

Programming in Modern C++
Professor Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur
Lecture 29

Polymorphism: Part 4: Staff Salary Processing using C

(Refer Slide Time: 00:35)

The slide is titled "Module Recap" and features a blue header with the IIT Kharagpur logo on the left and a small video inset of the professor on the right. The main content area is white and contains a bulleted list of topics discussed in the module. A vertical navigation menu is on the left side of the slide, and a footer at the bottom contains the text "Programming in Modern C++", "Partha Pratim Das", and "M29.2".

- Discussed why destructors must be `virtual` in a polymorphic hierarchy
- Introduced Pure Virtual Functions
- Introduced Abstract Base Class

The slide is titled "Module Objectives" and features a blue header with the IIT Kharagpur logo on the left and a small video inset of the professor on the right. The main content area is white and contains a bulleted list of learning objectives. A vertical navigation menu is on the left side of the slide, and a footer at the bottom contains the text "Programming in Modern C++", "Partha Pratim Das", and "M29.3".

- Understand design with ISA related concepts
- Understand the problems with C design

Welcome to Programming in Modern C++, we are going to discuss Module 29 now. In the earlier three modules, we have been developing different concepts of polymorphism, and the kinds of specific support that C++ provides over an inheritance hierarchy. We have seen different types of casting, static and dynamic binding, virtual functions. We have looked at why destructors must be virtual on a hierarchy. Why we need pure virtual function. What is abstract base class and so, on. So, in terms of dealing with a certain level of polymorphism,

which typically what we are dealing with is the runtime polymorphism right now along with some of the overloading which we have already learned, we have covered all these.

Now, we are equipped we are as an engineer, we are equipped with our tools. So, it is time for us to try out our tools on doing an actual design and implementation. So, the main objective of this module would be to understand design, where there is a prevalence of ISA-related concepts that is generalization specialization. And we will start with the C design and identify why C is not an appropriate platform for dealing with such design requirements. But before we will start doing this, we will first take a quick look at some exercise for practice.

(Refer Slide Time: 02:08)

The slide is titled "Module Outline" and features a dark blue header with a logo on the left and a small video feed of the presenter on the right. A vertical navigation menu on the left lists various topics, with "Binding: Exercise" highlighted. The main content area contains a numbered list of three items:

- 1 Binding: Exercise
 - Exercise 1
 - Exercise 2
- 2 Staff Salary Processing
 - C Solution
 - Engineer + Manager
 - Engineer + Manager + Director
 - Advantages and Disadvantages
- 3 Module Summary

At the bottom, the text "Programming in Modern C++" is on the left, "Partha Pratim Das" is in the center, and "M29.4" is on the right.

The slide is titled "Binding: Exercise" and features a dark blue header with a logo on the left and a small video feed of the presenter on the right. A vertical navigation menu on the left lists various topics, with "Binding: Exercise" highlighted. The main content area contains the text "Binding: Exercise" in a large, bold, red font.

At the bottom, the text "Programming in Modern C++" is on the left, "Partha Pratim Das" is in the center, and "M29.5" is on the right.

Binding: Exercise 1

Module: M29

Partha Pratim Das

Objectives & Outlines

Binding: Exercise 1

Staff Salary Processing

C. Science

Engineer - Manager

Engineer - Manager - Engineer

Advantages and Disadvantages

Module Summary

```

// 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;
    
```

