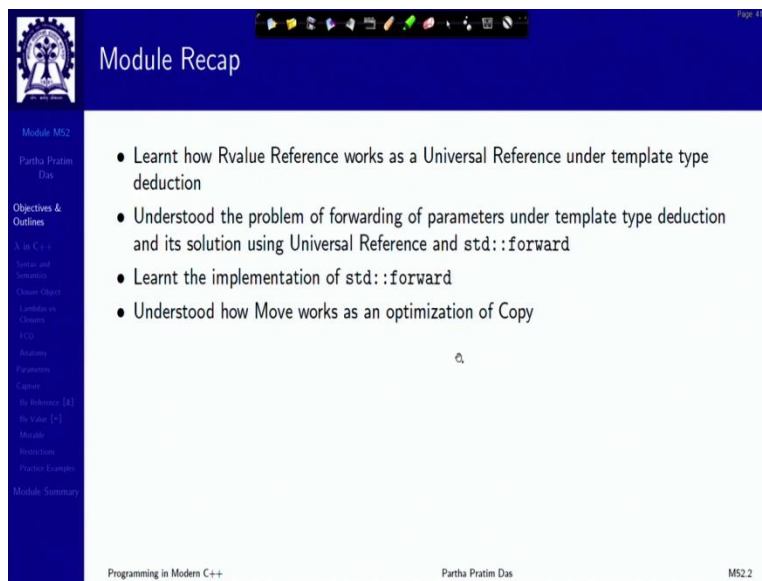


**Programming in Modern C++**  
**Professor. Partha Pratim Das**  
**Department of Computer Science and Engineering**  
**Indian Institute of Technology, Kharagpur**  
**Lecture 52**  
**C++11 and beyond: General Features: Part 7**  
**Lambda in C++/1**

Welcome to Programming in Modern C++ we are in week 11. And I am going to discuss module 52.

(Refer Slide Time: 00:36)



The screenshot shows a presentation slide titled "Module Recap" with a blue header. On the left, there is a vertical navigation menu with the following items: "Module M52", "Partha Pratim Das", "Objectives & Outlines", "3 in C++", "Value and Reference", "Class Object", "Lambda Expressions", "RVO", "Move Semantics", "Copy Semantics", "Move Semantics (R)", "Move Semantics (L)", "Move Semantics", "Move Semantics", "Module Summary". The main content area contains a bulleted list of four items:

- Learnt how Rvalue Reference works as a Universal Reference under template type deduction
- Understood the problem of forwarding of parameters under template type deduction and its solution using Universal Reference and `std::forward`
- Learnt the implementation of `std::forward`
- Understood how Move works as an optimization of Copy

At the bottom of the slide, there is a footer with the text "Programming in Modern C++", "Partha Pratim Das", and "M522".

In the last module, we have concluded the discussions on rvalue references as universal reference, we have talked about the perfect forwarding solution to the forwarding problem using `std::forward` and discussed certain points about move as an optimization of copy.

(Refer Slide Time: 00:59)

The slide is titled "Module Objectives" and features a blue header with a logo on the left and a small video feed of the presenter on the right. The main content area is white and contains a bulleted list of objectives. A vertical navigation menu is on the left side of the slide, and a footer with course information is at the bottom.

- To understand  $\lambda$  expressions (unnamed function objects) in C++
  - Closure Objects
  - Parameters
  - Capture

Programming in Modern C++ Partha Pratim Das M52.3

In the current module we will introduce another very very important general feature of C++11 that is to understand Lambda expressions in C++ that is in C++11 and specifically what are closure objects, what are the consequences in terms of parameters and capture.

(Refer Slide Time: 01:22)

The slide is titled "Module Outline" and features a blue header with a logo on the left and a small video feed of the presenter on the right. The main content area is white and contains a numbered list of topics. A vertical navigation menu is on the left side of the slide, and a footer with course information is at the bottom.

- 1  $\lambda$  in C++11, C++14, C++17, C++20
  - Syntax and Semantics
  - Closure Object
    - Lambdas vs. Closures
    - First Class Object
    - Anatomy
  - Parameters
  - Capture
    - By Reference [&]
    - By Value [=]
    - Mutable
    - Restrictions
    - Practice Examples
- 2 Module Summary

Programming in Modern C++ Partha Pratim Das M52.4

This is the outline which will be available on your left panel all the time as you know.



(Refer Slide Time: 02:33)

**λ in C++: Closure Object**

- A *λ expression* is a mechanism for specifying a *function object* or *functor* (Recall **Module 40**)
- The primary use for a  $\lambda$  is to specify a simple action to be performed by some function
- For example, consider a remainder operation `rem` that computes  $m \% n$ , that is,  $m$  modulo  $n$ . It has type `int -> int`. To write `rem` in C++, we define a function / functor:

<pre>int n = 7; int rem(int m) // Function { return m % n; } // Uses n in context</pre>	<pre>int n = 7; struct remainder {           // Functor     int mod;                // State     remainder(int n): mod(n) { // Ctor (n from context)     int operator()(int m)    // Function call operator         { return m % mod; } // Body };</pre>
<pre>rem(23); // 2</pre>	<pre>struct remainder rem(n); rem(23); // 2</pre>

$\lambda$ : `auto rem = [n](int m) -> int { return m % n; } // Captures n from context`  
`rem(23); // 2`

- Note that `[n]` Captures `n` from context to close `rem` and create the **Closure Object** in C++

**λ in C++: Closure Object**

- A *λ expression* is a mechanism for specifying a *function object* or *functor* (Recall **Module 40**)
- The primary use for a  $\lambda$  is to specify a simple action to be performed by some function
- For example, consider a remainder operation `rem` that computes  $m \% n$ , that is,  $m$  modulo  $n$ . It has type `int -> int`. To write `rem` in C++, we define a function / functor:

<pre>int n = 7; int rem(int m) // Function { return m % n; } // Uses n in context</pre>	<pre>int n = 7; struct remainder {           // Functor     int mod;                // State     remainder(int n): mod(n) { // Ctor (n from context)     int operator()(int m)    // Function call operator         { return m % mod; } // Body };</pre>
<pre>rem(23); // 2</pre>	<pre>struct remainder rem(n); rem(23); // 2</pre>

$\lambda$ : `auto rem = [n](int m) -> int { return m % n; } // Captures n from context`  
`rem(23); // 2`

- Note that `[n]` Captures `n` from context to close `rem` and create the **Closure Object** in C++

*Handwritten annotations:* A blue bracket groups the `int n = 7;` line and the `remainder` struct definition. A blue arrow points from the `int n = 7;` line to the `mod(n)` parameter in the constructor of the `remainder` struct.

So, lambdas are typically short form of lambda expression it is considered as an expression which specifies a function object or a functor you would recall that we had discussed about function objects, these are classes which has the function call operator overloaded this was discussed in module 40. If you are not very clear about functors please go back and revise module 40 because, this entire module and the next will rely heavily on the understanding of functors per se.

Now, the primary use of a lambda is to specify a simple action to be performed by some function. So, let us say we want to find out the remainder of a value of a number by another

number that is if you divide how much will be the remainder the modular operation which is from int to int.

So, in terms of C++ this rem operation can be written in 2 forms, I can write a function m rem which takes m as a parameter and uses the divisor n as a global variable from the context or I can define a functor say remainder this I put a data member modulus mod, which is the value by which the module operation will be done. So, the constructor sets the value of this mod which I take from the global again I can pass it directly as well. And then I overload the function call operator with the actual.

(Refer Slide Time: 04:55)

**λ in C++: Closure Object**

- A *λ expression* is a mechanism for specifying a *function object* or *functor* (Recall **Module 40**)
- The primary use for a *λ* is to specify a simple action to be performed by some function
- For example, consider a remainder operation **rem** that computes  $m \% n$ , that is, *m modulo n*. It has type `int -> int`. To write **rem** in C++, we define a function / functor:

<pre>int n = 7; int rem(int m) // Function { return m % n; } // Uses n in context</pre>	<pre>int n = 7; struct remainder { // Functor     int mod; // State     remainder(int h): mod(n) { } // Ctor (n from context)     int operator()(int m) // Function call operator     { return m % mod; } // Body };</pre>
<pre>rem(23); // 2</pre>	<pre>struct remainder rem(n); rem(23); // 2</pre>

`λ: auto rem = [n](int m) -> int { return m % n; } // Captures n from context`  
`rem(23); // 2`

- Note that `[n]` Captures *n* from context to close **rem** and create the **Closure Object** in C++

**λ in C++: Closure Object**

- A *λ expression* is a mechanism for specifying a *function object* or *functor* (Recall **Module 40**)
- The primary use for a *λ* is to specify a simple action to be performed by some function
- For example, consider a remainder operation **rem** that computes  $m \% n$ , that is, *m modulo n*. It has type `int -> int`. To write **rem** in C++, we define a function / functor:

<pre>int n = 7; int rem(int m) // Function { return m % n; } // Uses n in context</pre>	<pre>int n = 7; struct remainder { // Functor     int mod; // State     remainder(int n): mod(n) { } // Ctor (n from context)     int operator()(int m) // Function call operator     { return m % mod; } // Body };</pre>
<pre>rem(23); // 2</pre>	<pre>struct remainder rem(n); rem(23); // 2</pre>

`λ: auto rem = [n](int m) -> int { return m % n; } // Captures n from context`  
`rem(23); // 2`

- Note that `[n]` Captures *n* from context to close **rem** and create the **Closure Object** in C++

So, here this is the actual function computation here this is the actual function computation, so, I overload the function call operator with this. So, if it is a function I directly call it as a function, if it is a functor then I create an instance of the functor given the value of n to go here that is to be set as mod. And then I call the functor which actually calls the function call operator recap off this is what we have.

