

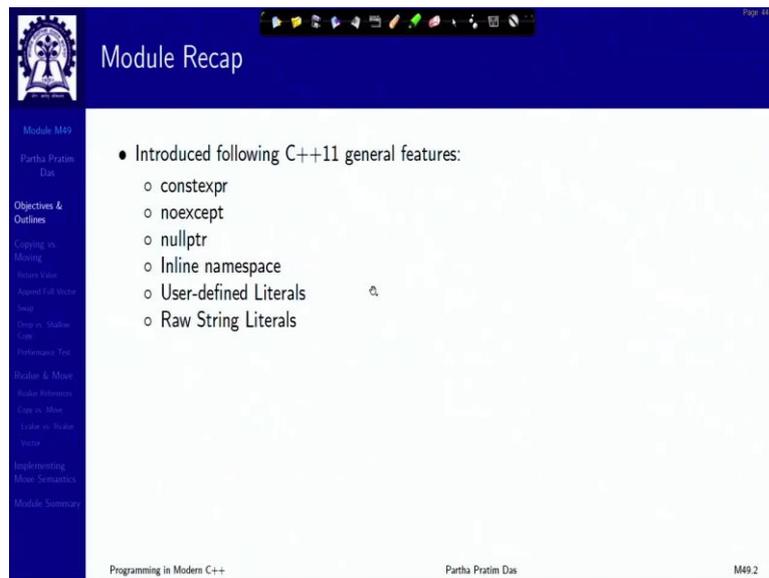
**Programming in Modern C++**  
**Professor Partha Pratim Das**  
**Department of Computer Science and Engineering**  
**Indian Institute of Technology, Kharagpur**

**Lecture 49**

**C++11 and beyond: General Features: Part 4:**  
**rvalue and Move/1**

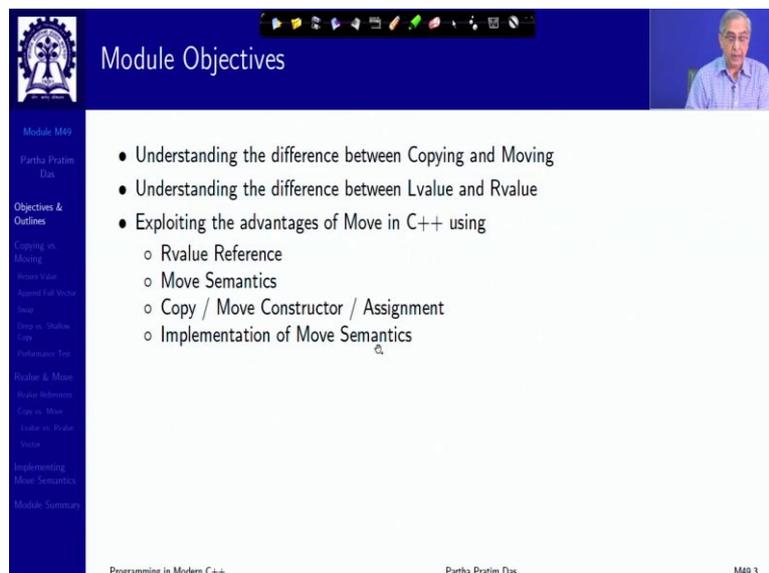
Welcome to Programming in Modern C++ we are in week 10. And we are going to discuss module 49.

(Refer Slide Time: 00:33)



The slide is titled "Module Recap" and is part of a presentation. It features a blue header with the IIT Kharagpur logo on the left and a navigation bar at the top. The main content area is white and contains a bulleted list of C++11 general features. A vertical sidebar on the left lists various topics, with "Objectives & Outlines" highlighted. The footer includes the text "Programming in Modern C++", "Partha Pratim Das", and "M49.2".

- Introduced following C++11 general features:
  - constexpr
  - noexcept
  - nullptr
  - Inline namespace
  - User-defined Literals
  - Raw String Literals



The slide is titled "Module Objectives" and is part of a presentation. It features a blue header with the IIT Kharagpur logo on the left and a navigation bar at the top. A small video inset of the professor is visible in the top right corner. The main content area is white and contains a bulleted list of learning objectives. A vertical sidebar on the left lists various topics, with "Objectives & Outlines" highlighted. The footer includes the text "Programming in Modern C++", "Partha Pratim Das", and "M49.3".

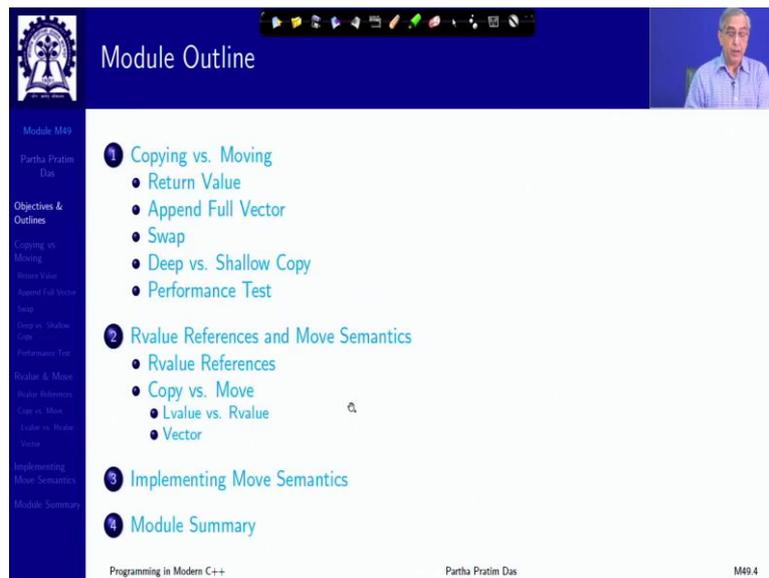
- Understanding the difference between Copying and Moving
- Understanding the difference between Lvalue and Rvalue
- Exploiting the advantages of Move in C++ using
  - Rvalue Reference
  - Move Semantics
  - Copy / Move Constructor / Assignment
  - Implementation of Move Semantics

In the last module, we have continued to discuss different general features of C++11, these six features were discussed, they are diverse and kind of supports different requirements. In this module and the next we are going to discuss something which is also a general feature,

but is fundamentally very, very significant for C++11 extension of the language, particularly for making it a lot more efficient in execution than it used to be.

So, we need to for this, we need to understand the difference between copying and moving something so fundamental. And the difference between lvalue and rvalue. And we take advantage of move in C++ using what is known as rvalue reference and to move semantics.

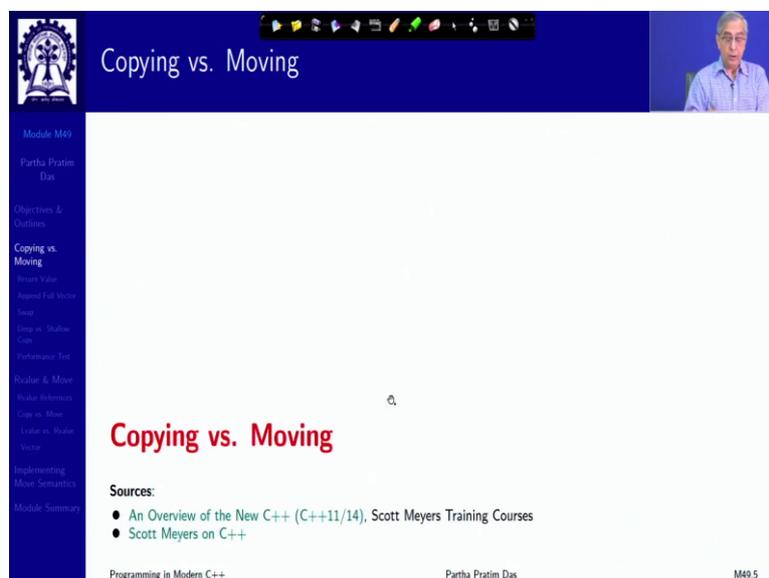
(Refer Slide Time: 01:23)



The slide titled "Module Outline" shows a table of contents for Module M49. The main content is a numbered list of topics:

- 1 Copying vs. Moving
  - Return Value
  - Append Full Vector
  - Swap
  - Deep vs. Shallow Copy
  - Performance Test
- 2 Rvalue References and Move Semantics
  - Rvalue References
  - Copy vs. Move
    - Lvalue vs. Rvalue
    - Vector
- 3 Implementing Move Semantics
- 4 Module Summary

Navigation icons are visible at the top, and a small video feed of the presenter is in the top right corner. The footer includes "Programming in Modern C++", "Partha Pratim Das", and "M49.4".



The slide titled "Copying vs. Moving" features the title in large red font. Below the title, it lists sources:

Sources:

- An Overview of the New C++ (C++11/14), Scott Meyers Training Courses
- Scott Meyers on C++

Navigation icons are visible at the top, and a small video feed of the presenter is in the top right corner. The footer includes "Programming in Modern C++", "Partha Pratim Das", and "M49.5".



## Copying vs. Moving



Module M09

Partha Pratim Das

Objectives & Outlines

Copying vs. Moving

Smart Value

Smart Full Vector

Smart

Smart vs. Smart Copy

Performance Test

Pointer & Move

Basic Performance

Copy vs. Move

State vs. State Value

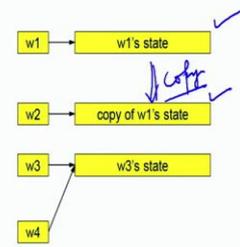
Implementing Move Semantics

Module Summary

- C++ has always supported copying object state:
  - *Copy* constructors, *Copy* assignment operators
- C++11 adds support for requests to *Move* object state:

```

Widget w1; ✓
...
// copy w1's state to w2
Widget w2(w1); ✓
Widget w3; ✓
...
// move w3's state to w4
Widget w4(std::move(w3));
            
```



- **Note:** w3 continues to exist in a valid state after creation of w4

So, this is the outline. So, let us see the difference between copying and moving. So, C++ has supported copying of states of objects, we know that and C++11 is providing support to request moving objects. So, what is the difference? Suppose, we have an object of object w1 of widget type. So, I can use either we can use this kind of way to construct object w2 where the copy of the state of w1 will be created. And w2 will basically mean that it is a copy of this.

