should also be discussed separately and motivated. So far example, one basic object oriented concept is aggregation or composition.

(Refer Slide Time: 06:52)



And what does this say? Well this says that we should group together simple variables to form or compose a larger variable as might be associated with an individual entity and this is nothing but a structure. So in some, some sense, structures are starting point of Object Oriented Programming, okay. In C++, we have seen that structures are classes and that is the reason. So classes are sort of the key Object Oriented Programming language construct and in fact, they generalize structures.

So the C++ designers say that, oh, we are not going to have much distinction between structures and classes and there is the principle that you must use or you must ask your students to use really, that you must create a struct for each important entity. Member functions, simply embody this idea that the code associated with that entity should be kept along with that entity and so we have the notion of member functions.

So this helps in decomposing or distributing the code as well. Now member functions also do something interesting which is, they allow us to hide data representation without affecting the user program. So for example, here is a structure definition. So we are defining a point. Now inside it, we have 2 member functions. x is a member function and y is a member function. Why do we have member functions rather than data members?

Well, if we do this, the user does not know whether internally we store the x y coordinates or we store the r theta coordinates and if we are storing r theta coordinates, we will have to generate x y when the user needs that but the user is calling the function anyway. So the function code will generate r theta. Now this is useful because it gives freedom to the implementer. The implementer may decide later on that no, no, no, I really want to store r theta for some reason.

Where the implementer is free to change the implementation without the user of the structure having to change his or her code. So the degree of couplings so to say, gets reduced because of this. Now the next notion is that of a copy constructor/destructor and assignment operator and in Object Oriented Programming, we are allowed to specify the behaviours of these operators as well, okay.

So basically we are defining these software components and we are saying that look we want these software components to be used in this manner. So we are passing an object by value to a function, when the copy constructor is invoked and that way we get to say what exactly happens when we pass an object to a function. Similarly, when an object goes out of scope, what exactly happens.

We get to tell the compiler that by writing a destructor, okay. Basically, this kind of customization is to be needed when our components use heap memory. So we talked about earlier that the heap memory has to be carefully managed. We need to delete things or maybe we make copies and our copy constructor or destructor and even the assignment operator will have to be overloaded to make these copies. So overloading other operators is also possible because it can provide compact and natural syntax.

(Refer Slide Time: 10:33)

OOP concepts contd.

Overloading other operators: Can provide compact and natural syntax.
Natural to overload +,* operators for class representing physics vectors.

```
struct pvect{
  ...
  pvect operator+(pvect &a){..}  pvect operator*(double t){..}
};
pvect u, a, s;   ...
s = u*t + a*(t*t/2);
```

Encapsulation: Hiding information from the user of the function.

- By creating public members we prevent the class/structure from being used in a potentially erroneous manner.
  We package electrical devices so that users dont get shocks...
- Developer can produce new implementation of class without user having to change her program.

So for example, it might be natural to overload + and * operators say for a class which represents vectors from physics. So here is an example. So we might have a physics vector class and inside that we have defined the operator+ on vectors as well as the operator* which is an operator between a vector and a double. So we might define vectors u, a, s and we might write the familiar formula s=ut+1/2at square.

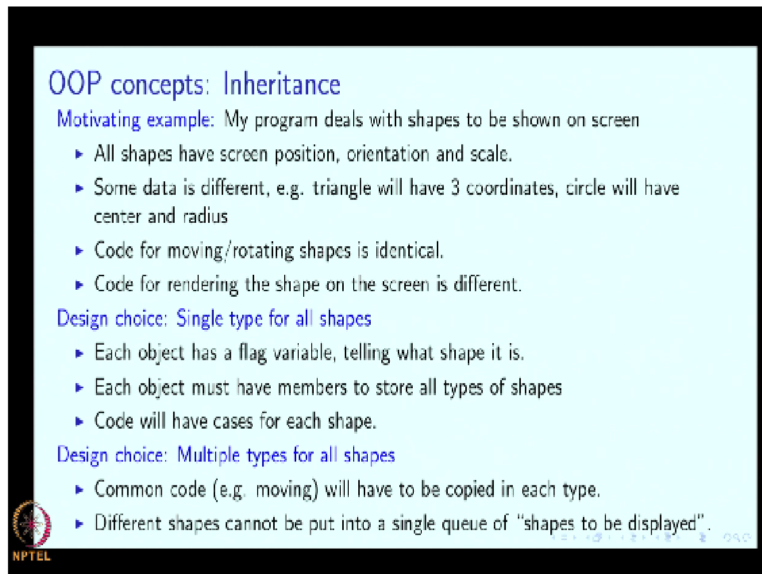
So what is happening over here is that t squared upon/2 is calculated, that is a double and that can be multiplied with the acceleration which is a physics vector, u can be multiplied with t which is also a physics vector and the 2 physics vector can be added but notice that this assignment operation looks like the physics formula. So, so it is, it is easier for us to write things like this if we have overloaded the operators in a suitable manner.

