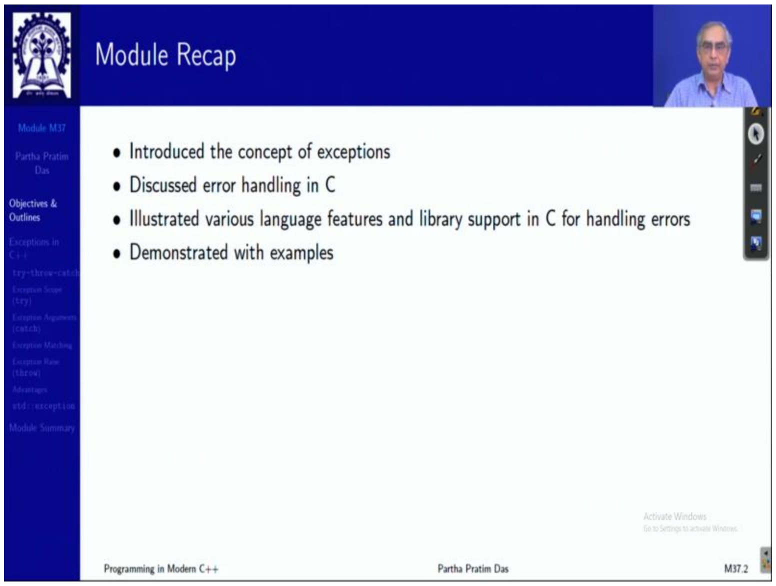
# Programming in Modern C++
**Professor Partha Pratim Das**
**Department of Computer Science and Engineering**
**Indian Institute of Technology, Kharagpur**
**Lecture 37**
**Exceptions (Error handling in C++): Part 2**

Welcome to Programming in Modern C++. We are in Week 8 and we are now going to discuss Module 37.

(Refer Slide Time: 00:37)

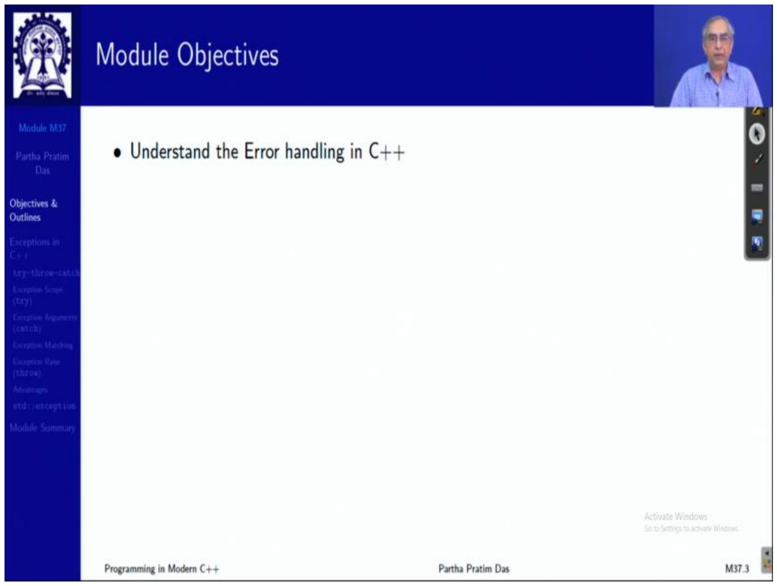


## Module Recap

- Introduced the concept of exceptions
- Discussed error handling in C
- Illustrated various language features and library support in C for handling errors
- Demonstrated with examples

In the last module, we started talking about error handling in C and C++ programs and we discussed the variety of mechanisms that the developers engage in handling errors in C, most of them come from a set of different standard library headers, and none of them are complete support, but a judicious combination of them, depending on different context, must be used to provide a good error handling support in C. We also introduced the concept of exceptions formally and making little distinction between errors in general and exceptions, which are unexpected and infrequent and demonstrated with examples.

(Refer Slide Time: 01:36)



## Module Objectives

- Understand the Error handling in C++

Module M37

Partha Pratim Das

Objectives & Outlines

Exceptions in C++

try-throw-catch

Exception Scope (try)

Exception Arguments (catch)

Exception Matching

Exception Raise (throw)

Advantages

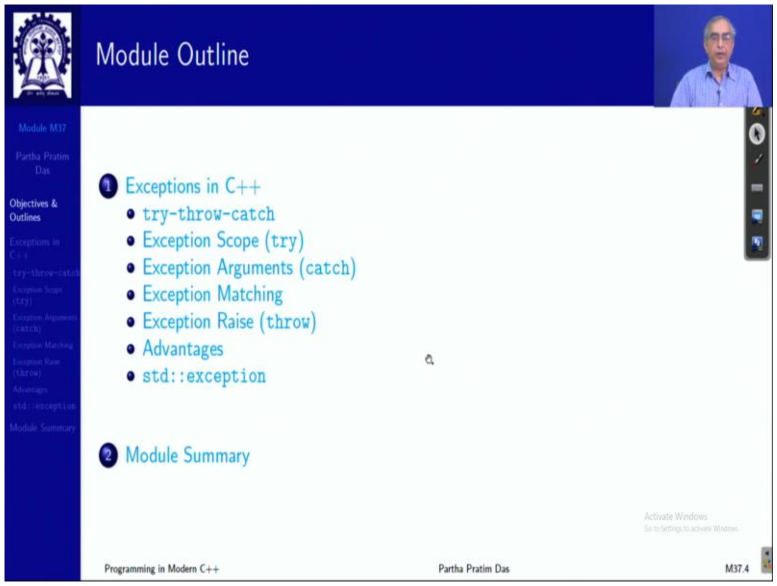
std::exception

Module Summary

Programming in Modern C++ Partha Pratim Das M37.3

We will continue on that in this second and concluding part of error handling discussions, particularly focusing on what C++ provides, how does C++ actually introduce a single exception handling mechanism to encompass all kind of situations that may arise in terms of synchronous, as well as asynchronous errors in a program.

(Refer Slide Time: 02:02)



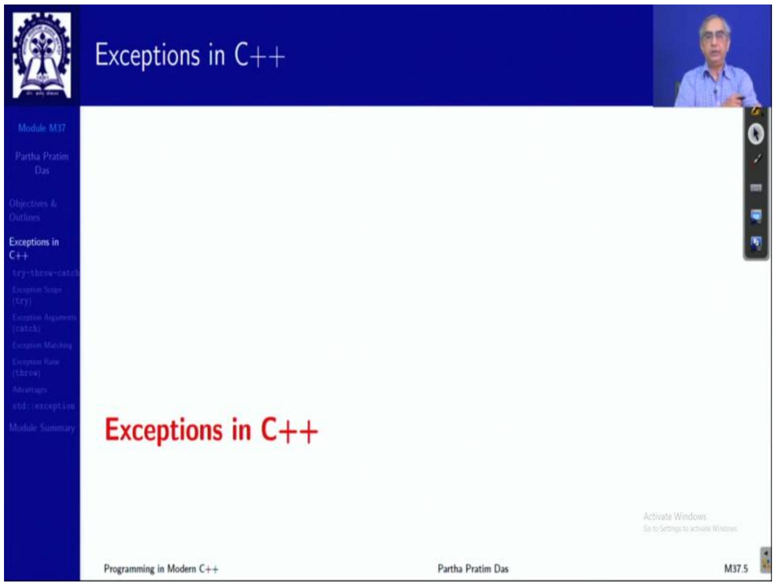
## Module Outline

1. Exceptions in C++
   - try-throw-catch
   - Exception Scope (try)
   - Exception Arguments (catch)
   - Exception Matching
   - Exception Raise (throw)
   - Advantages
   - std::exception
2. Module Summary

Programming in Modern C++ Partha Pratim Das M37.4

This outline will be available on the left panel as you know.