Now, let us create another widget object w3. And let us write something like this what it means we will come to that we say `std::move(w3)`, by that what we mean is we do not want to copy the state of w3, but we want to move the state of w3, which means that after this both w3 as well as w4 will share the same state. So, this this, this kind of sounds like the issue of deep copy and shallow copy, which will come to very soon.

(Refer Slide Time: 02:51)

**Copying vs. Moving: Return Value**

- C++ at times performs *extra copy*, while *temporary objects* are prime candidates for *move*:  

```
typedef std::vector<T> TVec; ✓  
TVec createTVec(); ✓ // factory function  
TVec vt;  
...  
vt = createTVec(); // in C++03, copy return value to vt, then destroy return value
```

The diagram shows a variable `vt` and a function `createTVec` that returns a `TVec` object. In the C++03 scenario, the return value is copied into `vt`, and the original return value is destroyed (marked with 'X's).

- *Moving* values would be cheaper and C++11 generally turns such copy operations into moves:  

```
TVec vt;  
...  
vt = createTVec(); // implicit move request in C++11. move data in return value object to vt  
// then destroy return value object
```

The diagram shows the same `vt` and `createTVec` function. In the C++11 scenario, the return value is moved into `vt`, and the original return value is destroyed.

Now, this becomes useful particularly for say a return value. So, in C++03, if you have a vector and you have a create vector function, which creates a vector and returns you then if you initialize this vt with the create(), the create vector is returning a vector, so that vector will be copied.

So, create vector is returning a vector and this vector will be copied. So, all elements as many as are there will be copied. Now, in C++11, if I can make use of the move semantics move request, then I can make it that whatever is returned by the vector will not be copied, but that returned object itself will be moved into vt. So, what is the advantage that two advantages one is -- see or rather one primary advantage is that for making the copy, I have to make copy of so, many objects the vector could be really large.

So, so many constructions and after that this is a return value, this is the return value of create. So, this cannot be used any further this cannot be accessed any further. So, I have to delete each one of them. So, I have to copy make copies and then delete them, which is useless. I can just use these objects, just take them directly. Because in any case, since is a return value does not have a name, I cannot reuse it in any other context. So, that is that is the insight between copy and move. I can move and get much better performance.

(Refer Slide Time: 04:44)

### Copying vs. Moving: Append a Full Vector

- Appending to a full vector causes much *copying before the append*. Moving would be efficient: assume vt lacks unused capacity

```
std::vector<T> vt;
...
vt.push_back(T object);
```

```
std::vector<T> vt;
...
vt.push_back(T object);
```

- vector and deque operations like insert, emplace, resize, erase, etc. would benefit too

### Copying vs. Moving: Append a Full Vector

- Appending to a full vector causes much *copying before the append*. Moving would be efficient: assume vt lacks unused capacity

```
std::vector<T> vt;
...
vt.push_back(T object);
```

```
std::vector<T> vt;
...
vt.push_back(T object);
```

- vector and deque operations like insert, emplace, resize, erase, etc. would benefit too

Think about appending to a full vector. Suppose I have a vector, say again. And suppose it has become full. What happens if the vector becomes full and I try to do a push\_back I am trying to do a push\_back of it, of a some T object, appropriate object. What we will have to do? It will have to create another new space for the vector, copy all the existing elements, and then put the push\_back element at the end.

So, this is our original state of the vector, all these T objects were existing, since it has run out of space, I take a bigger space, and then I copy each object, this entire vector, and then I add the new element, this is what is freaking. Extremely expensive, because as many elements are there, those many copies and there deletes all of these will have to happen.

Instead, the vector is going to change. And after this, this has been done, this old vector is of no use. So, if you are, in any case, going to delete it. So, why not, we do a move. That is we do not copy and construct the object and delete the earlier one, do not take this, say, I am taking this making a copy, deleting this, taking this making a copy deleting this, instead of I will just have the new vector allocation.

And let those existing objects be might, without actually doing copy, just moving that data thing, no creation, no destruction, and then have the new one. So, it can be tremendously great use if I could make use of this move. The question, obviously, is to decide when to copy and when to move.

(Refer Slide Time: 06:52)

Module M49  
Partha Pratim Das

Objectives & Outlines  
Copying vs. Moving  
Smart Pointers  
Swap  
Copy vs. Move  
Performance Test  
Riddle & More  
Notes Information  
Copy vs. Move  
Copy vs. Move  
Value  
Implementing Move Semantics  
Module Summary

### Copying vs. Moving: Swap

- Consider swapping two values:

**By Copy**

```
template<typename T> void swap(T& a, T& b) { // std::swap impl. by copy
void swap(T& a, T& b) {
    T tmp(a);           // copy a to tmp (=> 2 copies of a)
    a = b;             // copy b to a (=> 2 copies of b)
    b = tmp;           // copy tmp to b (=> 2 copies of tmp)
}                       // destroy tmp
```

**By Move**

```
template<typename T> void swap(T& a, T& b) { // std::swap impl. by move
    T tmp(std::move(a)); // move a's data to tmp
    a = std::move(b);   // move b's data to a
    b = std::move(tmp); // move tmp's data to b
}                       // destroy (eviscerated) tmp
```

Module M49  
Partha Pratim Das

Objectives & Outlines  
Copying vs. Moving  
Smart Pointers  
Swap  
Copy vs. Move  
Performance Test  
Riddle & More  
Notes Information  
Copy vs. Move  
Copy vs. Move  
Value  
Implementing Move Semantics  
Module Summary

### Copying vs. Moving: Swap

- Consider swapping two values:

**By Copy**

```
template<typename T> void swap(T& a, T& b) { // std::swap impl. by copy
void swap(T& a, T& b) {
    T tmp(a);           // copy a to tmp (=> 2 copies of a)
    a = b;             // copy b to a (=> 2 copies of b)
    b = tmp;           // copy tmp to b (=> 2 copies of tmp)
}                       // destroy tmp
```

**By Move**

```
template<typename T> void swap(T& a, T& b) { // std::swap impl. by move
    T tmp(std::move(a)); // move a's data to tmp
    a = std::move(b);   // move b's data to a
    b = std::move(tmp); // move tmp's data to b
}                       // destroy (eviscerated) tmp
```

The screenshot shows a presentation slide with the title "Copying vs. Moving: Swap". On the left, there is a navigation menu with items like "Module M09", "Partha Pratim Das", "Objectives & Outlines", "Getting in Swing", "Smart Value", "Smart Value", "Swap", "Using in Swaps", "Copy", "Performance Test", "Builder & Move", "Copy Semantics", "Copy in Move", "Using in Move", "Value", "Implementing", "Move Semantics", and "Module Summary". The main content area contains a bullet point: "Consider swapping two values:". Below this, there are two code snippets. The first is labeled "By Copy" and shows a templated swap function that uses direct assignments, resulting in multiple copies of the objects. The second is labeled "By Move" and shows the same function using `std::move` to transfer ownership, which is more efficient. The code for "By Move" has several lines underlined in blue.

So, that is that is the tricky part, which we will have to think about a very common function that we always write swap, swapping, two variables that templated function A and B have the same type it swaps. And this is a code that all of us have learnt initialize a temporary and use that to swap. So, what happens when you initialize when you create the temporary with a now we are copying a to the temporary, so there are two copies of a, one is in a one is in tmp.

Then you are making a copy assignment to b, you are copying b to a. So, what happens there are two copies of b one in a and other in b and finally you do copy tmp, copy assign tmp to b, so there are again two copies of tmp, one in b, another is in tmp and then you destroy them.

So, every time you can see that you have 2 copies, which is unnecessary. Because you did not, you did not want that all that you wanted is they just get swapped. So, kind of if I had pointers to them, logical thinking is if I had pointers to them, I can just swap these pointers in terms of actually changing the objects.

So, you can do that in C++11 by just saying this `std::move`, we just saw some time back, which tells that do not copy the object, but move the object. So, initially, I need this, I need this temporary to do the swap. So, I create the temporary, but I do not copy the state of it, I move the state of it into that. So, a becomes tmp is now holding a, a becomes free, it is not holding anything meaningful. So, then I move b into a.

So, a now holds that state of b and b becomes free does not hold anything meaningful. So, I then move tmp into b. So, b becomes takes the state of tmp which was the original state of a and tmp does not hold anything meaningful. So, I can destroy tmp without doing anything. I

do not just need to I did not do a copy. So, I did not take resources I did not create resources. So, the destruct here is just dummies just call.

(Refer Slide Time: 09:37)

**Copying vs. Moving: Deep vs. Shallow Copy**

- Moving most important when:
  - Object has data in separate memory (for example, on free store).
  - Copying is deep
- Moving copies only object memory
  - Copying copies object memory + separate memory
- Consider copying/moving A to B:
  - **Copying:** A<sub>c</sub> (object memory) is copied to copied A data (separate memory).
  - **Moving:** A<sub>m</sub> (object memory) is moved to B data (separate memory), which is circled in red.
- Moving never slower than copying, and often faster

So, copying versus move is a basic question, which is deep. If we have, if we see the distinction between deep copy and shallow copy. Copy is in our connotation always deep copy we have ensured that so if I have an object, then it is in memory and it has resources like it is pointing to 10 other things 10 different pointers to different other objects, so they are in a separate memory. So, when we talk about copy, we mean a deep copy that is copy the object memory as well as the separate memory.

So, if I have A then I copy, then I have A's data. And I will also have a copy of A's data in the sep..., from the separate memory that is the deep copy we say. So, we copy the pointed objects as well, where is a move, I will have the A's data the object memory and for move, I will not have the copy of the A's data, but I will use B's data itself and not create that separate memory again, just I take that memory. So, by that so move obviously invalidates the source, which copy does not. So, whenever the source from where I am I want to make a duplicate off is not required after this operation, I can certainly do move instead of doing copy, certainly it will be always faster to do that.