Now, with this I can write something which is known as a lambda expression. Its syntax is somewhat very similar to function but little bit different. Let us say this is my parameter. The way to tell the compiler that I am writing a lambda is this introducer which is a pair of square brackets, there may be something inside, this some captured maybe inside this, some captured may not be inside this, but this introduce that tells that lambda is going to start. Since, we do not have a function in function, what do we have, we have the name, we have the return type, and then we have the parameters, list of parameters and the function body for components.

(Refer Slide Time: 06:35)

**λ in C++: Closure Object**

- A *λ expression* is a mechanism for specifying a *function object* or *functor* (Recall **Module 40**)
- The primary use for a *λ* is to specify a simple action to be performed by some function
- For example, consider a remainder operation **rem** that computes  $m \% n$ , that is,  $m$  modulo  $n$ . It has type `int -> int`. To write **rem** in C++, we define a function / functor:

```

int n = 7;
int rem(int m) // Function
{ return m % n; }
// Uses n in context

int n = 7;
struct remainder { // Functor
    int mod; // State
    remainder(int n): mod(n) { } // Ctor (n from context)
    int operator()(int m) // Function call operator
    { return m % mod; } // Body
};

rem(23); // 2

struct remainder rem(n);
rem(23); // 2

λ: auto rem = [n](int m) -> int { return m % n; } // Captures n from context
rem(23); // 2

```

- Note that `[n]` Captures `n` from context to close `rem` and create the **Closure Object** in C++

**λ in C++: Closure Object**

- A *λ expression* is a mechanism for specifying a *function object* or *functor* (Recall **Module 40**)
- The primary use for a *λ* is to specify a simple action to be performed by some function
- For example, consider a remainder operation *rem* that computes  $m \% n$ , that is, *m* modulo *n*. It has type `int -> int`. To write *rem* in C++, we define a function / functor:

<pre>int n = 7; int rem(int m) // Function { return m % n; } // Uses n in context</pre>	<pre>int n = 7; struct remainder {           // Functor     int mod;                 // State     remainder(int n): mod(n) { // Ctor (n from context)     int operator()(int m)    // Function call operator         { return m % mod; } // Body };</pre>
<pre>rem(23); // 2</pre>	<pre>struct remainder rem(n); rem(23); // 2</pre>
<pre>λ: auto rem = [n](int m) -&gt; int { return m % n; } // Captures n from context rem(23); // 2</pre>	

- Note that `[n]` Captures *n* from context to close *rem* and create the **Closure Object** in C++

**λ in C++: Closure Object**

- A *λ expression* is a mechanism for specifying a *function object* or *functor* (Recall **Module 40**)
- The primary use for a *λ* is to specify a simple action to be performed by some function
- For example, consider a remainder operation *rem* that computes  $m \% n$ , that is, *m* modulo *n*. It has type `int -> int`. To write *rem* in C++, we define a function / functor:

<pre>int n = 7; int rem(int m) // Function { return m % n; } // Uses n in context</pre>	<pre>int n = 7; struct remainder {           // Functor     int mod;                 // State     remainder(int n): mod(n) { // Ctor (n from context)     int operator()(int m)    // Function call operator         { return m % mod; } // Body };</pre>
<pre>rem(23); // 2</pre>	<pre>struct remainder rem(n); rem(23); // 2</pre>
<pre>λ: auto rem = [n](int m) -&gt; int { return m % n; } // Captures n from rem(23); // 2</pre>	

- Note that `[n]` Captures *n* from context to close *rem* and create the **Closure Object** in C++

So, now if you look at this name part is not there, it is an unnamed function. The parameters come after the introducer. The introducer tells that well, I am going to create a lambda. The return type changes position it was prefixed in the function now it (is) comes as a suffix. So, it is a suffix return type, which you have already seen. So, the return type comes here.

And then the function body is directly here, this is what is called the lambda. And to be able to use this I can directly invoke a function or invoke it on a value that is I can put (23) on this and I will get a value 2, because this functor will be evaluated on 23 that is *m* will go as 23. Or if I want to define it somewhere and use it later, then I can give it a name for my convenience it is not a name of the function for the simple reason that if it is the name of the function here like

rem here, you cannot copy the name. Once, you have given a name of the function that is the function, but here it is not so. This is like a variable which has the function object like more like here, so, by that I can refer it.

(Refer Slide Time: 09:26)

**lambda in C++: Closure Object**

- A *lambda expression* is a mechanism for specifying a *function object* or *functor* (Recall **Module 40**)
- The primary use for a *lambda* is to specify a simple action to be performed by some function
- For example, consider a remainder operation *rem* that computes  $m \% n$ , that is,  $m$  modulo  $n$ . It has type  $\text{int} \rightarrow \text{int}$ . To write *rem* in C++, we define a function / functor:

```

int n = 7;
int rem(int m) // Function
{ return m % n; }
// Uses n in context

int n = 7;
struct remainder {           // Functor
    int mod;                // State
    remainder(int n): mod(n) { // Ctor (n from context)
    int operator()(int m)    // Function call operator
    { return m % mod; }     // Body
};

struct remainder rem(n);

rem(23); // 2

lambda: auto rem = [n](int m) -> int { return m % n; } // Captures n from context
rem(23); // 2
    
```

• Note that **[n]** Captures  $n$  from context to close *rem* and create the **Closure Object** in C++

**lambda in C++: Closure Object**

- A *lambda expression* is a mechanism for specifying a *function object* or *functor* (Recall **Module 40**)
- The primary use for a *lambda* is to specify a simple action to be performed by some function
- For example, consider a remainder operation *rem* that computes  $m \% n$ , that is,  $m$  modulo  $n$ . It has type  $\text{int} \rightarrow \text{int}$ . To write *rem* in C++, we define a function / functor:

```

int n = 7;
int rem(int m) // Function
{ return m % n; }
// Uses n in context

int n = 7;
struct remainder {           // Functor
    int mod;                // State
    remainder(int n): mod(n) { // Ctor (n from context)
    int operator()(int m)    // Function call operator
    { return m % mod; }     // Body
};

struct remainder rem(n);

rem(23); // 2

lambda: auto rem = [n](int m) -> int { return m % n; } // Captures n from context
rem(23); // 2
    
```

• Note that **[n]** Captures  $n$  from context to close *rem* and create the **Closure Object** in C++

Naturally, the variable needs say type which is something like  $\text{int}$  to  $\text{int}$  complicated. So, I do not want to talk about that I leave it to the *auto* feature of C++11 to take care of it. So, this is basically my lambda expression.

One more thing to note is in this lambda expression, there are two types of variables. One are parameters like  $m$ , these are in the lambda language these are called bound variables whereas one

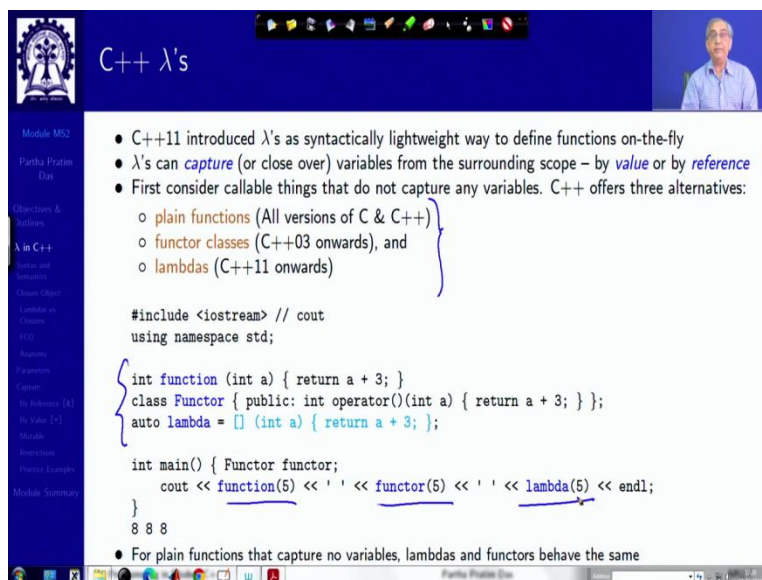


is this n which does not come as a parameter n does not come as a parameter it comes from the context.

But to be able to evaluate the function, I need to know the value of n. So, n is not bound in the function as parameter. So, I call it a free variable. These are terms in the lambda calculus from where the Lambda expressions are coming, you do not have to bother about that calculus, but just the terms.

So, if I have a free variable, then I have to specify how and where it is coming from. And that is what is called capture that it has to capture a value to be able to use it here it is coming from the global here what I do is in the introducer I can specify what I want to capture. Now, unless if I just keep the specifier introducer as empty, then this lambda is not well defined, because I do not know what is this n, I know what is this m, it is a bound variable, it is a parameter, but I do not know what is this n. So, n needs to be captured. That is the whole idea of the lambda.

(Refer Slide Time: 10:00)



The slide is titled "C++ λ's" and features a small video feed of a speaker in the top right corner. The main content is a list of bullet points and a code block. The bullet points describe C++11's introduction of lambdas as a syntactically lightweight way to define functions on-the-fly, their ability to capture variables from the surrounding scope, and the three alternatives for callable entities: plain functions, functor classes, and lambdas. The code block demonstrates these three alternatives by defining a plain function, a functor class, and a lambda, all of which perform the same computation (returning a + 3). The main function calls each of these three entities with the value 5, and the output shows that all three produce the same result: 8 8 8.

