

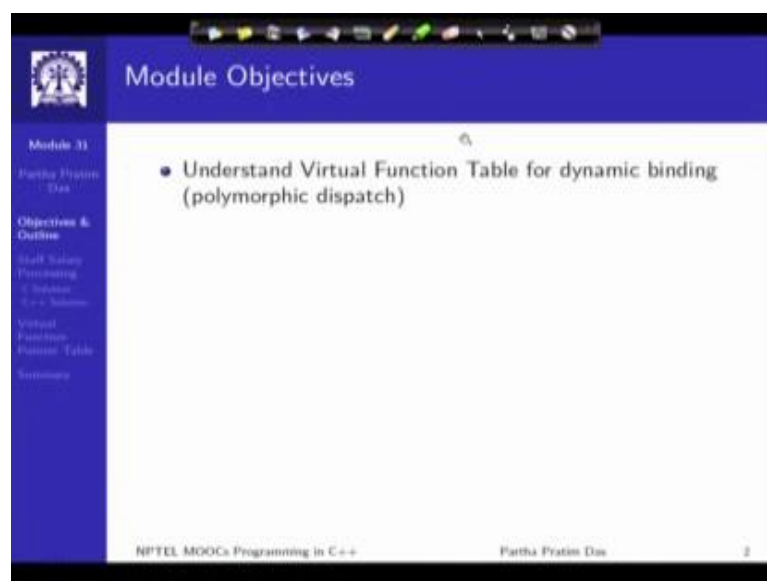
Programming in C++
Prof. Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture - 46
Virtual Function Table

Welcome to Module 31 of Programming in C++. In the last couple of modules we have discussed about Dynamic Binding, the Polymorphism, we have discussed what is polymorphic type and particularly the major new feature that we have learnt is about polymorphic dispatch.

It is a mechanism through which, when I call a member function of a class belonging to a polymorphic hierarchy. When I make the call through a pointer or through the reference of the base class type then the actual function invoked depends not on the type of the pointer or the reference, but it actually depends on the current object being pointed to or being refer to. It depends on the runtime and this mechanism is known as Polymorphism or this method of binding is known as Dynamic Binding as we have seen, and this whole thing is known as Polymorphic Dispatch in C++.

(Refer Slide Time: 01:37)

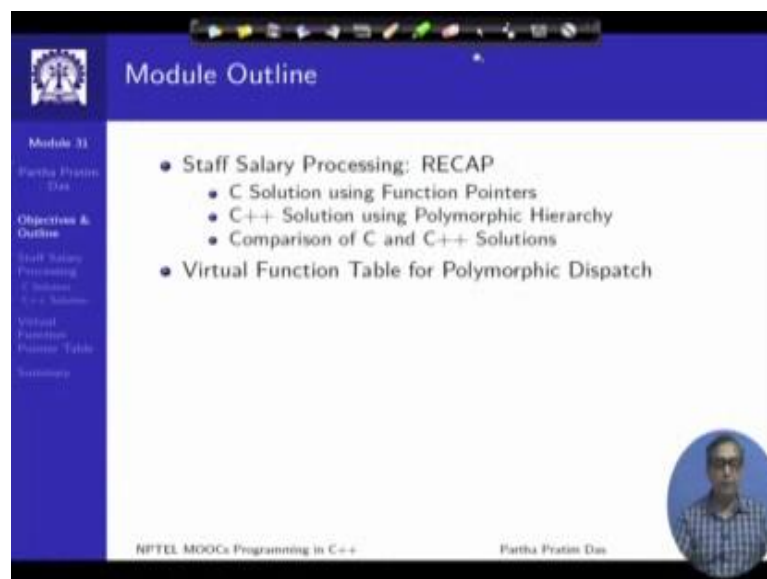


The image shows a presentation slide with a blue header and a white main content area. The header contains the text 'Module Objectives' and a small logo on the left. The main content area has a bulleted list with one item: 'Understand Virtual Function Table for dynamic binding (polymorphic dispatch)'. On the left side of the slide, there is a vertical navigation menu with the following items: 'Module 31', 'Partha Pratim Das', 'Objective & Outline', 'Self Study Programming in C++', 'Virtual Function Table', and 'Summary'. At the bottom of the slide, there is a footer with the text 'NPTEL MOOCs Programming in C++', 'Partha Pratim Das', and the number '2'.