A
A
B
A
C

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

Binding: Exercise 1

Module: M29

Partha Pratim Das

Objectives & Outlines

Binding: Exercise 1

Staff Salary Processing

C. Science

Engineer - Manager

Engineer - Manager - Engineer

Advantages and Disadvantages

Module Summary

```

// 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);	✓		
pA->g(3.2);	✓	✓	✓
pA->h(&a);	✓	✓	✓
pA->h(&b);	✓	✓	✓

So, this is your outline here will be available on the left. So, I have put two exercises actually one exercise on binding having two different instance requirements. So, there is a class A which is specialized in B which is further specialized in C and there are virtual functions here. There is a non-virtual function, some virtual functions like g() is inherited but not overridden. Some are inherited and overridden, some are inherited and made virtual hiding the function of the base class. Then again you have some which is overridden into class C others just iterated.

So, if you have this kind of a class hierarchy and if you construct objects of three classes A, B and C and you have two pointers, one is of type A and another is of type B then you should be able to work this out. There are four, there are three functions f(), g(), and h() of that you can see that f() does not have any overload, g() also does not have any overload, but if you

look at h() you find that there is a overload. So, there are four different function signatures to deal with.

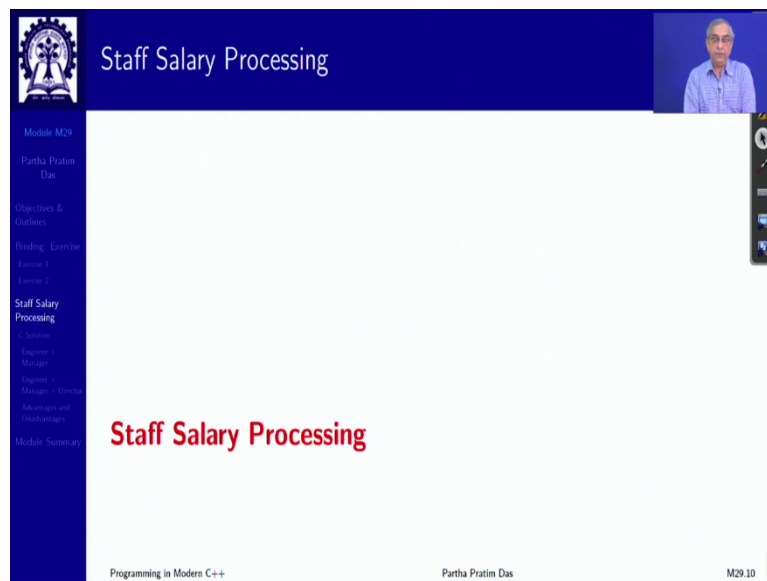
So, I would need you to fill up this matrix that which function will be called with pA->f(a) if pA is initialized with the address of the a object. So, similarly do that for each four. Again, a different instance is if you initialize if you assign pA with the address of the b object, which functions will be called and so on. So, please work this out and do not look at the answer it is on the next slide for your verification, but first work this out and then check that you have done it correctly.

This is I am putting this so that it is very, very important that you understand the static and dynamic binding of function calls in a very solid way that is overriding, overloading polymorphic virtual functions must all be very clear to you. So please work this out. This is the first part of the exercise. I will skip the solution slide which is after this.

The second part of the exercise has exactly the same problem, exactly the same instance except that now we are using pB earlier we are using pA now we are using pB which is pointed to the B class and all the three cases in all the three cases pB is assigned the address of a object address of b object and address of c object, you have to comment, you have to write what will happen in all these different 12 cases.

Mind you, out of the two kinds of instances of this exercise that I have just discussed, it is possible that some of the function calls or some of the instantiation may not actually compile. So, it may give a compile time error, it may give runtime error, you have to consider all of that. And if you think that certain cases have this error you write down that error itself. Again, like the first part, the solution is given in the next slide, which you referred to only after you have solved this, so I will skip the solution.

(Refer Slide Time: 06:31)



Staff Salary Processing

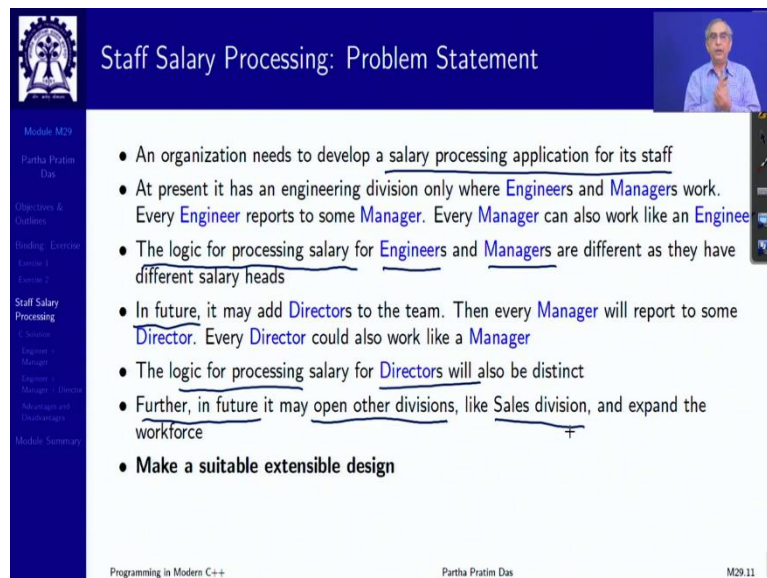
Module M29
Partha Pratim Das

Objectives & Outlines
Binding Exercise
Exercise 1
Exercise 2
Staff Salary Processing
C++
Engineer
Manager
Engineer Manager
Advantages and Disadvantages
Module Summary

Staff Salary Processing

Programming in Modern C++ Partha Pratim Das M29.10

This slide shows the title 'Staff Salary Processing' in red text on a white background. A navigation sidebar is on the left, and a video feed of the presenter is in the top right corner.



Staff Salary Processing: Problem Statement

Module M29
Partha Pratim Das

Objectives & Outlines
Binding Exercise
Exercise 1
Exercise 2
Staff Salary Processing
C++
Engineer
Manager
Engineer Manager
Advantages and Disadvantages
Module Summary

- 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**

Programming in Modern C++ Partha Pratim Das M29.11

This slide contains a bulleted list of requirements for a salary processing application. The text is black on a white background. A navigation sidebar is on the left, and a video feed of the presenter is in the top right corner.

Now, let me come to the meet of the discussion in this module. What I will do from here till the end of this week, that is covering the next module also, we will try to solve a simple yet quite illustrative problem, which will show us how really the inheritance and polymorphic strengths must be used. And we will do this through a process of iterative refinement, I would say, that is we will start with a very initial first cut solution, which easily come to our mind and we will try to implement that. Identify the problems they are in and try to rectify that using the proper modeling and again look at what are the gaps that still remain and keep on doing this till we come to a solution which is acceptable and which is nice.

