Programming in Modern C++
Professor. Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur
Lecture 51
C++11 and beyond: General Features : Part 6:
Rvalue & Perfect Forwarding

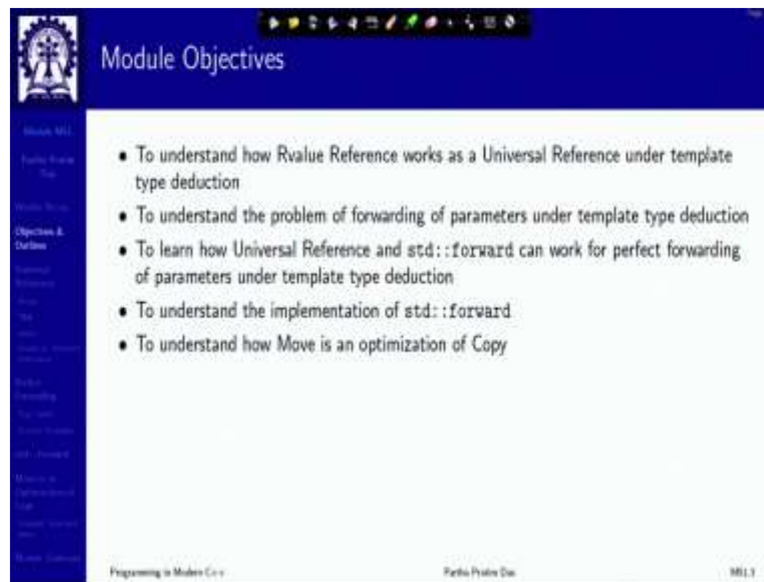
Welcome to programming in Modern C++. We are in week 11. And I am going to start discussing module 51.

(Refer Slide Time: 00:38)

In the last week, we have started talking about the modern part of C++ covering C++11 primarily, and there are several general features that we have covered which are listed here.

Then important general feature that we have discussed is the difference between copying and moving particularly, what is the difference between Lvalue and Rvalue and what is move semantics and how the move in C++ can take advantage of the move semantics to have better performance we have seen std::move function also in the standard library.

(Refer Slide Time: 01:25)

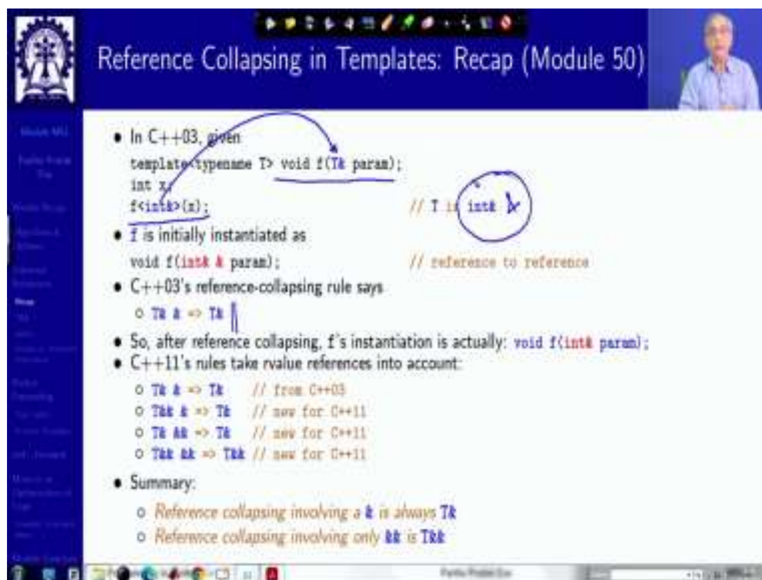


We will continue on that. In the current module we will try to understand how Rvalue reference works as a universal reference under template type deduction and the problem that arises due to forwarding of parameters known as forwarding problem that happens in the template type deduction we try to learn how universal reference and std::forward function can work for perfect forwarding of parameters. We will understand the implementation of std::forward and understand how move is an optimization of the copy what is a solution that C++11 is actually giving us.

(Refer Slide Time: 02:13)

So, this is the outline which will be available on the left. So, starting with discussion on Universal Reference, let me quickly specifically recap a few key concepts that we did in the last module.

(Refer Slide Time: 03:29)



One is of Reference Collapsing, that is, we observed that when we use the template in terms of C++, often we will have multiple references coming for a for a particular type. So, in C++03, there there was one reference collapsing rule so that you have T reference and another reference then this will be collapsed to a single reference. So, reference to reference does not make a sense its reference to the original.

So, this is what it is not only the in this kind of a template when we try to do an invocation and binding using this function f then T becomes int& and therefore we have in total we have the function having a type, T&& which is collapsed to the T&.

(Refer Slide Time: 04:06)

In C++11 We have Rvalue references. So, there are 4 possibilities 4 possible combinations and we learned the collapsing rule that reference collapsing involving any Lvalue reference will always collapse to Lvalue reference and reference collapsing when it involves only Rvalue reference, it will collapse to Rvalue reference. So, this is the only rule where Rvalue and Rvalue will collapse to Rvalue otherwise, it will always be Lvalue. So, we need to keep this in mind and this will be typically used here.