We have looked at polymorphic dispatch mechanism and we have used it extensively in designing a solution of staff salary processing. In the module today, we will discuss about Virtual Function Table that is we would like to take a little bit of understanding in terms of how is this polymeric dispatched; actually implemented by the compiler. Because you will have to recall that the compiler works at the static time when the code is being process the source is being processed at that time there is no execution that is happening.

So, compiler has no way of knowing as to at runtime given a pointer which is the type of object that it will actually point too, but still it has to generate a code that will work according to the dynamic type during the execution.

(Refer Slide Time: 02:44)



This is done through the use of virtual function table, which is key for the polymeric dispatch. For this we will take a quick recap of the salary processing application have we been discussing, we will introduce a new solution in C and show how that gives us the insight into understanding the virtual function table for the dispatch.

(Refer Slide Time: 03:06)

Staff Salary Processing: Problem Statement: RECAP (Module 29)

- 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 and expand the workforce
- **Make a suitable extensible design**

NPTEL MOOCs Programming in C++ Partha Pratim Das

So this was the problem, we had assume that there are different types employees for whom a salary needs to be processed and their salary processing algorithms are different. And the crux of the whole design is the fact that the design needs to be appropriately extensible it should be possible to extent add new classes to the hierarchy as and when we want to do that.

(Refer Slide Time: 03:32)

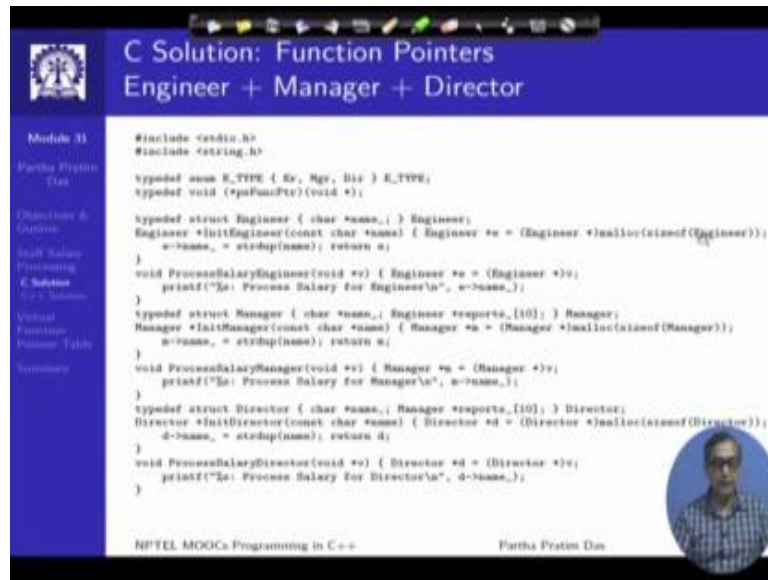
C Solution: Function Pointers Engineer + Manager: RECAP (Module 29)

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

NPTEL MOOCs Programming in C++ Partha Pratim Das

In C, we had taken these considerations and you can always refer to the earlier module, refer them have a module number also and check up on what are design considerations.

(Refer Slide Time: 03:48)



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

typedef union E_TYPE { E*, M*, D* } E_TYPE;
typedef void (*pFuncPtr)(void *);