(Refer Slide Time: 11:20)

Simple Performance Test

- Given

```
const std::string stringValue("This string has 29 characters");
```

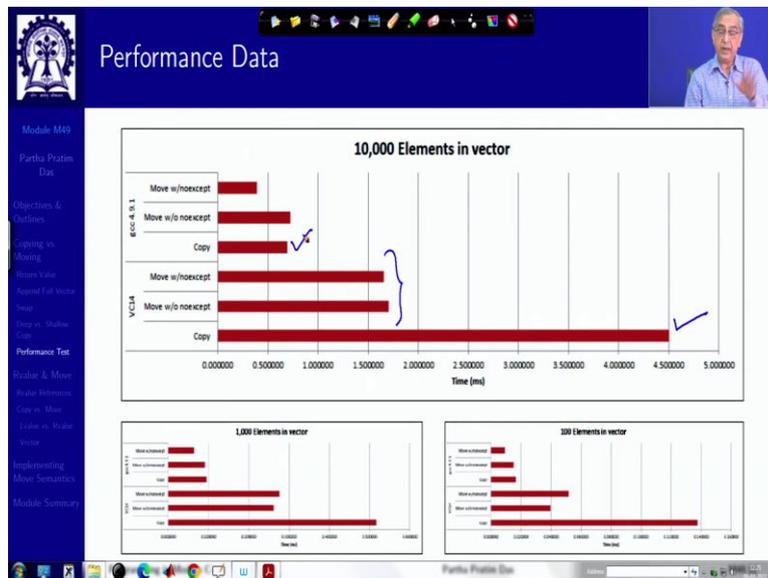
```
class Widget { std::string s;  
public:  
    Widget(): s(stringValue) { }  
    ...  
};
```

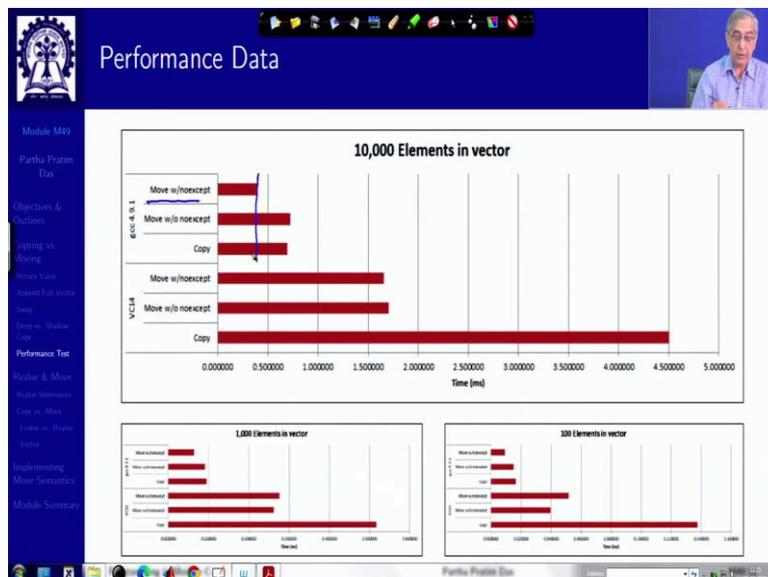
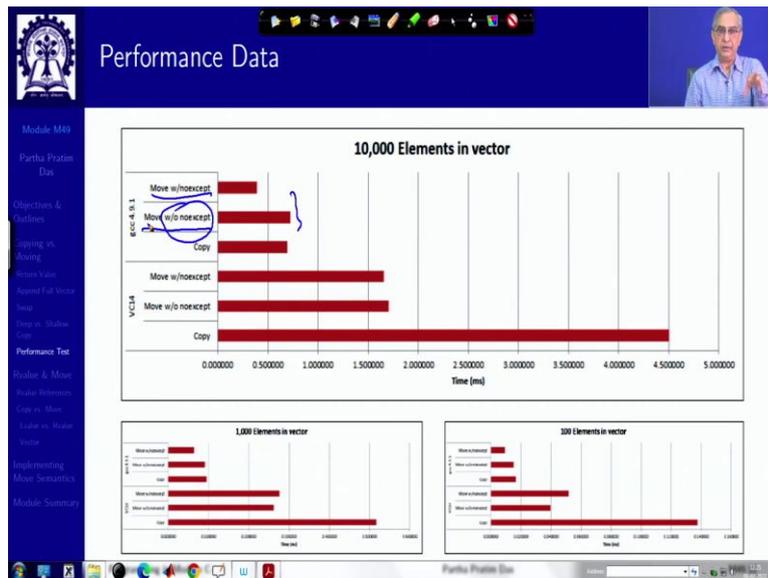
- Consider this `push_back`-based loop:

```
std::vector<Widget> vw;  
Widget w;  
for (std::size_t i = 0; i < n; ++i) { // append n copies of w to vw  
    vw.push_back(w);  
}
```

So, just a performance test, which was done decades back. So, here is a widget which has a string constant string value and it just constructs that and, it tries to do a `push_back` of the same value in the in a vector of widgets repeatedly it is repeatedly being pushed back is just a bulk workload to show that how does copy so in generally will copy the for the `push_back`, how does it impact.

(Refer Slide Time: 11:58)

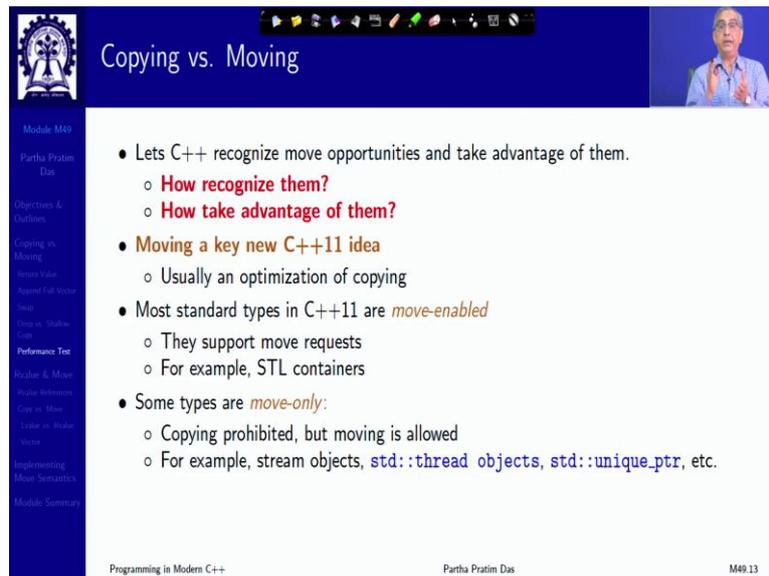




And, results are really stunning, you can see that with copy, if you take this much without copy with move you take this much less similar thing happens here obviously, in terms of Visual Studio compiler, you see a much bigger advantage there are some reasons for that, but the basic idea is being able to move instead of copy and you can also see the difference between move without except, move without noexcept and move with no exception.

You can see that in GCC if you do not have no except, then you may not get much benefit, because you have a lot of exception possibility of exception handling code a data structure that you need to deal with. But if you use no except as you are doing here, then you get almost about half the required time.

(Refer Slide Time: 13:04)



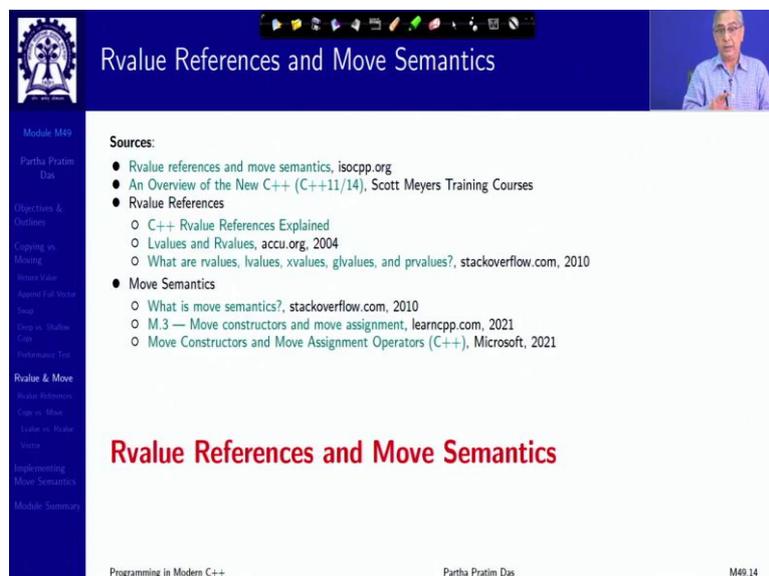
**Copying vs. Moving**

- Lets C++ recognize move opportunities and take advantage of them.
  - **How recognize them?**
  - **How take advantage of them?**
- **Moving a key new C++11 idea**
  - Usually an optimization of copying
- Most standard types in C++11 are *move-enabled*
  - They support move requests
  - For example, STL containers
- Some types are *move-only*:
  - Copying prohibited, but moving is allowed
  - For example, stream objects, `std::thread` objects, `std::unique_ptr`, etc.

Programming in Modern C++ | Partha Pratim Das | M49.13

So, performance gets so that is the objective with which C++11 has focused on the semantics of move along with the semantics of copy. Now, the question naturally is how to recognize them, how to take advantage of them. And for that C++ standard has made the standard types move enabled and some of the types are moved, you cannot copy them you can just move them we will talk about those more in future.