- C++11 introduced λ's as syntactically lightweight way to define functions on-the-fly
- λ's can *capture* (or close over) variables from the surrounding scope – by *value* or by *reference*
- First consider callable things that do not capture any variables. C++ offers three alternatives:
  - plain functions (All versions of C & C++)
  - functor classes (C++03 onwards), and
  - lambdas (C++11 onwards)

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

int function (int a) { return a + 3; }
class Functor { public: int operator()(int a) { return a + 3; } };
auto lambda = [] (int a) { return a + 3; };

int main() { Functor functor;
  cout << function(5) << ' ' << functor(5) << ' ' << lambda(5) << endl;
}
8 8 8
```

- For plain functions that capture no variables, lambdas and functors behave the same

So, let us see with this introduction, let us see what all it can lead to. So, C++11 has given this lambdas to actually define very lightweight functions which are heavily useful in variety of contexts. So, right now, therefore, in C++11, we have 3 kinds of main callable entities one are plain functions C and C++ all of functors C++03 Onwards and lambdas from C++11 I have just defined again 3 versions of the same computation as a function as a functor and as a lambda and shown that they can be used in the identical ways for the same result.

(Refer Slide Time: 10:47)

**C++ λ Syntax and Semantics**

- A λ expression consists of the following:  
`[capture list] (parameter list) -> return-type { function body }`
- The capture list and parameter list can be empty, so the following is a valid λ:  
`[] () { cout << "Hello, world!" << endl; }`
- **Parameter list** is a sequence of parameter types and variable names as for an ordinary function
- **Function body** is like an ordinary function body
- If the **function body** has only one return statement (which is very common), the **return type** is assumed to be the same as the type of the value being returned
- If there is **no return statement** in the function body, the return type is assumed to be **void**
  - Below λ has return type **void** – can be called without any use of the return value:  
`[] () { cout << "Hello from trivial lambda!" << endl; } ();`
  - However, trying to use the return type of the call is an error:  
`cout << [] () { cout << "Hello from trivial lambda!" << endl; } () << endl;`

**C++ λ Syntax and Semantics**

- A λ expression consists of the following:  
`[capture list] (parameter list) -> return-type { function body }`
- The capture list and parameter list can be empty, so the following is a valid λ:  
`[] () { cout << "Hello, world!" << endl; }`
- **Parameter list** is a sequence of parameter types and variable names as for an ordinary function
- **Function body** is like an ordinary function body
- If the **function body** has only one return statement (which is very common), the **return type** is assumed to be the same as the type of the value being returned
- If there is **no return statement** in the function body, the return type is assumed to be **void**
  - Below λ has return type **void** – can be called without any use of the return value:  
`[] () { cout << "Hello from trivial lambda!" << endl; } ();`
  - However, trying to use the return type of the call is an error:  
`cout << [] () { cout << "Hello from trivial lambda!" << endl; } () << endl;`

Coming to specific syntax much of it I have already discussed I have a parameter list. First, I have a capture list with the introducer capture list may or may not be there, then I have a parameter list I have a suffix return type and the function body this is the main structure of a lambda expression parameter capture list is optional.

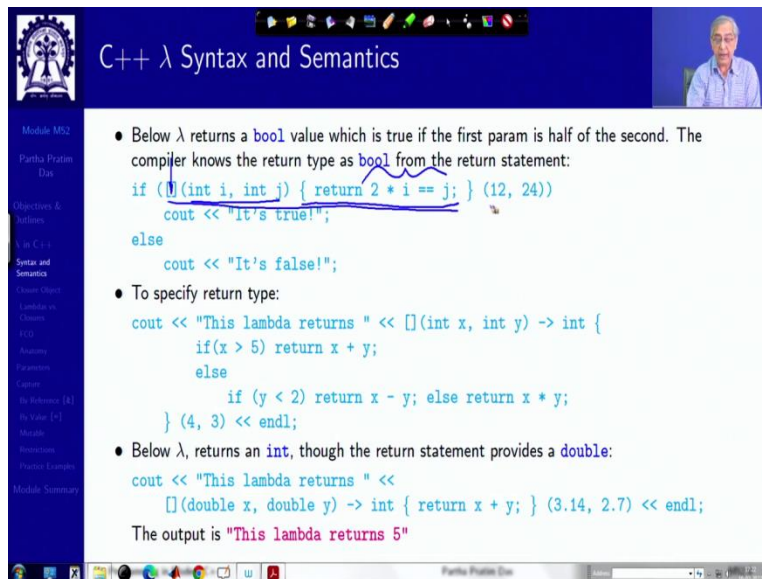
So, I may not have anything parameters are optional, so, it may not have anything, return type is optional in the way that if it is possible, if the function body has only one return statement, which is typical for a lambda expression, though not mandatory, but it is typically a lambda expression will have one return statement.

So, that return expression will have a type. So, from that the compiler may be able to deduce the type. If I want that deduced type to be the return type, then I may not provide the return type. And if there is no return statement in the function body, then the return type is taken to be void. So, this is the basic structure.

So, you can see that here I have a lambda expression which has no capture because there is no free variable there is no parameter there is no bound variable, and there is no return statement. So, it actually returns a void. So, I can directly take this entire lambda and apply it, invoke it as a functor. And if I invoke it as a functor, it will print this message.

But since, of course, this particular lambda does not return anything, I cannot use it as a part of another expression, because that will, there will be an error which is the same for any other function call as well.

(Refer Slide Time: 12:45)



The slide is titled "C++ λ Syntax and Semantics" and features a small video inset of a speaker in the top right corner. The main content consists of three bullet points explaining lambda function return types, each followed by a code snippet. The first bullet point discusses a lambda that returns a boolean value based on a condition, with a blue bracket highlighting the return statement. The second bullet point shows a lambda that returns an integer, with a blue bracket highlighting the return type and the return statement. The third bullet point shows a lambda that returns an integer despite having a double return statement, with a blue bracket highlighting the return type and the return statement. The output of the third lambda is shown as "This lambda returns 5".

- Below  $\lambda$  returns a `bool` value which is true if the first param is half of the second. The compiler knows the return type as `bool` from the return statement:  

```
if ( $\lambda$ (int i, int j) { return 2 * i == j; } (12, 24))  
    cout << "It's true!";  
else  
    cout << "It's false!";
```
- To specify return type:  

```
cout << "This lambda returns " <<  $\lambda$ (int x, int y) -> int {  
    if(x > 5) return x + y;  
    else  
        if (y < 2) return x - y; else return x * y;  
} (4, 3) << endl;
```
- Below  $\lambda$ , returns an `int`, though the return statement provides a `double`:  

```
cout << "This lambda returns " <<  
     $\lambda$ (double x, double y) -> int { return x + y; } (3.14, 2.7) << endl;
```

The output is "This lambda returns 5"

C++  $\lambda$  Syntax and Semantics

- Below  $\lambda$  returns a `bool` value which is true if the first param is half of the second. The compiler knows the return type as `bool` from the return statement:
 

```
if ([](int i, int j) { return 2 * i == j; }) (12, 24)
    cout << "It's true!";
else
    cout << "It's false!";
```

*compare(12, 24)*
- To specify return type:
 

```
cout << "This lambda returns " << [](int x, int y) -> int {
    if(x > 5) return x + y;
    else
        if (y < 2) return x - y; else return x * y;
} (4, 3) << endl;
```
- Below  $\lambda$ , returns an `int`, though the return statement provides a `double`:
 

```
cout << "This lambda returns " <<
    [](double x, double y) -> int { return x + y; } (3.14, 2.7) << endl;
```

The output is "This lambda returns 5"

C++  $\lambda$  Syntax and Semantics

- Below  $\lambda$  returns a `bool` value which is true if the first param is half of the second. The compiler knows the return type as `bool` from the return statement:
 

```
if ([](int i, int j) { return 2 * i == j; }) (12, 24)
    cout << "It's true!";
else
    cout << "It's false!";
```
- To specify return type:
 

```
cout << "This lambda returns " << [](int x, int y) -> int {
    if(x > 5) return x + y;
    else
        if (y < 2) return x - y; else return x * y;
} (4, 3) << endl;
```
- Below  $\lambda$ , returns an `int`, though the return statement provides a `double`:
 

```
cout << "This lambda returns " <<
    [](double x, double y) -> int { return x + y; } (3.14, 2.7) << endl;
```

The output is "This lambda returns 5"

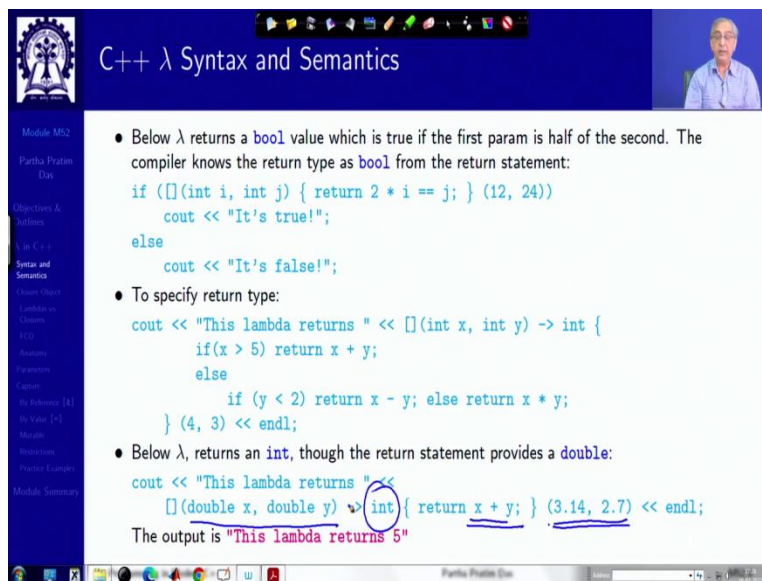
Here are some more examples of lambdas. I am checking a condition Boolean condition here, the condition I am checking for 2 parameters `i` and `j`, whether `j` is double of `i`. Therefore, the return type of this is can be inferred to be `bool`. So, I do not write it, there is nothing to capture because there is no free variable. And I take this entire functor. And I just apply it on a pair of values `i` and `j`. So, `i` becomes 12, `j` becomes 24 and this is evaluated.

You can see that this is this is actually pretty nice to do that, because if I had to do it in a in a different way, then possibly I would have said `compare 12 for 24` whereas, actually, the `compare` code is very simple. So, being able to write it in situ gives me a lot of understandability easy

understandability also, besides efficiency to know exactly what is being checked what is being compared.

Similar thing another is here, where I am trying to as a part of C out I am trying to write a lambda where I have given the return type just as an illustration, and then I have some code to compute return, I have to give return type here because as I said that in this case, there are multiple return statements. So, in that case, the compiler does not try one to deduce the return type, it would prefer the designer to provide the return type.

(Refer Slide Time: 14:34)



The screenshot shows a presentation slide with a blue header and a white content area. The header contains the title 'C++ λ Syntax and Semantics' and a small video feed of a speaker. The content area lists three bullet points with corresponding C++ code snippets and their outputs. The first bullet point shows a lambda returning a bool. The second shows a lambda returning an int. The third shows a lambda returning an int despite double parameters, with the output 'This lambda returns 5'.