(Refer Slide Time: 04:13)

Now, if we look at the T&& parameter deduction in templates, particularly if I have an Rvalue reference parameter in the template, then we know that this is, this will go through the template type deduction and the in terms of template type deduction, what it does, when it actually a parameter is passed, whether to this template f, this definition of the template f, if I pass if I call it with f(x), where x is an Lvalue, then actually an Lvalue will be passed.

So, this is a Lvalue reference and you have the Rvalue reference. So, due to collapsing this will become an Lvalue reference, the same thing will happen if I pass an Rvalue reference. So, I will have an Rvalue reference and that will collapse to Rvalue reference only similar thing this is for plain old data type, the same thing will happen for the user defined type.

So, we get a very specific feature that if the template parameter is an Rvalue reference, then when Lvalue is passed to that in place of that parameter, then it will become an Lvalue reference if an Rvalue is passed to that parameter, then it will become a non reference or a Rvalue reference.

(Refer Slide Time: 05:53)

Universal References

- T&& really is a magical reference type!
  - For lvalue arguments, T&& becomes T& => lvalues can bind
  - For rvalue arguments, T&& remains T&& => rvalues can bind
  - For const/volatile arguments, const/volatile becomes part of T
  - T&& parameters can bind *anything*
- Two conceptual meanings for T&& syntax:
  - **Rvalue reference**. Binds rvalues only

```
void f(Widget&& param);     // takes only non-const rvalue
```

  - **Universal reference**. Binds lvalues and rvalues

```
template<typename T>
void f(T&& param);          // takes lvalue or rvalue, const or non-const
```

  - Really an rvalues reference in a reference-collapsing context

So, this is a very nice property of the Rvalue reference template parameter which is kind of referred to often as a magical reference type so, that for Rvalue arguments, it binds with Rvalues for Lvalue arguments it binds with Lvalues. So, what we get if we summarize that if I have a plain function with Rvalue reference, then the calls to this function will bind only with Rvalue references that is non constant Rvalues.

Whereas, if we have a universal reference, what is universal reference, if we have the Rvalue reference in the context of a template parameter, then because of this property of reference collapsing, if I pass an Lvalue for this template, then it will behave like an Lvalue reference. If I pass an Rvalue, then it will behave like an Rvalue reference. So, it is kind of takes two different contexts. So, and that is the reason this is called a universal reference that it is adapting itself to the type of argument whether it is an Lvalue argument or it is an Rvalue argument according to that, the binding will become appropriately different.

(Refer Slide Time: 07:24)

In this connection, we also note that the auto feature that we had studied earlier that auto we mentioned that it follows the type deduction which is of templates type deduction. So, if I have auto&&, that is auto of Rvalue reference type, then those are also those will also behave like the universal reference.

So, here we have a Lvalue generated a function call which will give a value and here we have a variable which is an Lvalue. So, if I initialize the auto variable v1 with calcVal, which is an Rvalue, so, the type deduced by the auto&& going by the template type deduction will be int&&, that is it will deduce a Rvalue reference type whereas if I initialize it with a Lvalue, then it will deduce a Lvalue reference due to reference collapsing because auto follows the type deduction of the templates. So, auto&& also behaves like the universal reference.

(Refer Slide Time: 08:48)

But, if we try to do something equivalent using say a decltype, we have seen decltype I can take different type of each of these and then I in style of the auto if I try to put 2 ampersand, it will not behave like a universal reference because decltype as you know always extracts the actual type devoid of the references, it gets the actual type so, whether it is an Lvalue parameter or it is an whether it is an Rvalue parameter or it is an Lvalue parameter, the type deduced by decltype will be just the int.

Therefore, if I use two ampersand after this, it will both become Rvalue reference it will not behave like a universal reference. So, this is the key behavior that is important to understand for the universal references.

(Refer Slide Time: 09:43)

Universal and rvalue references are can be compared very easily both use the same && syntax, and it works as an Rvalue reference if it occurs after a plain old data type or a user defined type for Rvalue reference, but for a variable type T in a template && will work as a universal reference and there Rvalues will bind with Rvalue reference will bind with Rvalues and Lvalue and universal reference will bind with Lvalues and Rvalues. So, this is for just a plain definition. And this is for a template type definition.

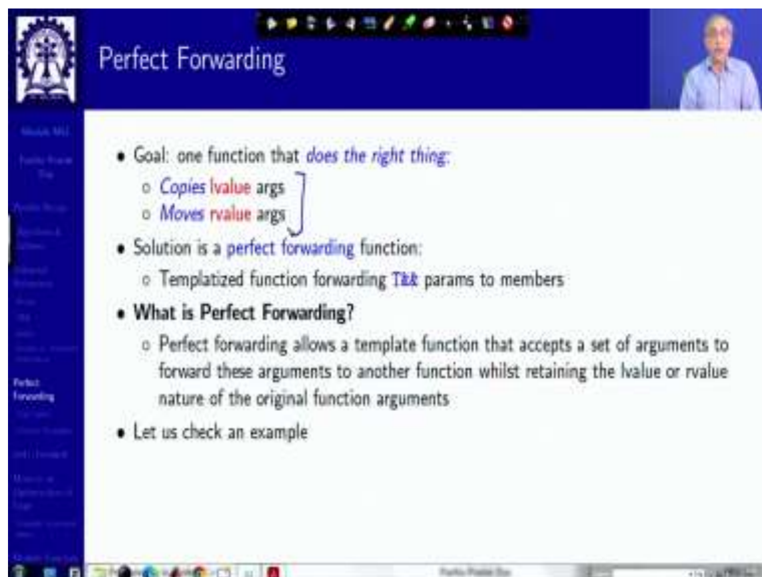
So, if I write int&& in a function, then this will bind as a Rvalue reference, but when I write T&& in a template situation, and then the template is invoked, then it will behave like a