(Refer Slide Time: 02:06)



## Exceptions in C++

Module M37

Partha Pratim Das

Objectives & Outlines

Exceptions in C++

try-throw-catch

Exception Scope (try)

Exception Arguments (catch)

Exception Matching

Exception Raise (throw)

Advantages

std::exception

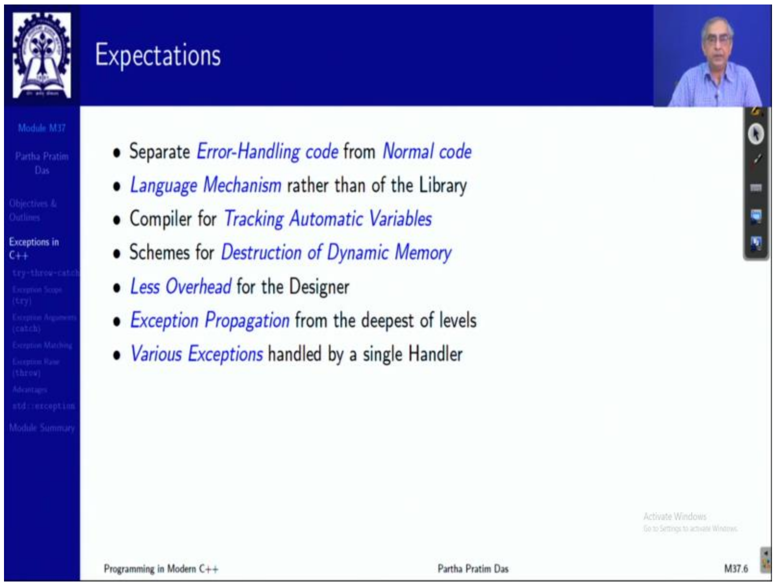
Module Summary

**Exceptions in C++**

Programming in Modern C++ Partha Pratim Das M37.5

Now, let us walk straight into discussing exceptions in C++.

(Refer Slide Time: 02:15)



## Expectations

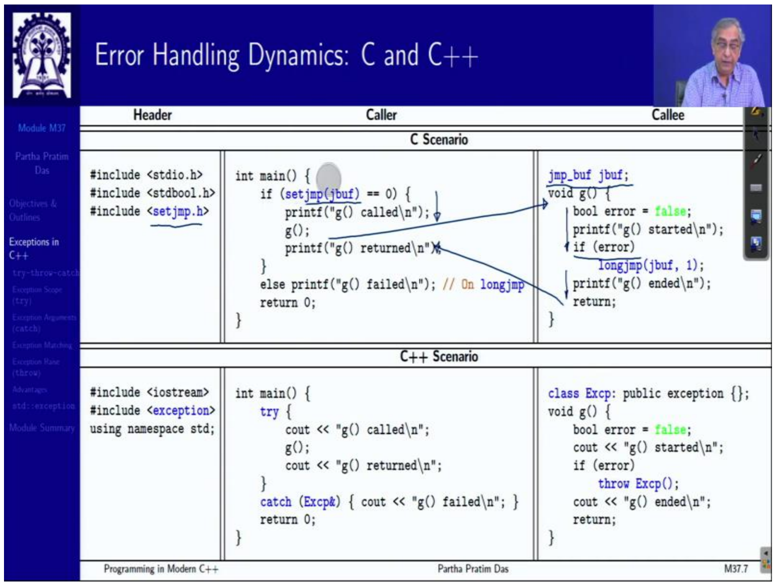
- Separate *Error-Handling code* from *Normal code*
- *Language Mechanism* rather than of the Library
- Compiler for *Tracking Automatic Variables*
- Schemes for *Destruction of Dynamic Memory*
- *Less Overhead* for the Designer
- *Exception Propagation* from the deepest of levels
- *Various Exceptions* handled by a single Handler

Programming in Modern C++ Partha Pratim Das M37.6

These are few characteristics that are desirable and incidentally are supported in C++. The first thing is it allows to separate that error handling flow code from the normal code. In C, this was not possible as we saw, they are all mixed up. Here exception flow and the normal flow are separated in the program at separate distinct spaces. It is a language mechanic rather than support from the standard library as we saw in C. The compilers can track automatic variables and as is required in C++, it can do the destruction of automatic variables when, because of error, without following the normal flow, the control has to go out of a certain scope.

It has schemes for destruction of dynamic memory. It provides naturally much less overhead for the designer, the developer, the programmer to take care of different error and exception situations in the program. It can come out of deep levels of nesting propagating the error in an appropriate way, which we saw we were kind of trying to do using local gotos. We do not need to do that anymore. Exceptions support this kind of proper exit from the deepest level of nesting and various exceptions can be handled by a single handler. So, these are the expectations and they are rightly met by the exception handling mechanism in C++.

(Refer Slide Time: 04:13)



## Error Handling Dynamics: C and C++

Module M37

Partha Pratim Das

Objectives & Outlines

Exceptions in C++

try-throw-catch

Exception Scope (try)

Exception Arguments (catch)

Exception Matching

Exception Raise (throw)

Advantages

std::exception

Module Summary

Header | Caller | Callee

**C Scenario**

Header:

```
#include <stdio.h>
#include <stdbool.h>
#include <setjmp.h>
```

Caller:

```
int main() {
    if (setjmp(jbuf) == 0) {
        printf("g() called\n");
        g();
        printf("g() returned\n");
    }
    else printf("g() failed\n"); // On longjmp
    return 0;
}
```

Callee:

```
jmp_buf jbuf;
void g() {
    bool error = false;
    printf("g() started\n");
    if (error)
        longjmp(jbuf, 1);
    printf("g() ended\n");
    return;
}
```

**C++ Scenario**

Header:

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

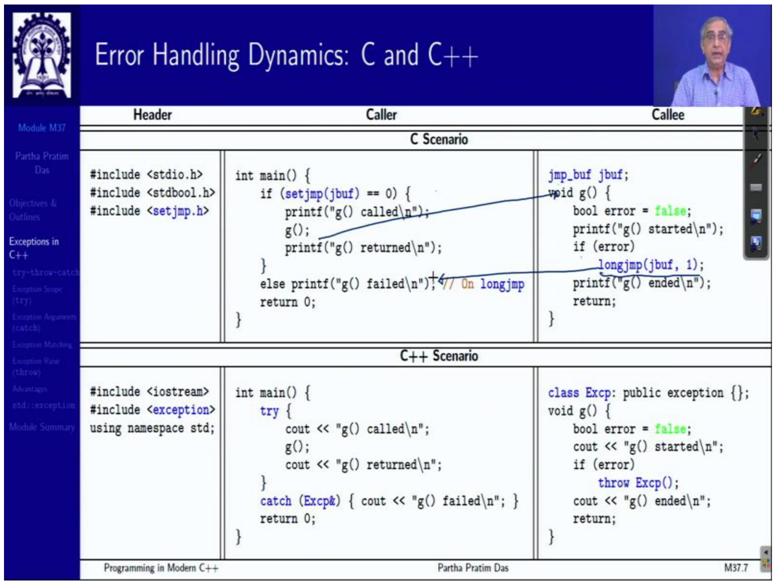
Caller:

```
int main() {
    try {
        cout << "g() called\n";
        g();
        cout << "g() returned\n";
    }
    catch (Excp&) { cout << "g() failed\n"; }
    return 0;
}
```