```
Module MS2
Partha Pratim Das
Objectives & Outcomes
1 in C++
Syntax and Semantics
Class Object
Compiler vs Interpreter
File
Namespace
Comments
1 in Interview (14)
1 in Quiz (1)
Review
Resources
Partha Pratim Das
Module Semantics
```

- Below  $\lambda$  returns a `bool` value which is true if the first param is half of the second. The compiler knows the return type as `bool` from the return statement:  
`if ([](int i, int j) { return 2 * i == j; } (12, 24))  
 cout << "It's true!";  
else  
 cout << "It's false!";`
- To specify return type:  
`cout << "This lambda returns " << [](int x, int y) -> int {  
 if(x > 5) return x + y;  
 else  
 if (y < 2) return x - y; else return x * y;  
} (4, 3) << endl;`
- Below  $\lambda$ , returns an `int`, though the return statement provides a `double`:  
`cout << "This lambda returns " <<  
 [](double x, double y) -> int { return x + y; } (3.14, 2.7) << endl;`  
The output is "This lambda returns 5"

And the return type could be different from for example; here I have a pair of double parameters for this lambda like applying it on 3.14 and 2.7. So,  $x + y$  will be deduced as a type double but I want an integer. So, I put it as an integer. So, here, when this lambda evaluates, then the  $x$  will be added to  $y$  and then the double result will be converted to `int` and passed back.

(Refer Slide Time: 15:03)

The slide is titled "C++ λ: Syntax and Semantics" and features a small video inset of a speaker in the top right corner. The main content is a list of bullet points explaining lambda capture rules, with code snippets and annotations. A blue bracket highlights the variable 'n' in the first code example, and a blue arrow points from the text 'n must be initialized before the construction of the closure' to the variable 'n' in the lambda body.

Module MS2  
Part 1: Primitives  
Day  
Objectives & Outcomes  
1. In C++  
Syntax and Semantics  
Closure Object  
Lambda as Function  
1.1.1  
1.1.2  
1.1.3  
1.1.4  
1.1.5  
1.1.6  
1.1.7  
1.1.8  
1.1.9  
1.1.10  
1.1.11  
1.1.12  
1.1.13  
1.1.14  
1.1.15  
1.1.16  
1.1.17  
1.1.18  
1.1.19  
1.1.20  
1.1.21  
1.1.22  
1.1.23  
1.1.24  
1.1.25  
1.1.26  
1.1.27  
1.1.28  
1.1.29  
1.1.30  
1.1.31  
1.1.32  
1.1.33  
1.1.34  
1.1.35  
1.1.36  
1.1.37  
1.1.38  
1.1.39  
1.1.40  
1.1.41  
1.1.42  
1.1.43  
1.1.44  
1.1.45  
1.1.46  
1.1.47  
1.1.48  
1.1.49  
1.1.50  
1.1.51  
1.1.52  
1.1.53  
1.1.54  
1.1.55  
1.1.56  
1.1.57  
1.1.58  
1.1.59  
1.1.60  
1.1.61  
1.1.62  
1.1.63  
1.1.64  
1.1.65  
1.1.66  
1.1.67  
1.1.68  
1.1.69  
1.1.70  
1.1.71  
1.1.72  
1.1.73  
1.1.74  
1.1.75  
1.1.76  
1.1.77  
1.1.78  
1.1.79  
1.1.80  
1.1.81  
1.1.82  
1.1.83  
1.1.84  
1.1.85  
1.1.86  
1.1.87  
1.1.88  
1.1.89  
1.1.90  
1.1.91  
1.1.92  
1.1.93  
1.1.94  
1.1.95  
1.1.96  
1.1.97  
1.1.98  
1.1.99  
1.1.100

- Below  $\lambda$  captures  $n$  by value to compute the value of remainder of  $m$ :  

```
int n = 7;  
auto rem = [n](int m) -> int { return m % n; };
```

  - $n$  is captured by  $[n]$  (value – a copy is made from the context) at the time of constructing the closure object. Hence  $n$  must be initialized before the construction of the closure
  - The value of  $n$  cannot be changed within the  $\lambda$  (for immutable  $\lambda$ 's)
  - The changes to  $n$  after the construction of the closure object are not reflected
- Below  $\lambda$  captures  $s$  by reference to accumulate the value of  $m$ :  

```
int s = 0;  
auto acc = [&s](int m){ s += m; };
```

  - $s$  is captured by  $[&s]$  (reference – a reference is set to the context) at the time of constructing the closure object. Hence it is optional to initialize  $s$  before the construction of the closure. However, it must be initialized before the use of the closure
  - The value of  $s$  can be changed within the  $\lambda$
  - The changes to  $s$  after the construction of the closure object will be reflected

Now, coming to capture, there are two ways to capture as we know that there are two ways to pass parameters to a function: call by value, call by reference, when a call by value a copy of that value is done from the actual to the formal parameter, and you make changes to that nothing is reflected to the actual parameters, in a call by reference, you create a reference to the actual parameters.

So, any changes you make in the formal parameter is reflected to the actual parameter a very similar concept exists for capture you can do captured by value. So, in the earlier case, when I just wrote  $n$ , I am capturing by value, which says that, when this particular functor the lambda is constructed, the value of  $n$  must be known, it will be copied and kept as a part of this lambda object, it will be copied from the context and therefore,  $n$  must have a valid value before initialization.

And what is important is, this capture is always taken to be as a constant parameter like if we, if I have to compare with the with the with the value it is constant in the lambda. So, if I have captured by value, then I cannot make changes, I cannot write something like within the body of the lambda function that is not allowed.

(Refer Slide Time: 16:43)

**C++ λ: Syntax and Semantics**

- Below  $\lambda$  captures  $n$  by value to compute the value of remainder of  $m$ :  

```
int n = 7;  
auto rem = [n](int m) -> int { return m % n; };
```

  - $n$  is captured by  $[n]$  (value – a copy is made from the context) at the time of constructing the closure object. Hence  $n$  must be initialized before the construction of the closure
  - The value of  $n$  cannot be changed within the  $\lambda$  (for immutable  $\lambda$ 's)
  - The changes to  $n$  after the construction of the closure object are not reflected
- Below  $\lambda$  captures  $s$  by reference to accumulate the value of  $m$ :  

```
int s = 0;  
auto acc = [&s](int m) { s += m; };
```

  - $s$  is captured by  $[&s]$  (reference – a reference is set to the context) at the time of constructing the closure object. Hence it is optional to initialize  $s$  before the construction of the closure. However, it must be initialized before the use of the closure
  - The value of  $s$  can be changed within the  $\lambda$
  - The changes to  $s$  after the construction of the closure object will be reflected

**C++ λ: Syntax and Semantics**

- Below  $\lambda$  captures  $n$  by value to compute the value of remainder of  $m$ :  

```
int n = 7;  
auto rem = [n](int m) -> int { return m % n; };
```

  - $n$  is captured by  $[n]$  (value – a copy is made from the context) at the time of constructing the closure object. Hence  $n$  must be initialized before the construction of the closure
  - The value of  $n$  cannot be changed within the  $\lambda$  (for immutable  $\lambda$ 's)
  - The changes to  $n$  after the construction of the closure object are not reflected
- Below  $\lambda$  captures  $s$  by reference to accumulate the value of  $m$ :  

