

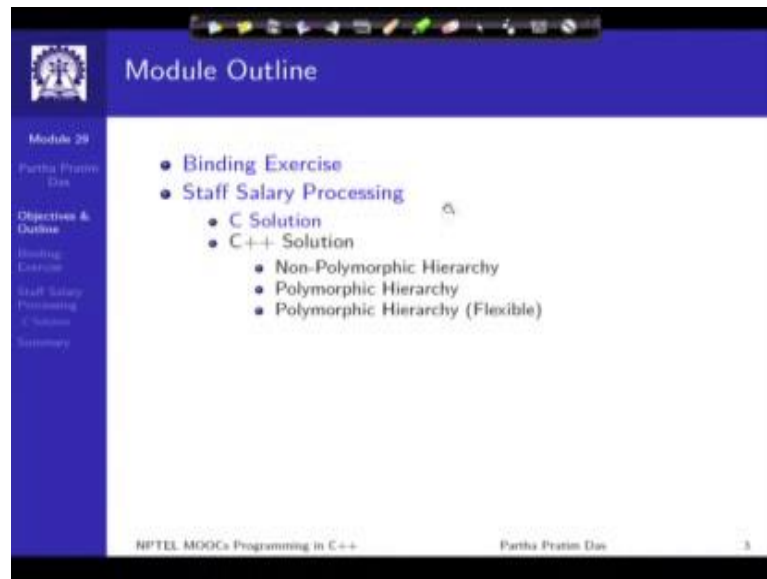
**Programming in C++**  
**Prof. Partha Pratim Das**  
**Department Computer Science and Engineering**  
**Indian Institute of Technology, Kharagpur**

**Lecture - 44**  
**Dynamic Binding (Polymorphism): Part IV**

Welcome to module 29 of Programming in C++. Since the last three modules, we have been discussing about polymorphism, the dynamic binding feature of C++. We have seen how on a hierarchy of classes, we can have virtual functions and how virtual functions dynamically bind to the object instance instead of binding to the pointer or reference to which a particular virtual function is being invoked. We have seen the differences between the semantics of non virtual functions and virtual functions; you have seen the behaviour under overloading.

We have also noted that virtual functions could be pure and in which case they do not need to have a body and in those cases the corresponding class containing such pure virtual functions are abstract base classes. So, we have more or less covered the whole of the theory of polymorphism that is involved in C++. In this module and the next we will try to use the knowledge that we have acquired in terms of the polymorphic feature and engage that in terms of solving problems and doing sample design. So, in view of this, in the current module and also the next the target would be to understand design with class hierarchy.

(Refer slide Time: 01:54)



The image shows a presentation slide titled "Module Outline" with a blue header and a white main area. On the left, there is a vertical navigation menu with a blue background and white text. The menu items are: "Module 29", "Partha Pratim Das", "Objectives & Outline", "Binding Exercise", "Staff Salary Processing", "C Solution", and "Summary". The "Staff Salary Processing" item is highlighted. The main content area contains a bulleted list: "• Binding Exercise", "• Staff Salary Processing", "• C Solution", "• C++ Solution", "• Non-Polymorphic Hierarchy", "• Polymorphic Hierarchy", and "• Polymorphic Hierarchy (Flexible)". At the bottom of the slide, there is a footer with the text "NPTEL MOOCs Programming in C++", "Partha Pratim Das", and a small icon.

Module Outline

- Binding Exercise
- Staff Salary Processing
  - C Solution
  - C++ Solution
    - Non-Polymorphic Hierarchy
    - Polymorphic Hierarchy
    - Polymorphic Hierarchy (Flexible)

NPTEL MOOCs Programming in C++ Partha Pratim Das

Specifically in this module, we will first take up and exercise with variety of binding combinations, when different kind of static and dynamic binding get mixed along with overloading how does it work. So, we will take one more little bit involved example to clear out the ideas and then will start off with an example of designing a staff salary application, which will run into the next module also. In the current module, you will show in how such an application can be designed in C and we will show the refinements C++ finally, using the polymorphic hierarchy in the next module.

