

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

Lecture 30

Polymorphism: Part 5: 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. A vertical navigation menu is on the left side, and a toolbar is on the right. The main content area contains two bullet points:

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

At the bottom, the footer includes "Programming in Modern C++", "Partha Pratim Das", and "MK12".

The slide is titled "Module Objectives" and features a blue header with the IIT Kharagpur logo on the left. A vertical navigation menu is on the left side, and a toolbar is on the right. The main content area contains two bullet points:

- Understand design with class hierarchy
- Understand the process of design refinement to get to a good solution from a starting one

At the bottom, the footer includes "Programming in Modern C++", "Partha Pratim Das", and "MK13".

The slide is titled "Module Outline" and features a blue header with a logo on the left and a small video feed of the presenter on the right. The main content is a list of topics:

- 1 Staff Salary Processing: C Solution
 - Flat C Solution: Recap
 - Advantages and Disadvantages
- 2 Staff Salary Processing: C++ Solution
 - Non-Polymorphic Hierarchy
 - Advantages and Disadvantages
 - Polymorphic Hierarchy
 - Advantages and Disadvantages
 - Polymorphic Hierarchy (Flexible)
 - Advantages and Disadvantages
- 3 Module Summary

At the bottom of the slide, there is a footer with the text "Programming in Modern C++" and "Part 6: Polymorphic Design".

Welcome to programming in Modern C++. We are in week 6, and we are now going to discuss Module 30. In the last module we started designing for a staff salary problem, and worked out a C solution. We would like to continue on that design to refine it with repeated set of observations of what is working well in the design, and what is not to get to a good solution from a starting one in C that we have already made.

So, it is more a design journey, you are not going to learn anything new for the language, but you are going to learn very, very useful thing in terms of how to do the design, how to refine and what are the pros and cons of doing designs in a different, various different ways. So, I suggest that you go through this module repeatedly to understand the basic design principles in C++. This is the module outline available on your left always.

(Refer Slide Time: 01:55)

The slide is titled "Staff Salary Processing: C Solution" and features a blue header with a logo on the left and a small video feed of the presenter on the right. The main content is a large white area with the text "Staff Salary Processing: C Solution" in red at the bottom. The footer contains the text "Programming in Modern C++" and "Part 6: Polymorphic Design".

Staff Salary Processing: Problem Statement: RECAP (I)

- 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++ Partho Pratim Das MK16

C Solution: Engineer + Manager + Director: RECAP (Module 2)

- 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++ Partho Pratim Das MK17

C Solution: Advantages and Disadvantages: RECAP (Module 2)

- **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 (**ProcessSalary/Manager** 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++ Partho Pratim Das MK18

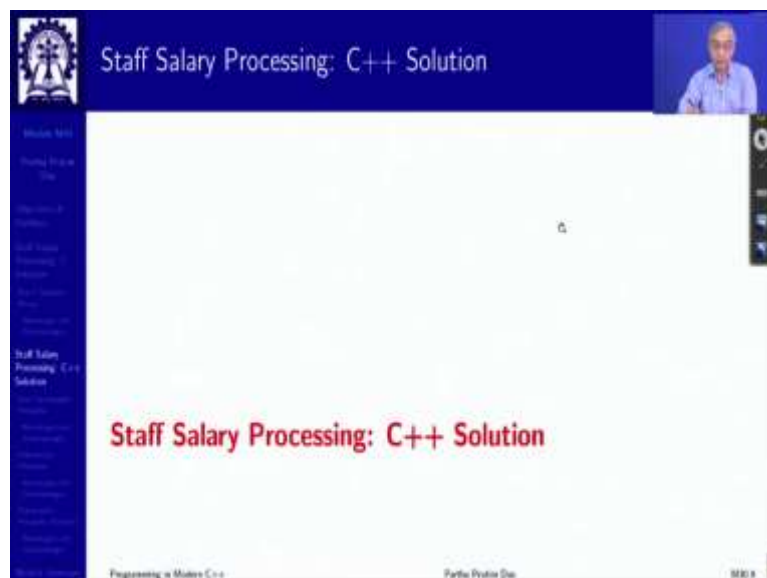
So, before I start on this let us just quickly recap on the C Solution. Here is a recap of the problem, the organization needs to process salary, it has engineering division with engineers and managers. They have different logic for their salary processing, in future directors may be added. The director's salary processing would also be different from all these, and further in future there may be new divisions and we need an extensible design.

So, based on this, we started by asking different questions of how to model the object, how to initialize, how to make collection of staff, what, how to represent the processing functions, how to invoke them and so on the common questions. And based on the answers, we made a design in C, and here is a summary of the advantages and disadvantages of the design.

Obviously, the main disadvantages are as we have noted, one is in terms of the objected design, there is no encapsulation duplication of fields and so on, and second is primarily in terms of management of different types of objects, which are different, but related in a strong way, an Engineer, Manager, Director all are related in a strong way. So, we needed to do an explicit management through an E_Type tag, and that makes the addition of new type of staff difficult that makes it error prone.

There is a function switch and all of that difficulties. We are not being able to use function pointers because of non-uniformity of processing function signature and so on. So, based on all this, the basic recommendation that we got ourselves is, we will use classes for encapsulation on a hierarchy. So, we are now going to get into that part of the design C++ Solutions based on the earlier advantages and disadvantages.

(Refer Slide Time: 04:09)



C++ Solution: Non-Polymorphic Hierarchy: Engineer + Manager



- How to represent Engineers and Managers?
 - Non-Polymorphic class hierarchy
- How to initialize objects?
 - Constructor / Destructor
- How to have a collection of mixed objects?
 - array of base class pointers
- How to model variations in salary processing algorithms?
 - Member functions
- How to invoke the correct algorithm for a correct employee type?
 - Function switch
 - Function pointers

C++ Solution: Non-Polymorphic Hierarchy: Engineer + Manager

```

#include <iostream>
#include <string>
using namespace std;

enum E_TYPE { Er, Mgr };

class Engineer {
protected:
    string name_; E_TYPE type_;
public:
    Engineer(const string& name, E_TYPE e = Er) : name_(name), type_(e) { }
    E_TYPE GetType() { return type_; }
    void ProcessSalary() { cout << name_ << ": Process Salary for Engineer" << endl; }
};

class Manager : public Engineer {
    Engineer *reports_[10];
public:
    Manager(const string& name, E_TYPE e = Mgr) : Engineer(name, e) { }
    void ProcessSalary() { cout << name_ << ": Process Salary for Manager" << endl; }
};
    
```