Callee:

```
class Excp: public exception {};
void g() {
    bool error = false;
    cout << "g() started\n";
    if (error)
        throw Excp();
    cout << "g() ended\n";
    return;
}
```

Programming in Modern C++ Partha Pratim Das M37.7



## Error Handling Dynamics: C and C++

Module M37

Partha Pratim Das

Objectives & Outlines

Exceptions in C++

try-throw-catch

Exception Scope (try)

Exception Arguments (catch)

Exception Matching

Exception Raise (throw)

Advantages

std::exception

Module Summary

**C Scenario**

Header:

```
#include <stdio.h>
#include <stdbool.h>
#include <setjmp.h>
```

Caller:

```
int main() {
    if (setjmp(jbuf) == 0) {
        printf("g() called\n");
        g();
        printf("g() returned\n");
    }
    else printf("g() failed\n"); // On longjmp
    return 0;
}
```

Callee:

```
jmp_buf jbuf;
void g() {
    bool error = false;
    printf("g() started\n");
    if (error)
        longjmp(jbuf, 1);
    printf("g() ended\n");
    return;
}
```

**C++ Scenario**

Header:

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

Caller:

```
int main() {
    try {
        cout << "g() called\n";
        g();
        cout << "g() returned\n";
    }
    catch (Excp&) { cout << "g() failed\n"; }
    return 0;
}
```

Callee:

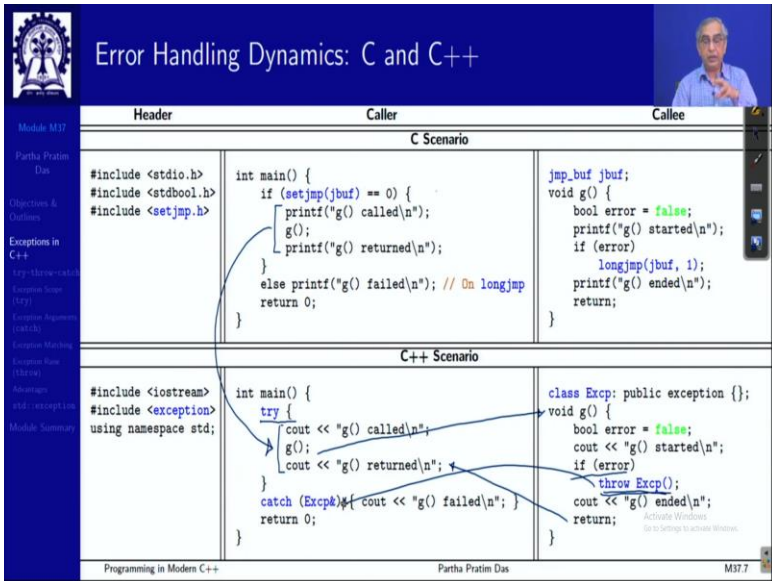
```
class Excp: public exception {};
void g() {
    bool error = false;
    cout << "g() started\n";
    if (error)
        throw Excp();
    cout << "g() ended\n";
    return;
}
```

Programming in Modern C++ Partha Pratim Das M37.7

So, here we start with the, putting in parallel the C solution for exception handling, which is closest to the C++, the non-local goto and the C++ solution. So, in the terms of non-local goto, we know that we need to include this header, define a buffer to keep the environment of the caller and in the caller, you do a setjmp to set the environment and then you proceed in terms of the normal flow. The function is called and the function continues.

Now, if there is an error, then a longjmp is executed. Otherwise, the function continues as it is and returns to this point. The statement after g(). But if there is a, if there is a longjmp, that is if there has been an instance of error which you are raising as in here, then the control that had gone from main to g() will come back to the exception part, to the handling part, which is what the jbuf, the jmp_buf environment context helps us to do. Naturally, this means that if we have different kind of errors, different functions, or within the same function, different long jumps to be done to different contexts, then we need to set different buffers.

(Refer Slide Time: 06:14)



## Error Handling Dynamics: C and C++

Module M37

Partha Pratim Das

Objectives & Outlines

Exceptions in C++

try-throw-catch

Exception Scope (try)

Exception Arguments (catch)

Exception Matching

Exception Raise (throw)

Advantages

std::exception

Module Summary

Header | Caller | Callee

**C Scenario**

```
#include <stdio.h>
#include <stdbool.h>
#include <setjmp.h>
```

```
int main() {
    if (setjmp(jbuf) == 0) {
        printf("g() called\n");
        g();
        printf("g() returned\n");
    }
    else printf("g() failed\n"); // On longjmp
    return 0;
}
```

```
jmp_buf jbuf;
void g() {
    bool error = false;
    printf("g() started\n");
    if (error)
        longjmp(jbuf, 1);
    printf("g() ended\n");
    return;
}
```

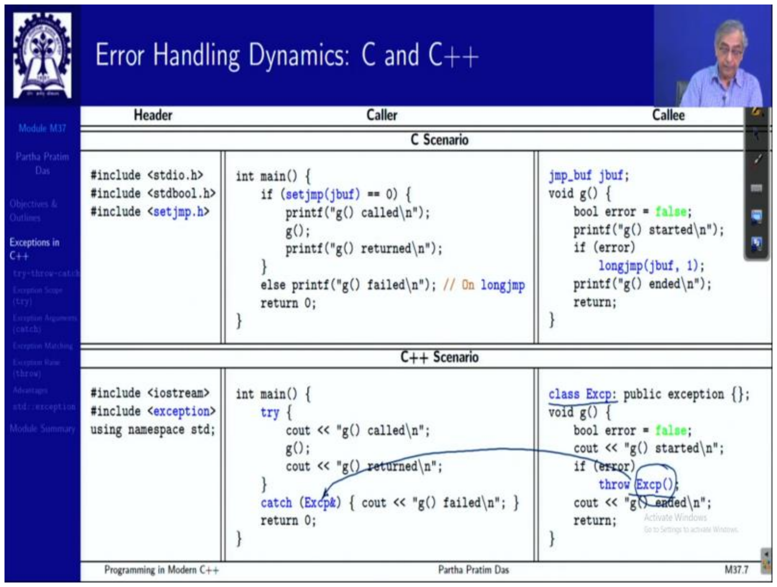
**C++ Scenario**

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

```
int main() {
    try {
        cout << "g() called\n";
        g();
        cout << "g() returned\n";
    }
    catch (Excp&) { cout << "g() failed\n"; }
    return 0;
}
```

```
class Excp: public exception {};
void g() {
    bool error = false;
    cout << "g() started\n";
    if (error)
        throw Excp();
    cout << "g() ended\n";
    return;
}
```

Programming in Modern C++ Partha Pratim Das M37.7



## Error Handling Dynamics: C and C++

Module M37

Partha Pratim Das

Objectives & Outlines

Exceptions in C++

try-throw-catch

Exception Scope (try)

Exception Arguments (catch)

Exception Matching

Exception Raise (throw)