(Refer slide Time: 02:44)

The slide displays C++ code for three classes: A, B, and C. Class A is the base class with virtual functions f, g, and h. Class B inherits from A, overriding f and h. Class C inherits from B, overriding g and h. Below the code is a table showing the initialization of pointers and the resulting function calls for four invocations of pA.

```
// Class Definitions
class A { public:
    virtual void f(int) { }
    virtual void g(double) { }
    int h(A *) { }
};
class B: public A { public:
    void f(int) { }
    virtual int h(B *) { }
};
class C: public B { public:
    void g(double) { }
    int h(B *) { }
};

// Application Codes
A a;
B b;
C c;

A *pA;
B *pB;
```

Invocation	Initialization		
	pA = &a;	pA = &b;	pA = &c;
pA->f(2);	A::f	B::f	B::f
pA->g(3.2);	A::g	A::g	C::g
pA->h(&a);	A::h	A::h	A::h
pA->h(&b);	A::h	A::h	A::h

The outline will be visible on the left of the slide as always. So, let us get started first with an example. So, in this example we have a simple structure. We have three classes; A is a base class, B is a specialization of A and C is a specialization of B. So, it is simple in that way. There are three functions; f, g and h. In class A of which, f and g are defined to be virtual. So, they are polymorphic and we have a class h which is non virtual. In class B, we have f and h which mean that we are overriding f and overriding h, but we are simply inheriting g. So, B does not have a signature of g and while we override h we do two things, one is we change the parameter type from A star to B star and also make it a polymorphic function. In C further, we override g and h from whatever C has inherited and just use inherited version of the function f in B in terms of C. So, this is the basic structure, we have constructed multiple objects A B C and we have two pointers; one pointer to A another is pointer to B.

In this context, what we need to answer is, if we have these four invocations that is p A this pointer invokes function f with the parameter, function g with a parameter, function h with two different parameters. If we look at these four invocations and if we try to see on to the different columns where p A, this pointer has been set with a certain objects address, so here it is set with the address of an A object, here of B object and here of C object the task is to fill up this matrix, which is already filled up here. So, what I will do I

will just quickly run you through as to why we should see the invocation the bindings in way that they are. So, to start with, we start with p A being assigned ampersand A which say that for p A in the static type is a star as we can easily see and also the dynamic type this is an a type object. So, the dynamic type of p A is also A.

So, since the static type is p A naturally it looks into the class A for the binding and the dynamic type being A clearly tell us that certainly, in all of this cases all these will actually invoke the class A functions. The only point to note here that in terms of the h function which takes A star pointer, whereas in the third case we have actually invoke it with the A star pointer, but in the last case we have invoked it with the B star pointer. So, if you look into this then this is p A invoking h with B star.

Now, we already know that B is A, and we already know that up cast is possible. So, this will also be possible because what will happen is this B star pointer; will get up cast to the A star pointer and the function will get invoked and that is the reason you have the same h function being invoked in both these cases. So, does not something very interesting let us look at the next one, the next is where p A has static type of A and the dynamic type of B because we have B object. So, naturally again p A being statically of type A, it will always look at this. So, now when you do p A pointer f, it looks at the function the class A, and then it finds that the function f is virtual, which means that it needs to be dynamically bound. It will be bound by the actual object that p A is pointing to and that object is B type and B overrides this function. So, earlier case it had invoked A colon colon f, Now, it will invoke B colon colon f which is straight forward.

In the second case again, when we look at p A invoking g, it invokes A colon colon g because even though g is virtual B does not override g. There is no g in class B. So, actually because it is virtual, this is actually trying to invoke B colon colon g, but B colon colon g is same as A colon colon g because b has simply inherited the function g. So, you have A colon colon g here. In last two cases, you do not have anything further because function h is in class A is non virtual. So, it is bound by the type of the pointer.