universal reference. Some more examples are given here from the standard library like vector if you do push_back ampersand that is how it is defined. So, this is a lvalue reference, there is a overload, which is an rvalue reference, whereas, emplace_back is a universal reference because it is templatized.

(Refer Slide Time: 11:25)

So, let us with this knowledge, let us move on to the next big problem that rvalue reference solves the first problem that it solved is that of move semantics and we have learned about it pretty well.

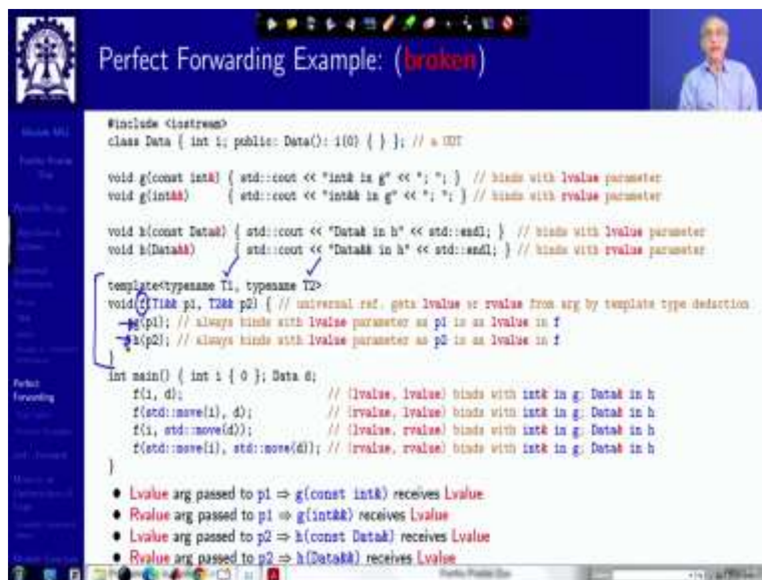
(Refer Slide Time: 11:39)



The next problem, as it is known, it is called the Perfect Forwarding problem. The perfect forwarding problem is arises when in a template function, I want to call another function and pass the parameters I have a template function and I want to call another function from that template function passing the parameters that the template function has received. So, this is what is called the parameter forwarding. So, you will receive parameters and you are forwarding it to another function.

Now, when you do this forwarding process, what you want is that the lvalue arguments should be should work as copies and rvalue arguments should work as move this property should remain so that the move semantics that we have created so, with so, much of care can be carried forward to all kinds of function calls.

(Refer Slide Time: 12:46)



So, let us see how what does that mean. So, to illustrate that, we have 1 template function right here, which has 2 type parameters T1, T2 both of them are given universal references, that is the , this is written as rvalue reference. So, in the template type deduction context, this will become a universal reference. So, this function f wants to call a function g and a function h, to g it passes parameter p1 to h it passes parameter p2, this is what it wants to do.

Now, for g and h, we have written 2 overloads for g we have written 1 overload, which takes const int& which means it takes an Lvalue parameter and it has another overload where it takes a Rvalue parameter. So, we want to check that when I forward this call from f to g using parameter

p1, which of these functions should get called. This is what is done for a plain old data type built in type int.

Similarly, I have defined another arbitrary class data and I have defined similar functions with l 1 with Lvalue parameter and another with Rvalue parameter for this user defined type data and I call this h this is just to show that the behavior is same for the built in type as well as for the user defined type in terms of f g is passed the parameter p2, this is the scenario.

Now, let us see if I try to create 2 variables I of int and d of Data, d has a default constructor so, that is what will get generated we are not really bothered about what the values are. We are more bothered about the actual binding of the functions. So, if I do f(i, d), that means that I am passing lvalue parameter as p2 as well as lvalue parameter as p2, because both an i and d are lvalues, I am passing lvalue arguments to both of them. So, I expect that P1 and P2, both will be received as lvalues and will be passed on as lvalues.

So, if it is passed on as lvalues, then naturally for g this will get should get called for h this should get called. So, you can see the respective statements given here to understand which function has been called. And I correctly find that those functions have been called.