Advantages

std::exception

Module Summary

Header | Caller | Callee

**C Scenario**

```
#include <stdio.h>
#include <stdbool.h>
#include <setjmp.h>
```

```
int main() {
    if (setjmp(jbuf) == 0) {
        printf("g() called\n");
        g();
        printf("g() returned\n");
    }
    else printf("g() failed\n"); // On longjmp
    return 0;
}
```

```
jmp_buf jbuf;
void g() {
    bool error = false;
    printf("g() started\n");
    if (error)
        longjmp(jbuf, 1);
    printf("g() ended\n");
    return;
}
```

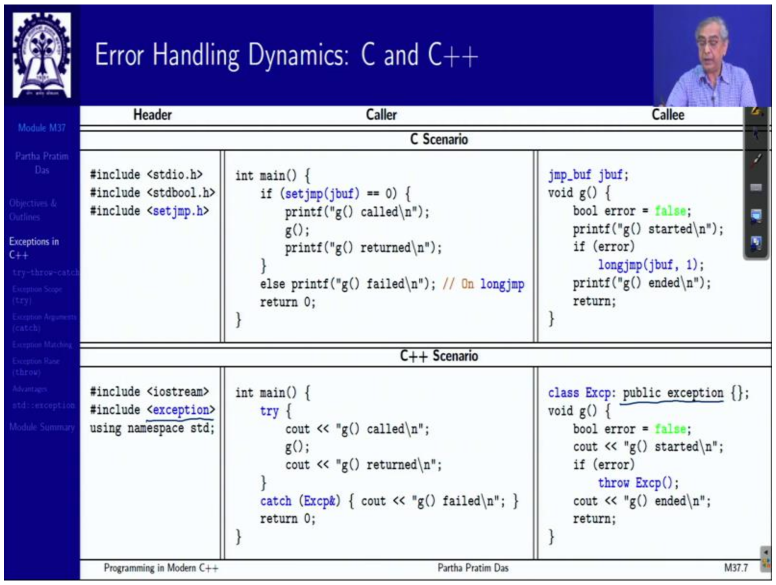
**C++ Scenario**

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

```
int main() {
    try {
        cout << "g() called\n";
        g();
        cout << "g() returned\n";
    }
    catch (Excp&) { cout << "g() failed\n"; }
    return 0;
}
```

```
class Excp: public exception {};
void g() {
    bool error = false;
    cout << "g() started\n";
    if (error)
        throw Excp();
    cout << "g() ended\n";
    return;
}
```

Programming in Modern C++ Partha Pratim Das M37.7



## Error Handling Dynamics: C and C++

Module M37

Partha Pratim Das

Objectives & Outlines

Exceptions in C++

try-throw-catch

Exception Scope (try)

Exception Arguments (catch)

Exception Matching

Exception Raise (throw)

Advantages

std::exception

Module Summary

Header | Caller | Callee

C Scenario

Header:

```
#include <stdio.h>
#include <stdbool.h>
#include <setjmp.h>
```

Caller:

```
int main() {
    if (setjmp(jbuf) == 0) {
        printf("g() called\n");
        g();
        printf("g() returned\n");
    }
    else printf("g() failed\n"); // On longjmp
    return 0;
}
```

Callee:

```
jmp_buf jbuf;
void g() {
    bool error = false;
    printf("g() started\n");
    if (error)
        longjmp(jbuf, 1);
    printf("g() ended\n");
    return;
}
```

C++ Scenario

Header:

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

Caller:

```
int main() {
    try {
        cout << "g() called\n";
        g();
        cout << "g() returned\n";
    }
    catch (Excp&) { cout << "g() failed\n"; }
    return 0;
}
```

Callee:

```
class Excp: public exception {};
void g() {
    bool error = false;
    cout << "g() started\n";
    if (error)
        throw Excp();
    cout << "g() ended\n";
    return;
}
```

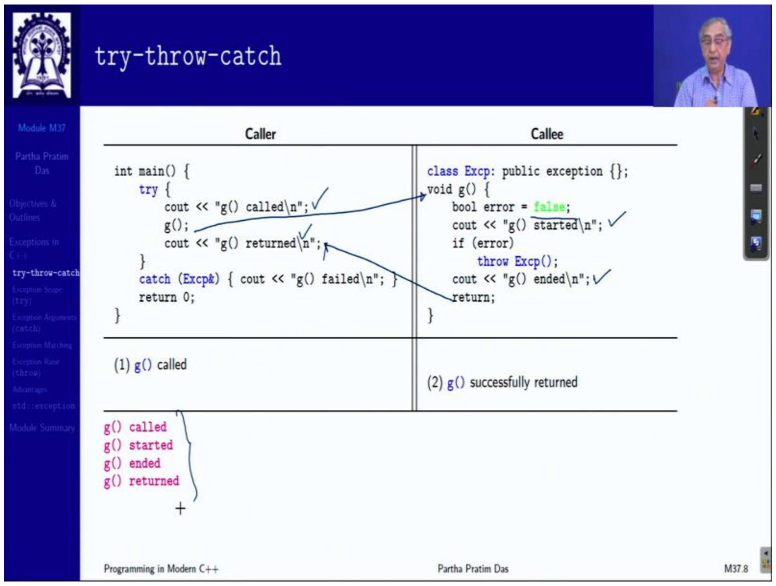
Programming in Modern C++ Partha Pratim Das M37.7

Now, in C++, this has all been made very simple. You just need to write the code, this equivalent code as in here within a pair of curly braces and prefix it by try. So, the sense is you are saying that I am trying this code out, there may be some issues. It goes normally calls g(). If there is an error, it will throw. If there is no error, it will return to the designated point. If there is an error, then it throws.

And when it throws, it directly comes to what is written as catch, known as a catch clause. So, we call this mechanism, try-throw-catch, you try out a code. If the code does not behave, it will throw. And if it throws, it will be caught by the handler to do subsequent things. This is the basic mechanism. Now, if we look at the throw, has to throw an exception object. This exception object is the one where the callee can put enough information for the caller to proceed. So, it will typically have a class defined for this with the appropriate behavior that you need and you construct. So, this is, as you can see, this is a kind of almost an explicit call to the constructor, which is, which will prepare and exception object, unnamed exception object, which will be thrown and be caught here by this type.

You can any kind of exception object can be thrown. It can be an integer, it can be a character. It can be your own class, but it is often customary to specialize from a library provided, standard library provided exception class, which has some well-defined behavior. We will see what those behaviors are, and use that. You can use your own exception class as well.