(Refer Slide Time: 13:31)



**Rvalue References and Move Semantics**

**Sources:**

- Rvalue references and move semantics, [isocpp.org](http://isocpp.org)
- An Overview of the New C++ (C++11/14), Scott Meyers Training Courses
- Rvalue References
  - C++ Rvalue References Explained
  - Lvalues and Rvalues, [accu.org](http://accu.org), 2004
  - What are rvalues, lvalues, xvalues, glvalues, and prvalues?, [stackoverflow.com](http://stackoverflow.com), 2010
- Move Semantics
  - What is move semantics?, [stackoverflow.com](http://stackoverflow.com), 2010
  - M.3 — Move constructors and move assignment, [learncpp.com](http://learncpp.com), 2021
  - Move Constructors and Move Assignment Operators (C++), Microsoft, 2021

**Rvalue References and Move Semantics**

Programming in Modern C++ | Partha Pratim Das | M49.14

So, for this what we need to understand and I would request you to be very, very attentive and really focus on this because this is something which is simple, but the core of C++ move 11 performance.

(Refer Slide Time: 13:44)

### Lvalues and Rvalues

- **Lvalues** are generally things we can take the address of:
  - In C, Expressions on *left-hand-side (LHS)* of an assignment
  - Named objects - variables
  - Legal to apply address of (&) operator
  - **Lvalue** references
- **Rvalues** are generally things we cannot take the address of:
  - In C, Expressions on *right-hand-side (RHS)* of an assignment
  - Typically unnamed temporary objects - expressions, return values from functions, etc.
  - **Rvalue** references
- Examples:

```
int x, *pInt; // x, pInt, *pInt are lvalues
std::size_t f(std::string str); // f and str are lvalues, f's return is rvalue
f("Hello"); // temp string("Hello") created for call is rvalue
std::vector<int> vi; // vi is lvalue
...
vi[5] = 0; // vi[5] is lvalue
```

  - Recall that `vector<T>::operator[]` returns `T&`

*@=b; lvs rvs lvalue rvalue*

### Lvalues and Rvalues

- **Lvalues** are generally things we can take the address of:
  - In C, Expressions on *left-hand-side (LHS)* of an assignment
  - Named objects - variables
  - Legal to apply address of (&) operator
  - **Lvalue** references
- **Rvalues** are generally things we cannot take the address of:
  - In C, Expressions on *right-hand-side (RHS)* of an assignment
  - Typically unnamed temporary objects - expressions, return values from functions, etc.
  - **Rvalue** references
- Examples:

```
int x, *pInt; // x, pInt, *pInt are lvalues
std::size_t f(std::string str); // f and str are lvalues, f's return is rvalue
f("Hello"); // temp string("Hello") created for call is rvalue
std::vector<int> vi; // vi is lvalue
...
vi[5] = 0; // vi[5] is lvalue
```

  - Recall that `vector<T>::operator[]` returns `T&`

*a+b+c*

### Lvalues and Rvalues

- **Lvalues** are generally things we can take the address of:
  - In C, Expressions on *left-hand-side (LHS)* of an assignment
  - Named objects - variables
  - Legal to apply address of (&) operator
  - **Lvalue** references
- **Rvalues** are generally things we cannot take the address of:
  - In C, Expressions on *right-hand-side (RHS)* of an assignment
  - Typically unnamed temporary objects - expressions, return values from functions, etc.
  - **Rvalue** references
- Examples:

```
int x, *pInt; // x, pInt, *pInt are lvalues
std::size_t f(std::string str); // f and str are lvalues, f's return is rvalue
f("Hello"); // temp string("Hello") created for call is rvalue
std::vector<int> vi; // vi is lvalue
...
vi[5] = 0; // vi[5] is lvalue
```

  - Recall that `vector<T>::operator[]` returns `T&`

*Handwritten annotations: checkmarks and circles around 'str' and 'string' in the code block.*

**Lvalues and Rvalues**

- **Lvalues** are generally things we can take the address of:
  - In C, Expressions on *left-hand-side (LHS)* of an assignment
  - Named objects - variables
  - Legal to apply address of (&) operator
  - **Lvalue** references
- **Rvalues** are generally things we cannot take the address of:
  - In C, Expressions on *right-hand-side (RHS)* of an assignment
  - Typically unnamed temporary objects - expressions, return values from functions, etc.
  - **Rvalue** references
- Examples:
 

```
int x, *pInt;           // x, pInt, *pInt are lvalues
std::size_t f(std::string str); // f and str are lvalues, f's return is rvalue
f("Hello");           // temp string("Hello") created for call is rvalue
std::vector<int> vi;   // vi is lvalue
...
vi[5] = 0;             // vi[5] is lvalue
```

  - Recall that `vector<T>::operator[]` returns `T&`

What is rvalue and what is move semantics? So, in this let us understand that what is an lvalue and what is an rvalue. So, lvalue, rvalue, this name were given in terms of C mode in terms of C programming lvalue is something that occurs on the left hand side of an assignment and rvalue is something which occurs on the right hand side of the assignment.

Now, what it means? Is if I am doing b assigned to a, then for b, I need the value of b but for a I need the address of a where it has to go I have to locate a. So, this is the left-hand side this is the hand side and this is what we call lvalue, this is what we call rvalue. Coming to C++ connotation this besides assignment there are several contexts where you need to talk about lvalue and rvalue. So, the this basic left hand side of an assignment connotation has reduced so often, now we talk about lvalue as locator value which can be located and rvalue is something which is not an lvalue.

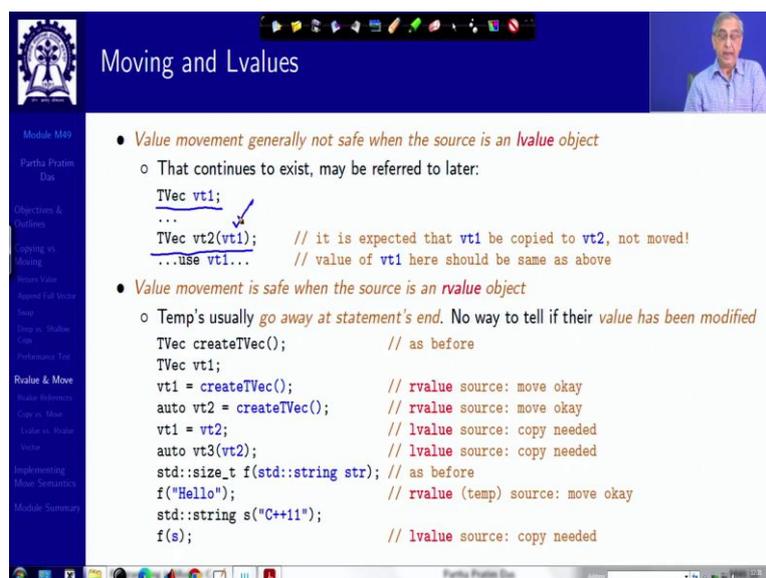
So, with that distinction, if we look at then these are lvalues are named objects variables, which you can catch hold off make computations with us the value in future. rvalues are typically unnamed temporary objects, which exists which has a value, but there is no way that you can catch hold of them like expressions  $a + b * c$ , naturally all of this cannot happen in one go, this happens and the value is generated with which you are adding a.

But can you access that value? No, this is an unnamed temporary thing. Similarly, return values from function these are all or different rvalues, and then you have lvalue reference and rvalue reference here are examples for your understanding `int x`, `int *pInt`, these are naturally lvalues you can catch hold of them, `f()` as a function is an lvalue, `str` as a parameter is an lvalue.

Whereas, f's return value if the return value is an rvalue, because either you copy and keep it or move and keep it or it actually gets lost at the end of the function called this will get lost. So, in this you have a, you have a Hello within double quotes, which means it is a constant char\*. Now, from that a std string has to get constructed to be called to that function, there is a construction involved. Now, that object which gets constructed and passed the str you have no hold on that object that object will get created will go to the function and will get destroyed after that after the function.

So, the str std::string of Hello, that object is not exist is a temporary and you will not be able to catch hold of it. So, it is an rvalue. Similarly, vi if I define it is an lvalue vi[5] is an lvalue. And that is the reason you will see that if you look at the in vector, if you look at the access operator operator, square brackets, it returns a reference because it is an lvalue.

(Refer Slide Time: 17:36)



The slide is titled "Moving and Lvalues" and features a video feed of a presenter in the top right corner. The main content consists of two bullet points and a code block. The first bullet point states that value movement is generally not safe when the source is an lvalue object, with a sub-point that such objects continue to exist and may be referred to later. A code snippet shows a TVec vt1; followed by TVec vt2(vt1); and a comment indicating that vt1 is expected to be copied to vt2, not moved. The second bullet point states that value movement is safe when the source is an rvalue object, with a sub-point that temporaries usually go away at the end of a statement. A code block follows, showing various operations on TVec objects and string literals, with comments indicating the source type (rvalue or lvalue) and the required action (move or copy).

```
Module M49
Partha Pratim Das
Objectives &
Outlines
Learning in
Moving
String Liter
Scoped Enum
Type
Using in: Public
Class
Performance Top
Rvalue & Move
Rvalue Reference
Copy in: Move
Using in: Public
Vector
Implementing
Move Semantics
Module Sources
```

### Moving and Lvalues

- *Value movement generally not safe when the source is an lvalue object*
  - That continues to exist, may be referred to later:

```
TVec vt1;
...
TVec vt2(vt1); // it is expected that vt1 be copied to vt2, not moved!
...use vt1... // value of vt1 here should be same as above
```
- *Value movement is safe when the source is an rvalue object*
  - Temp's usually go away at statement's end. No way to tell if their value has been modified

```
TVec createTVec(); // as before
TVec vt1;
vt1 = createTVec(); // rvalue source: move okay
auto vt2 = createTVec(); // rvalue source: move okay
vt1 = vt2; // lvalue source: copy needed
auto vt3(vt2); // lvalue source: copy needed
std::size_t f(std::string str); // as before
f("Hello"); // rvalue (temp) source: move okay
std::string s("C++11");
f(s); // lvalue source: copy needed
```