Let us move to the third, where I have an object of type C and I have pointer in to that. Naturally, if again the first place to look at is class A and we are invoking function f. So,

f is virtual, which means that it will get dispatched to class C, the type of the actually pointed object. So, if it gets invoked to in dispatch to class C then basically we should be invoking C colon colon f, but C does not override f. So, C has simply inherited the f from B that is why you get B colon colon f here. Whereas, when you try to do invoke function g then again you come here and you have a virtual function so that gets dispatch to certainly C colon colon g because you have C object. So, you have C colon colon g in this case C has over written whatever it had inherited. So, it has a separate C colon colon g function. So, that gets invoked and finally, in terms of the h function again you have the same behaviour because they are not virtual functions.

So, this how it behaves if these three objects, I invoke methods for these three objects using their address as if as a and A pointer to the A type of address. So, next what we look at what happens if we do the same way try to do the same invocation using the B type of pointer.

(Refer slide Time: 10:07)

The slide displays C++ code for classes A, B, and C, and application code. Below the code is a table summarizing the results of various pointer invocations.

```

// Class Definitions
class A { public:
    virtual void f(int) { }
    virtual void g(double) { }
    int h(A *) { }
};
class B: public A { public:
    void f(int) { }
    virtual int h(B *) { }
};
class C: public B { public:
    void g(double) { }
    int h(B *) { }
};

// Application Codes
A a;
B b;
C c;

A *pA;
B *pB;

```

Invocation	Initialization		
	pB = &a;	pB = &b;	pB = &c;
pB->f(2);	Error	B::f	B::f
pB->g(3.2);	Downcast	A::g	C::g
pB->h(&a);	(A *) to	No conversion (A *) to (B *)	
pB->h(&b);	(B *)	B::h	C::h

So, that is in the next slide. So, we are in this, there is no change on this side, what has changed is the invoke initialization the pointer B. Now, which we are using to invoke, earlier we were doing in terms of p A now, we are doing in terms of p B. Now, certainly the first thing that we see is this is a p B points to A. So, what do we expect? What is the

static type here? The static type is B star, and what is the dynamic type here is A star. So, if this assignment has to happen, if this assignment has to take place, then some A star object as to get cast to B star which happens to be a downcast. So, this down cast is not permitted. So, in all of these cases, actually we will not be able to do anything at all because this particular assignment itself will error out, because it is an error in terms of down cast. So, these cases are over.

Let us move to next one, which is using this and when we point to a B object. So, both the static and dynamic type is B. So, as we do that as in the previous case if I do p B pointer f then I start actually here because the type of the pointer is B here if should be invoked and f is a virtual function as we have seen it is a virtual function and it has a dynamic and it is pointing to a B object. So, it should invoke the function of the B object. Similarly, if we invoke g it should again start here and try to invoke in a similar manner B colon colon g because g is a virtual function again, but B has not overridden g. So, B has simply inherited the g function from A. So, you see that A colon colon g will be invoked.

Let us look at the next one where we are trying to invoke this function. Now, when we try to invoke this function what will happen we are here? So, we start looking in the class B and in class B do I have a function h which is actually a function which is overridden from the function in A. Therefore, when I try to do p B pointer h and if I pass ampersand A then the type of the actual parameter is A star, whereas the type of the formal parameter, what it expects is B star. So, it means that I need to perform a conversation from A star to B star, only then I will be able to do that call, but certainly as we know that this turns out be a down cast, this is a case of a down cast. So, this is not permitted. So, this particular function invocation will simply not compile.