So, let me state the problem for once. The organization, there is a staff salary processing requirement, this is an organization which has different staff, and it needs to process the

salary of the staff. So, let us assume that initially there is only an engineering division where engineers and managers work and every engineer reports to a manager, every manager can also work like an engineer, but primarily the manager's task is to take reporting from the engineer.

Now, naturally the logic of processing salary for an engineer and the manager would be different. Because managers would have different kinds of salary structure and so, not only they will have different salary, but they will have different kinds of salary structure. And maybe they have bonuses, maybe they have stock options and different heads and so on.

So, different functions are required for processing the salary of the engineer and processing the salary of the manager. This is at the present. Now, in future it may add the organization may add directors to the team, so that directors will be responsible for broader business goals and every manager will report to some directors, and a director could also work like a manager if needed. Naturally, the logic of processing salary for the directors will also be distinct from the managers as well as the engineers.

Even further down in future, what happens is the company, the organization may decide to open other divisions like sales division, like HR division and so on and expand the workforce. And again, there will be such generalization, specialization relationships and different processing requirements. So, the what we need to do is given this simple problem we need to make a suitable design, which can be properly extended as needed over a period of time.

(Refer Slide Time: 09:56)

C Solution: Function Switch: Engineer + Manager

- How to represent **Engineers** and **Managers**? ✓
 - Collection of **structs** ✓
- 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

Programming in Modern C++ Partha Pratim Das M29.12

C Solution: Function Switch: Engineer + Manager

- How to represent **Engineers** and **Managers**?
 - Collection of **structs**
- 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

Programming in Modern C++ Partha Pratim Das M29.12

Type Switch

So, we start by asking ourselves a couple of questions, which we will keep on asking repeatedly, because if we want to do the design then first you need to set certain parameters for the design, and some questions very commonly naturally will come, which we need to address. For example, the first thing obviously is how to represent engineers and managers.

Obvious, we are in the domain of C, so, they will have certain attributes, so we will put them into structure, so we will have a collection of structures, one for the engineer, one for the manager and so on. How do we initialize these objects, engineer or manager objects, we need to write initialization function. We have seen that before in terms of stack example and so on. Now, how do we have the total aggregation collection of staffs? There would be engineer objects, engineer's structures, there will be manager objects, managers structures, different

types, so how do we put that together. We have learned in C, the best way to do that is to put them as a union, so that it could be either a manager or an engineer but not both.

And then I need a collection for that so I have an array of union. I am just trying to ask the natural questions that will happen and try to identify what is the language feature what is a design feature that we will use to get to that solution. Now, the next question is how do we model various salary processing algorithms there will be different so, we need different functions for every structure type.

So, we need different salary pricing for engineer, different salary pricing for manager, we have to implement separate functions. Now, the final question is having done all this how do I uniformly treat the entire collection of staffs and process their salary in their respective way? So, how to invoke the correct algorithm? Like herein I have the collection of all the employees, all the staffs, but for each I have different processing algorithm.

So, to for processing the salary of all employees I have to pick up every employee from this array, check what is the type is that employee an engineer or is that employ a manager, and accordingly call the respective processing function. This kind of a requirement where we make decisions based on the type of an object is known as a type switch. So, we can implement that using a function switch or we can implement that using function pointers. So, this is a basic level of analysis that goes into solving this problem.

(Refer Slide Time: 13:23)

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

typedef enum E_TYPE { Ex, Mgr } E_TYPE; // Tag for type of staff

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("%s: 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("%s: Process Salary for Manager\n", m->name_); }

typedef struct Staff { // Aggregation of staffs
    E_TYPE type_;
    union { Engineer *pE; Manager *pM; };
} Staff;
```