Now another principle from object orientation is that of hiding information from the user of the function. So by creating public members, we allow users to use some objects and prevent the rest of it which are private and that way the user is forced to only use the public members and not other members and the user is forced to use the, use our software components or use our object in the correct manner.

So this is sort of like why we package electrical devices, I mean, we do not want the users to get shock. We want the users to use that device only in some specified proper manner and therefore,

we package them. So this public/private business is sort of like that. And again, the developer can provide a new implementation in what goes on inside that packaging can change so long as the same kind of controls are given (()) (12:55). So, so long as the public members do not change, signatures of the public member functions do not change, the user, he does not know but the implementer can change the implementation.

(Refer Slide Time: 13:15)



The last concept in Object Oriented Programming that should be talked about is inheritance and here is the motivating example. So suppose my program deals with shapes to be shown on the screen just like our simple CTP library and our simple CTP library in fact uses inheritance. So all shapes have screen position, orientation and scale but each shape might have some different data. So for example, a triangle will have 3 coordinates.

A circle will have a center and radius, okay. Code for moving or rotating shapes is identical in some ways. The code for drawing itself or rendering the shape on the screen is different. So as you can see different shapes might have some data which is the same and some code which is the same but they will also have some code which is different as well as some data which is different.

Now the question arises, should we declare all shapes to be the same type or same class or should we declare different shapes to be of different types? So if we choose a single type for all

shapes, what happens? Well, then we will have to have a flag variable for each object telling what shape it really is. So if the flag says, it is a circle, then we will access the center and radius members of that object.

If the flag says it is a triangle, then we will have to access the coordinates, the 3 pairs of coordinates which might be stored inside that object. So each object must have members to store, store all types of shapes, okay. So that object will have to store the 3 pairs of coordinates as well as center and radius and as well as other attributes that other shapes might have. And the code which you write, say for rendering, will have to have cases.

So if you will again check is the flag saying that this is a triangle, then I will execute the triangle drawing code, okay. On the other hand, you could have multiple types for all shapes. So you have a type for a triangle, you have a type for a circle but there is some disadvantage here as well. Namely the common code will have to be copied in each type, okay. Because after all the types are different.

So you will have to have copies of the code. You will have to have copies of the declarations of the common data as well, okay. And then here is an interesting difficulty. I would like to maybe have a queue of shapes which contains all the shapes that I want to display. So if the shapes are of different types, then I cannot have a single queue. I will have to have a separate queue for each type. So our management just becomes very cluttered. This is where inheritance helps.

(Refer Slide Time: 16:11)

Inheritance helps...

- Define a shape type/class
- Define circle, triangle as subtypes/subclasses.
- Code/data common to all types of shapes will be defined in shape class.
- Subtypes will automatically *inherit* common code and data.
- You can define additional code/data for each subtype.
- You can have queues which contain any shape...

"Polymorphism"

Many examples and detailed discussion in Ch 25, Ch 26 [Ran14].

So in inheritance, I can define a shape type or class, okay and then I can define circles and triangles as subtypes or subclasses, okay. Code or data common to all types or shapes will be defined in the shape class. Subtypes will automatically inherit the common code and data and you can define additional code or data for each subtypes. So the rendering code will be defined separately inside each subtype, okay.