C++ Solution: Non-Polymorphic Hierarchy: Engineer + Manager

```

#include <iostream>
#include <string>
using namespace std;

enum E_TYPE { Er, Mgr };

class Engineer {
protected:
    string name_; E_TYPE type_;
public:
    Engineer(const string& name, E_TYPE e = Er) : name_(name), type_(e) { }
    E_TYPE GetType() { return type_; }
    void ProcessSalary() { cout << name_ << ": Process Salary for Engineer" << endl; }
};

class Manager : public Engineer {
    Engineer *reports_[10];
public:
    Manager(const string& name, E_TYPE e = Mgr) : Engineer(name, e) { }
    void ProcessSalary() { cout << name_ << ": Process Salary for Manager" << endl; }
};
    
```

```

#include <iostream>
#include <string>
using namespace std;

enum E_TYPE { Hr, Mgr };

class Engineer {
protected:
    string name; E_TYPE type;
public:
    Engineer(const string& name, E_TYPE e = Hr) : name(name), type(e) {}
    E_TYPE GetType() { return type; }
    void ProcessSalary() { cout << name << ": Process Salary for Engineer" << endl; }
};

class Manager : public Engineer {
public:
    Manager(const string& name, E_TYPE e = Mgr) : Engineer(name, e) {}
    void ProcessSalary() { cout << name << ": Process Salary for Manager" << endl; }
};

```

So, we asked the same set of questions the answers of course, are different. Now, we formally have represented the ISA relationship, every manager is an engineer we know. So, what we decide is to use non-polymorphic class hierarchy. There is a simplest form of inheritance hierarchy that you can have. Initialization, release, constructor, destructor obvious choice.

Now, we have an array of base class pointers, we do not need the union anymore, because there is a specialization. So, I can have a base class pointer of Engineer type which can represent either an engineer or a manager. Naturally salary processing moves to member function, but dispatch in terms of which processing function to call will still come remain under function switch. So, this is our design refinement criteria. So, based on this let us write the code.

Easy? We have the staff type here like the enum. We have now made the struct Engineer as a class. So, what we are able to do is, we will, we are putting the string name as well as the struct type, I am sorry employee type as fields in this protected data. Then we have the constructor, we have a method to get the employee type, because we will need to make decisions based on that, and then we have a ProcessSalary.

The same thing we do for managers, which is a specialization from engineer, so these data fields are not data members are no more required to be specified in the Manager they are inherited. So, refactoring of data is automatically happening through the inheritance. We have the Manager constructor, the processing salary, because that will be different, this is overridden in the Manager class and getting E_TYPE tie function I do not need to override because it is going to be the same for the Engineer as well as the Manager. So, certainly you can see that the design is getting simpler cleaner.

(Refer Slide Time: 06:52)

C++ Solution: Non-Polymorphic Hierarchy Engineer + Manager

```
int main() {
    Engineer e1("Rohit"), e2("Navita"), e3("Shamboo");
    Manager m1("Kamala"), m2("Rajib");
    Engineer *staff[] = { &e1, &e2, &e3, &m1, &m2 };
    for (int i = 0; i < sizeof(staff) / sizeof(Engineer*); ++i) {
        E_TYPE t = staff[i]->GetType();
        if (t == Enr)
            staff[i]->ProcessSalary();
        else if (t == Mgr)
            ((Manager *)staff[i])->ProcessSalary();
        else cout << "Invalid Staff Type" << endl;
    }
}
```

Rohit: Process Salary for Engineer
Kamala: Process Salary for Manager
Rajib: Process Salary for Manager
Navita: Process Salary for Engineer
Shamboo: Process Salary for Engineer

Progressing in Modern C++
Partha Prasad Das
MSB 12

C++ Solution: Non-Polymorphic Hierarchy Engineer + Manager

```
int main() {
    Engineer e1("Rohit"), e2("Navita"), e3("Shamboo");
    Manager m1("Kamala"), m2("Rajib");
    Engineer *staff[] = { &e1, &e2, &e3, &m1, &m2 };
    for (int i = 0; i < sizeof(staff) / sizeof(Engineer*); ++i) {
        E_TYPE t = staff[i]->GetType();
        if (t == Enr)
            staff[i]->ProcessSalary();
        else if (t == Mgr)
            ((Manager *)staff[i])->ProcessSalary();
        else cout << "Invalid Staff Type" << endl;
    }
}
```

Rohit: Process Salary for Engineer
Kamala: Process Salary for Manager
Rajib: Process Salary for Manager
Navita: Process Salary for Engineer
Shamboo: Process Salary for Engineer

Progressing in Modern C++
Partha Prasad Das
MSB 12

C++ Solution: Non-Polymorphic Hierarchy Engineer + Manager

```
int main() {
    Engineer e1("Rohit"), e2("Navita"), e3("Shamboo");
    Manager m1("Kamala"), m2("Rajib");
    Engineer *staff[] = { &e1, &e2, &e3, &m1, &m2 };
    for (int i = 0; i < sizeof(staff) / sizeof(Engineer*); ++i) {
        E_TYPE t = staff[i]->GetType();
        if (t == Enr)
            staff[i]->ProcessSalary();
        else if (t == Mgr)
            ((Manager *)staff[i])->ProcessSalary();
        else cout << "Invalid Staff Type" << endl;
    }
}
```

Rohit: Process Salary for Engineer
Kamala: Process Salary for Manager
Rajib: Process Salary for Manager
Navita: Process Salary for Engineer
Shamboo: Process Salary for Engineer

Progressing in Modern C++
Partha Prasad Das
MSB 12

**C++ Solution: Non-Polymorphic Hierarchy
Engineer + Manager**

```

int main() {
    Engineer e1("Rohit"), e2("Kavita"), e3("Shamhu");
    Manager m1("Kamala"), m2("Rajib");
    Engineer *staff[] = { &e1, &m1, &m2, &e2, &e3 };

    for (int i = 0; i < sizeof(staff) / sizeof(Engineer*); ++i) {
        E_TYPE t = staff[i]->GetType();
        if (t == Er)
            staff[i]->ProcessSalary();
        else if (t == Mgr)
            staff[i]->ProcessSalary();
        else cout << "Invalid Staff Type" << endl;
    }
}

```

class Cent!!!