Programming in Modern C++ Partha Pratim Das M29.13

C Solution: Function Switch: Engineer + Manager

Module M29
Partha Pratim Das

Objectives & Outlines

Binding Exercise
Lesson 1
Lesson 2

Staff Salary Processing
C Solution

Engineer + Manager

Lesson 1
Lesson 2
Manager + Exercise
Advantages and Disadvantages

Module Summary

```

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

typedef enum E_TYPE { Er, Mgr } E_TYPE; // Tag for type of staff

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("%s: 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("%s: Process Salary for Manager\n", m->name_); }

typedef struct Staff { // Aggregation of staffs
    E_TYPE type_;
    union { Engineer *pE; Manager *pM; };
} Staff;

```



Programming in Modern C++
Partha Pratim Das
M29.13

C Solution: Function Switch: Engineer + Manager

Module M29
Partha Pratim Das

Objectives & Outlines

Binding Exercise
Lesson 1
Lesson 2

Staff Salary Processing
C Solution

Engineer + Manager

Lesson 1
Lesson 2
Manager + Exercise
Advantages and Disadvantages

Module Summary

```

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

typedef enum E_TYPE { Er, Mgr } E_TYPE; // Tag for type of staff

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("%s: 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("%s: Process Salary for Manager\n", m->name_); }

typedef struct Staff { // Aggregation of staffs
    E_TYPE type_;
    union { Engineer *pE; Manager *pM; };
} Staff;

```



Programming in Modern C++
Partha Pratim Das
M29.13

C Solution: Function Switch: Engineer + Manager

Module M29
Partha Pratim Das

Objectives & Outlines

Binding Exercise
Lesson 1
Lesson 2

Staff Salary Processing
C Solution

Engineer + Manager

Lesson 1
Lesson 2
Manager + Exercise
Advantages and Disadvantages

Module Summary

```

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

typedef enum E_TYPE { Er, Mgr } E_TYPE; // Tag for type of staff

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("%s: 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("%s: Process Salary for Manager\n", m->name_); }

typedef struct Staff { // Aggregation of staffs
    E_TYPE type_;
    union { Engineer *pE; Manager *pM; };
} Staff;

```



Programming in Modern C++
Partha Pratim Das
M29.13

Now, let us get down to some code. Say, the first thing I need to know is I need to know whether some employees are manager or an engineer. So, how do I keep that I create an enum `E_TYPE` with two tags, so that if some employees are engineer will say, Er, tag it as `Er` other will tag it as `Mgr`. Now, we need to have the structure. So, I define struct engineer for simplicity I have just kept the name, other fields could be there and typedef that as `Engineer` so that I do not have to write too much.

As already identified, I need an initialization function. So, that initialization function must take the name and return me an `Engineer` pointer. In C++, it would be the constructor, but here we do not have that. So, we dynamically allocate the memory, copy the name and return that pointer. Very simple thing to do.

Now, having done this, the next is the same. The next obviously is I need a processing function. So, I say `processSalaryEngineer()` which takes an engineer pointer and does the processing just as a placeholder, I am doing a `printf`. Similar design I do for the `Manager`. And then, so I have tags ready, I have objects and their processing ready, their initialization ready. Now, I have to put the collection of the staffs.

So, I need to, a staff can be an engineer or a staff can be a manager. So, I need to unify these two types, so I create a union, union of `Engineer` pointer and `Manager` pointer. But then that classical question as to how do I know which type of pointer it is. So, I have a tag which will say which type of pointer it is. Whether it is an `Engineer` pointer or a `Manager` pointer, and then I aggregate these into a structure. So, this becomes my `Staff` entity. This is my `Engineer` entity, this is my `Manager` entity, this is a wrapper, this structure with the `E_TYPE` tag becomes my `Employee` entity.

(Refer Slide Time: 16:22)

C Solution: Function Switch: Engineer + Manager

```
int main() {
    Staff allStaff[10];
    allStaff[0].type_ = Er; allStaff[0].pE = InitEngineer("Rohit");
    allStaff[1].type_ = Mgr; allStaff[1].pM = InitManager("Kamala");
    allStaff[2].type_ = Mgr; allStaff[2].pM = InitManager("Rajib");
    allStaff[3].type_ = Er; allStaff[3].pE = InitEngineer("Kavita");
    allStaff[4].type_ = Er; allStaff[4].pE = InitEngineer("Shambhu");

    for (int i = 0; i < 5; ++i) {
        E_TYPE t = allStaff[i].type_;
        if (t == Er)
            ProcessSalaryEngineer(allStaff[i].pE);
        else if (t == Mgr)
            ProcessSalaryManager(allStaff[i].pM);
        else printf("Invalid Staff Type\n");
    }
}
```

Rohit: Process Salary for Engineer
Kamala: Process Salary for Manager
Rajib: Process Salary for Manager
Kavita: Process Salary for Engineer
Shambhu: Process Salary for Engineer