(Refer Slide Time: 08:46)



## try-throw-catch

Module M37

Partha Pratim Das

Objectives & Outlines

Exceptions in C++

try-throw-catch

Exception Scope (try)

Exception Arguments (catch)

Exception Matching

Exception Raise (throw)

Advantages

std::exception

Module Summary

Caller

```
int main() {
    try {
        cout << "g() called\n";
        g();
        cout << "g() returned\n";
    }
    catch (Excp&) { cout << "g() failed\n"; }
    return 0;
}
```

(1) g() called

Callee

```
class Excp: public exception {};
void g() {
    bool error = false;
    cout << "g() started\n";
    if (error)
        throw Excp();
    cout << "g() ended\n";
    return;
}
```

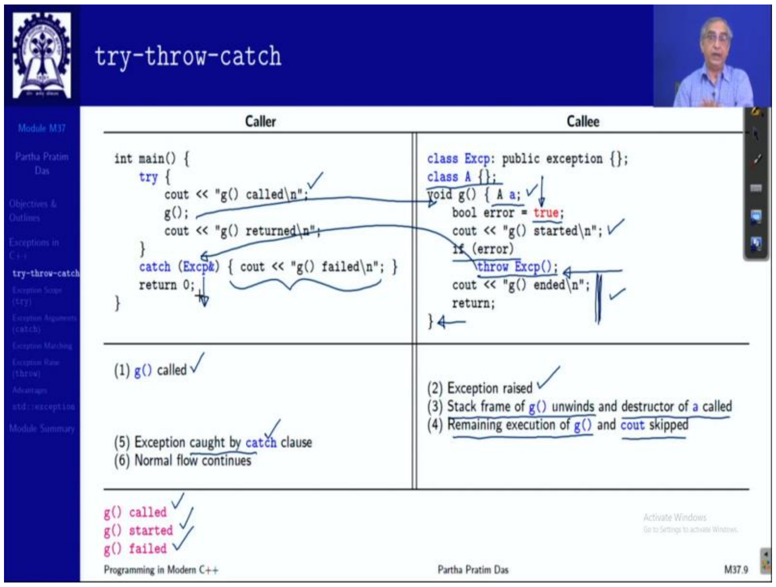
(2) g() successfully returned

```
g() called
g() started
g() ended
g() returned
```

Programming in Modern C++ Partha Pratim Das M37.8

Now, with this, let us see what will happen in terms of the try catch, try-throw-catch. Naturally, if there is no error, as I said, if there is no error, the simple thing is it calls it returns to the next statement. So, this will see out the called, cout started, cout ended and cout returned as you can see. g() is called, g() successfully returned. There is no surprise, there is no difference in terms of the normal code that you see.

(Refer Slide Time: 09:22)



## try-throw-catch

Module M37

Partha Pratim Das

Objectives & Outlines

Exceptions in C++

try-throw-catch

Exception Scope (try)

Exception Arguments (catch)

Exception Matching

Exception Raise (throw)

Advantages

std::exception

Module Summary

Caller

```
int main() {
    try {
        cout << "g() called\n";
        g();
        cout << "g() returned\n";
    }
    catch (Excp&) { cout << "g() failed\n"; }
    return 0;
}
```

(1) g() called

(5) Exception caught by catch clause
(6) Normal flow continues

Callee

```
class Excp: public exception {};
class A {};
void g() { A a;
    bool error = true;
    cout << "g() started\n";
    if (error)
        throw Excp();
    cout << "g() ended\n";
    return;
}
```

(2) Exception raised
(3) Stack frame of g() unwinds and destructor of a called
(4) Remaining execution of g() and cout skipped

```
g() called
g() started
g() failed
```

Activate Windows
Go to Settings to activate Windows.

Programming in Modern C++ Partha Pratim Das M37.9

However, if there is an error, now I am introducing as if there is an error. So, when the try, from the, within the try block, as I call g(), g() goes in, gets the error, executes, so called has happened, started has happened, and then some logic happens and there is an error. It constructs the exception object and throws it. As it throws, the remaining part of the code is not executed. So, the process is g() is called, exception is raised.

Now, at this point, the function will, would like to return the control back to the caller. Now, as you know, when the function is called, a stack frame is created for g(), which has a local variables and all that. And when it has to get out, when it completes a task and gets out of that function scope, that stack frame has to be removed, maybe logically removed, not actually removed in terms of values, but you change the pointers in a way so that it does not remain the active stack frame anymore.

So, when you are going out of the callee's context for a throw, an exception condition, the stack frame of g() will have to be unwinded. It has to be rolled back, because after this, the control will go to the caller and the stack frame of the caller needs to be activated. Now, what is the consequence of unwinding this stack frame? Is this just it a stack frame?

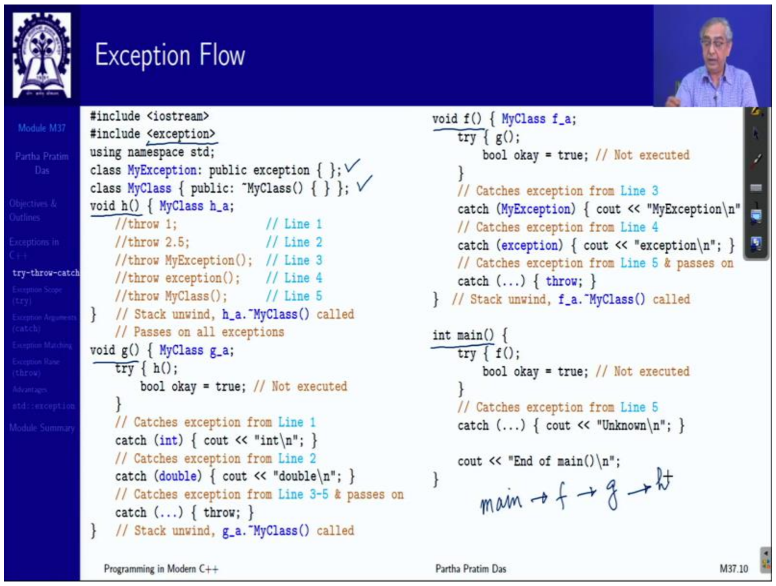
So, does this frame was on the top? Do I just put the top of the stack below this frame pointing to the frame of main, and am I done? In C, that is what we assume, but that is not, that is what longjmp will do. But in C++, the critical thing is there are several local automatic objects. So, I have just introduced some arbitrary class A and an object which is automatic in the function scope.

Now, this object, certainly when the control passed here, this object was constructed. If I had a normal flow at this point, this object would have been destructed. But I am not having a normal flow. This part is not going to get executed. So, what do I need to do before the throw happens is to also destruct the object a, because the stack frame is getting unwinded. So, wherever the object a was, that is not a part of the computation anymore. So, this is a very, very important behavior of throw in C++, which is critical for the correctness of all live objects in their lifetime.

Naturally, after that remaining execution of g() and the cout, these are skipped, and the control comes to the catch block, and the control as we, as it says, it is a, as if a ball has been thrown, and as the fielder does, we are catching that ball. We are catching that exception object thrown. And it is caught by the catch clause, catch clause does whatever handling is required, in this case, we have not put anything except for a statement. So, the failed is given on the output.

And then after this, the normal flow continues in main. It has to decide what to do about the incomplete execution of g(), whether it will call g() again, or call some other function and so on, all those are considerations of the developer. But this is all about the mechanism of try-catch-throw which is very, very important for any kind of error handling in C++.