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

**C++ Solution: Non-Polymorphic Hierarchy
Engineer + Manager**

```

int main() {
    Engineer e1("Rohit"), e2("Kavita"), e3("Shamhu");
    Manager m1("Kamala"), m2("Rajib");
    Engineer *staff[] = { &e1, &m1, &m2, &e2, &e3 };

    for (int i = 0; i < sizeof(staff) / sizeof(Engineer*); ++i) {
        E_TYPE t = staff[i]->GetType();
        if (t == Er)
            staff[i]->ProcessSalary();
        else if (t == Mgr)
            staff[i]->ProcessSalary();
        else cout << "Invalid Staff Type" << endl;
    }
}

```

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

With that, let us write the main function. These are the managers and engineers that I create. Finally, I need an aggregate the staff aggregate where we put the pointers of all these objects. It is an aggregate of engineers *, engineer pointer because that is a base class. So, for say &m1 or &m2, m1, m2 are manager object, so, here this will work to the upcast along that hierarchy.

So, with this array, we will be able to maintain all my processing all my, all the staff here. This is a for loop, instead of hardcoding we are using the size of this array to find out how many to be processed. Then what, how do I process? I have to know, which type of employee exist at a certain point in your staff[i], ith employ. So, on this pointer we do GetType.

Now GetType is there is only one function which is implemented at the root, so, it will get Employee type. In t like before I check if it is an engineer. If it is an engineer, we invoke

ProcessSalary, simple. The Engineer class has a ProcessSalary, so I invoke that function. If not, then we check if it is a manager.

Now, if it is a manager, then what do I need to do? I need to invoke the ProcessSalary for the manager class. But this pointer is of Engineer type. So, I need to change this pointer to Manager type. What am I doing? Can you identify what am I doing? I have a generalized class pointer Engineer and I am making that into Manager pointers, so I am actually doing a C style casting, which is a downcast, but it must be okay there because that is how I have set this tag.

So, if this tag is manager and if I cast it to Manager* it should be fine. So now, this pointer after casting, this pointer, the pointer after casting is a Manager type pointer. So, if I invoke, it will invoke the processing, ProcessSalary of the Manager class. So, it would be fine, otherwise, I do invalid. So, I made a substantial change and you can see, there are a lot of things that have got simplified, that have got structured organized, but still there are a lot to go.

(Refer Slide Time: 10:26)

**C++ Solution: Non-Polymorphic Hierarchy:
Engineer + Manager + Director**

```
graph LR; Director --> Manager; Manager --> Engineer;
```

- How to represent Engineers, Managers, and Directors?
 - Non-Polymorphic class hierarchy
- How to initialize objects?
 - Constructor / Destructor
- How to have a collection of mixed objects?
 - array of base class pointers
- How to model variations in salary processing algorithms?
 - Member functions
- How to invoke the correct algorithm for a correct employee type?
 - Function switch
 - Function pointers

Programming in Modern C++ | Partha Prasad Das | 10:26



C++ Solution: Non-Polymorphic Hierarchy Engineer + Manager + Director



```
#include <iostream>
#include <string>
using namespace std;
enum E_TYPE { En, Mgr, Dir };

class Engineer {
protected:
    string name_; E_TYPE type_;
public:
    Engineer(const string& name, E_TYPE e = En) : name_(name), type_(e) {}
    E_TYPE GetType() { return type_; }
    void ProcessSalary() { cout << name_ << ": Process Salary for Engineer" << endl; }
};

class Manager : public Engineer {
    Engineer *reports_[10];
public:
    Manager(const string& name, E_TYPE e = Mgr) : Engineer(name, e) {}
    void ProcessSalary() { cout << name_ << ": Process Salary for Manager" << endl; }
};

class Director : public Manager {
    Manager *reports_[10];
public:
    Director(const string& name) : Manager(name, Dir) {}
    void ProcessSalary() { cout << name_ << ": Process Salary for Director" << endl; }
};
```



C++ Solution: Non-Polymorphic Hierarchy Engineer + Manager + Director



```
#include <iostream>
#include <string>
using namespace std;
enum E_TYPE { En, Mgr, Dir };

class Engineer {
protected:
    string name_; E_TYPE type_;
public:
    Engineer(const string& name, E_TYPE e = En) : name_(name), type_(e) {}
    E_TYPE GetType() { return type_; }
    void ProcessSalary() { cout << name_ << ": Process Salary for Engineer" << endl; }
};

class Manager : public Engineer {
    Engineer *reports_[10];
public:
    Manager(const string& name, E_TYPE e = Mgr) : Engineer(name, e) {}
    void ProcessSalary() { cout << name_ << ": Process Salary for Manager" << endl; }
};

class Director : public Manager {
    Manager *reports_[10];
public:
    Director(const string& name) : Manager(name, Dir) {}
    void ProcessSalary() { cout << name_ << ": Process Salary for Director" << endl; }
};
```



C++ Solution: Non-Polymorphic Hierarchy Engineer + Manager + Director



```
#include <iostream>
#include <string>
using namespace std;
enum E_TYPE { En, Mgr, Dir };

class Engineer {
protected:
    string name_; E_TYPE type_;
public:
    Engineer(const string& name, E_TYPE e = En) : name_(name), type_(e) {}
    E_TYPE GetType() { return type_; }
    void ProcessSalary() { cout << name_ << ": Process Salary for Engineer" << endl; }
};

class Manager : public Engineer {
    Engineer *reports_[10];
public:
    Manager(const string& name, E_TYPE e = Mgr) : Engineer(name, e) {}
    void ProcessSalary() { cout << name_ << ": Process Salary for Manager" << endl; }
};

class Director : public Manager {
    Manager *reports_[10];
public:
    Director(const string& name) : Manager(name, Dir) {}
    void ProcessSalary() { cout << name_ << ": Process Salary for Director" << endl; }
};
```

**C++ Solution: Non-Polymorphic Hierarchy
Engineer + Manager + Director**

```

int main() {
    Engineer e1("Rohit"), e2("Kavita"), e3("Shambhu");
    Manager m1("Kamala"), m2("Rajib");
    Director d("Ranjana");
    Engineer *staff[] = { &e1, &m1, &m2, &e2, &e3, &d };

    for (int i = 0; i < sizeof(staff) / sizeof(Engineer*); ++i) {
        E_TYPE t = staff[i]->GetType();
        if (t == Ee)
            staff[i]->ProcessSalary();
        else if (t == Mgr)
            ((Manager *)staff[i])->ProcessSalary();
        else if (t == Dir)
            ((Director *)staff[i])->ProcessSalary();
        else cout << "Invalid Staff Type" << endl;
    }
}

```

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++ Part 6: Polymorphic Design 58/15

Now, before discussing that, let us try to add the Director type here. So, if we add the Director type, really none of these questions will have a different answer. So, we will just have to go and do it in the code. So, let us do it.