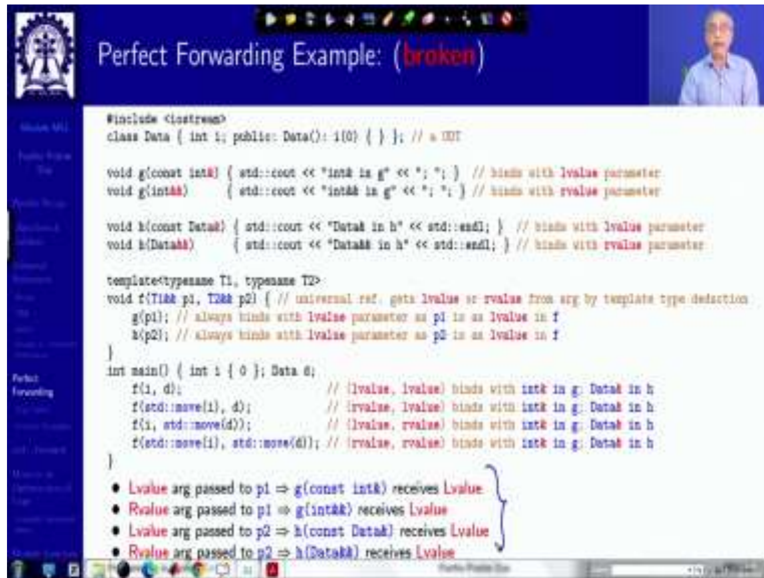
(Refer Slide Time: 15:51)

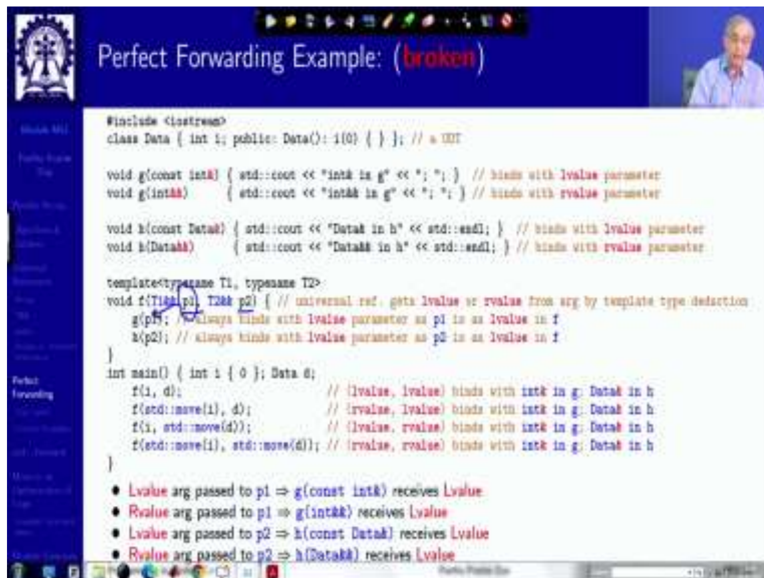
But now say, suppose I keep the lvalue for the second parameter, but the first parameter I changed to rvalue and I could have done it by, having a function which returns integer also, but I have used that simple technique that we have learnt is you can take any lvalue and call std::move on that which converts strips up the Lvalue reference and gives it the Rvalue reference. So, this becomes an Rvalue.

So, this is an Rvalue I am passing and this is an Lvalue I am passing so p1 is Lvalue, Rvalue, p2 is Lvalue. So, I would have expected that why for h it is an Lvalue. So, this function should get called this function should get called but for g this is a Rvalue. So, I would have expected that for the g this function should have been called. But no, this function does not get called g also

binds with the Lvalue version of the function. And you can continue this combination see both for, if you have an Rvalue reference in terms of the function h that is the function using the UDT parameter even then, you get the similar result, if both are Rvalues, then also you get the similar result.

(Refer Slide Time: 17:22)





In short, what you find that irrespective of whether the parameter is actually an Lvalue or an Rvalue, in spite of the universal reference, it is passed always as an Lvalue. So, the forwarding is not correct, the forwarding is not happening in the proper way, the Lvalue, Rvalue -ness of the parameter is getting lost. Why is this happening, very simple.

The reason it happens is when I get the parameter p1 or p2 here, p1 is a name. So, it is a named parameter. So, when I pass p1 here, irrespective of the fact that I had got p1, possibly as an Lvalue, but the fact that it has a name, the compiler deduces that p1 is an Lvalue, it forgets that it has an Rvalue. So, it does not matter whether p1 was received as an Lvalue or an Rvalue, the sheer fact that it has a name will make it a Lvalue and therefore the Lvalue part will happen. So, that is why this solution of forwarding gets broken.

It is liking the move semantics it is very easy to fix that and for that in the utility again you get another function which is known as std::forward and you specify the type you want. What it does is it checks from the T1 by the template deduction it checks what is the reference type is it Lvalue reference or an Rvalue reference and accordingly from this pure Lvalue, it either maintains the Lvalue reference if it is received as an Lvalue, but it makes it the Rvalue reference if it is received as an Rvalue.

So, STD forward actually is a pair of functions, which converts an Lvalue to an Rvalue if required or keeps an Rvalue as an Rvalue.

(Refer Slide Time: 19:35)

So, with that, when we try, we see that the first case has no difference, but then the second case where the first parameter is an Rvalue, we see that actually the Rvalue version of the function g is getting called. Similarly say if you take the last one, both of them are Rvalues and in both cases, the Rvalue version Rvalue reference version of the functions are getting called.