```
int s = 0;  
auto acc = [&s](int m) { s += m; };
```

  - $s$  is captured by  $[&s]$  (reference – a reference is set to the context) at the time of constructing the closure object. Hence it is optional to initialize  $s$  before the construction of the closure
  - The value of  $s$  can be changed within the  $\lambda$
  - The changes to  $s$  after the construction of the closure object will be reflected

The other is captured by reference. So, I am trying to, there is a variable  $s$ , and I am trying to do  $s +=$  parameter  $m$ , you can make out from this part of the function body that what I am trying to do is every time this lambda is called this functor is invoked, the parameter will get added to  $s$ .

So, like it says some accumulator that that I am doing. So, I need  $s$  to change  $s$  is not a parameter of this functor. So,  $s$  is a free variable, therefore,  $s$  needs a capture and I need to change  $s$  and want that to get reflected that is why I write it as  $\&s$ . So, this becomes captured by reference.

And when I do capture by reference, it is not necessary that the particular reference variable must have a value at that point, it is only before I use it, it must have a value because any, if it is

captured by reference then I can make changes to it, any change I make is reflected to the variable that I have captured and any change you make to the variable will be reflected to the lambda. So, this is captured by value and capture by reference and that is the core idea of the entire thing.

(Refer Slide Time: 18:10)

**Lambdas vs. Closures**

- Closure**
  - A *closure (lexical / function closure)*, is a technique for implementing *lexically scoped name binding* in a language with first-class functions
  - Operationally, a closure is a *record storing a function* together with an *environment*
  - The *environment* is a mapping associating (*binding*) each *free* variable of the function with the *value* or *reference* to which the name was bound when the closure was created
  - Unlike a plain function, a closure allows the function to access captured variables through the closure's copies of their values or references, even in its invocations outside their scope
- Lambdas vs. Closures** (From *Lambdas vs. Closures* by Scott Meyers, 2013)
  - A  $\lambda$  expression `auto f = [&](int x, int y){ return fudgeFactor * (x + y); };` exists only in a program's source code. A lambda does not exist at runtime
  - The runtime effect of a  $\lambda$  expression is the generation of an object, called *closure*
  - Note that *f* is not the closure, it is a *copy of the closure*. The actual closure object is a *temporary* that's typically destroyed at the end of the statement
  - Each  $\lambda$  expression causes a unique class to be generated (during compilation) and also causes an object of that class type – a closure – to be created (at runtime)
  - Hence, *closures are to lambdas as objects are to classes*

**Lambdas vs. Closures**

- Closure**
  - A *closure (lexical / function closure)*, is a technique for implementing *lexically scoped name binding* in a language with first-class functions
  - Operationally, a closure is a *record storing a function* together with an *environment*
  - The *environment* is a mapping associating (*binding*) each *free* variable of the function with the *value* or *reference* to which the name was bound when the closure was created
  - Unlike a plain function, a closure allows the function to access captured variables through the closure's copies of their values or references, even in its invocations outside their scope
- Lambdas vs. Closures** (From *Lambdas vs. Closures* by Scott Meyers, 2013)
  - A  $\lambda$  expression `auto f = [&](int x, int y){ return fudgeFactor * (x + y); };` exists only in a program's source code. A lambda does not exist at runtime
  - The runtime effect of a  $\lambda$  expression is the generation of an object, called *closure*
  - Note that *f* is not the closure, it is a *copy of the closure*. The actual closure object is a *temporary* that's typically destroyed at the end of the statement
  - Each  $\lambda$  expression causes a unique class to be generated (during compilation) and also causes an object of that class type – a closure – to be created (at runtime)
  - Hence, *closures are to lambdas as objects are to classes*

Now, you will find in the literature that I am talking about lambda expressions all the time, but you will find in the literature that the term more commonly used in C++ is called a closure object. So, what is the difference, similarity or connection between a lambda expression and closure object. It is like this -- this expression that I have written with the parameters return type,



capture variables capture mode everything is a program for the unnamed function that is called a lambda expression.

Now, this corresponds to as I said is a function object. This corresponds therefore, it has to correspond to a function object class. So, this actually is as if the definition of that function object class. Now, when I instantiate it the function object gets created and that function object is called the closure object.

Why is it called the closure object? Because in a lambda, you often have free variables which needs to be captured to close the entire meaning of what the lambda expression has only when you have all the free variables captured only then you have a closed lambda, which can be evaluated. So, that is the reason this has been given the name closer object.

So, to very clearly whatever you write here is the lambda expression which is only in the program source code, it does not exist at the runtime that gets translated into a functor class declaration internally. And at the runtime, an instance of that class that functor object gets instantiated created, constructed, and that object is called the functor object, that object is called the closure object.

Now, when I do `auto f`, this, so this entire thing is an object, it is an object. So, if I do `auto f`, initialized with an object, then what am I doing, I am actually copy constructing `f` that is how you will do, you have already an object the temporary object created. So, `f` is actually not the closure object, but it is a copy of the closure object, the closure object per se, what has got constructed at this point is a temporary object.

And at the end of the statement, it gets deleted by the execution. It is like a any temporary object, but in this, I am keeping a copy of that, and I can keep on copying it, it is a it is a, it is a as we will see, it is a first class object, which can be copied not like a normal C function which cannot be copied, it is a function cannot be copied, it can just be invoked, but a functor created from a lambda expression the closure object can be freely copied. So, that copy of the closure, so, this is something which is which is important to understand in terms of what is a lambda expression and what is a closure object.

(Refer Slide Time: 21:56)

**Closure Objects: Implementing  $\lambda$ 's**

- A  $\lambda$ -expression generates a **Closure Object** at *run-time*
- A closure object is *temporary*
- A closure object is *unnamed*
- For a  $\lambda$ -expression, the compiler creates a *functor class* with:
  - *data members*:
    - ▷ a value member each for each value capture
    - ▷ a reference member each for each reference capture
  - a *constructor* with the captured variables as parameters
    - ▷ a value parameter each for each value capture
    - ▷ a reference parameter each for each reference capture
  - a *public inline const function call operator()* with the parameters of the lambda as parameters, generated from the body of the lambda
  - *copy constructor, copy assignment operator, and destructor*
- A *closure object* is constructed as an instance of this class and *behaves like a function object*
- A  $\lambda$ -expression without any capture behaves like a function pointer

Source: C++ Lambda Under the Hood, 2019

**Closure Objects: Implementing  $\lambda$ 's**

- A  $\lambda$ -expression generates a **Closure Object** at *run-time*
- A closure object is *temporary*
- A closure object is *unnamed*
- For a  $\lambda$ -expression, the compiler creates a *functor class* with:
  - *data members*:
    - ▷ a value member each for each value capture
    - ▷ a reference member each for each reference capture
  - a *constructor* with the captured variables as parameters
    - ▷ a value parameter each for each value capture
    - ▷ a reference parameter each for each reference capture
  - a *public inline const function call operator()* with the parameters of the lambda as parameters, generated from the body of the lambda
  - *copy constructor, copy assignment operator, and destructor*
- A *closure object* is constructed as an instance of this class and *behaves like a function object*
- A  $\lambda$ -expression without any capture behaves like a function pointer

Source: C++ Lambda Under the Hood, 2019

So, lambda expressions generate closure objects at runtime, they are temporary, they are unnamed, and they correspond to a functor class where there are data members corresponding to what you have captured, because you have to remember them, you have to keep referring to them. So, you will have value members for value capture, you will have reference member for reference capture, you will have a constructor which will construct these data members in the appropriate way.

And the most important thing is you need to have the function call operator and this operator has to be public. So, that it can be freely used it is inline for the purpose of optimization, because it is

already there, you have the entire thing so, it is possible to inline it always. And (it is treated) by default it is treated like a constant function. And for this closure object type, the copy constructor, copy assignment, operator, destructor all are defined by default by the compiler, you do not have to write anything.

(Refer Slide Time: 23:08)

**Closure Objects: Implementing  $\lambda$ 's: Example**

```
#include <iostream> // lambda & closure object
using namespace std;
int main() {
    int val = 0; // for value capture init. must
    int ref;    // for ref. capture init. opt.

    auto check = [val, &ref](int param){
        cout << "val = " << val << ", ";
        cout << "ref = " << ref << ", ";
        cout << "param = " << param << endl;
    };
    // lambda to show captured values
    // constructed with value capture of val
    // and reference capture of ref
    // Also, has a parameter param

    ref = 2; // init. will be reflected
    check(5); // val = 0, ref = 2, param = 5
    val = 3; // change will not be reflected
    check(5); // val = 0, ref = 2, param = 5
    ref = 4; // change will be reflected
    check(5); // val = 0, ref = 4, param = 5
}
```

```
#include <iostream> // Possible functor by compiler
using namespace std;
int main() {
    int val = 0; // for value capture init. must
    int ref;    // for ref. capture init. opt.

    struct check_f { // functor to show captured values
        int val_f; // value member for value capture
        int& ref_f; // ref. member for ref. capture
        check_f(int v, int& r): // Ctor with
            val_f(v), ref_f(r) {} // value & ref param
        void operator()(int param) const { // param
            cout << "val = " << val_f << ", ";
            cout << "ref = " << ref_f << ", ";
            cout << "param = " << param << endl;
        };
    };
    auto check = check_f(val, ref); // Instantiation

    ref = 2; // init. will be reflected
    check(5); // val = 0, ref = 2, param = 5
    val = 3; // change will not be reflected
    check(5); // val = 0, ref = 2, param = 5
    ref = 4; // change will be reflected
    check(5); // val = 0, ref = 4, param = 5
}
```

**Closure Objects: Implementing  $\lambda$ 's: Example**

```
#include <iostream> // lambda & closure object
using namespace std;
int main() {
    int val = 0; // for value capture init. must
    int ref;    // for ref. capture init. opt.

    auto check = [val, &ref](int param){
        cout << "val = " << val << ", ";
        cout << "ref = " << ref << ", ";
        cout << "param = " << param << endl;
    };
    // lambda to show captured values
    // constructed with value capture of val
    // and reference capture of ref
    // Also, has a parameter param

    ref = 2; // init. will be reflected
    check(5); // val = 0, ref = 2, param = 5
    val = 3; // change will not be reflected
    check(5); // val = 0, ref = 2, param = 5
    ref = 4; // change will be reflected
    check(5); // val = 0, ref = 4, param = 5
}
```

```
#include <iostream> // Possible functor by compiler
using namespace std;
int main() {
    int val = 0; // for value capture init. must
    int ref;    // for ref. capture init. opt.

    struct check_f { // functor to show captured values
        int val_f; // value member for value capture
        int& ref_f; // ref. member for ref. capture
        check_f(int v, int& r): // Ctor with
            val_f(v), ref_f(r) {} // value & ref param
        void operator()(int param) const { // param
            cout << "val = " << val_f << ", ";
            cout << "ref = " << ref_f << ", ";
            cout << "param = " << param << endl;
        };
    };
    auto check = check_f(val, ref); // Instantiation

    ref = 2; // init. will be reflected
    check(5); // val = 0, ref = 2, param = 5
    val = 3; // change will not be reflected
    check(5); // val = 0, ref = 2, param = 5
    ref = 4; // change will be reflected
    check(5); // val = 0, ref = 4, param = 5
}
```

```

#include <iostream> // lambda & closure object
using namespace std;
int main() {
    int val = 0; // for value capture init. must
    int ref;    // for ref. capture init. opt.

    auto check = [val, &ref](int param){
        cout << "val = " << val << ", ";
        cout << "ref = " << ref << ", ";
        cout << "param = " << param << endl;
    };
    // lambda to show captured values
    // constructed with value capture of val
    // and reference capture of ref
    // Also, has a parameter param

    ref = 2; // init. will be reflected
    check(5); // val = 0, ref = 2, param = 5
    val = 3; // change will not be reflected
    check(5); // val = 0, ref = 2, param = 5
    ref = 4; // change will be reflected
    check(5); // val = 0, ref = 4, param = 5
}

#include <iostream> // Possible functor by compiler
using namespace std;
int main() {
    int val = 0; // for value capture init. must
    int ref;    // for ref. capture init. opt.