And this is the interesting thing. Our objects will behave sometimes like it is a triangle but it might sometimes behave like it is a shape and so you can have a queue in which you can say look I am going to put shapes rather than saying I am going to put circles and inside that, you will be able to put circle, objects as well as triangle objects, okay. So this is called polymorphism and this will be implementable under inheritance, okay. So many examples detailed discussions of this are given in the book and certainly this is the topic but at the Tier 2 level.

(Refer Slide Time: 17:23)

Create opportunities for using Inheritance

Simplecpp provides Composite class for creating composite objects Ch 26, [Ran14]

```
class Wheel : public Composite{
  Circle *rim;
  Line *spoke[10];
public:
  Wheel(double x, double y, Composite* owner=NULL) :
  Composite(x,y,owner){
    rim = new Circle(0,0,RADIUS,this);
    for(int i=0; i<10; i++){
      spoke[i] = new Line(0, 0, RADIUS*cos(i*PI/5),
      RADIUS*sin(i*PI/5), this);
    }
  }
};
```

Many students will want to create new shapes, and will need to use inheritance.

Now once you create inheritance, you should also give users opportunities for using inheritance not just tell them that the simple CTP library has been done in this manner. So simple CTP itself contains a class called composite and the composite class is used for creating composite objects. So here is how you can use it. You can create a class called Wheel by inheriting from this composite class and inside the composite class, you can use simple CTP objects like circles and lines or other composite objects that you might have created.

And this, this composite class contains code which will put together all these objects and it will allow you to instantiate Wheel classes wherever you are. So I am not going to discuss the details of it but this is how you should do things that if you discuss a concept, student should have opportunities for using that concept, okay. So in this case, many students will want to create new shapes because they will want to create complicated objects. So they will naturally feel that they must learn inheritance and how to use inheritance.

(Refer Slide Time: 18:40)

Concluding remarks:

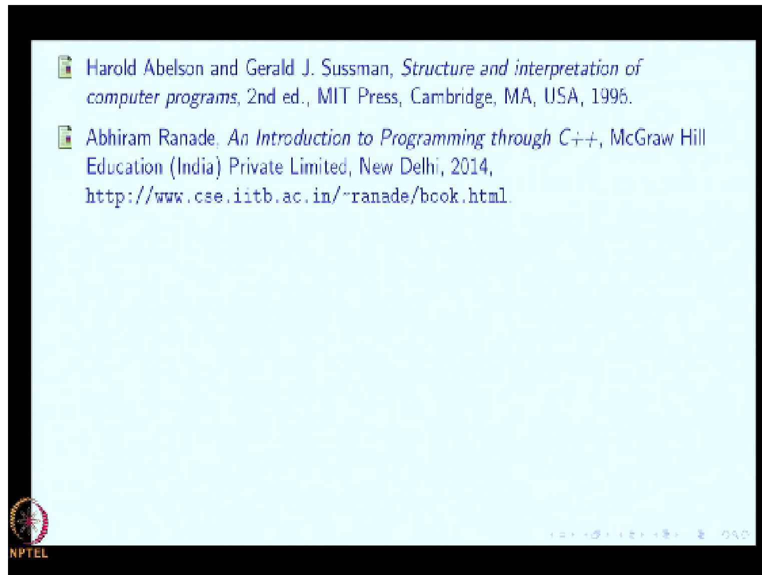
- If you have time in your course, you should select some of the advanced programming topics discussed here.
- Advanced programming topics will be useful for CS majors as well as non-majors.
- Algorithm design related topics could also be discussed if you have time.

Chapters 16, 19, 23, 24, 27, 29 of [Ran14]

Alright, so this concludes what I wanted to say in this sequence of lectures about advanced programming topics and let me just make a few concluding remarks. So if you have time in your course, you should select some of the advanced programming topics which we discussed, okay. Advanced programming topics will be useful for CS majors as well as for non-majors, okay. These days even non-CS majors will do things like graphics.

They might do things like even using parallel computing. So advanced programming topics and the notion of says managing heaps will be important to them. And of course, standard libraries are important for everyone. So certainly you should talk about those, okay. Another possibility is to discuss algorithm design topics, okay. So they have been discussed in so many different chapters. So those are also a possibility.

(Refer Slide Time: 19:46)



Here are the references that I used for the book, for the 2 books and with that I will stop. Thank you.