So, by using std::forward, we are able to preserve the Rvalue, Lvalue -ness under the template type deduction and forward perfectly that is the reason this is called perfect forwarding solution or perfect forwarding to say, so, you can, in summary you can get exactly the behavior that you had wanted. So, that is the basic solution of the forwarding problem like in move semantics we

could achieve the move semantics with the help of function std::move here we can get perfect forwarding by std::forward, very simple solution.

(Refer Slide Time: 20:45)

Perfect Forwarding

- Despite T&& parameters, code fully type-safe:
- Type compatibility verified upon instantiation
  - Only int-compatible types valid for call to g()
  - Only Data-compatible types valid for call to h(). For example in the context of

```
...
class DerivedData: public Data { public: DerivedData(): Data() { } };
...
int main() { ... DerivedData d; ... }
```

The code works exactly as before. Whereas for

```
...
class OtherData { int i; public: OtherData(): i(0) { } }; // another UDT
...
int main() { ... OtherData d; ... }
```

The code fails compilation: error: no matching function for call to h(OtherData&)

Perfect Forwarding

- Despite T&& parameters, code fully type-safe:
- Type compatibility verified upon instantiation
  - Only int-compatible types valid for call to g()
  - Only Data-compatible types valid for call to h(). For example in the context of

```
...
class DerivedData: public Data { public: DerivedData(): Data() { } };
...
int main() { ... DerivedData d; ... }
```

The code works exactly as before. Whereas for

```
...
class OtherData { int i; public: OtherData(): i(0) { } }; // another UDT
...
int main() { ... OtherData d; ... }
```

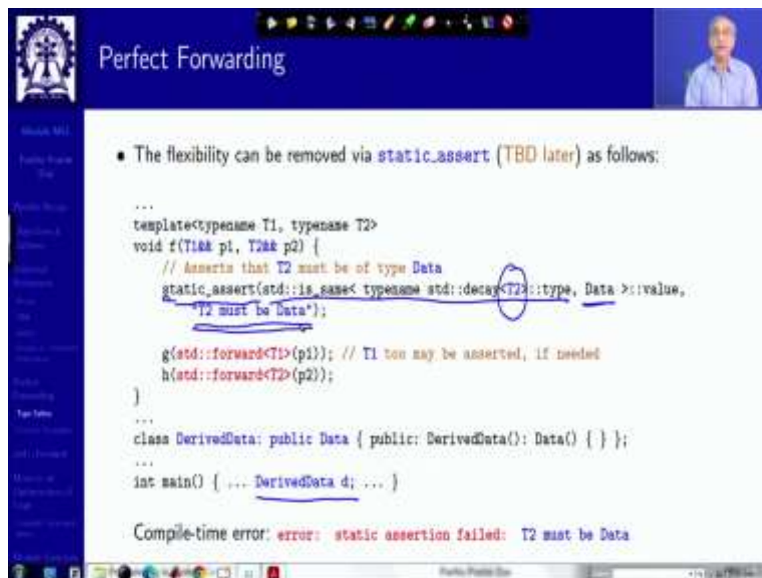
The code fails compilation: error: no matching function for call to h(OtherData&)

Now, this this entire process is very typesafe in the sense that when it does the forwarding it not only forwards to the corresponding type, but it does the same if or any type that is convertible to the given type. So, if I have a type h that that we have and if we have any type u which is derived from h we know that an object of type you can be passed in place of a parameter of type T in terms of reference and the same thing is allowed here.

So, if I define a derived data by specializing data then and we use the object of the derive data and pass it exactly the same way either by Rvalue reference or by by Lvalue reference the entire code will work exactly as fine. But in the contrary, if I have some other class, which is not related to which is not a specialization of data, then try to do this parameter take make an object and try to pass it as a parameter it will fail as it is expected to fail because it cannot bind to a unrelated class.

So, that tells us that this mechanism of universal reference along with the std::forward is not only solving the perfect forwarding problem, but it is fully typesafe solution.

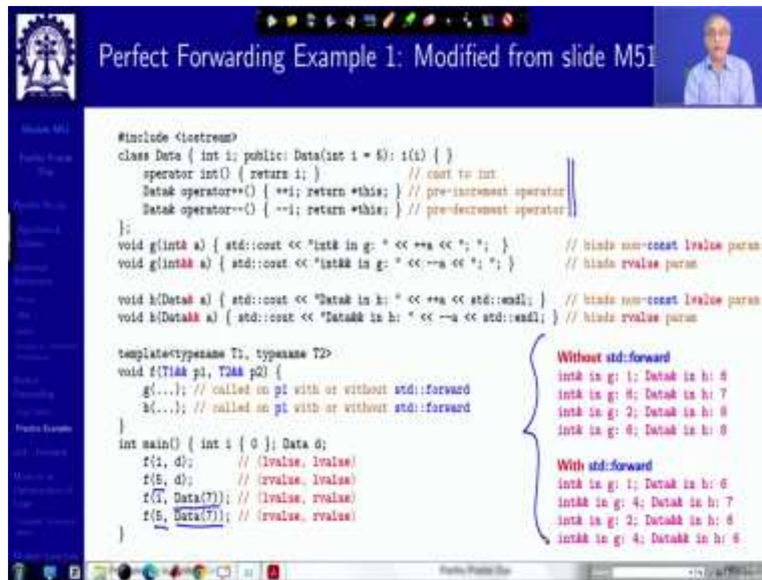
(Refer Slide Time: 22:17)



Now, in fact, you can you can also make sure that if you want that compatible types will not be forwarded even that can be done at a compile time using a specific feature called static assert, I will talk about it in one of the later modules where you can specify this is a do not try to understand the details of this entire thing we will come to that when we do static assert, but the basic thing is you are trying to say that the type of T1 what is the type that you will allow.