    struct check_f { // functor to show captured values
        int val_f; // value member for value capture
        int& ref_f; // ref. member for ref. capture
        check_f(int v, int& r): // Ctor with
            val_f(v), ref_f(r) { // value & ref param
        };
        void operator()(int param) const { // param
            cout << "val = " << val_f << ", ";
            cout << "ref = " << ref_f << ", ";
            cout << "param = " << param << endl;
        };
    };
    auto check = check_f(val, ref); // Instantiation

    ref = 2; // init. will be reflected
    check(5); // val = 0, ref = 2, param = 5
    val = 3; // change will not be reflected
    check(5); // val = 0, ref = 2, param = 5
    ref = 4; // change will be reflected
    check(5); // val = 0, ref = 4, param = 5
}

```

So, it is just an illustration of how the implementation might look like is I have a lambda expression here with two free variables. So, two capture variables one is captured by value other by reference and one parameter. So, in the main function, I have two variables, which I captured in this check and print them.

So, when I get into the corresponding possibly the corresponding function object class, well, this is not exactly what the compiler will do, but compiler in principle does something. Let us say it I define a class `check_f`, which has for each one corresponding data members, mind you, this is captured by value so, it is a value member, this is captured by reference so this a reference member, then I have the constructor of it, that is to construct the function object that is to construct the closure object.

So, it will have a value parameter and a reference parameter to set that two capture values there could be separate, other local variables now, I do not care. And finally, the parameter of the lambda expression is set as a parameter of the function call operator, you can see that that is defined as `const` that it is constant function it can work only with constant objects and the rest of the entire body simply comes in here.

So, if you keep this, this kind of, context in mind, then you will find it very, very easy to understand what is going on with the with the lambda expression and closer objects, here there are a number of examples, values for which it is done. So, when this closure object was created

ref did not have a value because it is a reference parameter, val has a value and but before invoking the closure object, I must set a value to ref.

So, I set it to 2, I have done check(5), so, val = 0, ref = 2, param = 5 as expected, I have changed val to 3, but it is captured by value. So, it is not expected to change it does not, but I change the reference I can see that change coming on here coming on here.

(Refer Slide Time: 26:01)

Closure Objects: First Class Objects (FCOs)

```
struct trace { int i;
  trace() : i(0) { std::cout << "construct\n"; }
  trace(trace const &t) { std::cout << "copy construct\n"; }
  ~trace() { std::cout << "destroy\n"; }
  trace& operator=(trace&t) { std::cout << "assign\n"; return *this; }
};
```

Code Snippets	Outputs
<pre>{ trace t; // t not used so not captured   int i = 8;   auto m1 = [=](){ return i / 2; }; }</pre>	<pre>construct destroy</pre>
<pre>{ trace t; // capture t by value   auto m1 = [=](){ int i = t.i; };   std::cout &lt;&lt; "-- make copy --" &lt;&lt; std::endl;   auto m2 = m1; }</pre>	<pre>construct copy construct -- make copy -- copy construct destroy destroy destroy</pre>
<pre>{ trace t; // capture t by reference   auto m1 = [&amp;](){ int i = t.i; };   std::cout &lt;&lt; "-- make copy --" &lt;&lt; std::endl;   auto m2 = m1; }</pre>	<pre>construct -- make copy -- destroy</pre>

Closure object has implicitly-declared copy constructor / destructor

So, this is the basic idea of the lambdas they work as first class objects. So, they can be easily copied and this is just an illustration using a class, we will just show that what happens if in a context you have the object of a class which you do not capture, or you captured by value, as in here or you capture by reference as in here.

And by tracing these messages, you will be able to see the difference between captured by value and the capture by reference if you capture by value, then naturally at the capture time a copy has to happen when you copy the closure object, in other copy has to happen, these are all happening for t and naturally the corresponding destroy will also have to happen whereas, if you capture by reference, no such copies will be required. That is a simple thing.

(Refer Slide Time: 27:04)

**Closure Objects: Anatomy**

```

1 2 3 4 5
|  |  |  |  |
[=] () mutable throw() -> int
{
  int n = x + y;
  x = y;
  y = n;
  return n;
}
6
  
```

- [1] Capture Clause (*introducer*)
- [2] Parameter List (Opt.) (*declarator*)
- [3] Mutable Specs. (Opt.)
- [4] Exception Specs. (Opt.)
- [5] (Trailing) Return Type (Opt.)
- [6]  $\lambda$  body

$\lambda$  Expression::  $\mathcal{E} \vdash \text{my\_mod} : \text{Int}, \lambda(v : \text{Int}). v \% \text{my\_mod} : \text{Int}$   
 Closure Object:: `[my_mod](int v) -> int { return v % my_mod; }`

- **Introducer:** `[my_mod]`
- **Capture:** `my_mod`
- **Parameters:** `(int v)`
- **Declarator:** `(int v) -> int`
- **Mutable Spec:** Skipped
- **Exception Spec:** Skipped
- **Return Type:** `-> int`
- **$\lambda$  Body:** `{ return v % my_mod; }`

This is for details that, formally, how does what does the syntax of the lambdas look like everything else, we have already explained, the two things that can happen between the parameter list and the suffix return type is you can write the keyword mutable which has certain meaning, and you can write the exception specification.

(Refer Slide Time: 27:31)

**Closure Objects: Parameters**

Parameter Passing	Remarks
<code>[] () { std::cout &lt;&lt; "foo" &lt;&lt; std::endl; }();</code>	foo
<code>[] (int v) { std::cout &lt;&lt; v &lt;&lt; " *6*" &lt;&lt; v*6 &lt;&lt; std::endl; }(7);</code>	7*6=42
<pre> int i = 7; [] (int &amp;v) { v += 6; } (i); std::cout &lt;&lt; "the correct value is: " &lt;&lt; i &lt;&lt; std::endl;           </pre>	the correct value is: 42
<pre> int j = 7; [] (int const &amp;v) { v += 6; } (j); std::cout &lt;&lt; "the correct value is: " &lt;&lt; j &lt;&lt; std::endl;           </pre>	// error: // assignment of read-only reference 'v'
<pre> int j = 7; [] (int v) { v += 6; std::cout &lt;&lt; "v: " &lt;&lt; v &lt;&lt; std::endl; }(j);           </pre>	v: 42
<pre> int j = 7; [] (int &amp;v, int j) { v += j; } (j, 6); std::cout &lt;&lt; "j: " &lt;&lt; j &lt;&lt; std::endl;           </pre>	// lambda parameters do not affect // the namespace j: 42
<pre> [] std::cout &lt;&lt; "foo" &lt;&lt; std::endl; (); is same as [] () std::cout &lt;&lt; "foo" &lt;&lt; std::endl; ();           </pre>	// lambda expression without a // declarator acts as if it were ()

We will look into. So, here are examples of different kind of way that you can have parameters and you can if you follow the semantics of call by value and call by reference, you will be able to understand all of that these very easily. The only one that I would like to mention about is here

you have a constant parameter, constant reference parameter `v` and you are trying to change that. So, naturally, you get an error, which is these are all exactly same, like what happens in a normal function or a functor.

(Refer Slide Time: 28:13)

**Closure Objects: Capture**

- The *captures* is a comma-separated list of zero or more captures, optionally with default
- The capture list defines the outside variables that are accessible from the  $\lambda$  function body
- The only capture defaults are
  - `&` (implicitly capture the used automatic variables *by reference*) and
  - `=` (implicitly capture the used automatic variables *by copy / value*)
- The *current object* (`*this`) can be *implicitly captured* if either capture default is present
- If implicitly captured, it is always captured by reference, even for `=`. Deprecated since C++20

Capture	Meaning	C++
<code>identifier</code>	simple by-copy capture	C++11
<code>identifier ...</code>	simple by-copy capture that is a pack expansion	C++11
<code>identifier init</code>	by-copy capture with an initializer	C++14
<code>&amp; identifier</code>	simple by-reference capture	C++11
<code>&amp; identifier ...</code>	simple by-reference capture that is a pack expansion	C++11
<code>&amp; identifier init</code>	by-reference capture with an initializer	C++14
<code>this</code>	simple by-reference capture of the current object	C++11
<code>*this</code>	simple by-copy capture of the current object	C++17
<code>... identifier init</code>	by-copy capture with an initializer that is a pack expansion	C++20
<code>&amp; ... identifier init</code>	by-reference capture with an initializer that is a pack expansion	C++20

Source: Lambda capture, cppreference.com

For capture, you have two generic ones capture all stuff where you can capture a variable by name that is either that will be value by `&name` that will be by reference or you can just write `&` in the capture or `=` in the capture. In this case, if I write `&` in the capture, then it basically means that all variables that need to be captured in the body of the lambda expression that is all free variables will be captured by reference here it means that all will be captured by value and then I can have added exceptions created to them.

So, these are the different capture rules. This is I do not expect you to learn them by heart, these are just for your reference to see that how regularly from C++11 till C++20 the semantics of lambda is getting enhanced by every release.

(Refer Slide Time: 29:16)

**Closure Objects: Capture**

- Optional captures of  $\lambda$  expressions are (C++11):
  - *Default all by reference* ✓  
`[&]() { ... }`
  - *Default all by value* ✓  
`[=]() { ... }`
  - *List of specific identifier(s) by value or reference and/or this*  
`[identifier]() { ... }`  
`[&identifier]() { ... }`  
`[foo, &bar, gorp]() { ... }`
  - *Default and specific identifiers and/or this*  
`[&, identifier]() { ... }`  
`[=, &identifier]() { ... }`

Source: Lambda capture, cppreference.com

So, you have a default, all by reference, you have a default all by value, or you can specifically list different cases of capture.

(Refer Slide Time: 29:32)

**Closure Objects: Capture: Simple Examples**