(Refer Slide Time: 13:50)



## Exception Flow

Module M37

Partha Pratim Das

Objectives & Outlines

Exceptions in C++

try-throw-catch

Exception Scope (try)

Exception Arguments (catch)

Exception Matching

Exception Raise (throw)

Advantages

std::exception

Module Summary

```
#include <iostream>
#include <exception>
using namespace std;
class MyException: public exception { };
class MyClass { public: ~MyClass() { } };
void h() { MyClass h_a;
    //throw 1;              // Line 1
    //throw 2.5;            // Line 2
    //throw MyException();  // Line 3
    //throw exception();    // Line 4
    //throw MyClass();      // Line 5
}   // Stack unwind, h_a.~MyClass() called
    // Passes on all exceptions
void g() { MyClass g_a;
    try { h();
        bool okay = true; // Not executed
    }
    // Catches exception from Line 1
    catch (int) { cout << "int\n"; }
    // Catches exception from Line 2
    catch (double) { cout << "double\n"; }
    // Catches exception from Line 3-5 & passes on
    catch (...) { throw; }
}   // Stack unwind, g_a.~MyClass() called
```

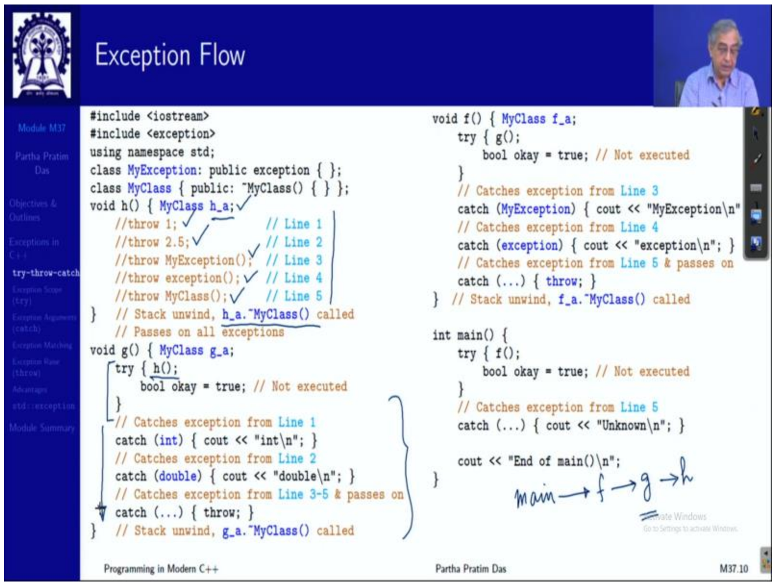
```
void f() { MyClass f_a;
    try { g();
        bool okay = true; // Not executed
    }
    // Catches exception from Line 3
    catch (MyException) { cout << "MyException\n"
    // Catches exception from Line 4
    catch (exception) { cout << "exception\n"; }
    // Catches exception from Line 5 & passes on
    catch (...) { throw; }
}  // Stack unwind, f_a.~MyClass() called

int main() {
    try { f();
        bool okay = true; // Not executed
    }
    // Catches exception from Line 5
    catch (...) { cout << "Unknown\n"; }

    cout << "End of main()\n";
}
```

$main \rightarrow f \rightarrow g \rightarrow h$

Programming in Modern C++ Partha Pratim Das M37.10



## Exception Flow

Module M37

Partha Pratim Das

Objectives & Outlines

Exceptions in C++

try-throw-catch

Exception Scope (try)

Exception Arguments (catch)

Exception Matching

Exception Raise (throw)

Advantages

std::exception

Module Summary

```
#include <iostream>
#include <exception>
using namespace std;
class MyException: public exception { };
class MyClass { public: ~MyClass() { } };
void h() { MyClass h_a;
    //throw 1;              // Line 1
    //throw 2.5;            // Line 2
    //throw MyException();  // Line 3
    //throw exception();    // Line 4
    //throw MyClass();      // Line 5
}   // Stack unwind, h_a.~MyClass() called
    // Passes on all exceptions
void g() { MyClass g_a;
    try { h();
        bool okay = true; // Not executed
    }
    // Catches exception from Line 1
    catch (int) { cout << "int\n"; }
    // Catches exception from Line 2
    catch (double) { cout << "double\n"; }
    // Catches exception from Line 3-5 & passes on
    catch (...) { throw; }
}   // Stack unwind, g_a.~MyClass() called
```

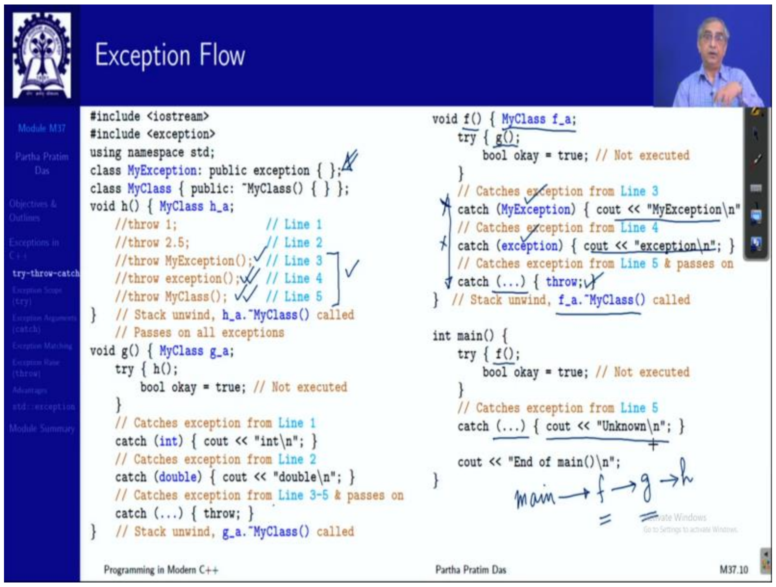
```
void f() { MyClass f_a;
    try { g();
        bool okay = true; // Not executed
    }
    // Catches exception from Line 3
    catch (MyException) { cout << "MyException\n"
    // Catches exception from Line 4
    catch (exception) { cout << "exception\n"; }
    // Catches exception from Line 5 & passes on
    catch (...) { throw; }
}  // Stack unwind, f_a.~MyClass() called

int main() {
    try { f();
        bool okay = true; // Not executed
    }
    // Catches exception from Line 5
    catch (...) { cout << "Unknown\n"; }

    cout << "End of main()\n";
}
```

$main \rightarrow f \rightarrow g \rightarrow h$

Programming in Modern C++ Partha Pratim Das M37.10


## Exception Flow

```
#include <iostream>
#include <exception>
using namespace std;
class MyException: public exception { };
class MyClass { public: ~MyClass() { } };
void h() { MyClass h_a;
    //throw 1;              // Line 1
    //throw 2.5;            // Line 2
    //throw MyException();  // Line 3
    //throw exception();    // Line 4
    //throw MyClass();      // Line 5
}   // Stack unwind, h_a.~MyClass() called
    // Passes on all exceptions
void g() { MyClass g_a;
    try { h();
        bool okay = true; // Not executed
    }
    // Catches exception from Line 1
    catch (int) { cout << "int\n"; }
    // Catches exception from Line 2
    catch (double) { cout << "double\n"; }
    // Catches exception from Line 3-5 & passes on
    catch (...) { throw; }
}   // Stack unwind, g_a.~MyClass() called
```

```
void f() { MyClass f_a;
    try { g();
        bool okay = true; // Not executed
    }
    // Catches exception from Line 3
    catch (MyException) { cout << "MyException\n"
    // Catches exception from Line 4
    catch (exception) { cout << "exception\n"; }
    // Catches exception from Line 5 & passes on
    catch (...) { throw; }
}  // Stack unwind, f_a.~MyClass() called

int main() {
    try { f();
        bool okay = true; // Not executed
    }
    // Catches exception from Line 5
    catch (...) { cout << "Unknown\n"; }

    cout << "End of main()\n";
}
```

$main \rightarrow f \rightarrow g \rightarrow h$

Module M37

Partha Pratim Das

Objectives & Outlines

Exceptions in C++

try-throw-catch

Exception Scope (try)

Exception Arguments (catch)

Exception Matching

Exception Raise (throw)

Advantages

std::exception

Module Summary

Programming in Modern C++ Partha Pratim Das M37.10

Here I have given a kind of small but detailed example with different cases. As I always say that I try to put some, one slide summary of the important concept, so it is that kind of a slide. So, let us spend a little bit of time here. I have a MyException class derived from public exception, which is a standard library exception class coming in this standard library exception. I have another class, MyClass, which is, you know, just to create objects.