Programming in Modern C++ Partha Pratim Das M29.14

C Solution: Function Switch: Engineer + Manager

```
int main() {
    Staff allStaff[10];
    allStaff[0].type_ = Er; allStaff[0].pE = InitEngineer("Rohit");
    allStaff[1].type_ = Mgr; allStaff[1].pM = InitManager("Kamala");
    allStaff[2].type_ = Mgr; allStaff[2].pM = InitManager("Rajib");
    allStaff[3].type_ = Er; allStaff[3].pE = InitEngineer("Kavita");
    allStaff[4].type_ = Er; allStaff[4].pE = InitEngineer("Shambhu");

    for (int i = 0; i < 5; ++i) {
        E_TYPE t = allStaff[i].type_;
        if (t == Er)
            ProcessSalaryEngineer(allStaff[i].pE);
        else if (t == Mgr)
            ProcessSalaryManager(allStaff[i].pM);
        else printf("Invalid Staff Type\n");
    }
}
```

Rohit: Process Salary for Engineer
Kamala: Process Salary for Manager
Rajib: Process Salary for Manager
Kavita: Process Salary for Engineer
Shambhu: Process Salary for Engineer

Programming in Modern C++ Partha Pratim Das M29.14

C Solution: Function Switch: Engineer + Manager

```
int main() {
    Staff allStaff[10];
    allStaff[0].type_ = Er; allStaff[0].pE = InitEngineer("Rohit");
    allStaff[1].type_ = Mgr; allStaff[1].pM = InitManager("Kamala");
    allStaff[2].type_ = Mgr; allStaff[2].pM = InitManager("Rajib");
    allStaff[3].type_ = Er; allStaff[3].pE = InitEngineer("Kavita");
    allStaff[4].type_ = Er; allStaff[4].pE = InitEngineer("Shambhu");

    for (int i = 0; i < 5; ++i) {
        E_TYPE t = allStaff[i].type_;
        if (t == Er)
            ProcessSalaryEngineer(allStaff[i].pE);
        else if (t == Mgr)
            ProcessSalaryManager(allStaff[i].pM);
        else printf("Invalid Staff Type\n");
    }
}
```

Rohit: Process Salary for Engineer
Kamala: Process Salary for Manager
Rajib: Process Salary for Manager
Kavita: Process Salary for Engineer
Shambhu: Process Salary for Engineer

Programming in Modern C++ Partha Pratim Das M29.14

C Solution: Function Switch: Engineer + Manager

```

int main() {
    Staff allStaff[10];
    allStaff[0].type_ = Er; allStaff[0].pE = InitEngineer("Rohit");
    allStaff[1].type_ = Mgr; allStaff[1].pM = InitManager("Kamala");
    allStaff[2].type_ = Mgr; allStaff[2].pM = InitManager("Rajib");
    allStaff[3].type_ = Er; allStaff[3].pE = InitEngineer("Kavita");
    allStaff[4].type_ = Er; allStaff[4].pE = InitEngineer("Shambhu");

    for (int i = 0; i < 5; ++i) {
        E_TYPE t = allStaff[i].type_;
        if (t == Er)
            ProcessSalaryEngineer(allStaff[i].pE);
        else if (t == Mgr)
            ProcessSalaryManager(allStaff[i].pM);
        else
            printf("Invalid Staff Type\n");
    }
}

```

Rohit: Process Salary for Engineer
 Kamala: Process Salary for Manager
 Rajib: Process Salary for Manager
 Kavita: Process Salary for Engineer
 Shambhu: Process Salary for Engineer

Programming in Modern C++ | Partha Pratim Das | M29.14

Let us move on. So, once we have that, then my actual staff would be just an array. So, I put an array keeping a maximum number of staff to create every engineer. Suppose Rohit is an engineer, so what do I have to do, I have to do at create the Rohit object. So do InitEngineer for Rohit, because he is an engineer put it to pE part of the union and tag it as Er.

Kamala is a manager, so I initialize that, put the tag as manager. I do that for all the employees, so my collection is ready then finally, to process the salary. To process the salary what do I have to do? I have to do it for all the staff. I have just hard coded it here to keep things simple. Then, as I go along this array of staff, I pick up the first element and check what is the type. So, I take that in E_TYPE. What are the possibilities? It could be an Er or it could be Mgr.

So, I check with whatever I have got, I check with Er, if it is true I call the, I know that this is an engineer record, so I call the processSalaryEngineer() with the respective Engineer pointer. If it is not, I check whether it is a manager. If it is, then I do processSalaryManager with the Manager pointer. Otherwise, for some reason it should not happen.