I need to add the tag type, nothing changes in the Engineer and the Manager, but I just specialize from the Manager to create a Director have its constructor, have its ProcessSalary implement. So, the addition has still needs care, but that has become substantially less now. Let us look at the situation in the main function.

In the main function, we still collect all these employees in an array of pointers, do everything else in the same way, and we have a type switch, now I have to add this part, if t is Director then now, the cast will be again another downcast. I am doing one risky thing after the other. I have to cast it to Director* because now I have a director and invoke the ProcessSalary for a Director object, different from the Engineering and Manager, but it is not yet totally clean, but certainly it is cleaner than what we had earlier.

(Refer Slide Time: 12:20)

C++ Solution: Non-Polymorphic Hierarchy: Advantages and Disadvantages

- **Advantages**
 - Data is encapsulated
 - Hierarchy factors common data members
 - Constructor / Destructor to manage lifetime
 - struct-specific functions made member function (overridden)
 - E.Type subsumed in class - no need for union
 - Code reuse evidenced
- **Disadvantages**
 - 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`
 - Application code need to have a new case (`if-else`) based on the new type
 - ▷ Error prone because the application programmer has to cast to right type to call `ProcessSalary`
- **Recommendation**
 - Use a polymorphic hierarchy with dynamic dispatch

Programming in Modern C++ Part 8: From 100

C++ Solution: Non-Polymorphic Hierarchy: Advantages and Disadvantages

- **Advantages**
 - Data is encapsulated
 - Hierarchy factors common data members
 - Constructor / Destructor to manage lifetime
 - struct-specific functions made member function (overridden)
 - E.Type subsumed in class - no need for union
 - Code reuse evidenced
- **Disadvantages**
 - 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`
 - Application code need to have a new case (`if-else`) based on the new type
 - ▷ Error prone because the application programmer has to cast to right type to call `ProcessSalary`
- **Recommendation**
 - Use a polymorphic hierarchy with dynamic dispatch

Programming in Modern C++ Part 8: From 100

So, let us take a stock. What all advantages that we got by doing this change? First is data is encapsulated. That is a first object-oriented principle is satisfied, data is encapsulated. Hierarchy has factored the common data members like the Employee type and the name of the employee and so on. On whatever common data members are there between the employees like their employee code, address, date of birth, date of joining, all of that will be there will similarly get factored into the base class.

Constructor destructors to manage lifetime which is a clean solution in C++ we know. struct type this struct-specific member functions that we had they have been struct-specific functions that we have they have been made into member functions. So, what is the big advantage in that? The big advantage in that is earlier all these functions were having

different names SalaryProcessingEngineer, SalaryProcessingManager, SalaryProcessingDirector.

Now all of them are salary processing. Depending on which class they belong to as a member, they have different implementation they are overridden. So, now we have engineer::salary processing, ProcessSalary, Manager::ProcessSalary, Director::ProcessSalary and so on that becomes a lot more nice and uniform in terms of the design through this overriding mechanism.

I do not need the union anymore because what I am replacing the union with one is, E_TYPE is subsumed in the class, it is a part of the class itself, so it is inside. So, the class itself remembers. And what am I substituting union for? I am substituting union by upcast. I have an array of pointers where the upcast combines everything in terms of base pointer, base class pointer, but it still can dig into that object and get the E_TYPE to decide which type of object it is.

So, the code reuse, the amount of code reuse has substantially increased, it is directly evidenced. But it still continues with the disadvantage the type of the object is still explicitly managed by E_TYPE. I am putting an E_TYPE when I had to add the Director, I had to add a new type to the enum, I had to add a specific case, I had to do a case-specific downcast, very risky, if I make any mistake that is the application programmer has cast to right type of call right type to call the process salary.

So, all risky things, lot of risky things are still remaining. So, what is the recommendation we gained? We have gotten this advantages by the mere use of C++ classes, and by the use of hierarchy, but we have so far kept that hierarchy non-polymorphic. So, let us make it a polymorphic hierarchy with dynamic dispatch, and see what advantages we gain over the design that we have done so, far.

(Refer Slide Time: 16:09)

C++ Solution: Polymorphic Hierarchy Engineer + Manager + Director

```
classDiagram
    class Director
    class Manager
    class Engineer
    Director --|> Manager
    Manager --|> Engineer
```

- How to represent **Engineers, Managers, and Directors**?
 - Polymorphic class hierarchy
- How to initialize objects?
 - Constructor / Destructor
- How to have a collection of mixed objects?
 - array of base class pointers
- How to model variations in salary processing algorithms?
 - Member functions
- How to invoke the correct algorithm for a correct employee type?
 - Virtual Functions

Programming in Modern C++ Partha Pratim Das 16:17

C++ Solution: Polymorphic Hierarchy Engineer + Manager + Director

```
#include <iostream>
#include <string>
using namespace std;

class Engineer {
protected:
    string name_;
public:
    Engineer(const string& name) : name_(name) {}
    virtual void ProcessSalary() { cout << name_ << ": Process Salary for Engineer" << endl; }
};

class Manager : public Engineer {
    Engineer *reports_[10];
public:
    Manager(const string& name) : Engineer(name) {}
    void ProcessSalary() { cout << name_ << ": Process Salary for Manager" << endl; }
};

class Director : public Manager {
    Manager *reports_[10];
public:
    Director(const string& name) : Manager(name) {}
    void ProcessSalary() { cout << name_ << ": Process Salary for Director" << endl; }
};
```

Programming in Modern C++ Partha Pratim Das 16:18

C++ Solution: Polymorphic Hierarchy Engineer + Manager + Director

```
int main() {
    Engineer e1("Rohit"), e2("Kavita"), e3("Shamika");
    Manager m1("Kamala"), m2("Rajib");
    Director d("Ranjana");
    Engineer *staff[] = { &e1, &m1, &m2, &e2, &e3, &d };

    for (int i = 0; i < sizeof(staff) / sizeof(Engineer); ++i)
        staff[i]->ProcessSalary();
}
```

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

Programming in Modern C++ Partha Pratim Das 16:18

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

```

int main() {
    Engineer e1("Rohit"), e2("Kavita"), e3("Shambhu");
    Manager m1("Kamala"), m2("Rajib");
    Director d("Ranjana");
    Engineer *staff[] = { &e1, &m1, &m2, &e2, &e3, &d };

    for (int i = 0; i < sizeof(staff) / sizeof(Engineer); ++i)
        staff[i]->ProcessSalary();
}

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

```

So, what we have in terms of the hierarchies are same, the change is in the first answer to the first question, how to represent these objects we say that polymorphic class hierarchy. Rest of this is same, array of base class pointer is same, member function is same. As I do polymorphic class hierarchy naturally my dispatch mechanism or the way to call object-specific processing function now do not need a function switch, it can use function pointers in the form of virtual functions.