```
int x = 2, y = 3; // Global Context
const auto 10 = []() { return 1; }; // No capture
typedef int (*11) (int); // Function pointer
const 11 f = [](int i){ return i; }; // Converts to a func. ptr. w/o capture
const auto 12 = [=]() { return x; }; // All by value (copy)
const auto 13 = [&]() { return y; }; // All by ref
const auto 14 = [x]() { return x; }; // Only x by value (copy)
const auto 1x = [=x]() { return x; }; // wrong syntax, no need for
// = to copy x explicitly
const auto 15 = [&y]() { return y; }; // Only y by ref
const auto 16 = [x, &y]() { return x * y; }; // x by value and y by ref
const auto 17 = [=, &x]() { return x + y; }; // All by value except x
// which is by ref
const auto 18 = [&, y]() { return x - y; }; // All by ref except y which
// is by value
const auto 19 = [this]() { } // capture this pointer
const auto 1a = [*this]() { } // capture a copy of *this
// since C++17
```

Programming in Modern C++ Partha Pratim Das MS2.19

Now, here again, is a set of simple examples showing you different types of capture. Try them out understand meanings are given by the site.



(Refer Slide Time: 29:45)

The slide is titled "[&]()->rt{...}: Capture" and features a small video inset of a speaker in the top right corner. The main content is a list of bullet points and code snippets:

- *Capture default all by reference*  

```
int total_elements = 1;
for_each(cardinal.begin(), cardinal.end(),
    [&](int i) { total_elements *= i; }); // total_elements
// can be changed
```

A blue arrow points from the `&` in the lambda capture to the `total_elements` variable in the lambda body.
- **Errors**  

```
[=](int i) { total_elements *= i; };
```

error C3491: 'total\_elements': a by-value capture cannot be modified in a non-mutable lambda

```
[](int i) { total_elements *= i; };
```

error C3493: 'total\_elements' cannot be implicitly captured because no default capture mode has been specified

If you look at one example, say here I have a variable `total_elements` and I am doing a STL algorithm we have talked about this earlier, I have a iterated here, another iterated here and the operation is given as a lambda expression. If you create a closure object where I am doing `total_elements *= i` so like this is a multiplicative accumulation, that is what will happen.

Now, obviously, here the capture will have to be referenced because I am changing this. So, if I capture is value, (this is) this will not compile, if the capture is not there also this will not compile. So, this is these are very simple, this is just rules of basic rules that gets extended.

(Refer Slide Time: 30:37)

The slide is titled "[&] ()->rt{...}: Capture: Scope & Lifetime". It is divided into two columns: "Wrong Capture by Reference" and "Correct Capture by Reference".

**Wrong Capture by Reference:**

- Closures may outlive their creating function

```
std::function<bool>(int)>
returnClosure(int a) { // returns bool
    int b, c;
    // won't compile but assume it would
    return [] (int x) {
        { return a*x*x + b*x + c == 0; };
    };
}
```

// f is essentially a copy of  
// lambda's closure  
auto f = returnClosure(10);  
...  
if (f(22)) // invoke the closure

- What are the values of a, b, c in the call?
  - returnClosure no longer active!
- Non-static locals referenceable only if captured

**Correct Capture by Reference:**

- This version has no such problem

```
int a; // now at global or namespace scope
std::function<bool>(int)>
returnClosure() {
    static int b, c; // now static ...
    // now compiles
    return [] (int x) {
        { return a*x*x + b*x + c == 0; };
    };
}
```

// as before  
auto f = returnClosure();  
...  
if (f(22)) // as before

- a, b, c outlive returnClosure's invocation
- Variables of static storage duration always referenceable

Similarly, here, I have a lambda as a part of a functor definition, this lambda has 3 free variables a, b, c, which are taken from a parameter here and from the local variables of this. Now, mind you, this will not be permitted. Why you will this not be permitted, this will not be permitted, because when I create the corresponding class, the values that I get these are all local, so, they will be available only at that time, they will not be available at a later point of time. So, how do I use them without proper capture?

So, one option is if the values are either global or static, then I will be able to do this. This is a simple consequences of the scoping rule, and so on.

(Refer Slide Time: 31:44)

```
Module MS2
Partha Pratim Das
Objectives & Outlines
A in C++
Syntax and Semantics
Class Object
Lambda in C++
C++
Features
Parameters
Capture
By Reference [&]
By Value [ ]
Mutable
Restrictions
Practical Examples
Module Summary
```

**[&]() ->rt{...}: Capture**

```
// #include <iostream>, <algorithm>, <vector>
template< typename T >
void fill(std::vector<int>& v, T done) { int i = 0; while (!done()) { v.push_back(i++); } }
int main() {
    std::vector<int> stuff; // Fill the vector with 0, 1, 2, ... 7
    fill(stuff, [&]{ return stuff.size() >= 8; }); // [=] compiles but is infinite loop
    for(auto it = stuff.begin(); it != stuff.end(); ++it) std::cout << *it << ' ';
    std::cout << std::endl;

    std::vector<int> myvec; // Fill the vector with 0, 1, 2, ... till the sum exceeds 10
    fill(myvec, [&]{ int sum = 0; // [=] compiles but is infinite loop
                 std::for_each(myvec.begin(), myvec.end(), [&](int i){ sum += i; });
                 // [=] is error: assignment of read-only variable 'sum'
                 return sum >= 10;
    });
    for(auto it = myvec.begin(); it != myvec.end(); ++it) std::cout << *it << ' ';
    std::cout << std::endl;
}
0 1 2 3 4 5 6 7
0 1 2 3 4
```

Programming in Modern C++ Partha Pratim Das MS2.22

```
Module MS2
Partha Pratim Das
Objectives & Outlines
A in C++
Syntax and Semantics
Class Object
Lambda in C++
C++
Features
Parameters
Capture
By Reference [&]
By Value [ ]
Mutable
Restrictions
Practical Examples
Module Summary
```

**[=]() ->rt{...}: Capture: Mutable**

- Consider  

```
int h = 10;
auto two_h = [=] () { h *= 2; return h; };
std::cout << "2h:" << two_h() << " h:" << h << std::endl;
```

error C3491: 'h': a by-value capture cannot be modified in a non-mutable lambda
- λ closure objects have a *public inline function call operator* that:
  - Matches the parameters of the lambda expression
  - Matches the return type of the lambda expression
  - Is declared *const*
- Make mutable*  

```
int h = 10;
auto two_h = [=] () mutable { h *= 2; return h; };
std::cout << "2h:" << two_h() << " h:" << h << std::endl;
```

2h:20 h:10

And here again, there are more examples of capture that I would like you to go through and execute and understand one more. As always, I have given a number of examples here, which you can try out on your own.

Mutable is a concept for captured by value, suppose you have captured by value, this is default. So, here, there is no parameters so, h is a only free variable. So, if I try to do this, if I try to double h within the lambda expression, then certainly I will get an error because h is captured by value and if I captured by value, I am not allowed to change it, I told earlier. So, an exception to

this can be created if I use the key word mutable, if I say it is mutable, the lambda as a whole is mutable, then any variable which is captured by value can also be changed.

But, under the mutable condition, the changes being made in this h, and changes and this h are different. This is not the same case as reference where they are same variable but here, because it is captured by value, a copy of this global h has been done here. And by doing mutable, I am allowing that to be modified, the concept is very similar to the mutable members of constant objects.

(Refer Slide Time: 33:22)

The screenshot shows a presentation slide with a blue header and a white content area. The header contains the title "[=]()->rt{...}: Capture: Mutable" and a small video feed of a speaker. The content area contains two code snippets. The first snippet is for a mutable lambda function capturing by value, and the second is for a mutable lambda function capturing by reference. Both snippets include output lines showing the value of 'h' before and after the lambda function is executed. In the first case, 'h' remains 10. In the second case, 'h' changes to 20.

```
int h = 10;
auto f = [=] () mutable { h *= 2; return h; }; // h changes locally
std::cout << "2h:" << f() << std::endl;
std::cout << " h:" << h << std::endl;
```

2h:20  
h:10

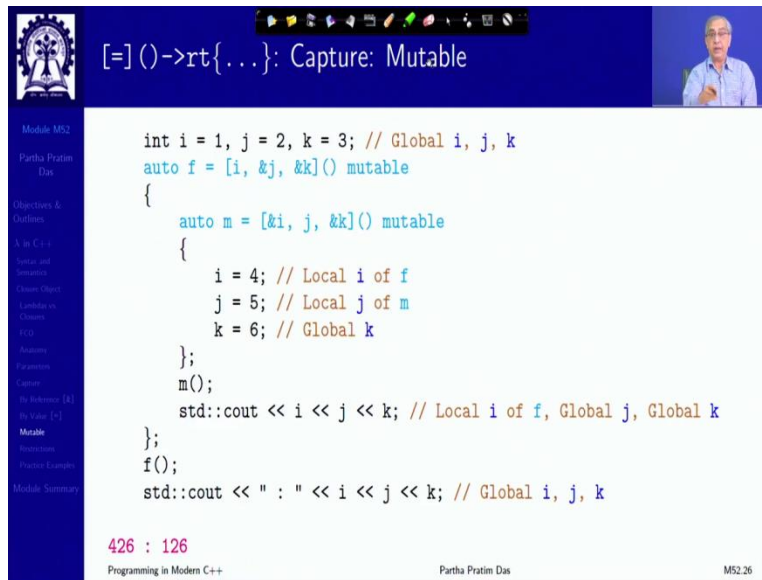
---