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(void *e) { Engineer *e = (Engineer *)e;
printf("E: Process Salary for Engineer\n", e->name);
}
typedef struct Manager { char *name; Engineer *reports; } Manager;
Manager *InitManager(const char *name) { Manager *m = (Manager *)malloc(sizeof(Manager));
m->name = strdup(name); return m;
}
void ProcessSalaryManager(void *m) { Manager *m = (Manager *)m;
printf("M: Process Salary for Manager\n", m->name);
}
typedef struct Director { char *name; Manager *reports; } Director;
Director *InitDirector(const char *name) { Director *d = (Director *)malloc(sizeof(Director));
d->name = strdup(name); return d;
}
void ProcessSalaryDirector(void *d) { Director *d = (Director *)d;
printf("D: Process Salary for Director\n", d->name);
}
```

Now, while we discussed the solution we had used the mechanism of a function switch, that is we explicitly maintained the type and at the time of processing the record of every employee we checked on what is the type maintaining that in a union and then based on the type we call the appropriate salary processing function. Now we would try to look at the same solution with a little bit of different flavor using function pointers. The main differences that I would highlight; certainly the explicit maintenance of types stills remains.

I am just assuming that there are three types; engineer, manager, and director. Earlier if you recall you may open up the earlier video or presentation to check that the salary processing routines, say the salary processing for the engineer was taking a pointer to engineer. Salary processing for manager was taking a pointer to manager, and salary processing for the of director was taking a pointer to director. Now I have changed it a little bit all of them now take a void star pointer, because I want all of these functions to have the same signature.

Once we get it as void star naturally we do not know the void star does not tell me what type of object it is, but since I know the salary processing for engineer has been invoked I know that void star pointer actually points to an engineer record so I cast it to the engineer. Or if it is a manager function called I cast it to manager or director function called I cast it to director and then do the processing. I can still manage without actually explicitly having different parameter types here. But in this process what a gain is all this functions now have the same signature.

Since they have the same signature I can try to combine them in terms of a uniform function pointer type which I say is a ptr is a function pointer which takes a void star and does not written anything. Note this is a function pointer so this basically is a function pointer type this is not a function itself. We will define function.

So, any of this functions for engineer for manager or for the director will actually match with this function pointer type. Rest of the design remain same the different structure types to define different types of objects and with this let us see how do we actually combine the whole thing into the application.

(Refer Slide Time: 06:33)

```
typedef struct Staff {
    E_Type type;
    void *p;
} Staff;

int main() {
    pfFuncPtr pptray[] = { ProcessSalaryEngineer,
                        ProcessSalaryManager,
                        ProcessSalaryDirector };

    Staff staff[] = { { E, InitEngineer("Rohit") },
                    { Mgr, InitEngineer("Kamla") },
                    { Mgr, InitEngineer("Rajib") },
                    { E, InitEngineer("Karita") },
                    { E, InitEngineer("Shashhu") },
                    { Dir, InitEngineer("Sanjana") } };

    for (int i = 0; i < sizeof(staff) / sizeof(Staff); ++i)
        pptray[staff[i].type](staff[i].p);

    return 0;
}
```

Output:

```
Rohit: Process Salary for Engineer
Kamla: Process Salary for Manager
Rajib: Process Salary for Manager
Karita: Process Salary for Engineer
Shashhu: Process Salary for Engineer
Sanjana: Process Salary for Director
```

NPTEL MOOCs Programming in C++ Partha Pratim Das

In terms of the application what we do is, we now maintain a record where on one it say it is basically a doublet this is where I keep the type which would be something like Er or Mgr and so on. The other is a pointer of void star that is this is a pointer of some kind. Now what we do? We populate an array of function pointer. So this is an array of function pointer. I have already defined a function pointer type. So this is an array of function pointers 0 1 and 2, and in each one of these we actually assign three different functions that we have written for the three different structure types.

These have three different function pointers actually deciding in the array. Then you populate the collection of objects, so collection of objects is the object type and the object instance. It is a pointer to the object instance so in place of void star this will now have. If I look into this array this is like doublet, so I have Er then an Er object, I have Mgr, I have a manager object and so on. So we will have in this way there are six different of them. This stuff is my total collection.

Now, when I want to process that I will go over a loop and this is what is critical is I will use this function pointer array and using the type of i-th staff. Say if i is 0 when the type is Er. I will pick up from this function pointer array that is this array, I will index it by this type which is Er that means 0 and then we will call that function. So, this p s array if I just use a different color, say this particular one is a function pointer, because p s array is this array. I have index it with the stuff i dot type which is Er in this case, the first one is an engineer. So what I get is a function pointer which is a function of this is the type of function, it takes a void star and the function is called. And what it passes is the second parameter here which is staff i dot p the pointer to the object.

When I call it with Er here, so I have got this function that is process salary engineer this will get called by the engineer record, that is this will get called with the in it engineer that have done with the Rohit. When i becomes 1 I get the second record where I have the type as Mgr and which is 1, so I get the second function pointer and I invoke it with the pointer to the corresponding employee which is the manager record. In this way one after the other all of this functions will get called with the appropriate type with the appropriate employee record and once we are inside the record we have already seen that

if the engineer function has been invoked and I have got a to the pointer to the engineer record here then I will cast and execute that.

This is the basic style of doing this. The change that we have made is we have made this whole thing into array of function pointers here and we are invoking them through a simple code. And we do not any more need the switch conditional switch that we had been doing all that is taken care of by the indexing in this function pointer array. This could be another solution in C which is a smart solution in C which will be very useful.

(Refer Slide Time: 11:26)

**C++ Solution: Polymorphic Hierarchy: RECAP
Engineer + Manager + Director: (Module 30)**

Module 31
Partha Pratim Das

Objectives & Outline
Staff Salary Processing
C Solution
C++ Solution
Virtual Functions
Passing Tables
Summary

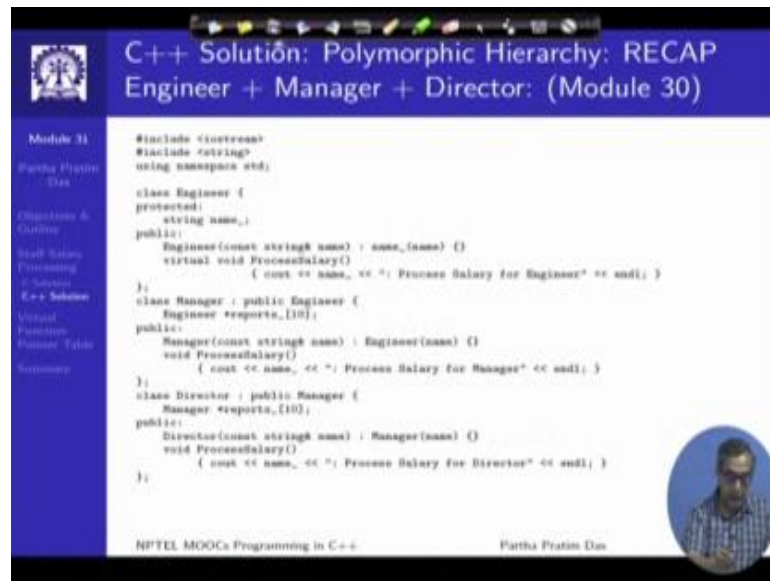
Director — Manager — Engineer

- 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

NPTEL MOOCs Programming in C++ Partha Pratim Das

If we just quickly compare it with the polymorphic C++ solution this was basic approach to the solution and this was a solution.

(Refer Slide Time: 11:36)



C++ Solution: Polymorphic Hierarchy: RECAP
Engineer + Manager + Director: (Module 30)

```
#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; }
};

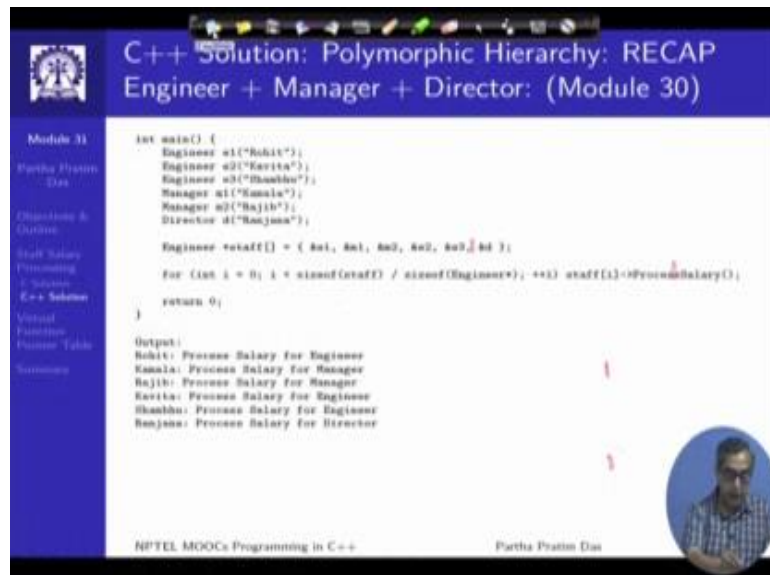
class Manager : public Engineer {
    Manager(const string& name) : Engineer(name) {}
    void ProcessSalary()
        { cout << name_ << ": Process Salary for Manager" << endl; }
};

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

NPTEL MOOCs Programming in C++ Partha Pratim Das

If you look at then you will see that we have all the same things the destructor, the constructor, the different overridden process salary functions and so on.