So, these are the two main differences after the analysis of disadvantages that we have seen and we are going to try this in the next iteration of the design, next refinement of the design. So, now, this entire E_TYPE staff is gone, we have the hierarchy, Manager is an Engineer, Director is a Manager, every constructors, processing functions everything. The polymorphism comes through making the process salary virtual in the base class root class engineer. So, as such the class part of the design is more or less similar except for the E_TYPE is erased and this is ProcessSalary function member function is virtualized.

Let us look at the main goal. I still need the collection to be built, of course, the for loop this does not change, but look at this. What has happened to that type switch if t is equal to Er do this, if t is equal to Mgr do this, if t is equal to Dir do this, you have 20 types long chain or switch all that is replaced by only one call. How? Because, now we have a polymorphic hierarchy where ProcessSalary is a virtual function in the base class Engineer of which the pointer type I have got.

So, this pointer has the current staff i pointer has a static type which is Engineer, but its dynamic type will keep on changing at it traverses. For the first object it is Engineer, so it will invoke the ProcessSalary for Engineer, for the second object it is Manager, so it will invoke

the ProcessSalary for Manager, third, again Manager fourth, again Engineer, fifth Engineer, sixth is Director. So, it will invoke ProcessSalary for the Director. So, you can see how drastically things have got simplified, compacted and the kind of there is a step jump in terms of the quality of design that we have got with this refinement.

(Refer Slide Time: 19:45)

C++ Solution: Polymorphic Hierarchy: Advantages and Disadvantages

- **Advantages**
 - Data is fully encapsulated
 - Polymorphic Hierarchy removes the need for explicit `E_Type`
 - Application code is independent of types in the system (virtual functions manage types through polymorphic dispatch)
 - High Code reuse – code is short and simple
- **Disadvantages**
 - Difficult to add an employee type that is not a part of this hierarchy (for example, employees of Sales Division)
- **Recommendation**
 - Use an abstract base class for employees

Progressing to Modern C++ Partha Pratim Das 19:45

C++ Solution: Polymorphic Hierarchy: Advantages and Disadvantages

- **Advantages**
 - Data is fully encapsulated
 - Polymorphic Hierarchy removes the need for explicit `E_Type`
 - Application code is independent of types in the system (virtual functions manage types through polymorphic dispatch)
 - High Code reuse – code is short and simple
- **Disadvantages**
 - Difficult to add an employee type that is not a part of this hierarchy (for example, employees of Sales Division)
- **Recommendation**
 - Use an abstract base class for employees

Progressing to Modern C++ Partha Pratim Das 19:45

We have to assess ourselves for further refinement, naturally in terms of advantages. What gets added at two things one is a couple of things. One is polymorphic hierarchy, which is removed the explicit use of `E_TYPE`, application code for independent type are not needed virtual function takes care of it, substantial code reuse and code shortening.

Does it solve all the problems? Unfortunately, no. You will still have to remember that as soon as a, you want to add a new hierarchy, new Employee type that is not a part of this

hierarchy, you will have to do a lot of things. So, what is that we are missing out? If you look into the entire hierarchy, the entire hierarchy is of concrete classes, which says that everything that we have is what exists but, that does not keep a scope for the future where we open the gates for whatever can come in future for that we need an abstraction. And for that, we need an abstract base class kind of an employee at the root, so that we can anytime go and add classes wherever we need them to be added. Let me explain this a little bit more with the diagram.

(Refer Slide Time: 21:36)

C++ Solution: Polymorphic Hierarchy (Flexible)
Engineer + Manager + Director + Others

```

classDiagram
    class Director
    class Manager
    class Engineer
    class SalesExecutive
    class Employee
    Director --|> Manager
    Manager --|> Engineer
    SalesExecutive --|> Employee
    Employee --|> Engineer
  
```

- How to represent Engineers, Managers, Directors, etc.?
 - Polymorphic class hierarchy with an Abstract Base Employee
- How to initialize objects?
 - Constructor / Destructor
- How to have a collection of mixed objects?
 - array of base class pointers
- How to model variations in salary processing algorithms?
 - Member functions
- How to invoke the correct algorithm for a correct employee type?
 - Virtual Functions (Pure in Employee)

Programming in Modern C++ Partha Pratim Das MBZ

C++ Solution: Polymorphic Hierarchy (Flexible)
Engineer + Manager + Director + Others

```

#include <iostream>
#include <string>
using namespace std;
class Employee {
protected: string name_;
public:
    virtual void ProcessSalary() = 0;
    virtual "Employee()" {}
};
class Engineer: public Employee { public:
    Engineer(const string& name) { name_ = name; }
    void ProcessSalary() { cout << name_ << ": Process Salary for Engineer" << endl; }
};
class Manager : public Engineer { Engineer *reports_[10]; public:
    Manager(const string& name) : Engineer(name) {}
    void ProcessSalary() { cout << name_ << ": Process Salary for Manager" << endl; }
};
class Director : public Manager { Manager *reports_[10]; public:
    Director(const string& name) : Manager(name) {}
    void ProcessSalary() { cout << name_ << ": Process Salary for Director" << endl; }
};
class SalesExecutive : public Employee { public:
    SalesExecutive(const string& name) { name_ = name; }
    void ProcessSalary() { cout << name_ << ": Process Salary for Sales Executive" << endl; }
};
  
```

Programming in Modern C++ Partha Pratim Das MBZ

```

#include <iostream>
#include <string>
using namespace std;
class Employee {
protected: string name_;
public:
    virtual void ProcessSalary() = 0;
    virtual ~Employee() {}
};
class Engineer: public Employee { public:
    Engineer(const string& name) { name_ = name; }
    void ProcessSalary() { cout << name_ << ": Process Salary for Engineer" << endl; }
};
class Manager : public Engineer { Engineer *reports_[10]; public:
    Manager(const string& name) : Engineer(name) {}
    void ProcessSalary() { cout << name_ << ": Process Salary for Manager" << endl; }
};
class Director : public Manager { Manager *reports_[10]; public:
    Director(const string& name) : Manager(name) {}
    void ProcessSalary() { cout << name_ << ": Process Salary for Director" << endl; }
};
class SalesExecutive: public Employee { public:
    SalesExecutive(const string& name) { name_ = name; }
    void ProcessSalary() { cout << name_ << ": Process Salary for Sales Executive" << endl; }
};