**Moving and Lvalues**

- *Value movement generally not safe when the source is an lvalue object*
  - That continues to exist, may be referred to later:
 

```
TVec vt1;
...
TVec vt2(vt1); // it is expected that vt1 be copied to vt2, not moved!
...use vt1... // value of vt1 here should be same as above
```
- *Value movement is safe when the source is an rvalue object*
  - Temp's usually *go away at statement's end*. No way to tell if their *value has been modified*

```
TVec createTVec(); // as before
TVec vt1;
vt1 = createTVec(); // rvalue source: move okay
auto vt2 = createTVec(); // rvalue source: move okay
vt1 = vt2; // lvalue source: copy needed
auto vt3(vt2); // lvalue source: copy needed
std::size_t f(std::string str); // as before
f("Hello"); // rvalue (temp) source: move okay
std::string s("C++11");
f(s); // lvalue source: copy needed
```

**Moving and Lvalues**

- *Value movement generally not safe when the source is an lvalue object*
  - That continues to exist, may be referred to later:
 

```
TVec vt1;
...
TVec vt2(vt1); // it is expected that vt1 be copied to vt2, not moved!
...use vt1... // value of vt1 here should be same as above
```
- *Value movement is safe when the source is an rvalue object*
  - Temp's usually *go away at statement's end*. No way to tell if their *value has been modified*

```
TVec createTVec(); ✓ // as before
TVec vt1; ✓
vt1 = createTVec(); ✓ // rvalue source: move okay
auto vt2 = createTVec(); ✓ // rvalue source: move okay
vt1 = vt2; ✓ // lvalue source: copy needed
auto vt3(vt2); ✓ // lvalue source: copy needed
std::size_t f(std::string str); // as before
f("Hello"); // rvalue (temp) source: move okay
std::string s("C++11");
f(s); // lvalue source: copy needed
```

Now, how do you move what is the consequence of moving movement of values in terms of lvalues and rvalues. So, if you move a value, when it is an lvalue, that is often generally not safe. So, you have a vector vt1, you are creating another vector with vt2 with vt1. Now, there are two choices as we have seen, we can copy the state so that vt1 remains valid, and vt2 is a copy of that vt1 or we can move the state so that we do not have to duplicate both vt1, vt2 is the same state.

Now, the risk of doing that is vt1 is an lvalue, I have a name, it is a named object. So, subsequent to this construction, subsequent to this construction of vt2, I can still use vt1, so vt1 and vt2 needs to be different. So, move here is not something which is smart. Whereas if I have values which are rvalues, which are temporary, it is safe to move them. I have this function, I have an object vt1, and I have called this function and the returned object from

createTVec. I am assigning to vt1. Now, the return of TVec is a temporary object which will get destroyed anyway at the end of this call. So, if I move that to vt1, I do not lose anything. So, it is an rvalue and naturally move is okay.

Similarly, if I do an initialization of vt2 not an assignment using the return value of TVec it is again an rvalue and moving is okay. Whereas, if I assign vt2 to vt1, then vt2 is an lvalue. So, I need a copy. If I do a copy construction of vt3 from vt2, it is again an lvalue I need a copy, If I have this function f() as before as we have seen Hello is a rvalue, a temporary value.

So, move is okay if I have a string like this then f() calling f(s) need to have a copy because I have s as a named object representing the string within double quotes C++11. It represents a std::string and I can use it subsequently for that. So, that is a judgement point that you would have to see that is there a way to use that object, subsequently if it is then it cannot be moved, it should not be moved, then it should be copied. So, move is not good for lvalues, but move is excellent for rvalues.

(Refer Slide Time: 20:49)

The image shows a presentation slide titled "Rvalue References" with a blue header and a white content area. The slide is part of a presentation by Partha Pratim Das, as indicated by the logo and name in the top left. The content area contains a list of bullet points with handwritten annotations in blue ink. The first bullet point states "C++11 introduces rvalue references" with sub-points for syntax and normal references. The second bullet point states "Rvalue references identify objects that may be moved from". The third bullet point is "Reference Binding Rules" with sub-points for important for overloading resolution, "As always:", and "In addition:". The "As always:" sub-point has two sub-points: "Lvalues may bind to lvalue references" and "Rvalues may bind to lvalue references to const". The "In addition:" sub-point has two sub-points: "Rvalues may bind to rvalue references to non-const" and "Lvalues may not bind to rvalue references". A final note states "Otherwise lvalues could be accidentally modified".

- C++11 introduces **rvalue references**
  - Syntax: **T&&**
  - **Normal references** now known as **lvalue references**
    - ▷ **Rvalue references** behave similarly to **Lvalue references**
  - Must be initialized, cannot be rebound, etc.
- **Rvalue references identify objects that may be moved from**
- Reference Binding Rules
  - Important for overloading resolution ✓
  - As always:
    - ▷ **Lvalues** may bind to **lvalue references** ✓
    - ▷ **Rvalues** may bind to **lvalue references to const**
  - In addition:
    - ▷ **Rvalues** may bind to **rvalue references to non-const** ✓
    - ▷ **Lvalues** may **not** bind to **rvalue references** ✓
  - Otherwise lvalues could be accidentally modified

So, to be able to detect that a different kind of reference is introduced in C++ it is known as rvalue reference in the syntax, it just uses the ampersand twice. So, normal references as we have known them is now known as lvalue reference. So, rvalue reference behaves very similar to lvalue reference, rvalue reference will identify if I hold an rvalue reference to an object I know that I can move from that object that is I should move from that object.

So, it can be used in overload resolution, rvalue and lvalue references. lvalue may bind to lvalue references, rvalue may bind to lvalue references to constant that we have seen that you

cannot pass an expression where there is an reference, but you can pass a constant expression, because you need to have that reference to be identifiable by the by the parameter name. In addition, rvalues may refer to rvalue references to non-constant and lvalues may not bind to rvalue references, because if they do, then the possibility of the move or possibility of further changes will be accidental.

(Refer Slide Time: 22:21)

**Rvalue References**

- Examples:
 

```

void f1(const TVec&); // takes const lvalue ref
TVec vt;
f1(vt); // fine (as always)
f1(createTVec()); // fine (as always)
void f2(const TVec&); // #1: takes const lvalue ref
f2(vt); // #2: takes non-const rvalue ref
// lvalue => #1
f2(createTVec()); // both viable, non-const rvalue => #2
void f3(const TVec&&); // #1: takes const rvalue ref
void f3(TVec&&); // #2: takes non-const rvalue ref
f3(vt); // error! lvalue
f3(createTVec()); // both viable, non-const rvalue => #2

```

*Handwritten annotations in blue ink:* checkmarks above f1, f1(vt), and f1(createTVec()); arrows pointing to f2(const TVec&); and f2(vt); a bracket grouping f3(const TVec&&); and f3(TVec&&);

**Rvalue References**

- Examples:
 

```

void f1(const TVec&); // takes const lvalue ref
TVec vt;
f1(vt); // fine (as always)
f1(createTVec()); // fine (as always)
void f2(const TVec&); // #1: takes const lvalue ref
void f2(TVec&&); // #2: takes non-const rvalue ref
f2(vt); // lvalue => #1
f2(createTVec()); // both viable, non-const rvalue => #2
void f3(const TVec&&); // #1: takes const rvalue ref
void f3(TVec&&); // #2: takes non-const rvalue ref
f3(vt); // error! lvalue
f3(createTVec()); // both viable, non-const rvalue => #2

```

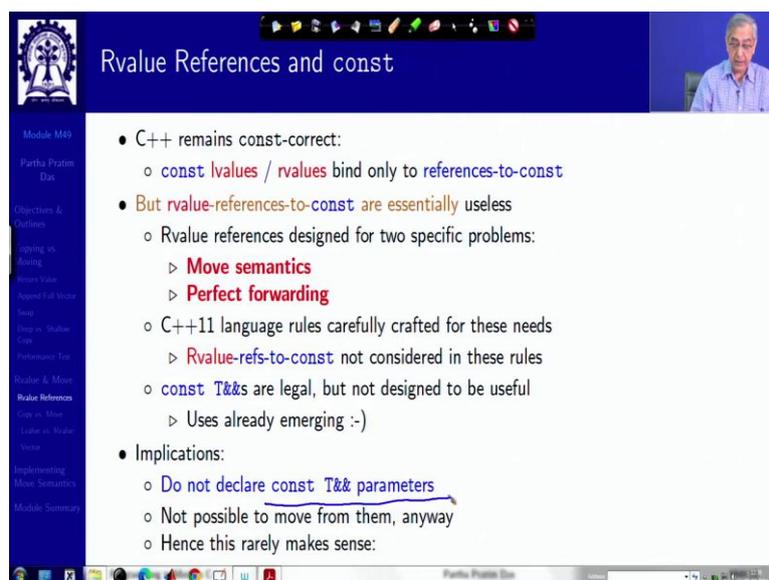
*Handwritten annotations in blue ink:* a bracket grouping f3(const TVec&&); and f3(TVec&&); a checkmark above f3(createTVec());

So, again a couple of example, here, you have this as a constant lvalue reference in f1. So, if you pass vt to that, it is fine, if you pass this to createTVec call to that also is fine. So, what we are doing is are basically taking a you need a lvalue and you have lvalue reference and you have passed it rvalue. So, rvalues can be converted to lvalue constant lvalues.

If I have `f2()` of two times which one which takes constant lvalue reference and other which takes non constant rvalue reference, then if I do `f2(vt)` it will be taking an lvalue. So, it will call this form if I do `f2(createTVec())` then both are viable, it will take the rvalue because it is an rvalue. So, it can take an rvalue reference. So, having this overload allows me to differentiate between whether I can treat it as an lvalue or I can treat it as an rvalue. See more here you have a constant `TVec` rvalue reference and non-constant.

So, a constant rvalue reference and non-constant rvalue reference and you try to pass an lvalue to that now, this is an error because it tells you to actually move the object, but being an lvalue, you cannot move that object. So, this conversion will give you an error. Whereas if you do `f3(createTVec())`, then you have a rvalue, so both of these are possible non-constant one that will be preferred for reason that we will come to very soon.