But otherwise, if there is there is some other tag that has come then I say it is invalid, so that is an exception. Do that, try to run this. This is given below is output that you will get to see the salary has been processed. So, we have a first cut design for the problem.

(Refer Slide Time: 18:30)

C Solution: Function Switch: Engineer + Manager + Director

- How to represent **Engineers, Managers, and Directors**?
 - Collection of **structs**
- 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

Programming in Modern C++ Partha Pratim Das M29.15

C Solution: Function Switch: Engineer + Manager + Director

```
#include <stdio.h>
#include <stdlib.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("%s: 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("%s: 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("%s: Process Salary for Director\n", d->name_);
}

typedef struct Staff { E_TYPE type_; union { Engineer *pE; Manager *pM; Director *pD; };
} Staff;
```

Programming in Modern C++ Partha Pratim Das M29.16

C Solution: Function Switch: Engineer + Manager + Director

```
int main() { Staff allStaff[10];
allStaff[0].type_ = Er; allStaff[0].pE = InitEngineer("Rohit");
allStaff[1].type_ = Mgr; allStaff[1].pM = InitManager("Kamala");
allStaff[2].type_ = Mgr; allStaff[2].pM = InitManager("Rajib");
allStaff[3].type_ = Er; allStaff[3].pE = InitEngineer("Kavita");
allStaff[4].type_ = Er; allStaff[4].pE = InitEngineer("Shambhu");
allStaff[5].type_ = Dir; allStaff[5].pD = InitDirector("Ranjana");

for (int i = 0; i < 6; ++i) { E_TYPE t = allStaff[i].type_;
if (t == Er)
ProcessSalaryEngineer(allStaff[i].pE);
else if (t == Mgr)
ProcessSalaryManager(allStaff[i].pM);
else if (t == Dir)
ProcessSalaryDirector(allStaff[i].pD);
else printf("Invalid Staff Type\n");
}
}
```

Rohit: Process Salary for Engineer
Kamala: Process Salary for Manager
Rajib: Process Salary for Manager
Kavita: Process Salary for Engineer
Shambhu: Process Salary for Engineer
Ranjana: Process Salary for Director

Programming in Modern C++ Partha Pratim Das M29.17

C Solution: Function Switch: Engineer + Manager + Director

Instead of if-else chain, we can use switch to explicitly switch on the type of employee

```

int main() { Staff allStaff[10];
  allStaff[0].type_ = Er; allStaff[0].pE = InitEngineer("Rohit");
  allStaff[1].type_ = Mgr; allStaff[1].pM = InitManager("Kamala");
  allStaff[2].type_ = Mgr; allStaff[2].pM = InitManager("Rajib");
  allStaff[3].type_ = Er; allStaff[3].pE = InitEngineer("Havita");
  allStaff[4].type_ = Er; allStaff[4].pE = InitEngineer("Shambhu");
  allStaff[5].type_ = Dir; allStaff[5].pD = InitDirector("Ranjana");

  for (int i = 0; i < 6; ++i) { E_TYPE t = allStaff[i].type_;
    switch (t) {
      case Er: ProcessSalaryEngineer(allStaff[i].pE); break;
      case Mgr: ProcessSalaryManager(allStaff[i].pM); break;
      case Dir: ProcessSalaryDirector(allStaff[i].pD); break;
      default: printf("Invalid Staff Type\n"); break;
    }
  }
}

```

Rohit: Process Salary for Engineer
 Kamala: Process Salary for Manager
 Rajib: Process Salary for Manager
 Havita: Process Salary for Engineer
 Shambhu: Process Salary for Engineer
 Ranjana: Process Salary for Director

Programming in Modern C++ Partha Pratim Das M29.18

Now, let us extend it a little bit. Let us also introduce directors and see what we have all we have to do. We have solved this for engineer and manager now, and we are told that soon directors are going to come. So, as we introduce directors there is no, none of these questions will have a different answer. So, we have the same answers, same design perspectives, all that we need to do is to actually go ahead and implement.

So, what do I need to do, now directors are there, so I need a third type of employee so I introduce Dir here. My Engineer does not change, my Manager does not change, but I will have the Director created in the same way defined in the same way with the structure with the manager reporting with the director initialization and the processSalaryDirector() a new function.

What do I need to do for the staff's collection? I earlier had engineer and manager now have to introduce this. So, this is what I need to introduce. This is what I need to introduce, this is what I need to introduce. These are the changes from my earlier design that I am having to do, distributed at different places.

Having done that, let us see. Now there is no specific change in terms of this array in the similar way, I create different staff instances. But coming back here I need to add another if condition this is my type switch or by function switch. Because now I have a new type of object, the director object, requiring a new type of processing function. So, if else, if else, if else like this, it will, so, the total changes a changes that we had to do earlier and in the earlier slide at this one, and then this design now works for Engineer, Manager and Director.