(Refer Slide Time: 11:53)



C++ Solution: Polymorphic Hierarchy: RECAP
Engineer + Manager + Director: (Module 30)

```
int main() {
    Engineer e1("Rohit");
    Engineer e2("Kavita");
    Engineer e3("Shanbh");
    Manager m1("Kamala");
    Manager m2("Rajib");
    Director d1("Banjara");

    Engineer *staff[] = { &e1, &e1, &e2, &e2, &e3, &e3 };

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

    return 0;
}
```

Output:
Rohit: Process Salary for Engineer
Kamala: Process Salary for Manager
Rajib: Process Salary for Manager
Kavita: Process Salary for Engineer
Shanbh: Process Salary for Engineer
Banjara: Process Salary for Director

NPTEL MOOCs Programming in C++ Partha Pratim Das

And with that if we look at our code is simply for loop with this call being made. Where this each and every one of this is a pointer to the engineer type and depending upon the polymorphic dispatch it will call the appropriate salary process function.

(Refer Slide Time: 12:21)

C Solution

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

C++ Solution

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

NPTEL MOOCs Programming in C++ Partha Pratim Das

If we compare this to side by side this is just comparing the design features.

(Refer Slide Time: 12:26)

C Solution (Function Pointer)

```
#include <stdio.h>
#include <string.h>
typedef enum E_TYPE { E, Mgr, Dir } E_TYPE;
typedef void (*pFuncPtr)(void *);
typedef struct { E_TYPE type; void *p; } Staff;

typedef struct { char *name; } Engineer;
Engineer *InitEngineer(const char *name);
void ProcessSalaryEngineer(void *p);
typedef struct { char *name; } Manager;
Manager *InitManager(const char *name);
void ProcessSalaryManager(void *p);
typedef struct { char *name; } Director;
Director *InitDirector(const char *name);
void ProcessSalaryDirector(void *p);

int main() { pFuncPtr pFuncArray[] = {
    ProcessSalaryEngineer,
    ProcessSalaryManager,
    ProcessSalaryDirector };

    Staff staff[] = {
        { E, InitEngineer("Rohit") },
        { Mgr, InitManager("Kamala") },
        { Dir, InitDirector("Ranjana") } };

    for (int i = 0; i <
        sizeof(staff)/sizeof(Staff); ++i)
        pFuncArray[staff[i].type_](staff[i].p);
    return 0;
}
```

C++ Solution (Virtual Function)

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

class Engineer { protected: string name;
public: Engineer(const string name);
    virtual void ProcessSalary(); };
class Manager : public Engineer {
public: Manager(const string name);
    void ProcessSalary(); };
class Director : public Manager {
public: Director(const string name);
    void ProcessSalary(); };

int main() {
    Engineer e1("Rohit");
    Manager m1("Kamala");
    Director d1("Ranjana");
    Engineer *staff[] = { &e1, &
    m1, &d1 };
    for (int i = 0; i <
        sizeof(staff)/sizeof(Engineer); ++i)
        staff[i]->ProcessSalary();
    return 0;
}
```

NPTEL MOOCs Programming in C++ Partha Pratim Das

But if we just compare this to solution side by side you will see that this is the C solution this is the corresponding C++ class. So, you have the in it engineer, you have the constructor here, you have this particular processing function, you have the over load member function which is a polymorphic in nature. Similarly, I have for the next type the director type and these are three classes that we have.

Now, if we look at the processing part see how similar this two processing look like. So here I take the pointer to that object and based on the dynamic type I dispatch to any one of this functions depending on its corresponding type. Here we do the same thing using the array p s array which is the array of the function pointers that can be used. The only difference is I need to explicitly maintain the type and the relationship between the type and the particular object that I am using.

So based on the type I pick up the function and then for that particularly the second part tells me what is the object pointer and we pass the object pointer. So, here I do not need to do that because as we know all that we need is to basically pick up the right type, the object pointer implicit as this pointer and that will work. The reason I show you this example or show you this way doing it in using function pointer in C is a fact that this is how actually in reality the C++ compiler takes care of the virtual functions or the polymorphic function.

(Refer Slide Time: 14:22)

The slide displays the following code and object layouts:

```
class B {
    int i;
public:
    B(int i, j): x(i, j) {}
    void f(int); // B::f(const, int)
    virtual void g(int); // B::g(const, int)
}
B b(100);
B *p = &b;
```

```
class D: public B {
    int j;
public:
    D(int i, int j): B(i, j), j(j) {}
    void f(int); // D::f(const, int)
    void g(int); // D::g(const, int)
}
D d(200, 500);
D *q = &d;
```

b Object Layout

Offset	VFP
0	B::g(const, int)
4	
8	

Source Expression: b.f(10);
Compiled Expression: B::f(b, 10);

Source Expression: p->f(20);
Compiled Expression: B::f(p, 20);

Source Expression: b.g(30);
Compiled Expression: B::g(b, 30);

Source Expression: p->g(40);
Compiled Expression: p->x[0](p, 40);

d Object Layout

Offset	VFP
0	B::g(const, int)
4	
8	

Source Expression: d.f(10);
Compiled Expression: D::f(d, 10);

Source Expression: q->f(20);
Compiled Expression: B::f(q, 20);

Source Expression: d.g(30);
Compiled Expression: D::g(d, 30);

Source Expression: p->q(40);
Compiled Expression: p->x[0](p, 40);

So to bring it to the actual details, suppose I have a base class B, so I have for this base class I have two functions; one is function f and this function g. Which are one is of polymorphic type g is polymorphic type, f is a non polymorphic type, it is a non-virtual function. And based on this base class I have a derived class here which over writes the non-polymorphic function as well as the polymorphic function.

Now, what you do in this case is? For a non-polymorphic function let us say if we just talk about the function f which is non-polymorphic in nature, so how will the invocations look like? Suppose if I do sample b dot f which is directly accessing it with object. So, how will the compiler call it? Compiler will know from the fact that b is a class B type of object the compiler knows that there is a function f so that is been called. So, it calls b colon-colon f which is this function, and what is required? It requires this pointer which is the address of b, so it puts the address of b and it puts the parameter. So, it puts the address of b here and the parameter and this will get called. This is a static binding; this is a simple static binding.

If I do that using a pointer I get the same thing doing it with a pointer. So, if I am calling this then from the type of the pointer which is of b type it knows that it has to look in the b class we have seen this before and in the b class it has f function which is non-virtual,

so it will directly have to call that function. So, it is statically puts b colon-colon f passes the pointer value and the parameter. These are the static ways this is absolutely fine.

Now, let us look at a third invocation also where I am using the object and calling the non-virtual function g. Again the compiler does the same thing. We know that if we call it with the object it is the function of that class which will get called, so b colon-colon g is what we called. Address of b is passed here and the parameter is passed here. So, up to this point for these three all of these are basically static binding. So, if I instead of b if I have a derive class object d and I try to do this for the derive class object I will have similar static binding code here, where in each one of these the compiler knows that the type is d and whether it is directly by the object or it is through a pointer, but the function is a non virtual one so it will put the static calls.

The issue arrives when I want to actually look at calling something like p pointer g, that is I am a calling virtual function and I am using a pointer so this is where I need the dynamic binding. I am calling p pointer g, I am using a pointer and I am using a virtual function. In this case the p is a pointer to a b object, in the case on right hand side b is pointer to the d object. So, what I will need here? I will need here that p this call will actually resolve to this function.

Whereas I will need that in this case this call will resolve to this function. Even though both of this calls at the call site looks same. How do I do that? how do I make that change? So what I do is something very very simple. Think about the layout of the object, the object has a data member i, so I have a data member i this was constructed with hundred. So, I have a data member i b colon-colon i which is the field here.

Similarly, if I look at the derive class object, it has added a data member j so it will have one base class part which is 200 and the additional data member that has added which is 500. So these are point. Now what we do is we add one more field, this is invisible field we add one more field to the object.

This field is a pointer field and what does it point to, it points to a table of function pointers and this table is known as the VFT or the Virtual Function Table. So what it does

is, when say I am in class b and I am trying to look at the codes, I have a virtual function pointer and I have one virtual function. So I put that virtual function in this table that is I put this pointer in this table. And whenever I will get a call for this function g this virtual function g through a pointer I will not generate a code like this, I will not generate this static code rather I will generate a code like this. Do not get confused with this syntax, what is being said? P is a pointer. So p is a pointer, so p points here.