(Refer Slide Time: 24:40)



The screenshot shows a presentation slide with a blue header containing the title "Rvalue References and const" and a small video feed of the presenter in the top right corner. The slide content is as follows:

- C++ remains const-correct:
  - `const lvalues / rvalues` bind only to `references-to-const`
- But `rvalue-references-to-const` are essentially useless
  - Rvalue references designed for two specific problems:
    - ▷ **Move semantics**
    - ▷ **Perfect forwarding**
  - C++11 language rules carefully crafted for these needs
    - ▷ `Rvalue-refs-to-const` not considered in these rules
  - `const T&&` are legal, but not designed to be useful
    - ▷ Uses already emerging :-)
- Implications:
  - Do not declare `const T&&` parameters
  - Not possible to move from them, anyway
  - Hence this rarely makes sense:

Now, the question is, what is what about const-ness? The `const lvalue` or `rvalue` bind only to `references-to-const` but `rvalue references to const` are essentially useless. Why? Because why did we identify rvalue because we should be able to move. Now, if we are saying that I have a reference to an object which is an rvalue, but to be treated as a constant, then naturally I cannot move it. Because to be able to move, I need to make changes in that source object. So, it kind of contradicts the semantics of it is it is legal, though its semantic use is really not understood so well till this time.

But it is typical that `rvalue reference to const` is not considered right now, but it is not illegal to write it will, but it will not let you do that move that is the consequence. So, you should not

do that. And this rvalue references solve two specific problems, which is move semantics and perfect forwarding. So, remember this part that you should never declare the const reference rvalue reference parameter.

(Refer Slide Time: 26:10)

The slide is titled "Distinguishing Copying from Moving" and features a small video inset of a speaker in the top right corner. The main content includes a C++ class definition for 'Widget' and a list of explanatory bullet points.

```
class Widget { public:
    Widget(const Widget&); // copy constructor
    Widget(Widget&&) noexcept; // move constructor
    Widget& operator=(const Widget&); // copy assignment op
    Widget& operator=(Widget&&) noexcept; // move assignment op
    ...
};
Widget createWidget(); // factory function
Widget w1;
Widget w2 = w1; // lvalue src => copy required
w2 = createWidget(); // rvalue src => move okay
w1 = w2; // lvalue src => copy required
```

- Overloading exposes move-instead-of-copy opportunities:
- Move operations need not be `noexcept`, but it is preferable
  - Moves should be fast, and `noexcept` => more optimizable
  - Some contexts require `noexcept` moves (for example, `std::vector::push_back`)
  - Move operations often have natural `noexcept` implementations
- We declare move operations `noexcept` by default

So, differentiate this we have a widget this is a typical copy constructor with lvalue. We write another with rvalue and we call that a move constructor. So, the difference is in the copy constructor, the source object state will be copied and the object created in the move constructor source object state will not be copied it will be moved and we need noexcept for optimization.

Similarly, this is copy assignment operator, this is move assignment operator, in the copy assignment, you make a copy of the source object in a move assignment you do not make that. So, if we have this function if we have the w1, we are creating w2 with that, so, w1 is an lvalue. So, copy is required. But if I assign create widget result into w2 then the result is an rvalue. So, move is okay. Whereas, if I directly assigned w2 to w1 then a copy will be required. So, that is the basic difference.

(Refer Slide Time: 27:30)

### Copy vs. Move : Lvalue vs. Rvalue

```

class A { public: A() { std::cout << "Defa Ctor" << endl; } // Defa Constructor
A(const A&) { std::cout << "Copy Ctor" << endl; } // Copy Constructor
A(A&&) noexcept { std::cout << "Move Ctor" << endl; } // Move Constructor
A& operator=(const A&) { cout << "Copy =" << endl; return *this; } // Copy =
A& operator=(A&&) noexcept { cout << "Move =" << endl; return *this; } // Move =
friend A operator+(const A& a, const A& b) { A t; return t; } // Temp. obj. ret.-by-value
};
  
```

	Only Copy		Copy & Move	
	Debug	Release	Debug	Release
A a; // lvalue	Defa Ctor	Defa Ctor	Defa Ctor	Defa Ctor
A b = a; // lvalue	Copy Ctor	Copy Ctor	Copy Ctor	Copy Ctor
A c = a + b; // rvalue	Defa Ctor	Defa Ctor	Defa Ctor	Defa Ctor
// RVO in a + b for release build	Copy Ctor	// RVO	Move Ctor	// RVO
A d = std::move(a); // rvalue	Copy Ctor	Copy Ctor	Move Ctor	Move Ctor
b = a; // lvalue	Copy =	Copy =	Copy =	Copy =
c = a + b; // rvalue	Defa Ctor	Defa Ctor	Defa Ctor	Defa Ctor
	Copy Ctor	// RVO	Move Ctor	// RVO
	Copy =	Copy =	Move =	Move =

- Return Value Optimization (RVO) eliminates the temp. obj. created to hold a function's return value
- std::move(t) produces a rvalue from t to indicate that the object t may be moved from

### Copy vs. Move : Lvalue vs. Rvalue

```

class A { public: A() { std::cout << "Defa Ctor" << endl; } // Defa Constructor
A(const A&) { std::cout << "Copy Ctor" << endl; } // Copy Constructor
A(A&&) noexcept { std::cout << "Move Ctor" << endl; } // Move Constructor
A& operator=(const A&) { cout << "Copy =" << endl; return *this; } // Copy =
A& operator=(A&&) noexcept { cout << "Move =" << endl; return *this; } // Move =
friend A operator+(const A& a, const A& b) { A t; return t; } // Temp. obj. ret.-by-value
};
  
```

	Only Copy		Copy & Move	
	Debug	Release	Debug	Release
A a; // lvalue	Defa Ctor	Defa Ctor	Defa Ctor	Defa Ctor
A b = a; // lvalue	Copy Ctor	Copy Ctor	Copy Ctor	Copy Ctor
A c = a + b; // rvalue	Defa Ctor	Defa Ctor	Defa Ctor	Defa Ctor
// RVO in a + b for release build	Copy Ctor	// RVO	Move Ctor	// RVO
A d = std::move(a); // rvalue	Copy Ctor	Copy Ctor	Move Ctor	Move Ctor
b = a; // lvalue	Copy =	Copy =	Copy =	Copy =
c = a + b; // rvalue	Defa Ctor	Defa Ctor	Defa Ctor	Defa Ctor
	Copy Ctor	// RVO	Move Ctor	// RVO
	Copy =	Copy =	Move =	Move =

- Return Value Optimization (RVO) eliminates the temp. obj. created to hold a function's return value
- std::move(t) produces a rvalue from t to indicate that the object t may be moved from

### Copy vs. Move : Lvalue vs. Rvalue

```

class A { public: A() { std::cout << "Defa Ctor" << endl; } // Defa Constructor
A(const A&) { std::cout << "Copy Ctor" << endl; } // Copy Constructor
A(A&&) noexcept { std::cout << "Move Ctor" << endl; } // Move Constructor
A& operator=(const A&) { cout << "Copy =" << endl; return *this; } // Copy =
A& operator=(A&&) noexcept { cout << "Move =" << endl; return *this; } // Move =
friend A operator+(const A& a, const A& b) { A t; return t; } // Temp. obj. ret.-by-value
};
  
```

	Only Copy		Copy & Move	
	Debug	Release	Debug	Release
A a; // lvalue	Defa Ctor	Defa Ctor	Defa Ctor	Defa Ctor
A b = a; // lvalue	Copy Ctor	Copy Ctor	Copy Ctor	Copy Ctor
A c = a + b; // rvalue	Defa Ctor	Defa Ctor	Defa Ctor	Defa Ctor
// RVO in a + b for release build	Copy Ctor	// RVO	Move Ctor	// RVO
A d = std::move(a); // rvalue	Copy Ctor	Copy Ctor	Move Ctor	Move Ctor
b = a; // lvalue	Copy =	Copy =	Copy =	Copy =
c = a + b; // rvalue	Defa Ctor	Defa Ctor	Defa Ctor	Defa Ctor
	Copy Ctor	// RVO	Move Ctor	// RVO
	Copy =	Copy =	Move =	Move =

- Return Value Optimization (RVO) eliminates the temp. obj. created to hold a function's return value
- std::move(t) produces a rvalue from t to indicate that the object t may be moved from

And so, here I have given a couple worked out a couple of examples in detail to show you how does this copy and move constructor as well as assignment operator work. So, here is a class A which has a default constructor, it has a copy constructor, it has a move constructor, copy assignment operator, move assignment operator and binary operator which does not as such do anything it just creates a temporary the result object and returns that object in between of course, you will have the actual computation to be done.

Now, if you look at you will see that if you do only copy, then the calls will happen like this, forget about the release part initial. So, A is declared. So, default constructor if you have only copy a is assigned to b, the lvalue copy constructor a plus b assigned to c. So, what will happen first this operator will be called. So, A t will be created and then it will be copied.

So, A t is created and then it is copied if I put this forget about this now, suppose I do an assignment of a to b the copy assignment suppose I do a + b and assign it to c default construction of t copy construction for doing the return by copy return by value you need a copy construction and finally the copy assignment happening here. So, this is what happens if you just have the copy constructor and copy assignment operator.

Now, let us say we have move constructor and move assignment operator also. We have not commented them out we also so this is same. This is same because it I have an lvalue. But when I create this I have a default constructor for this t but after that, for the return earlier I was having a copy construction. Now, I have a move construction because this return value is a temporary object, it is an lvalue.

So, I can just move the state, I do not need to copy the state. Then I have `std::move`, which says that take this lvalue and a is an lvalue, but convert that to, rvalue converted to an rvalue reference. So, you can, with that, you can, you will have a move constructor, which will forcibly convert the lvalue to an rvalue and we will move.

Similarly, in this assignment, I have an lvalue. So, copy copy. In this computation, I have a default, for constructing this, I have a move constructor for the return and then move assignment operator. So, this is how you start to benefit because you get a lot more of move construction and move assignment happening, which are much less expensive than you, actually.