Let us say we have a global, I mean, for simplicity, I have not gone into member functions, I am just dealing with global functions. So, there are three global functions here, f(), g(), and h(), and naturally, main. So, the flow is main will call f(), which in turn, will call g(), which in turn will call h(). And on this flow, we want to see what is the effect of having different exceptions.

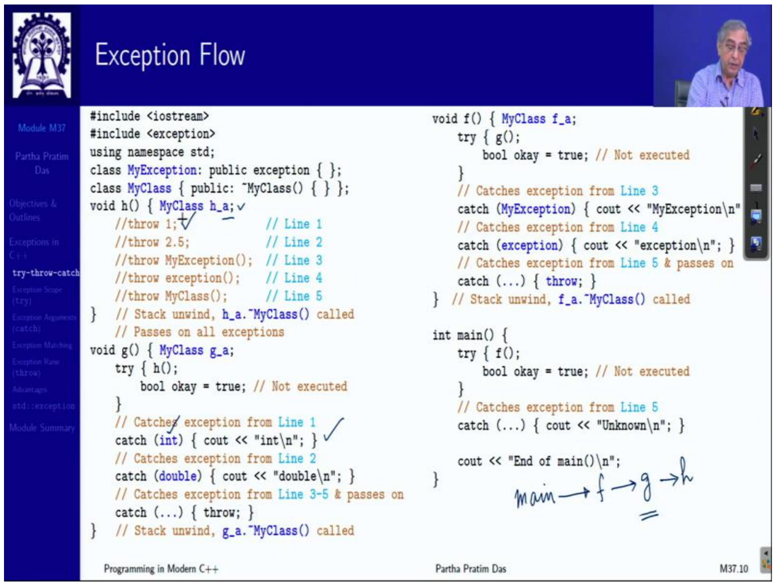
Now, so, h(), I am sorry, I should not erase this. This is important for you to keep in mind. So, h() is the, in this code, h() is the most deep call that I have. Within h(), within the scope, I construct an object of MyClass. This is just for illustration of automatic variable handling. Then I have put in terms of comments, so that you uncomment any one of them and really check out throwing different types of exception objects. This is integer, this is double, this is MyException, which is specialized from exception. This is unspecialized exception, the standard library exception, and even throwing the, a MyClass object. In every case, you will have to construct the object and throw if it is a built-in type, you can just throw the literal.

Now, one thing common that will happen when any one of the throw happens is, since the control goes out from the context of h(), the object which was local, automatic here will have to be destructed, and this will be destructed at this point. So, if you look at the caller of g(), caller of h(), because h() was called from g().

So, if h throws, it will go to g(), that has put h() in a try block. And it has put a number of catch clauses, catch, multiple catch clauses can be put, these are alternates to check what type of error has actually come, which type of exception object has actually come. And they are executed, they are tried out, not executed. They are tried out in this order.

So, you first try to match the first catch clause. If it matches your type of the object thrown by the callee, if it matches the type given in the catch clause, that catch clause is catches that exception and rest of the clauses are skipped. But if it does not, then it moves on to the next catch clause. And this mapping does not use any implicit conversion or anything. It has to be either a perfect match or it can match a specialized class for a, class object for a generalized class object that is upcast is allowed, but implicit conversions are not allowed. So, what will happen? If I think I will reduce this clutter a little bit.

(Refer Slide Time: 18:18)



## Exception Flow

```
#include <iostream>
#include <exception>
using namespace std;
class MyException: public exception { };
class MyClass { public: ~MyClass() { } };
void h() { MyClass h_a;
    //throw 1;             // Line 1
    //throw 2.5;           // Line 2
    //throw MyException(); // Line 3
    //throw exception();   // Line 4
    //throw MyClass();     // Line 5
}   // Stack unwind, h_a.~MyClass() called
    // Passes on all exceptions
void g() { MyClass g_a;
    try { h();
        bool okay = true; // Not executed
    }
    // Catches exception from Line 1
    catch (int) { cout << "int\n"; }
    // Catches exception from Line 2
    catch (double) { cout << "double\n"; }
    // Catches exception from Line 3-5 & passes on
    catch (...) { throw; }
}   // Stack unwind, g_a.~MyClass() called
```

```
void f() { MyClass f_a;
    try { g();
        bool okay = true; // Not executed
    }
    // Catches exception from Line 3
    catch (MyException) { cout << "MyException\n"
    // Catches exception from Line 4
    catch (exception) { cout << "exception\n"; }
    // Catches exception from Line 5 & passes on
    catch (...) { throw; }
}  // Stack unwind, f_a.~MyClass() called

int main() {
    try { f();
        bool okay = true; // Not executed
    }
    // Catches exception from Line 5
    catch (...) { cout << "Unknown\n"; }

    cout << "End of main()\n";
}
```

$\text{main} \rightarrow f \rightarrow \underline{\underline{g}} \rightarrow h$

Module M37

Partha Pratim Das

Objectives & Outlines

Exceptions in C++

try-throw-catch

Exception Scope (try)

Exception Arguments (catch)

Exception Matching

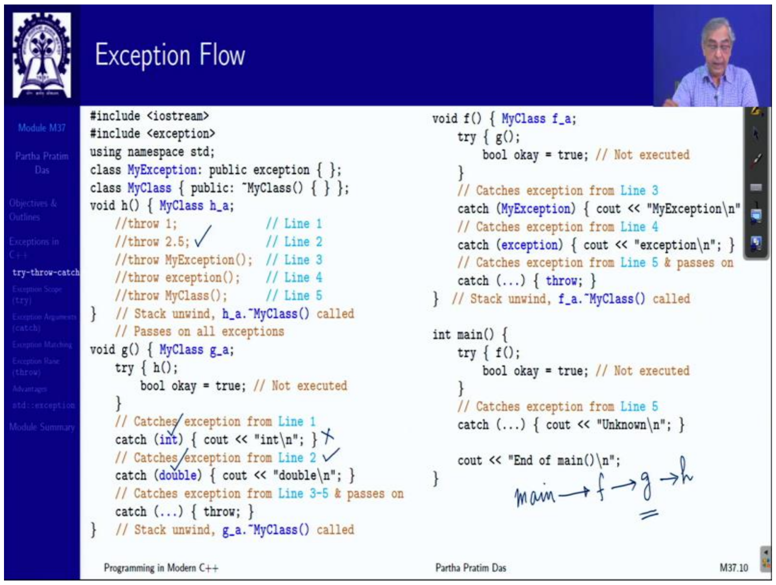
Exception Raise (throw)

Advantages

std::exception

Module Summary

Programming in Modern C++ Partha Pratim Das M37.10



## Exception Flow

Module M37

Partha Pratim Das

Objectives & Outlines

Exceptions in C++

try-throw-catch

Exception Scope (try)

Exception Arguments (catch)

Exception Matching

Exception Raise (throw)