Coming to the last, when you invoke h in terms of this p B and you try to see it will turn out to be B colon colon h because you certainly look here, you have B type object. So, certainly you will have B colon colon h. Finally, when if you have the C object, this is these cases prevail because you start here and this function f is a virtual therefore, it gets relegated dispatch to C colon colon f and C as not overridden f. So, it what actually invokes is the f that C inherits, which is B colon colon f, so that part is easy. Now, if you

invoke function g you again start here now in this what happens you do not see a g but what is a g that you see here that is actually this g, which is which b has inherited from the class A and that function is a virtual. So, actually B colon colon g the function g in class B is same as function g in class A and that actually is a virtual function.

So, this will delegate according to the type of the object that dynamically exist. So, it will try to listen through this and it will try to invoke C colon colon g that is this function so you get C colon colon g. The next case is a similar it will actually since this function as become virtual so you start looking in here. So, you see that h is a virtual function. So, you delegate dispatch that to the class C because you have C type object. So, you try to invoke this function and then you find that the actual parameter type is A star and the formal parameter type is B star. So, the conversation is a down cast. So, it here again you get and compilation error on this code. Finally, when you try to invoke h certainly you have p B. So, you start here you find that h is a virtual function. So, you dispatch according to the type of the dynamic object which brings you to this and you have c colon colon h.

I have quickly run it through this, maybe it was little fast for some of you. If it is then please spend some more time try to understand re run the video understand the logic in every case, but this kind of I have tried to cover all different cases of static and non static binding along with the what can be overloaded and what can be cast in terms of the upcast and the down cast. So, with this we will now move on.

(Refer slide Time: 16:20)

The image is a screenshot of a presentation slide. The title is "Staff Salary Processing: Problem Statement". The slide contains a list of bullet points describing the requirements for a salary processing application. The slide also includes a navigation menu on the left, a small video inset of a speaker, and footer information.

**Staff Salary Processing: Problem Statement**

- An organization needs to develop a salary processing application for its staff
- At present it has an engineering division only where Engineers and Managers work. Every Engineer reports to some Manager. Every Manager can also work like an Engineer
- The logic for processing salary for Engineers and Managers are different as they have different salary heads
- In future, it may add Directors to the team. Then every Manager will report to some Director. Every Director could also work like a Manager
- The logic for processing salary for Directors will also be distinct
- Further, in future it may open other divisions, like Sales division, and expand the workforce
- **Make a suitable extensible design**

Module 29  
Partha Prasin Das  
Objectives & Outline  
Reading  
Course  
Staff Salary Processing  
1. Problem Statement  
Summary

NPTEL MOOCs Programming in C++ Partha Prasin Das 5

With this we will take up a problem which we will try to solve in the remaining of this module and the next it is a staff salary processing problem. So, let us think of an organization that needs to develop a salary processing application for a staff. So, what is the information that we have we have that the organization has only one engineering division, where engineers and managers work every engineer reports to a manager and a manager can also work as a engineer. But what is different is the salary processing logic for the engineer and for the managers are different, possibly managers have some bonus and all that. So, the same function cannot be used to process the salary for both them.

Further what the organization wants that foresee that in some near future they will also possibly appoint directors in this division and then some managers will report to different directors and directors will would, if required would also work as managers and so on and directors will have their own processing logic for their salary. Further down in future, the organization wants to say that further down in future which is not in their view right now, but they are they could also open other divisions like they could open a sales division and expand their work force in a completely different type. So, what is required is in this context we need to make a suitable extensible design, design so that we can as a required add new employee types and new processing logic for the processing of salary without making significant changes to the existing code or if possible not making any change at all in the application that finally process the salary, this is the basic target.