```

```

int main() {
    Engineer e1("Rohit"), e2("Kavita"), e3("Shamika");
    Manager m1("Kamala"), m2("Rajib");
    SalesExecutive s1("Hari"), s2("Rishan");
    Director d("Ranjana");

    Employee *staff[] = { e1, e2, e3, m1, m2, s1, s2, d };
    for (int i = 0; i < sizeof(staff) / sizeof(Employee); ++i)
        staff[i]->ProcessSalary();
}

```

Rohit: Process Salary for Engineer
Kamala: Process Salary for Manager
Rajib: Process Salary for Manager
Kavita: Process Salary for Engineer
Hari: Process Salary for Sales Executive
Shamika: Process Salary for Engineer
Ranjana: Process Salary for Director
Rishan: Process Salary for Sales Executive

So, earlier we had only this part. Now, we add an abstract base class and SalesExecutive which you want to add is naturally not on the engineering cadar, so, it goes in a different way. So, what changes to our answer is polymorphic class hierarchy with an abstract base class Employee which was not there. The moment you have abstract base class or the moment you have a class, which can be extended in any way at the root you do not know how to process the salary of that employee because it is just a concept, it is not a physical employee, it is a concept.

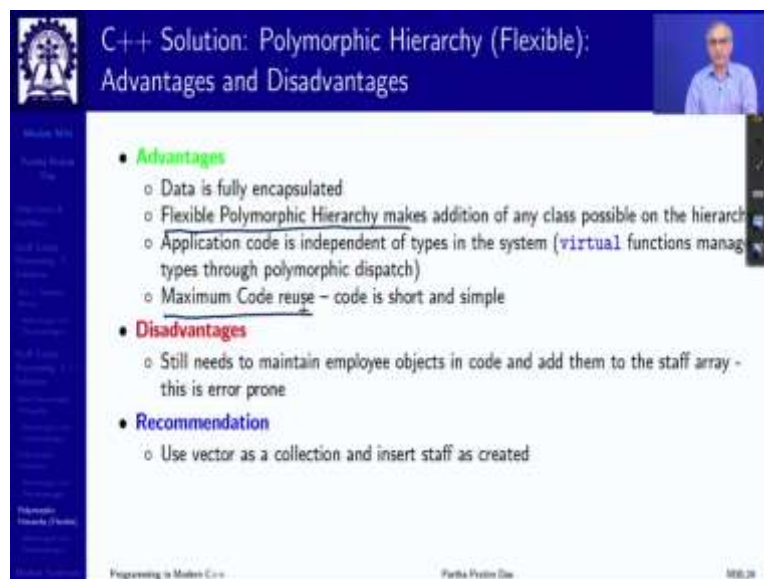
So, which means that your processing function will need to become purely virtual at the employee. Everything else remains the same, we are just opening up the polymorphic hierarchy for easy extension and additions in future.

So, we have this base class created, employee. We continue to refactor, we know, whatever the employee is, the employee is going to have name, address, date of birth, date of joining and all that we can put all of that refracted into the base class. This is the key point that we provide the ProcessSalary here as a pure virtual function. Because we do not know what is the logic of processing the salary of this employee who by himself is an abstract concept herself is an abstract concept. Keeping to our learning from the last module, we make the destructor of this employee class virtual so that we do not have any slicing. So, this is your base class design.

Then the rest of it is extremely modular in the way it goes along the along the hierarchy that you have. So, Engineer is an Employee, Manager is an Engineer, Director is a Manager, SalesExecutive is an Employee, and you have constructor, you have ProcessSalary overridden in every case to be able to process these different cases.

The processing code it has, it now has seamlessly been able to add a SalesExecutive and note, very importantly, this code does not need to change, no change in that code. Your hierarchy has extended and you are making this call from the abstract base class pointer Employee which will get dispatched to either an Engineer or a Manager or a SalesExecutive or a Director, as the case would be, and you do not need any changes there, beautiful, is it not.

(Refer Slide Time: 25:19)



C++ Solution: Polymorphic Hierarchy (Flexible): Advantages and Disadvantages

- **Advantages**
 - Data is fully encapsulated
 - Flexible Polymorphic Hierarchy makes addition of any class possible on the hierarchy
 - Application code is independent of types in the system (virtual functions manage types through polymorphic dispatch)
 - Maximum Code reuse – code is short and simple
- **Disadvantages**
 - Still needs to maintain employee objects in code and add them to the staff array - this is error prone
- **Recommendation**
 - Use vector as a collection and insert staff as created

Programming in Modern C++ Partha Pratim Das 988.28

C++ Solution: Polymorphic Hierarchy (Flexible)
Engineer + Manager + Director + Others

```

int main() {
    Engineer e1("Rohit"), e2("Kavita"), e3("Shamhu");
    Manager m1("Kamala"), m2("Ajith");
    SalesExecutive se1("Hari"), se2("Rishu");
    Director d1("Ranjana");

    Employee *staff[] = { &e1, &m1, &m2, &se1, &se2, &d1, &d2 };

    for (int i = 0; i < sizeof(staff) / sizeof(Employee*); ++i)
        staff[i]->ProcessSalary();
}

```

Rohit: Process Salary for Engineer
Kamala: Process Salary for Manager
Ajith: Process Salary for Manager
Kavita: Process Salary for Engineer
Hari: Process Salary for Sales Executive
Shamhu: Process Salary for Engineer
Ranjana: Process Salary for Director
Rishu: Process Salary for Sales Executive

Progressing to Modern C++ Partha Pratim Das 18B.23

Assessment, again, there is always scope for refinement possibly. So, what we see is advantages remain the same, we have added flexibility by adding this abstract base class. We have almost a maximum code reuse, no code is being duplicated, replicated anywhere, but we still have a disadvantage. The disadvantage is, we still need to maintain the aggregate of the employees through an array. If I can just go back to that previous, yes, this is what I mean. I have to maintain this array.

So, if there is an object instance, Employee instance created, which is not added to this array will have problem. If an instance is added twice you can understand what will happen, two salaries will be, salaries will be paid to the same employee. So, this remains a pretty vulnerable part of the design. And when you have thousands of employees would you think of doing this hard coding in your code. Every time an employee joins he will go and change the code, and add that, not possible. So, this needs to be cleaned up, that is basic.