This is just to show you a different implementation of the main function. Instead of the if else chain, you can also use a switch, which at least looks cleaner. If you do if else and every value of this tag is a constant, so that is an ideal case for writing a switch, so this at least will look same if there are a number of 10, 20 different types of employees otherwise, if else chain will become very, very long code, but that is just a you know, kind of management, code management issue.

(Refer Slide Time: 21:28)

C Solution: Advantages and Disadvantages

- **Advantages**
 - Solution exists! ✓
 - Code is well structured – has patterns
- **Disadvantages**
 - Employee data has scope for better organization
 - ▷ No encapsulation for data
 - ▷ Duplication of fields across types of employees – possible to mix up types for them (say, `char` and `string`)
 - ▷ Employee objects are created and initialized dynamically through `Init...` functions. How to release the memory?
 - Types of objects are managed explicitly by `E.Type`:
 - ▷ Difficult to extend the design – addition of a new type needs to:
 - Add new type code to `enum E.Type`
 - Add a new pointer field in `struct Staff` for the new type
 - Add a new case (`if-else` or `case`) based on the new type
 - ▷ Error prone – developer has to decide to call the right processing function for every type (`ProcessSalaryManager` for `Mgr` etc.)
 - Unable to use Function Pointers as each processing function takes a parameter of different type - no common signature for dispatch
- **Recommendation**
 - Use classes for encapsulation on a hierarchy

Handwritten notes: Name, E-name

Module M29
Partha Pratim Das
Objectives & Outcomes
Binding Exercise
Lesson 1
Lesson 2
Staff Salary Processing
C Solution
Lesson 1
Lesson 2
Lesson 3
Lesson 4
Lesson 5
Advantages and Disadvantages
Module Summary

Programming in Modern C++ Partha Pratim Das M29.19

C Solution: Advantages and Disadvantages

- **Advantages**
 - Solution exists!
 - Code is well structured – has patterns
- **Disadvantages**
 - Employee data has scope for better organization
 - ▷ No encapsulation for data
 - ▷ Duplication of fields across types of employees – possible to mix up types for them (say, `char` and `string`)
 - ▷ Employee objects are created and initialized dynamically through `Init...` functions. How to release the memory?
 - Types of objects are managed explicitly by `E.Type`:
 - ▷ Difficult to extend the design – addition of a new type needs to:
 - Add new type code to `enum E.Type`
 - Add a new pointer field in `struct Staff` for the new type
 - Add a new case (`if-else` or `case`) based on the new type
 - ▷ Error prone – developer has to decide to call the right processing function for every type (`ProcessSalaryManager` for `Mgr` etc.)
 - Unable to use Function Pointers as each processing function takes a parameter of different type - no common signature for dispatch
- **Recommendation**
 - Use classes for encapsulation on a hierarchy

Module M29
Partha Pratim Das
Objectives & Outcomes
Binding Exercise
Lesson 1
Lesson 2
Staff Salary Processing
C Solution
Lesson 1
Lesson 2
Lesson 3
Lesson 4
Lesson 5
Advantages and Disadvantages
Module Summary

Programming in Modern C++ Partha Pratim Das M29.19

C Solution: Advantages and Disadvantages

- **Advantages**
 - Solution exists!
 - Code is well structured – has patterns
- **Disadvantages**
 - Employee data has scope for better organization
 - ▷ No encapsulation for data
 - ▷ Duplication of fields across types of employees – possible to mix up types for them (say, `char` and `string`)
 - ▷ Employee objects are created and initialized dynamically through `Init...` functions. How to release the memory?
 - Types of objects are managed explicitly by `E.Type`:
 - ▷ Difficult to extend the design – addition of a new type needs to:
 - Add new type code to `enum E.Type`
 - Add a new pointer field in `struct Staff` for the new type
 - Add a new case (`if-else` or `case`) based on the new type
 - ▷ Error prone – developer has to decide to call the right processing function for every type (`ProcessSalaryManager` for `Mgr` etc.)
 - Unable to use Function Pointers as each processing function takes a parameter of different type - no common signature for dispatch
- **Recommendation**
 - Use classes for encapsulation on a hierarchy