So, I am, if I say that for type of T2 here, I will allow only data then when I try to pass a reference to a derived object in Lvalue or Rvalue, then this code will not compile rather I will get a compilation error message that T2 must be of data. So, in this way you can also statically control what you want whether you want the compatibility to extend or how much you want the compatibility to extend and so on.

(Refer Slide Time: 23:24)



So, in the in the following I have given a number of examples based on perfect forwarding this one the first one is based on a little variant of the earlier example where instead of using std::move to generate your Rvalues we have directly used constants or direct temporary objects here and you can see the how the code similarly behaves.

I have given some functionality in the class Data also I would want you to read and run this and see that you actually get these outputs first is without std::forward in which case everything will be forwarded as Lvalue and in the second case with std::forward where appropriate Lvalue or Rvalue forwarding will happen.

(Refer Slide Time: 24:22)

There are two other examples, please go through them. One is a generic factory method here what I want is I want to write a factory kind of function, which takes a create object function which takes a type and and a value and gives me an object of that type. So, it is kind of as if it is in the in the factory and these all should be generic in nature. And in the next couple of slides slowly this whole code has been developed and I do not want to go through it in the presentation. I would want you to run and study and understand this.

(Refer Slide Time: 25:05)

So, this was, this is example 2, which spans over 4 versions and finally gives you a complete solution to see how perfect forwarding can be effectively used in terms of achieving in a really powerful compact generic code.

(Refer Slide Time: 25:24)

Perfect Forwarding Example 3: apply Functor/1

- Let us design an apply functor that would take a function and its arguments and apply the function on the arguments

```
template<typename F, typename... Ts> // Using variadic template (TBD later)
auto apply(std::ostream& os, F&& func, Ts&&... args)
    -> decltype(func(args...)) { // may not preserve rvalue-ness
    os << "Forwarding:: ";
    return func(args...);        // may not preserves rvalue-ness
}
```

- args... are lvalues, but apply's caller may have passed rvalues:
  - Templates can distinguish rvalues from lvalues
  - apply might call the wrong overload of func

```
class Data { };
Data myData() { return Data(); }

class DataDispatcher { public:
    void operator()(const Data&) { std::cout << "operator()(const Data&) called\n\n"; } // takes lvalue
    void operator()(Data&&) { std::cout << "operator()(Data&&) called\n\n"; }          // takes rvalue
};
int main() { Data d = myData();
    apply(std::cout, DataDispatcher(), d);        // Forwarding:: operator()(const Data&) called
    apply(std::cout, DataDispatcher(), myData()); // Forwarding:: operator()(const Data&) called
                                                  // rvalue forwarded as lvalue!
}
```

Programming in Modern C++ Partha Pratim Das M41.22

The next example is about applying, we call it a apply functor, which takes a function and its arguments and applies the function on the arguments. Properly distinguishing the Lvalues and Rvalues again, the solution is developed in two steps, first you just take the name solution and it will not work because everything will be forwarded as Lvalue when the apply tries to take the function and pass the argument to it. But, if you when you use std::forward on the parameter, you will be able to forward it in the proper way. So, please go through and repair on this. The complete solution is finally given in this slide 24.

(Refer Slide Time: 26:16)

Moving forward, what is this std::forward, it is a it is a simple function in the utility component of the Standard Template Library, which which has this property that if it gets a Lvalue reference, then it will give you an Lvalue reference if it gets an Rvalue reference, it will give you a Rvalue so, that is what we wanted.

(Refer Slide Time: 26:45)

The implementation is pretty straightforward, it is templatized by the type. So, for Lvalues, you will get T would be T& for Rvalues T will be simple T and in both cases, we want a universal reference to be returned. Because then under template deduction, it will be Rvalue or Lvalue appropriately, what you do is something very, very simple, you just do the Remove reference, which we had done earlier, take the type of that and make an Lvalue. Do the same thing, make it an Rvalue.

So, you can see that the forward has two overloads one for an Lvalue parameter, Lvalue reference type one for Rvalue reference type. So, depending on the reference type, it has actually got which through template deduction you know, you will be able to choose the right one and then you cast it always to the universal reference and return that. So, with this, your std::forward also is a very simple code, which will work always.