(Refer Slide Time: 26:51)

C++ Solution: Polymorphic Hierarchy (Flexible): Advantages and Disadvantages

- **Advantages**
 - Data is fully encapsulated
 - Flexible Polymorphic Hierarchy makes addition of any class possible on the hierarchy
 - Application code is independent of types in the system (virtual functions manage types through polymorphic dispatch)
 - Maximum Code reuse – code is short and simple
- **Disadvantages**
 - Still needs to maintain employee objects in code and add them to the staff array - this is error prone
- **Recommendation**
 - Use vector as a collection and insert staff as created

C++ Solution: Polymorphic Hierarchy (Flexible) Engineer + Manager + Director + Others

```
#include <iostream>
#include <string>
#include <vector>
using namespace std;
class Employee { protected: string name; // Base of the employee
vector<Employee*> reports; // Collection of reportees aggregated
public: virtual void ProcessSalary() = 0; // Processing salary
virtual ~Employee() {}
static vector<Employee*> staffs; // Collection of all staffs
void AddStaff(Employee* e) { staffs.push_back(e); }; // Add a staff to collection
};
class Engineer : public Employee { public:
Engineer(const string& name) { name_ = name; // Why init like name_(name) won't work?
AddStaff(this); } // Add the staff
void ProcessSalary() { cout << name_ << ": Process Salary for Engineer" << endl; }
};
class Manager : public Engineer { public: Manager(const string& name) : Engineer(name) {}
void ProcessSalary() { cout << name_ << ": Process Salary for Manager" << endl; }
};
class Director : public Manager { public: Director(const string& name) : Manager(name) {}
void ProcessSalary() { cout << name_ << ": Process Salary for Director" << endl; }
};
class SalesExecutive : public Employee { public:
SalesExecutive(const string& name) { name_ = name; AddStaff(this); } // Add the staff
void ProcessSalary() { cout << name_ << ": Process Salary for Sales Executive" << endl; }
};
```

C++ Solution: Polymorphic Hierarchy (Flexible) Engineer + Manager + Director + Others

```
#include <iostream>
#include <string>
#include <vector>
using namespace std;
class Employee { protected: string name; // Base of the employee
vector<Employee*> reports; // Collection of reportees aggregated
public: virtual void ProcessSalary() = 0; // Processing salary
virtual ~Employee() {}
static vector<Employee*> staffs; // Collection of all staffs
void AddStaff(Employee* e) { staffs.push_back(e); }; // Add a staff to collection
};
class Engineer : public Employee { public:
Engineer(const string& name) { name_ = name; // Why init like name_(name) won't work?
AddStaff(this); } // Add the staff
void ProcessSalary() { cout << name_ << ": Process Salary for Engineer" << endl; }
};
class Manager : public Engineer { public: Manager(const string& name) : Engineer(name) {}
void ProcessSalary() { cout << name_ << ": Process Salary for Manager" << endl; }
};
class Director : public Manager { public: Director(const string& name) : Manager(name) {}
void ProcessSalary() { cout << name_ << ": Process Salary for Director" << endl; }
};
class SalesExecutive : public Employee { public:
SalesExecutive(const string& name) { name_ = name; AddStaff(this); } // Add the staff
void ProcessSalary() { cout << name_ << ": Process Salary for Sales Executive" << endl; }
};
```

```

#include <iostream>
#include <string>
#include <vector>
using namespace std;
class Employee { protected: string name_; // Name of the employee
                  vector<Employee*> reports_; // Collection of reports aggregated
public: virtual void ProcessSalary() = 0; // Processing salary
        virtual ~Employee() { }
        static vector<Employee*> staffs; // Collection of all staffs
        void AddStaff(Employee* e) { staffs.push_back(e); } // Add a staff to collection
};
class Engineer : public Employee { public:
    Engineer(const string& name) : Employee(name) {
        name_ = name; // Why init it? name_ (name) won't work?
        AddStaff(this); // Add the staff
    }
    void ProcessSalary() { cout << name_ << ": Process Salary for Engineer" << endl; }
};
class Manager : public Engineer { public: Manager(const string& name) : Engineer(name) { }
    void ProcessSalary() { cout << name_ << ": Process Salary for Manager" << endl; }
};
class Director : public Manager { public: Director(const string& name) : Manager(name) { }
    void ProcessSalary() { cout << name_ << ": Process Salary for Director" << endl; }
};
class SalesExecutive : public Employee { public:
    SalesExecutive(const string& name) { name_ = name; AddStaff(this); } // Add the staff
    void ProcessSalary() { cout << name_ << ": Process Salary for Sales Executive" << endl; }
};

```

```

vector<Employee*> Employee::staffs; // Collection of all staffs

int main() {
    Engineer e1("Rohit"), e2("Kavita"), e3("Shambhu");
    Manager m1("Kamala"), m2("Rajib");
    SalesExecutive s1("Hari"), s2("Bishnu");
    Director d1("Ranjana");

    vector<Employee*>::const_iterator it; // Iterator over staffs
    for (it = Employee::staffs.begin();
         it < Employee::staffs.end();
         ++it)
        (*it)->ProcessSalary(); // Process respective salary
}

Rohit: Process Salary for Engineer
Kavita: Process Salary for Engineer
Shambhu: Process Salary for Engineer
Kamala: Process Salary for Manager
Rajib: Process Salary for Manager
Hari: Process Salary for Sales Executive
Bishnu: Process Salary for Sales Executive
Ranjana: Process Salary for Director

```

So, what he says let us do this. The in it, let us look at it in a little different way. We are saying that these are the objects created and then the application is adding them. Why? When the object is getting created, it could add itself to the collection. It could add itself to the collection. If I need to do that, I need an active collection. I need just not an array, but a data structure where I can just invoke a method and get myself added collected.

So, my recommendation our recommendation is we will use vector as a collection and insert staff as they are created to get rid of this disadvantage, refine design. So, vector included, now in the abstract base class we introduce a collection, we make collection we move that from the application code to the class hierarchy code and make vector of Employee pointers a member in the Employee class.

Obviously, this collection will be unique one, therefore, this is static. This is next, I need a way to add this. So, we provide a method `AddStaff`, which takes this vector and pushes the current employee pointer that will add the staff to the collection. So now what changes is when I am doing class `Engineer`, which is an `Employee`, I need to `AddStaff`. And this because in the constructor it is now constructed, so I add myself to this staff collection.

Similarly, I do that for class `executive AddStaff`. Remember this is where, remember the structure, this is your employee, this is your `Engineer`, this is your `SalesExecutive`, this is your `Manager`, this is your `Director`. So, this `AddStaff` is needed in these classes, which are after the abstract base class, which are the root of the concrete classes. Naturally you do not need it here.

Because to construct a `Manager` you will obviously construct an `Engineer`, and object as a base and `AddStaff` will happen. So, with this, I have the advantage that I will not need to manage the collection anymore. All that will happen automatically as an when objects are created, they will get added to this collection. So, all that I need to do is go over this collection.