Programming in Modern C++ Partha Pratim Das M29.19

C Solution: Advantages and Disadvantages

- **Advantages**
 - Solution exists!
 - Code is well structured – has patterns
- **Disadvantages**
 - Employee data has scope for better organization
 - ▷ No encapsulation for data
 - ▷ Duplication of fields across types of employees – possible to mix up types for them (say, `char` and `string`)
 - ▷ Employee objects are created and initialized dynamically through `Init...` functions. How to release the memory?
 - Types of objects are managed explicitly by `E.Type`:
 - ▷ Difficult to extend the design – addition of a new type needs to:
 - Add new type code to `enum E.Type`
 - Add a new pointer field in `struct Staff` for the new type
 - Add a new case (`if-else` or `case`) based on the new type
 - ▷ Error prone – developer has to decide to call the right processing function for every type (`ProcessSalaryManager` for `Mgr` etc.)
 - Unable to use Function Pointers as each processing function takes a parameter of different type - no common signature for dispatch
- **Recommendation**
 - Use classes for encapsulation on a hierarchy

Programming in Modern C++ Partha Pratim Das M29.19

Now, let us evaluate what have we got. So, let us look at the advantages and disadvantages. The first advantage is that the solution exists, that is the first thing. If you are working, the first thing you need to do is to make sure that it can be solved. Then comes good solution, extendable solution, robust solution, testable solution, verifiable solution all of that will come later. So, the first thing we have got the solution exist.

The second we observe is, the way we have been analyzing very clearly, and writing the code step by step based on our analysis, it has a well structured pattern. You have a tag, you have a structure, initializer, a processing function, you have a wrapper for the union of different types then an array for this and finally a type switch.

So, there is a certain nice pattern in this. But it has a lot of disadvantages. For example, the employee data has a lot of scope for better organization, the first thing missing is there is no

encapsulation, C has no encapsulation. So, structure is all everything is public. Then, we are having to duplicate fields across types of employees. For example, every employee has a name, so every structure must have a field called name.

We may mix up in the name of that field. I mean, I may call something as name, someone later on may want to call it a e-name, someone may call it as name_ so there is confusion. Some might want to say that this is a char *, some might want to say that this is and this is a const char * and so on so forth. So, this kind of duplication of fields same fields, which mean the same thing across different types are really difficult in the object model.

Objects are created and initialized dynamically through Init functions. So, what we have missed out is how to clean them up. So, I just wanted to keep the idea simple, but you see the as you need the Init you will also need to provide a way to clean up the memory. You need a releasing function, and that releasing function like the processing function will again have to be called based on the specific type. You will come across another type switch there to release because all structures will not have the same set of fields and therefore, they cannot be released by the same code, lot of issues.

So, this is about the overall data modeling. Then the more severe problems are objects are managed, types of objects are managed explicitly by E_Type. We are putting a tag. So, what did we have to do to extend the design, to add a new type like Director we did, I needed to add a separate code E_Type. Add a new pointer filled in the struct Staff in the beginning so that I can point to that and add a new case in the if else or the case switch.

So, if all of these and if we have the releasing function will have yet another switch to deal with. If all of these are not correctly done in every case of addition of a type we are going to get into severe problem. So, this makes the developer has to decide to call the right processing function for every type. All of these will have a need a lot of development care. So, that is going to be a serious drawback for this design.

Finally, here, we have seen earlier that function pointers because we have different types of functions for processing. So, instead of function switch could we not have used function pointers. We would have liked to, but the difficulty here is that the employee processing, the engineer processing function takes an attribute, takes a parameter which is Engineer*. The manager processing function takes an attribute takes an argument which is Manager*, Director*, so on. So, all of their arguments are different. So, I cannot have a common signature through which function pointers can be directly dispatched.

So, function pointer-based approach on this design the way we have done the design will also not work. So, we are losing out on that ability. Now, after the analysis, what comes is the recommendation. So, the recommendation is to use classes for encapsulation on a hierarchy and that is the refinement we will start working on.

(Refer Slide Time: 27:08)

The image shows a presentation slide titled "Module Summary". The slide has a dark blue header with the title in white. Below the header, there is a white area containing two bullet points:

- Practiced exercise with binding – various mixed cases
- Started designing for a staff salary problem and worked out C solutions

On the left side of the slide, there is a vertical navigation menu with the following items: "Module M29", "Partha Pratim Das", "Objectives & Outlines", "Binding - Exercise", "Exercise 1", "Exercise 2", "Staff Salary", "Processing", "1. Summary", "2. Summary", "Message", "Exercise", "Message - Exercise", "Advantages and Disadvantages", and "Module Summary". At the bottom of the slide, there is a footer with the text "Programming in Modern C++", "Partha Pratim Das", and "M29.20".

So, to summarize, in this module, we have provided some practice exercise, but most importantly, we have started designing a staff salary problem. We have worked out a C solution and we have shown what are the various shortcomings of that design, starting from modeling of the object, as well as processing through types switch, as well as being unable to use function pointers and so on, which in the next module will keep on refining with C++, and go to a better design. Thank you very much for your attention. See you in the next module.