What is p pointer VFT? P pointer VFT is a pointer to this table. And if p pointer VFT is pointed to this table p pointer VFT 0 is a 0th entry of this table, and what is that entry? That entry is a function pointer. So, p pointer VFT 0 is this 0th function pointer in the table. So, I say that whenever you get p pointer g you actually call whatever function exists in this location. So you pick up this function and call it with the parameters, with parameters up to with this pointer which is p and the actual parameter value which is 45.

The compiler knowing that this is a virtual function and knowing that this is been called through a pointer does not generate the static time code like, hard codes the function, but it puts the call as if through the virtual function table. How does that help? The way it helps is now you have one level of indirection. So what happens is, when you have specialize this class b into class d and you construct a object for that that object also has a virtual function table pointer. That pointer to virtual function table this table is now of class d, this table was of class b. Now what I have done in class d? In the class d this function virtual function g has been overridden a new definition has been given so that is now become d colon-colon g.

So, while we specialize following the specialization we remove this function from the virtual function table of b and through the overriding we put the function that we have written new that is a function for d, d colon-colon g in the virtual function table.

And for this call, the compiler generates the same indirected code. Now what will happen? Now think off. This was the call and this is what the function is generated the compiler has generated these are call what has being generated this is what is been generating the static time. Now what happens, the two scenarios are that p is actually pointing to a b object. If it points to a b object it has the virtual function table pointer it

goes to this virtual function table of b this is what it has got, it picks of function 0 the 0th function and calls that, so it is calling b colon-colon g.

But if p is pointing to a d object here then it has a different function pointer, virtual function table pointer. When it traverses at that pointer v p pointer VFT it actually gets the virtual function table of d. It picks up the 0th function which is this function, which happens to be now d colon-colon g because it actually points to a d object so this table is different. Then it passes the parameters and call that naturally d colon-colon g will get a called.

With this, simple mechanism of table of function pointer, so what we will learn from this, is if a class is polymorphic if a type is polymorphic that if it has at least one a virtual function then for that class there will be a virtual function table, which will have all the virtual functions listed one after the other in the table in the order in which they have been defined. Here we have only one entry because we have only one virtual function. If there are more then there will be more entries and as the class is specialized the compiler checks does the class redefine the virtual function if it does then it changes the corresponding entry in its own virtual function table and replaces with the version that has override.

And then with always makes a call instead of making a direct hard coded call like this a static type call like this where it clearly says what the function, it says that I do not know the about the function at the runtime go to the virtual function table pick up the 0th function and whatever function pointer is your function.

So, depending on the type of the object the virtual function table will be different, but by the same mechanism the current entry the runtime entry in the 0th location for this function will appropriately a point to either the function in the base class or the function in the derive class depending on which kind of object I have and therefore which kind of virtual function table I am pointing to. The basic virtual function I am pointer table mechanism by which we can do things.

(Refer Slide Time: 26:00)

Virtual Function Pointer Table

- Whenever a class defines a virtual function a hidden member variable is added to the class which points to an array of pointers to (virtual) functions called the **Virtual Function Table (VFT)**
- VFT pointers are used at run-time to invoke the appropriate function implementations, because at compile time it may not yet be known if the base function is to be called or a derived one implemented by a class that inherits from the base class
- VFT is class-specific – all instances of the class has the same VFT
- VFT carries the **Run-Time Type Information (RTTI)** of objects

NPTEL MOOCs Programming in C++ Partha Pratim Das 14

As I said whenever a class defines a virtual function a hidden member variable is added to the class which points here. And at the runtime the invocation is indirectly through this and this carries what is known as RTTI. We will talk about RTTI more that is Run-Time Type Information of the whole polymorphic system.

(Refer Slide Time: 26:23)

Virtual Function Pointer Table