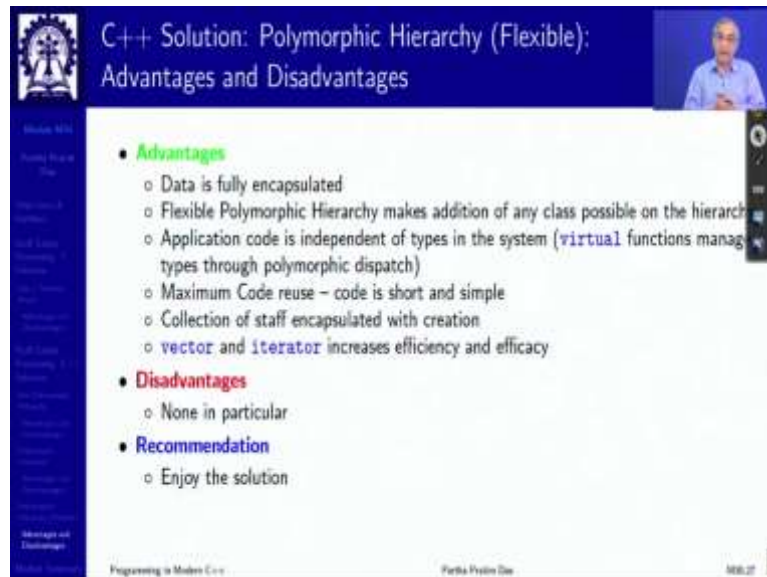
There is a small nuance which I would like to mention here. For example, here in creating the name I have done it in the body not in the initialize list. Normally we have said that always do it in the initializer list like this. Please note, that that is not going to work in this case. The reason is simple, is name is inherited from the employee class, which is abstract base class.

So, `Engineer` does not have directly a name field, it will come from that abstract base class, but being an abstract base class it will never get constructed. Being an abstract base class it will never get constructed, so `Engineer` cannot, `engineer`, constructor cannot refer to that. So, we have to go through an empty initializer here get the default object constructed and then actually set this name field. So, this is a small understanding in terms of the object layout that you need to keep.

So, having done this, now we have this global vector. So, which is `Employee::staffs` so I need to define that in the global space I do that because it is a static. And I am just now I am just creating the objects, I am no more having to add them, they are getting automatically added. Then finally, I have to go over this vector so what I do is the standard vector mechanism I will construct a constant iterator. Why constant, because I do not expect to change any employee object just process that, so and do a start to go up to begin to plus plus and on every item I will come `ProcessSalary`.

Absolutely compacted. Now it does not depend this part of the main application function does not depend on the classes, instances or anything, they are just generic, and I can freely keep on adding classes as I want and I have everything that I actually looking for.

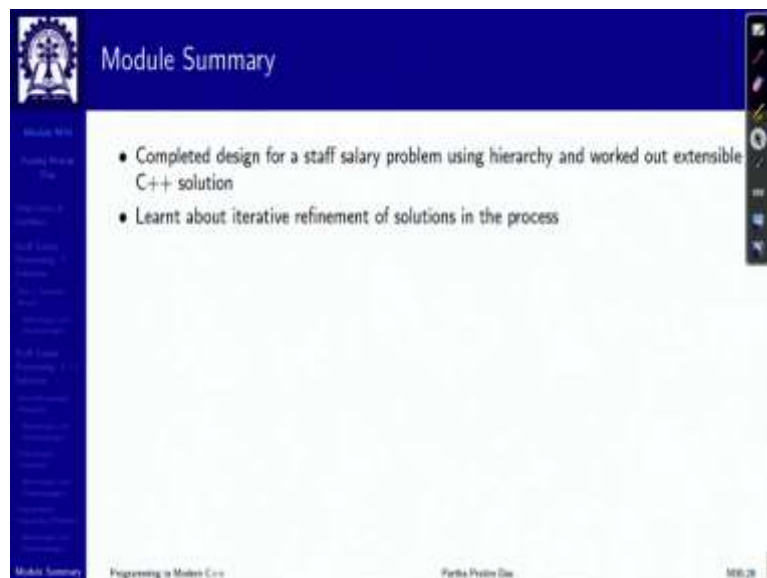
(Refer Slide Time: 32:43)



C++ Solution: Polymorphic Hierarchy (Flexible): Advantages and Disadvantages

- **Advantages**
 - Data is fully encapsulated
 - Flexible Polymorphic Hierarchy makes addition of any class possible on the hierarchy
 - Application code is independent of types in the system (virtual functions managing all polymorphic dispatch)
 - Maximum Code reuse – code is short and simple
 - Collection of staff encapsulated with creation
 - `vector` and `iterator` increases efficiency and efficacy
- **Disadvantages**
 - None in particular
- **Recommendation**
 - Enjoy the solution

Programming in Modern C++ | Part 8: Polymorphic Hierarchy | 32:43



Module Summary

- Completed design for a staff salary problem using hierarchy and worked out extensible C++ solution
- Learnt about iterative refinement of solutions in the process

Programming in Modern C++ | Part 8: Polymorphic Hierarchy | 32:43

Data is fully encapsulated flexible polymorphic hierarchy where free addition of any class is possible the any stage of the hierarchy, application code is independent of types in the system, virtual functions, managing all polymorphic dispatch, maximum code reuse short simple, collection of staff encapsulated, flexible polymorphic hierarchy where free addition of any class is possible the any stage of the hierarchy.

Application code is independent of types in the system, virtual functions, managing all polymorphic dispatch, maximum code reuse, short simple, collection of staff encapsulated for

creation, vector and iteration increases the efficiency and efficacy of the solution, no particular disadvantage as such so we can enjoy the solution.

So, this was to summarize. This is an exercise, I mean, I did not want to just ask the problem and give you the solution. But we made this work through, to make you realize that you cannot, given a problem you cannot jump to the best solution in one stage, it is never possible. You will have to go through stages of refinement.

Now, some of you maybe, grasping it well so that you start with the non-polymorphic hierarchy as a first design, skipping the C part structure-based design, some of you maybe, even more mature to start with the polymorphic hierarchy, it depends.

There could be other ways of doing that, but what we will have to remember is you always need to start at an initial solution and ask design questions repeatedly to see how it can be improved, keep on doing this till you meet all your design objectives.

Thank you, very much. With this I conclude this module. Thank you for your attention, and we also conclude the discussions for this week which was being primarily your exposure to polymorphism both in terms of the language features as well as the design practices which will be the core of take back from the objected-oriented side of design in C++. Thank you very much, see you again, in the next module.