(Refer slide Time: 18:09)

**C Solution:  
Engineer + Manager**

- How to represent Engineers and Managers?
  - struct
- How to initialize objects?
  - Initialization functions
- How to have a collection of mixed objects?
  - Array of union
- How to model variations in salary processing algorithms?
  - struct-specific functions
- How to invoke the correct algorithm for a correct employee type?
  - Function switch
  - Function pointers

NPTEL MOOCs Programming in C++ Partha Pratim Das 7

So, let us get started. So, I will take the clue from a solution which we will do in C and let us assume that we just have C at hand and let say how would have solve this problem assuming the very first version where we have just two types of employees; engineers and manager. So, there are several questions that you will need to answer before you can get started with the design and actually code, for example, how do you represent these concepts of engineers and managers, they will have to be represented somehow. So, possible since you are in C then the choices is almost trivial that you will use the struct that is not a problem. How do you initialize these objects with their name, their designation, their basic PA and so on? Certainly, you will need to have certain initialization functions that work for every structure type.

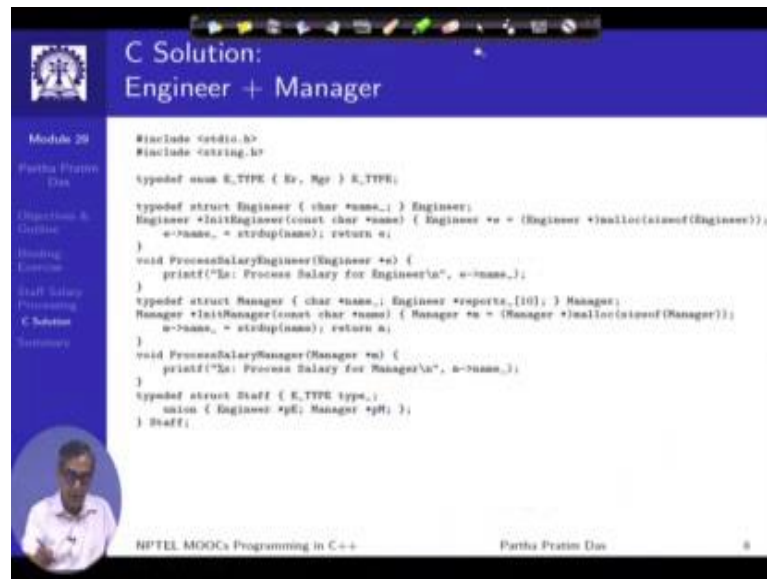
Third is a deeper question is finally, see if we have structures then we will have some structure for engineer and some structure for managers. These are two different kinds of structure types. So, if we have that then certainly we cannot make an array of engineers and managers. So, we will need a container because we can just make of an array of engineers or make an array of managers, but if you do that then the processing will get fragmented every time we add a new kind of employee. So, we need to have a container which can have any kind of objects that we can put together.

So, the solution in C is basically using array of union. So, what we can do we can say that I have a struct which has a fixed field, say type and then within that it has a union which has multiple different fields, say this is a pointer to engineer, this is a pointer to manager like this. So, what happens is when you want to keep that, you when you want keep an engineer record you put a specific type value here, say the type value is er and then set this pointer which is meant for to point to engineer object make this point to the particular instance of struct, but when you have to say do with deal with the manager then you set this type as say some manager value, manager type and set this pointer to the manager instance. Then depending on any element in this if you first check the type whether it is er or it is mgr you can decide which of the fields in the union to use and use it appropriately. So, this is a nice design rapper which is c uses extensively to keep a container of objects which are of mixed types. So, we will have to use that.

Now, the certainly the question of how do we have different salary processing, this is more like the initialization. So, which we will need to have for every structure type, we will need to have some structure of a specific function a function for engineer a function for manager and so on. Then finally, comes the question that if we have an array of a union. So, we have different records, this record could mean for an engineer, this record could be for manager, this could be for an engineer, this could be again for an engineer and so on, this could be for manager and so on.

How do I given any object, how do I decide what is a correct algorithm for the correct employee type. So, there is some kind of a switching which is involved that is I have to know because the correct employee the correct algorithm is encoded in tempt of the structure specific function. So, I have and in the array of union I have the employee type. So, I have to combine them in some way and create a switch and C provides typically two different options that I can have a function switch that is if else if else kind of structure or I can use a set of function pointers.