(Refer Slide Time: 27:52)

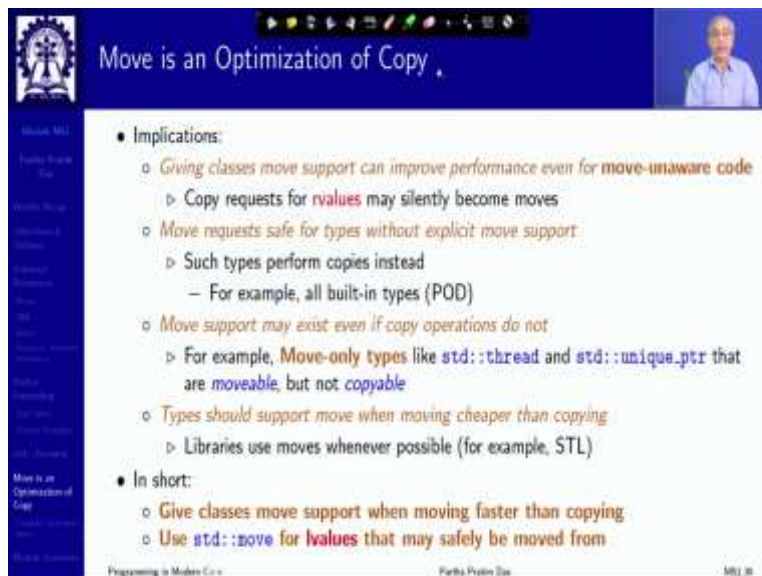
Now, before I close a couple of comments about finally, move as an Optimization of Copy, it is not something which is different. But so the the task of move can be taken up by a copy when it is not possible to move that is that is the essence of the whole thing. So, if my resource class has only copy constructor, and no move support, then even though I might ask the resource in the copy constructor of the class, you would remember these classes we had discussed earlier in in module 50.

So, I am trying to do STD move and trying to move that but since there is no move constructor, this will also use the copy constructor, it will that Rvalue will automatically be converted to the

Lvalue and will be passed to the copy constructor of move my resource. Whereas, if I have the move support added that is if I have given a move constructor, the similar passing the Rvalue reference Rvalue type will give me actually a binding to the move constructor of the my resource class.

So, it is very flexible, that it is not something different that you need to do, you are doing the same, your intention is to move in both cases, but depending on the support given by the target class, if it has moved it will move if it does not have move it will seamlessly without bothering you fall back to the copy version.

(Refer Slide Time: 29:42)



So, that that way it makes it always sensible that you write the move versions as and when it is possible or as and when you feel that well move is will be beneficial than copying. So, you provide that support and in terms of using the other classes it will translate seamlessly because you will be able to take that work with move if the move support is there, if it is not there, it will seamlessly fall back on the copy support.

So, this is this is the basic principle of design that it should be. Of course, note that there are some classes near where the copy operation have forcibly been blocked. So that they can there they are called move only types where you can only move you cannot copy an object, they are only movable not copyable. We will talk about these kinds of classes and objects at a later point

of time. But in general, the types should support move when it is cheaper than copy. That is the general principle that we learn from this support.

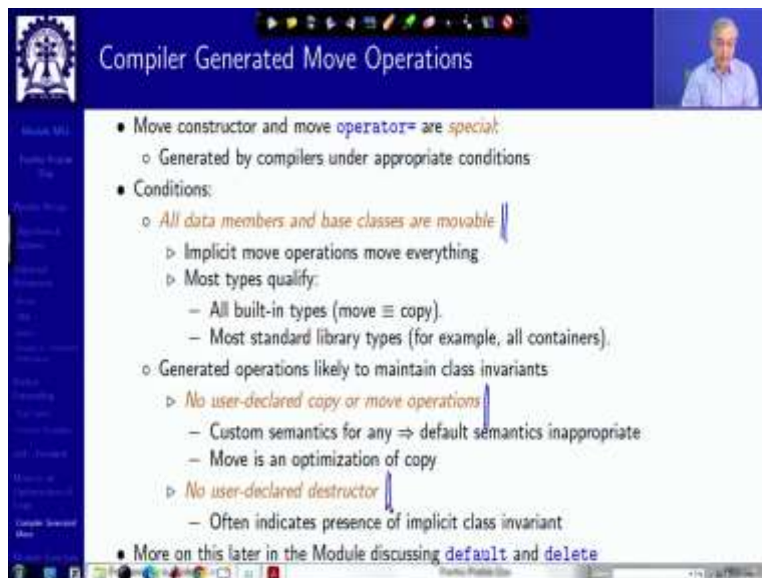
(Refer Slide Time: 30:55)

And just to realize that, we have always been talking about move benefits in case of move constructor move assignment operator, but move in place of copy is not limited to these 2 functions, you can any function can have a move version, if it is meaningful. For example, here I am I am trying to I have a class my list which has a list ID and a vector of values integer values given. So, I have a setter function a function to set an ID given an ID it will set an ID, it I have a value setter also given a vector of values it will set that vector. So, these codes are trivial.

But what it what these codes will do, we we have known always that these codes will copy either this ID string or this vector of values and so on. But we can also if we want, we can also move versions of that by simply changing the parameter of this function to a Rvalue reference non constant Rvalue reference. So, you just overload that function.

And instead of copying, here, you write a move. So, what will happen if your set ID is being set with an Rvalue, then that Rvalue will not have to be copied, which it would in case these were not there. So, this is what makes can make any function which needs to move around data more efficient to work with the copy.