```
int h = 10;
auto g = [&] () { h *= 2; return h; }; // h changes globally
std::cout << "2h:" << g() << std::endl;
std::cout << " h:" << h << std::endl;
```

2h:20  
h:20

So, here are examples of what will happen if you capture all captured default by value and make mutable what will be the effect and if you capture by reference, what will be the effect as you can see, in both cases, you can change h, in the first case, the original h does not change. In the second case, the original h also has changed.

(Refer Slide Time: 33:49)



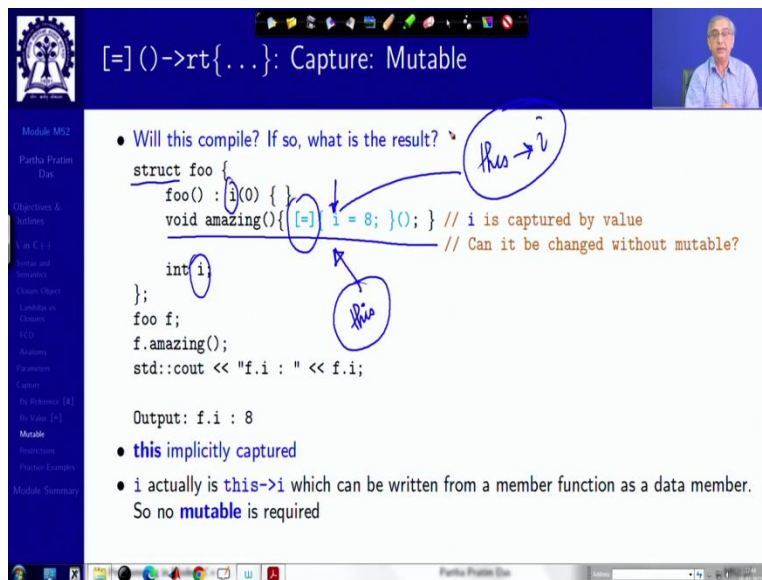
[=]()->rt{...}: Capture: Mutable

```
int i = 1, j = 2, k = 3; // Global i, j, k
auto f = [i, &j, &k]() mutable
{
    auto m = [&i, j, &k]() mutable
    {
        i = 4; // Local i of f
        j = 5; // Local j of m
        k = 6; // Global k
    };
    m();
    std::cout << i << j << k; // Local i of f, Global j, Global k
};
f();
std::cout << " : " << i << j << k; // Global i, j, k
```

426 : 126  
Programming in Modern C++ Partha Pratim Das MS2.26

This is another nested example of mutable which I will leave for you to read, execute and understand.

(Refer Slide Time: 33:59)



[=]()->rt{...}: Capture: Mutable

- Will this compile? If so, what is the result?

```
struct foo {
    foo() : i(0) {}
    void amazing(){ [=] i = 8; }(); // i is captured by value // Can it be changed without mutable?
    int i;
};
foo f;
f.amazing();
std::cout << "f.i : " << f.i;
```

Output: f.i : 8

- **this** implicitly captured
- **i** actually is **this->i** which can be written from a member function as a data member. So no **mutable** is required

Handwritten annotations: "this" circles around the struct name and the member function name. Arrows point from "this" to the lambda capture [=] and to the variable i in the member function.

One more of this, the interesting thing about this example is this is a structure and this is a member function and I am changing within this member function the value i, the question is, do I need mutable here? I am changing a value which is captured by a variable which is captured by value. The answer is no, I do not need a mutable here.

Because if I capture by default, then in the context of a class, that this pointer gets captured by default. So, when I write i, it actually means this->i and a member function using this pointer i can always change the data members of the class. So, here though it looks like a case of mutable this does not get mutable, it works according to the class rules.

(Refer Slide Time: 35:05)

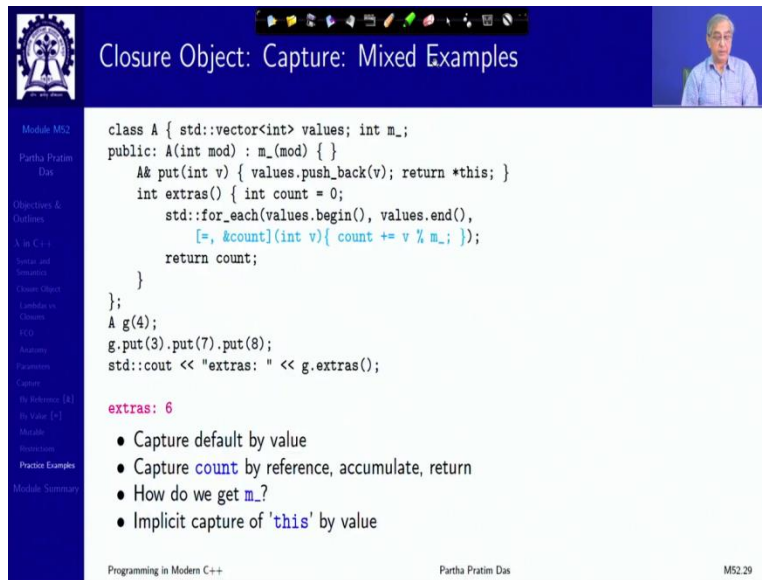
**Capture: Restrictions**

- Capture restrictions
  - Identifiers must only be listed once
    - `[i,j,&z]()` {...} // Okay
    - `[&a,b]()` {...} // Okay
    - `[z,&i,z]()` {...} // Bad, z listed twice
  - Default by value, explicit identifiers by reference
    - `[=,&j,&z]()` {...} // Okay
    - `[=,this]()` {...} // Bad, no this with default =
    - `[=,&i,z]()` {...} // Bad, z by value
  - Default by reference, explicit identifiers by value
    - `[&,j,z]()` {...} // Okay
    - `[&,this]()` {...} // Okay
    - `[&,i,&z]()` {...} // Bad, z by reference
- Scope of Capture
  - Captured entity must be defined or captured in the immediate enclosing lambda expression or function

Programming in Modern C++ Partha Pratim Das MS2.28

Here are some restrictions on capture, like, you cannot have the same variable captured twice or you cannot capture this directly, well, these are these are changing with the as a language dialects are moving, but these are the basic rules of exception that exist in terms of capturing in lambdas.

(Refer Slide Time: 35:37)



The slide displays a C++ class `A` with a `values` vector and a `put` method. The `extras` method uses a lambda function to iterate over the vector and calculate a sum. The output shows `extras: 6`.

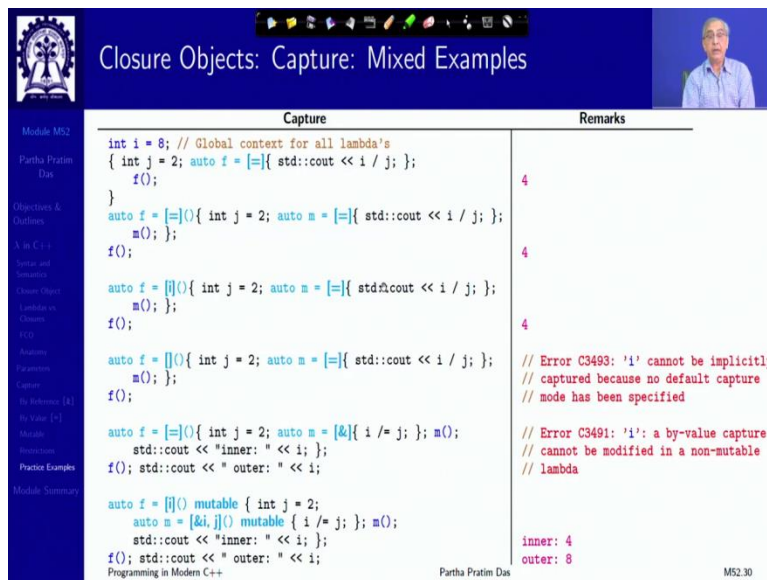
```
class A { std::vector<int> values; int m_;
public: A(int mod) : m_(mod) { }
A& put(int v) { values.push_back(v); return *this; }
int extras() { int count = 0;
std::for_each(values.begin(), values.end(),
[=, &count](int v){ count += v % m_; });
return count;
}
};
A g(4);
g.put(3).put(7).put(8);
std::cout << "extras: " << g.extras();
```

**extras: 6**

- Capture default by value
- Capture `count` by reference, accumulate, return
- How do we get `m_`?
- Implicit capture of `'this'` by value

And finally, there are a couple of mixed examples. So, here is a capture by default by value and a specific variable by reference and see what happens in that case.

(Refer Slide Time: 35:54)



Capture	Remarks
<pre>int i = 8; // Global context for all lambda's { int j = 2; auto f = [=]{ std::cout &lt;&lt; i / j; }; f(); }</pre>	4
<pre>auto f = [=](){ int j = 2; auto m = [=]{ std::cout &lt;&lt; i / j; }; m(); }; f();</pre>	4
<pre>auto f = [](){ int j = 2; auto m = [=]{ std::cout &lt;&lt; i / j; }; m(); }; f();</pre>	4
<pre>auto f = [](){ int j = 2; auto m = [=]{ std::cout &lt;&lt; i / j; }; m(); }; f();</pre>	// Error C3493: 'i' cannot be implicitly // captured because no default capture // mode has been specified
<pre>auto f = [=](){ int j = 2; auto m = [&amp;]{ i /= j; }; m(); std::cout &lt;&lt; "inner: " &lt;&lt; i; }; f(); std::cout &lt;&lt; " outer: " &lt;&lt; i;</pre>	// Error C3491: 'i': a by-value capture // cannot be modified in a non-mutable // lambda
<pre>auto f = [](){ mutable { int j = 2; auto m = [&amp;i, j](){ mutable { i /= j; }; m(); std::cout &lt;&lt; "inner: " &lt;&lt; i; }; f(); std::cout &lt;&lt; " outer: " &lt;&lt; i;</pre>	inner: 4 outer: 8

There are a couple of different examples of default capture, nested capture, mutable capture, with mutable and so on. So, in each case, you will study understand what is happening try to execute in the compiler and understand the total semantics.

(Refer Slide Time: 36:14)

Module Summary

- Understood  $\lambda$  expressions (unnamed function objects) in C++ with
  - o Closure Objects
  - o Parameters
  - o Capture

Programming in Modern C++ Partha Pratim Das M52.31

So, this brings us to the end of this module. In summary, we have understood the Lambda Expressions in C++11, the basic foundation of that the closure object the parameters and the capture. In the next module we will continue to discuss about some more features of lambdas in C++11. Thank you for your attention and see you in the next module.