(Refer slide Time: 22:25)



The screenshot shows a presentation slide with a blue header and a white content area. The header contains the text 'C Solution: Engineer + Manager'. The content area displays C code for defining and managing Engineer and Manager structures. The code includes headers for stdio.h and string.h, defines an enum E\_TYPE for Engineer and Manager, and defines structures for Engineer, Manager, and Staff. It also includes initialization functions (InitEngineer, InitManager) and processing functions (ProcessSalaryEngineer, ProcessSalaryManager). A union Staff is defined to hold either an Engineer or a Manager pointer. The slide footer includes the NPTEL MOOCs logo, the text 'Programming in C++', the name 'Partha Pratim Das', and the number '8'.

```
#include <stdio.h>
#include <string.h>

typedef enum E_TYPE { Er, Mgr } E_TYPE;

typedef struct Engineer { char *name_; } Engineer;
Engineer *InitEngineer(const char *name) { Engineer *e = (Engineer *)malloc(sizeof(Engineer));
e->name_ = strdup(name); return e;
}

void ProcessSalaryEngineer(Engineer *e) {
printf("Er: Process Salary for Engineer\n", e->name_);
}

typedef struct Manager { char *name_; Engineer *reports_[10]; } Manager;
Manager *InitManager(const char *name) { Manager *m = (Manager *)malloc(sizeof(Manager));
m->name_ = strdup(name); return m;
}

void ProcessSalaryManager(Manager *m) {
printf("Mr: Process Salary for Manager\n", m->name_);
}

typedef struct Staff { E_TYPE type_;
union { Engineer *e; Manager *m; };
} Staff;
```

So, let us start and look at that we will initially use a function switch. So, just to quickly run through that I need to define the types of employees that I have, er for engineer; mgr. So, I defining an enumerate type these are structure which defines and engineer type I had just shown name any other attributes can be put in there. We have structures for manager where we have name and I have also optionally put that since engineers will report to managers. I will need to keep a list of engineers who report to the manager we need to initialize.

So, there is an initialization function which certainly takes the name and then allocates the space for that particular structure object through malloc, sets the name by copying and returns that. So, if I invoke the init engineer function with a name then it will return me the pointer to an engineer which I can then put in the store. Similarly, for the manager I have init function like this. We have separate processing function for engineer, which takes an engineer pointer and manager which takes a manager pointer and finally, these are collection store as I said.

So, I have a structure where one field is a type which will be one of these because it is e type and I have a union of two pointers. So, at any point of time any one of them will actually carry a meaningful value the other one will because they will be overlap because

it is a union both of them do not exist at the same time. So, finally, I call this type as staff. So, that turns out to be my final collection of employees.

(Refer slide Time: 24:13)

```
int main() {
    Staff allStaff[10];
    allStaff[0].type = E;
    allStaff[0].pF = InitEngineer("Rohit");
    allStaff[1].type = M;
    allStaff[1].pF = InitManager("Kamala");
    allStaff[2].type = M;
    allStaff[2].pF = InitManager("Rajib");
    allStaff[3].type = E;
    allStaff[3].pF = InitEngineer("Kacita");
    allStaff[4].type = E;
    allStaff[4].pF = InitEngineer("Shanku");

    for (int i = 0; i < 6; ++i) {
        E_TYPE s = allStaff[i].type;
        if (s == E) ProcessSalaryEngineer(allStaff[i].pF);
        else if (s == M) ProcessSalaryManager(allStaff[i].pF);
        else printf("Invalid Staff Type\n");
    }
    return 0;
}
-----
Output:
Rohit: Process Salary for Engineer
Kamala: Process Salary for Manager
Rajib: Process Salary for Manager
Kacita: Process Salary for Engineer
Shanku: Process Salary for Engineer
```

So, given this now I can certainly try to write the application. So, staff is my collection. So, I create an array of this union type. So, 10 staff can be kept in this all staff record then the next, this is just for making the whole program work. It just showed that how we would create the different engineer and manager and put them in the collection. So, if we just think about one in the first one then what we are trying to do we have this. So, this is all staff the 0th location, this as a type field.