(Refer Slide Time: 32:46)



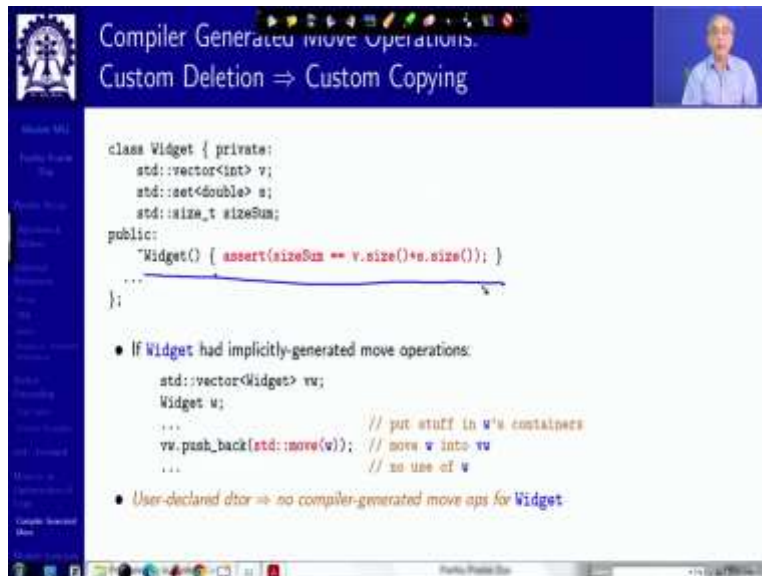
Now, naturally, compilers like compilers to give you support in terms of constructors destructors, copy constructors, etc, that you have missed out that you have not read. Move constructor and move assignment operators are also special in that way that compiler generates the move operations if you have not provided but you are asking for it.

But there are certain conditions under which (the) this will be done one is all data members of the class must be movable only then the compiler will generate otherwise, if you write it will be there otherwise it will not be there. The second condition is they should not be any user declared copy or move operation.

If similar rule applies for copy free copy functions also that if you define a copy constructor, then no free copy constructor will be provided. But here no matter we will have any move operation

defined or any copy operation defined, then this free move functions will not be available. Similarly, if you have a user declared destructor the move operation will not be made freely available, you will have to write it if you need them.

(Refer Slide Time: 34:03)



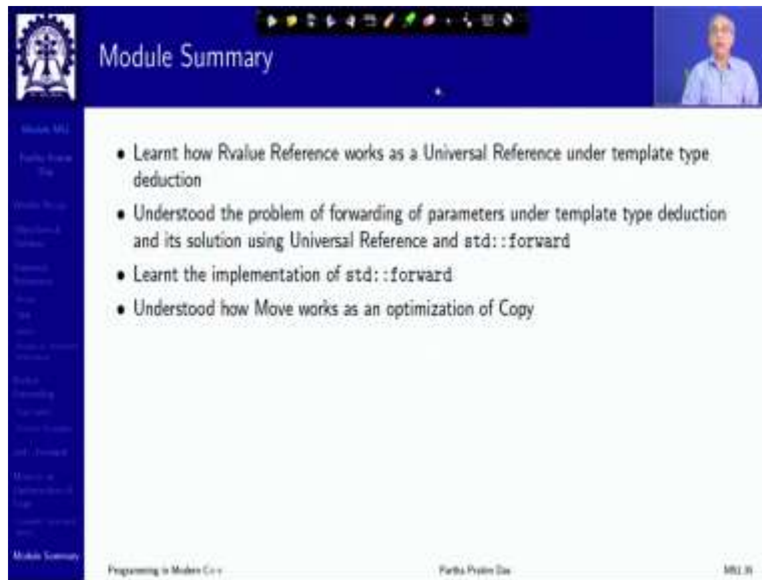
So, these are some of the rules of compiler generated version. So, here I have just shown some examples that if you have a destructor that is if you are doing a custom deletion, then you have to decide what kind of copying you will do because the free move will not be available. Unfortunately, free copy is available which is a bad thing. We will talk about that (on a) at a later point.

(Refer Slide Time: 34:28)

Similarly, if you are doing custom moving then you have to do custom copying also, because you have if you have provided for say, these are, this is a class widget one, which has 2 parameters which are copyable and movable. So, you just have provided a constructor no copy construction operation, no copy operations, no move operations compiler will give you both move and copy operations in terms of construction and assignment.

But, if you add in this Widget2 if you add a user declared copy constructor, then no implicit move operation will be provided though implicit copy assignment operator will be provided. So, the custom move if you have custom move semantics, then you must have custom copy semantics. That is that is what it all relates to. And that is so, so, default is to try to define the move semantics in every case, make advantage of compiler generated move constructor and assignment operator whenever it is possible. But make keep this in mind that whenever possible, moving is cheaper than doing copies and that can give you great optimization in terms of your code.

(Refer Slide Time: 35:49)



So, in summary, we have learnt how Rvalue references work as universal reference and how to solve the perfect forwarding problem and what is the way to design with move as an optimization of the copy? Thank you very much for your attention. We will meet in the next module.