```
class A { public:
    virtual void f(int) { }
    virtual void g(double) { }
    int hA * { }
};
class B: public A { public:
    void f(int) { }
    virtual int hB * { }
};
class C: public B { public:
    void g(double) { }
    int hC * { }
};
A a; B b; C c;
A *pA; B *pB;
```

Source Expression

```
pA->f(2);
pA->g(3.2);
pA->h(a);
pA->h(b);

pB->f(2);
pB->g(3.2);
pB->h(a);
pB->h(b);
```

Compiled Expression

```
pA->vft[0](pA, 2);
pA->vft[1](pA, 3.2);
A::h(pA, a);
A::h(pA, b);

pB->vft[0](pB, 2);
pB->vft[1](pB, 3.2);
pB->vft[2](pB, a);
pB->vft[2](pB, b);
```

a Object Layout

Object	VFT
vft → 0	A::f(int, int)
1	A::g(double, double)

b Object Layout

Object	VFT
vft → 0	B::f(int, int)
1	A::g(double, double)
2	B::h(int, B*)

c Object Layout

Object	VFT
vft → 0	B::f(int, int)
1	C::g(double, double)
2	C::h(int, C*)

NPTEL MOOCs Programming in C++ Partha Pratim Das 15

This is one more example, the example that we had solved as a binding exercise so you can see here this is just for your further illustration. This is primarily for your own working out, but we have a class A and specialized from that we have class B, specialized from that we have class C. And class A has a polymorphic function it has a virtual function so naturally the whole hierarchy is polymorphic and therefore, the all objects on any of this classes on this hierarchy will have a virtual function table pointer.

So if I have the object A, say object A here then it will have data members those are not interesting so we have not listed those, but it will have a virtual function table pointer which will point to VFT of class A. And how many entries will be there? There is one virtual function f and one virtual function g. So there is a virtual function f there is a virtual function g.

But when it specializes to B you have made h also a virtual function. So what will happen? A third function will get added in the location two. And what will happen to 0 and 1? F was defined in A and there is over head name B. So, the 0th entry that was for f gets over hidden. Now, in place of a colon-colon f you have b colon-colon f. G was also defined as a virtual function in a it resides in location one, but class b has not overridden g. So, when you inherit you actually get the same a colon-colon g as function number one. And h you have made it a virtual function a fresh so the virtual function table gets added with an additional function pointer.

As you come to C object what you have done you have overridden g. So, what you get, you have not done anything with f, so f is simply inherited. So you got b colon-colon f in the 0th entry continuous to be b colon-colon f. But this entry number one was a colon-colon g, but now you have over written that, so that gets over written with c colon-colon g.

Entry number two was b colon-colon h. The h function in class b that have been over written here so you get c colon-colon h, this is how the virtual function table will keep on growing and that basically will tell you why we said that once of function becomes virtual on this hierarchy it will have to remain virtual because once it that the compiler as

a decision to make and here I have shown what will be the compile version of this codes are.

So you can say this goes to virtual function table entry 0, g goes to table entry 1, but where as if I do p a pointer h then I have static binding, the explicit function calls because this function was non-virtual. Whereas when I do it with p b, that is a pointed to the class b type then they have brought out it through the virtual function table because here in class b h has become a virtual function.

So please work this out, please try to understand this construction very carefully and work through this source expression and against the compile expression of where you have static binding and where you have dynamic binding so that you will be able to understand this as well.

(Refer Slide Time: 30:13)

The image shows a presentation slide with a blue header and footer. The header contains the text 'Module Summary'. The main content area is white and contains two bullet points. The left sidebar is blue and contains a list of navigation items. At the bottom right, there is a small circular portrait of a man. The footer contains the text 'NPTEL MOOCs Programming in C++' and 'Partha Pratim Das'.

Module Summary

- Leveraging an innovative solution to the Salary Processing Application in C using function pointers, we compare C and C++ solutions to the problem
- The new C solution is used to explain the mechanism for dynamic binding (polymorphic dispatch) based on virtual function tables

NPTEL MOOCs Programming in C++ Partha Pratim Das

To summarize it leveraging and innovative solution to the staff salary application which uses function pointers in C we have laid the foundation for explaining how virtual functions are implemented using the virtual function pointer table. Please try to understand this more clearly so that any confusion about dynamic dispatch would be clear in your mind.