So, there I put the engineer type because I want an engineer and then I invoke init engineer and as you saw that if I invoke it with the name of the engineer then you returns me pointer to the engineer. So, in the union I am basically looking at a engineer pointer which will get set. In this way, the manager pointer and the other engineer pointers also get set having done this then this is where I actually do the actual processing since what I will have to do; I will go over this whole collection. So, I have to go over this whole collection. Here, I have just hard coded the number 6 ignore that this could have been tracked also by counting. I just hard coded that there are 6 or possibly that it should have been 5. There are 5 such and so what I do is I have to go over.

So, I have an array of all this union records. So, I go to the 0th one, first I look at the type int. So, because I have to know which kind of, whether I have an engineer or a manager. So, you pick up the type here from all staff I the type and then depending on the type if the type is er, I invoke the processing salary engineering function with the corresponding pointer which is the pe pointer, if not otherwise I check if it is a manager type then I invoke the process salary manager function with the corresponding manager pointer and if it is even not even that then there must be some error. So, I just error print a error message.

If I do that then we have these 5 employees, 1 engineer and 3 engineer and 2 managers will get this particular output in terms of processing. So, this output is equivalent to as if the salary has been processed. So, this is how this could be written. So, naturally you can see there are some 9 points, one is a union container itself, one is a way you have to initialize and create this object and one certainly is this. So, if we if we just go forward and try to now take the first step in the future that you want add the director type here or design remain same. So, all that all this design details remain same, but only thing we are adding only thing we are adding a director type now. So, we want to add that.

(Refer slide Time: 27:17).

**C Solution: Engineer + Manager + Director**

```
#include <stdio.h>
#include <string.h>

typedef enum E_TYPE { Er, Mgr, Dir } E_TYPE;

typedef struct Engineer { char *name_; } Engineer;
Engineer *InitEngineer(const char *name) { Engineer *e = (Engineer *)malloc(sizeof(Engineer));
e->name_ = strdup(name); return e;
}
void ProcessSalaryEngineer(Engineer *e) {
printf("Er: Process Salary for Engineer\n", e->name_);
}

typedef struct Manager { char *name_; Engineer *reports_[10]; } Manager;
Manager *InitManager(const char *name) { Manager *m = (Manager *)malloc(sizeof(Manager));
m->name_ = strdup(name); return m;
}
void ProcessSalaryManager(Manager *m) {
printf("Ma: Process Salary for Manager\n", m->name_);
}

typedef struct Director { char *name_; Manager *reports_[10]; } Director;
Director *InitDirector(const char *name) { Director *d = (Director *)malloc(sizeof(Director));
d->name_ = strdup(name); return d;
}
void ProcessSalaryDirector(Director *d) {
printf("Da: Process Salary for Director\n", d->name_);
}

typedef struct Staff { E_TYPE type_;
union { Engineer *e; Manager *m; Director *d; };
} Staff;
```

NPTEL MOOCs Programming in C++ Partha Pratin Das 13

So, if we do that then all that it means that I have to add a structure type for the director. The initializer for director, the processing routine for the director in this there will have to be a code for the director. They will have to be another field which points to the director type. So, that would basically be all in terms of the representation.