(Refer Slide Time: 31:05)

```

// C++ program with the copy and the move constructors
class C { int* data; // Declare the raw pointer as the data member of class
public:
    C(int d) { // Constructor
        data = new int(d); // Declare object in the heap
        cout << "Ctor: " << d << endl;
    };
    C(const C& src) : myClass{ *src.data } { // Copy Constructor by delegation
        // Copying the data by making deep copy
        cout << "C-Ctor: " << *src.data << endl;
    }
    C(C&& src) : data{ src.data } noexcept { // Move Constructor
        cout << "M-Ctor: " << *src.data << endl;
        src.data = nullptr;
    }
    ~C() { // Destructor
        if (data != nullptr) // If pointer is not pointing to nullptr
            cout << "Dtor: " << *data << endl;
        else // If pointer is pointing to nullptr
            cout << "Dtor: " << "nullptr" << endl;
        delete data; // Free up the memory assigned to the data member of the object
    }
};
    
```

So, here I have given this detailed explanation for your self-study. Here is another example also using a class C, which has a resource in terms of an int pointer. So, with that you have a default constructor, copy constructor, move constructor and destructor.

(Refer Slide Time: 31:34)

```

int main() { vector<C> v; // Create vector of C Class
            v.push_back(C{10}); // Inserting object of C class
            v.push_back(C{20});
        }
    
```

	Only Copy			Copy & Move		
	Debug	Release	Remark	Debug	Release	Remark
{ vector<C> v; ✓						
// v.size()		Ctor: 10	Ctor: 10	Ctor: 10	Ctor: 10	
v.push_back(C{10});	Ctor: 10	Ctor: 10	// Delegate	M-Ctor: 10	M-Ctor: 10	// Add 10 to v
// v.size() = 1	C-Ctor: 10	C-Ctor: 10	// C-Ctor	Dtor: nullptr	Dtor: nullptr	
	Dtor: 10	Dtor: 10				
// Move C{10}	Ctor: 20	Ctor: 20		Ctor: 20	Ctor: 20	
// for C{20}	C-Ctor: 10	C-Ctor: 10	// Delegate	M-Ctor: 10	M-Ctor: 10	// Move 10 in v
v.push_back(C{20});	C-Ctor: 10	C-Ctor: 10	// C-Ctor	Dtor: nullptr	Dtor: nullptr	
// v.size() = 2	Ctor: 20	Ctor: 20	// Delegate	M-Ctor: 20	M-Ctor: 20	// Add 20 to v
	C-Ctor: 20	C-Ctor: 20	// C-Ctor	Dtor: nullptr	Dtor: nullptr	
	Dtor: 20	Dtor: 20				
// End of scope	Dtor: 10	Dtor: 10	// Release	Dtor: 10	Dtor: 10	// Release
// Release v	Dtor: 20	Dtor: 20	// Vector v	Dtor: 20	Dtor: 20	// Vector v

### Copy vs. Move : Vector

```

int main() { vector<C> v; // Create vector of C Class
            v.push_back(C{10}); // Inserting object of C class
            v.push_back(C{20});
        }
    
```

	Only Copy			Copy & Move		
	Debug	Release	Remark	Debug	Release	Remark
{ vector<C> v;						
// v.size() = 0	Ctor: 10	Ctor: 10		Ctor: 10	Ctor: 10	
v.push_back(C{10});	Ctor: 10	Ctor: 10	// Delegate	M-Ctor: 10	M-Ctor: 10	// Add 10 to v
// v.size() = 1	C-Ctor: 10	C-Ctor: 10	// C-Ctor	Dtor: nullptr	Dtor: nullptr	
	Dtor: 10	Dtor: 10				
// Move C{10}	Ctor: 20	Ctor: 20		Ctor: 20	Ctor: 20	
// for C{20}	Ctor: 10	Ctor: 10	// Delegate	M-Ctor: 10	M-Ctor: 10	// Move 10 in v
v.push_back(C{20});	C-Ctor: 10	C-Ctor: 10	// C-Ctor	Dtor: nullptr	Dtor: nullptr	
// v.size() = 2	Dtor: 10	Dtor: 10				
	Ctor: 20	Ctor: 20	// Delegate	M-Ctor: 20	M-Ctor: 20	// Add 20 to v
	C-Ctor: 20	C-Ctor: 20	// C-Ctor	Dtor: nullptr	Dtor: nullptr	
	Dtor: 20	Dtor: 20				
// End of scope	Dtor: 10	Dtor: 10	// Release	Dtor: 10	Dtor: 10	// Release
} // Release v	Dtor: 20	Dtor: 20	// Vector v	Dtor: 20	Dtor: 20	// Vector v

### Copy vs. Move : Vector

```

int main() { vector<C> v; // Create vector of C Class
            v.push_back(C{10}); // Inserting object of C class
            v.push_back(C{20});
        }
    
```

	Only Copy			Copy & Move		
	Debug	Release	Remark	Debug	Release	Remark
{ vector<C> v;						
// v.size() = 0	Ctor: 10	Ctor: 10		Ctor: 10	Ctor: 10	
v.push_back(C{10});	Ctor: 10	Ctor: 10	// Delegate	M-Ctor: 10	M-Ctor: 10	// Add 10 to v
// v.size() = 1	C-Ctor: 10	C-Ctor: 10	// C-Ctor	Dtor: nullptr	Dtor: nullptr	
	Dtor: 10	Dtor: 10				
// Move C{10}	Ctor: 20	Ctor: 20		Ctor: 20	Ctor: 20	
// for C{20}	Ctor: 10	Ctor: 10	// Delegate	M-Ctor: 10	M-Ctor: 10	// Move 10 in v
v.push_back(C{20});	C-Ctor: 10	C-Ctor: 10	// C-Ctor	Dtor: nullptr	Dtor: nullptr	
// v.size() = 2	Dtor: 10	Dtor: 10				
	Ctor: 20	Ctor: 20	// Delegate	M-Ctor: 20	M-Ctor: 20	// Add 20 to v
	C-Ctor: 20	C-Ctor: 20	// C-Ctor	Dtor: nullptr	Dtor: nullptr	
	Dtor: 20	Dtor: 20				
// End of scope	Dtor: 10	Dtor: 10	// Release	Dtor: 10	Dtor: 10	// Release
} // Release v	Dtor: 20	Dtor: 20	// Vector v	Dtor: 20	Dtor: 20	// Vector v

And using that, if you create a vector and start doing push back, the main thing you will see is certainly, if you want to do a push\_back, what will happen, initially, the vector has size 0, its got nothing, just this vector. Now, you do a push\_back. So, it has you have to first construct, you have to first construct the temporary object C(10), that will have to be constructed.

And then you have to do a copy construction. So, here, the copy construction texts, uses the default constructor to construct and that that copied value will be put in the vector and the original lvalue, this one will get destroyed. Because this is an lvalue this is an rvalue, the temporary gets destroyed, that is fine.

Now, if you want to insert the second one, here, the vector is already full. So, as we have seen, we have to make a move. So, create an object for 20. And then you have to create a

copy of the existing value 10. And then destroy the earlier value, create an existing copy of existing, destroy the earlier value, and then make a copy of 20.

And you will have that so you have that additional task. In contrast, if you had just done move, then instead of copy construction here, you can just do with move construction, instead of doing a release, you do not need to do that, because you have already taken that resource. Similar thing, you will benefit in terms of moving the existing object instead of copying it. So, that is a basic advantage that you gain.

(Refer Slide Time: 33:33)

The slide is titled "Copy vs. Move : Vector: Explanation" and features a small video inset of the presenter in the top right corner. The main content is a C++ code snippet for a class `C` and a `vector<C>` `v`. The code includes a default constructor, a copy constructor, a move constructor, and a destructor. It also shows two `push_back` operations: one for `C{10}` and one for `C{20}`. The slide provides a detailed, line-by-line explanation of the execution flow, including the construction of temporary objects, the use of `rvalue` and `lvalue` references, and the destruction of objects that are no longer needed. The execution flow is as follows:

- `vector<C> v;` ⇒ `v` is default constructed as an empty vector of `C`. `v.size() = 0`
- `v.push_back(C{10});` ⇒ Construct `C{10}`, copy/move & place in `v[0]`, and destruct. `v.size() = 1`
  - `Ctor: 10` /\* Ctor for `C{10}` ⇒ `t10`, Temp.obj. and `rvalue` \*/ `Ctor: 10`
  - `Ctor: 10` /\* delegated from C-Ctor \*/
  - `C-Ctor: 10` /\* C-Ctor for `t10` ⇒ `v10 = v[0]`, `lvalue` to place in `v` \*/ `M-Ctor: 10`
  - `Dtor: 10` /\* Dtor for `t10` \*/ `Dtor: nullptr`
- `v.push_back(C{20});` ⇒ Construct `C{20}`. Copy/move `v[0]` and destruct old `v[0]`. Copy/move & place `C{20}` in `v[1]`, and destruct. `v.size() = 2`
  - `Ctor: 20` /\* Ctor for `C{20}` ⇒ `t20`, Temp.obj. & `rvalue` \*/ `Ctor: 20`
  - `Ctor: 10` /\* delegated from C-Ctor \*/
  - `C-Ctor: 10` /\* C-Ctor for `v10` ⇒ `v10_1 = v[0]`, `lvalue` to place in `v` \*/ `M-Ctor: 10`
  - `Dtor: 10` /\* Dtor for `v10` \*/ `Dtor: nullptr`
  - `Ctor: 20` /\* delegated from C-Ctor \*/
  - `C-Ctor: 20` /\* C-Ctor for `t20` ⇒ `v20 = v[1]`, `lvalue` to place in `v` \*/ `M-Ctor: 20`
  - `Dtor: 20` /\* Dtor for `t20` \*/ `Dtor: nullptr`
- `}` ⇒ Automatic `v` going out of scope. Destruct `v[0]` and `v[1]`
  - `Dtor: 10` /\* Dtor for `v10_1 = v[0]` \*/ `Dtor: 10`
  - `Dtor: 20` /\* Dtor for `v20 = v[1]` \*/ `Dtor: 20`

The slide footer includes "Programming in Modern C++", "Partha Pratim Das", and "M49 25".