(Refer slide Time: 27:40)



```
int main() { Staff allStaff[10];
allStaff[0].type = En;
allStaff[0].pH = InitEngineer("Rohit");
allStaff[1].type = Mgr;
allStaff[1].pH = InitManager("Eamala");
allStaff[2].type = Mgr;
allStaff[2].pH = InitManager("Rajith");
allStaff[3].type = En;
allStaff[3].pH = InitEngineer("Kavita");
allStaff[4].type = En;
allStaff[4].pH = InitEngineer("Shambhu");
allStaff[5].type = Dir;
allStaff[5].pH = InitDirector("Sanjana");

for (int i = 0; i < 6; ++i) {
    E_TYPE t = allStaff[i].type;
    if (t == En) ProcessSalaryEngineer(allStaff[i].pH);
    else if (t == Mgr) ProcessSalaryManager(allStaff[i].pH);
    else if (t == Dir) ProcessSalaryDirector(allStaff[i].pH);
    else printf("Invalid Staff Type\n");
}
return 0;
}
```

Output:  
Rohit: Process Salary for Engineer  
Eamala: Process Salary for Manager  
Rajith: Process Salary for Manager  
Kavita: Process Salary for Engineer  
Shambhu: Process Salary for Engineer  
Sanjana: Process Salary for Director  
NPTEL MOOCs Programming in C++ Partha Pratim Das 12

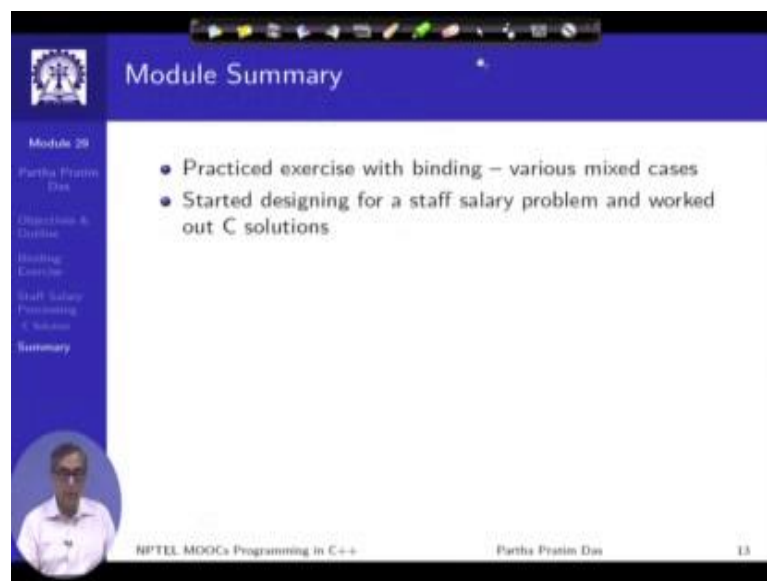
Then, if we actually look at the code then all this were there, then I can also create a director and put that, but here if you look at then in the processing routing, if you look at then you will need to have another condition which will check if somebody is a director and then call the corresponding process salary director function with the pointer of the direct. So, if you just see, we are getting into some bit of uncomfortable situation here because more and more types the basic design specification it has to be flexible, but more and more we add different types, we on one side here, this switch will keep on expanding.

So, the application code if I add a type then the application code has to know this is the application code main routine right. So, the application code as is to know that a new type as been added and it as to add some code there it as to add some more switch there. So, that is one a knowing factor we having to maintain the type information through an enumerated value separately, which is another disturbing factor and if you just look at if

you just look at the earlier one, we are creating lot of duplications, for example, here we have to do a union based collection which is also quite combustion because it is not guarantee that the value that I keep in the type and the particular field that I read in the union is no way checked by compiler.

So, it we can always make mistakes in that and we see lot of issues because see this name see if you look into this typedef this part in the director. This part all of these are basically code you know repeated things and if you look into these codes all these codes the copy of name and all that they are very similar in structure, but still we are just having to copy and you know paste and edit into that and creating lot of possible issues in terms of the code maintenance and expansion. So, by using C we are able to get to a solution we can survive, but we need to have better solutions for this problem, which is what we will discuss in the next module.

(Refer slide Time: 30:02)



So, here to summarise we have practiced a binding exercise and we have started designing for a staff salary application, where we have just worked out C solution and observed what are the difficulties lacunae in that solution.