And again, I have given a very line by line step by step explanation for what is going on. So, use it in your self-study to get more insight.

(Refer Slide Time: 33:45)

**Copy vs. Move : Vector: Performance Trade-off**

- Since, class `C` has no default constructor, `vector<C> v` is constructed as an empty vector with `v.size() = 0`. Hence, every time a `push_back` (insert at the `end()`) is done, we need to expand the allocation of the vector by copying / moving the existing elements
- For `v.push_back(C{10})`, `C{10}` is constructed as a temporary object (**rvalue**). So, it needs to be copied / moved for `push_back` to the vector as **lvalue**. Same for `v.push_back(C{20})`
- Further, for `v.push_back(C{20})`, fresh allocation and copy / movement of existing element is needed for `push_back`
- To `push_back` the  $n^{\text{th}}$  element, we need to copy / move existing  $n - 1$  elements. This means:
  - **Using Copy**
    - ▷  $n - 1$  resource allocations (**new int**) and de-allocations (**delete**)
    - ▷ For  $n$  elements this adds to  $\sum_{i=0}^{n-1} i = \frac{n(n-1)}{2} = O(n^2)$  total allocations / de-allocations
  - **Using Move**
    - ▷ **0** resource allocations (**new int**) and de-allocations (**delete**)
    - ▷ For  $n$  elements this adds to  $\sum_{i=0}^{n-1} 0 = 0$  total allocations / de-allocations. **Huge Benefit!**

Now, if we just try to analyze, what, what advantage do we get. So, you can see that for the first copy, we can just we need to copy that object temporarily. In the vector for the second, I need to move the earlier object because the vector is full, and then copy the new one. For the third, I need to move, copy both of these existing objects, and then copy the new one. So, using copy every time for inserting the  $i$  plus first object I need to make  $i$  copies.

So, if I look at insertion of  $n$  objects, then there are order  $n$  square, total copy and release. That is allocation and de-allocation I have to do. Instead, if I use if I could use move, then whatever exists. I am just moving. I am not creating any new `int*` data resource, neither I am releasing them. So, it is simply 0. So, a huge huge benefit. And that is the incentive of why we should use the move semantics.

(Refer Slide Time: 34:50)

Module M49  
Partha Pratim Das

## Implementing Move Semantics

- Move operations take source's value, but leave source in valid state:

```
class Widget {  
public:  
    Widget(Widget&& rhs) noexcept : pds(rhs.pds) // take source's value  
    { rhs.pds = nullptr; } // leave source in valid state  
  
    Widget& operator=(Widget&& rhs) noexcept {  
        delete pds; // get rid of current value  
        pds = rhs.pds; // take source's value  
        rhs.pds = nullptr; // leave source in valid state  
        return *this;  
    }  
    ...  
  
private:  
    struct DataStructure;  
    DataStructure *pds;  
};
```

Diagram: :Widget → :DataStructure

- Easy for built-in types (for example, pointers). Trickier for UDTs...

Now, implementing the move semantics is simple. So, I will just give you a glimpse here. And we will discuss more again in the next discussion that if you have a move constructor, then from the source, all that you are doing is you are just moving the source, take the sources value. And then you release it, let us set it to the null pointer. Similarly, free is an assignment operator, you can do the same thing release what you had copy and set the source to null. This will be simple.

(Refer Slide Time: 35:39)

Module M49  
Partha Pratim Das

## Implementing Move Semantics

- Widget's move operator= fails given move-to-self:  

```
Widget w;  
w = std::move(w); // undefined behavior!
```
- It may be harder to recognize, of course:  

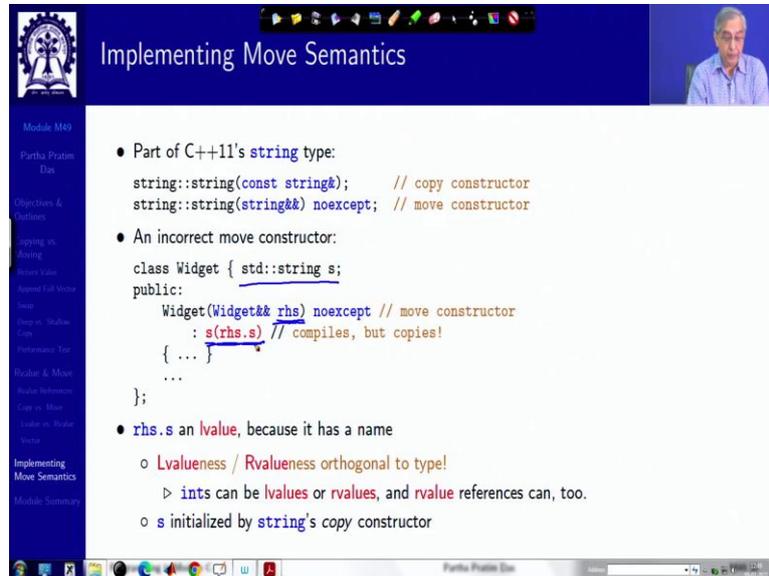
```
Widget *pw1, *pw2;  
...  
*pw1 = std::move(*pw2); // undefined if pw1 == pw2
```
- C++11 condones this
  - In contrast to copy operator=
- A fix is simple, if you are inclined to implement it:  

```
Widget& Widget::operator=(Widget&& rhs) noexcept {  
    if (this == &rhs) return *this; // or assert(this != &rhs);  
    ...  
}
```

Now, the problem with this is a problem we had seen earlier in the copy assignment operator is that the self copy is a problem. So, the same thing will happen in terms of move also, that it is correct undefined behavior if you are moving from this object to itself. So, also in move,

what you will have to do is to check that your source object and the target object are not the same. You already know this.

(Refer Slide Time: 36:10)



The slide is titled "Implementing Move Semantics" and features a video feed of the presenter in the top right corner. The main content includes:

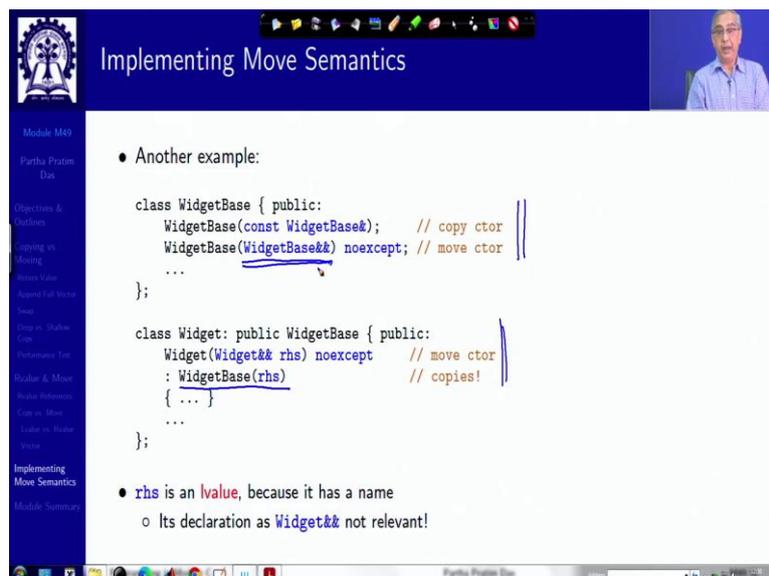
- Part of C++11's `string` type:  

```
string::string(const string&); // copy constructor  
string::string(string&&) noexcept; // move constructor
```
- An incorrect move constructor:  

```
class Widget { std::string s;  
public:  
    Widget(Widget&& rhs) noexcept // move constructor  
        : s(rhs.s) // compiles, but copies!  
    { ... }  
    ...  
};
```
- `rhs.s` is an `lvalue`, because it has a name
  - `Lvalueness / Rvalueness` orthogonal to `type!`
    - ▷ `ints` can be `lvalues` or `rvalues`, and `rvalue` references can, too.
  - `s` initialized by `string`'s copy constructor

Rest of it is simple. But remember, this I will leave as questions in this, I will answer them in the next module, because I want you to think about this, that in a move constructor from moving from a source, I say I have a string `s` and I have written this this will compile. But this will copy you have to find the justification for why it will copy and how to solve that problem.

(Refer Slide Time: 36:48)



The slide is titled "Implementing Move Semantics" and features a video feed of the presenter in the top right corner. The main content includes:

- Another example:  

```
class WidgetBase { public:  
    WidgetBase(const WidgetBase&); // copy ctor  
    WidgetBase(WidgetBase&&) noexcept; // move ctor  
    ...  
};  
  
class Widget: public WidgetBase { public:  
    Widget(Widget&& rhs) noexcept // move ctor  
        : WidgetBase(rhs) // copies!  
    { ... }  
    ...  
};
```
- `rhs` is an `lvalue`, because it has a name
  - Its declaration as `Widget&&` not relevant!

Similarly, if I take a different one, there is a there is a base type. And there is a derived type. And I am passing this RHS as a parameter to the base type, and I expect the move construction to happen, but you will find that it is not happening it will it will still copy. So, think over that as to reasoning as to what is the problem here particularly that the variables we are copying from our lvalues. So, that is that is this knave implementation will not work. So, we will have to see what more we will have to do.

(Refer Slide Time: 37:31)

The image shows a presentation slide titled "Module Summary" for "Module M49" by "Partha Pratim Das". The slide content is as follows:

- Understood the difference between Copying and Moving
- Understood the difference between Lvalue and Rvalue
- Learnt the advantages of Move in C++ using
  - Rvalue Reference
  - Move Semantics
  - Copy / Move Constructor / Assignment
  - Implementation of Move Semantics

The slide footer contains "Programming in Modern C++", "Partha Pratim Das", and "M49.32".

And I will talk about that, but before that, I would expect that you have thought through this. So, here we have in this module, we have introduced something very very fundamentally important there is difference between copying and moving particularly lvalue and rvalue and the advantage of moving in C++ the rvalue semantics and the move rvalue reference and the move semantics. Thank you very much for your attention and we meet in the next module.