Advantages

std::exception

Module Summary

```
#include <iostream>
#include <exception>
using namespace std;
class MyException: public exception { };
class MyClass { public: ~MyClass() { } };
void h() { MyClass h_a;
    //throw 1;            // Line 1
    //throw 2.5; ✓        // Line 2
    //throw MyException(); // Line 3
    //throw exception();   // Line 4
    //throw MyClass();     // Line 5
}   // Stack unwind, h_a.~MyClass() called
    // Passes on all exceptions
void g() { MyClass g_a;
    try { h();
        bool okay = true; // Not executed
    }
    // Catches exception from Line 1
    catch (int) { cout << "int\n"; } ✗
    // Catches exception from Line 2 ✓
    catch (double) { cout << "double\n"; }
    // Catches exception from Line 3-5 & passes on
    catch (...) { throw; }
}   // Stack unwind, g_a.~MyClass() called
```

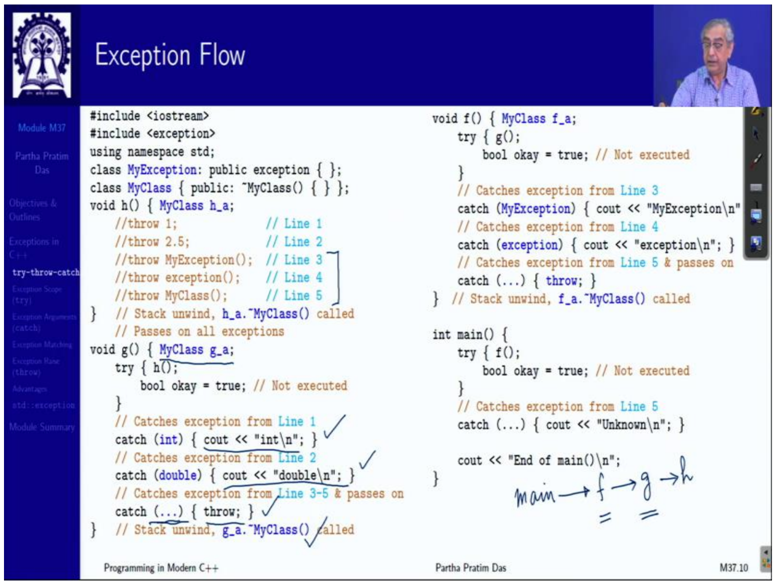
```
void f() { MyClass f_a;
    try { g();
        bool okay = true; // Not executed
    }
    // Catches exception from Line 3
    catch (MyException) { cout << "MyException\n"
    // Catches exception from Line 4
    catch (exception) { cout << "exception\n"; }
    // Catches exception from Line 5 & passes on
    catch (...) { throw; }
}  // Stack unwind, f_a.~MyClass() called

int main() {
    try { f();
        bool okay = true; // Not executed
    }
    // Catches exception from Line 5
    catch (...) { cout << "Unknown\n"; }

    cout << "End of main()\n";
}
```

$$main \rightarrow f \rightarrow \underline{\underline{g}} \rightarrow h$$

Programming in Modern C++ Partha Pratim Das M37.10



## Exception Flow

```
#include <iostream>
#include <exception>
using namespace std;
class MyException: public exception { };
class MyClass { public: ~MyClass() { } };
void h() { MyClass h_a;
    //throw 1;            // Line 1
    //throw 2.5;          // Line 2
    //throw MyException(); // Line 3 ]
    //throw exception();   // Line 4 ]
    //throw MyClass();     // Line 5 ]
}   // Stack unwind, h_a.~MyClass() called
    // Passes on all exceptions
void g() { MyClass g_a;
    try { h();
        bool okay = true; // Not executed
    }
    // Catches exception from Line 1 ✓
    catch (int) { cout << "int\n"; }
    // Catches exception from Line 2 ✓
    catch (double) { cout << "double\n"; }
    // Catches exception from Line 3-5 & passes on
    catch (...) { throw; } ✓
}   // Stack unwind, g_a.~MyClass() called ✓
```

```
void f() { MyClass f_a;
    try { g();
        bool okay = true; // Not executed
    }
    // Catches exception from Line 3
    catch (MyException) { cout << "MyException\n"
    // Catches exception from Line 4
    catch (exception) { cout << "exception\n"; }
    // Catches exception from Line 5 & passes on
    catch (...) { throw; }
}  // Stack unwind, f_a.~MyClass() called

int main() {
    try { f();
        bool okay = true; // Not executed
    }
    // Catches exception from Line 5
    catch (...) { cout << "Unknown\n"; }

    cout << "End of main()\n";
}
```

$$main \rightarrow \underline{\underline{f}} \rightarrow \underline{\underline{g}} \rightarrow h$$

Programming in Modern C++ Partha Pratim Das M37.10

Now, let us say, if I have thrown one, Line 1, the first catch clause is int, so it will match here and this handling will be done, that is int will be printed. If you did not uncomment this, naturally, if you keep this uncommented, then the second throw, you will never be able to see. But if you keep it commented and uncomment this one, then you have a double object coming in. So, in the catch clause, int will not match. Rather, int will go to the next, it is double, so it will match here.

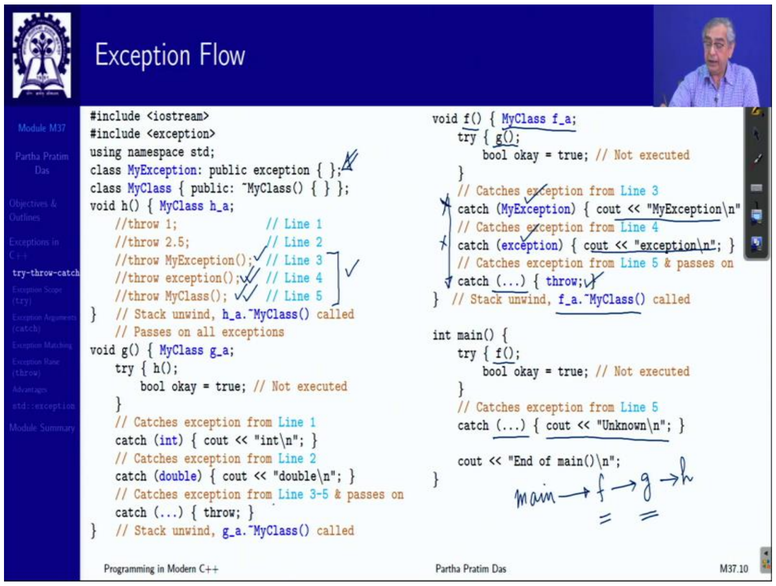
So, the throw from the first line matches catch int, throw from the second line matches double. Then we have something which is interesting, which is called the catch all clause, this is called catch all. That is after everything, everything we have specified, at the end, you can put a catch all clause which will match any type of exception object that has been thrown.

So, there are three possibilities, MyException, exception, and MyClass object, so that they will not match here, they will not match here, so all of them will match in this clause. This is the catch all clause. Now, obviously at this point, with this catch all clause, you really do not know what type of exception object you have got and what type of exception has actually happened.

So, what I show is that it is not necessary for the caller to be able to handle the exception in every case, which it could do for int and for double, but it could just throw it again, it could just re-throw, this is called a re-throw. Just throw, it does not say anything about the object, because it does not know the object type. It has not resolved the object type. So, whatever object it has got, whatever type it has, that object will be passed on.

So, g() will also have now an unwinding of the stack, because it is throwing. So, if g() unwinds, the control will go back to f(), and before doing that, the local automatic object in g() as a part of the stack unwinding process will get distracted. You can see the nice pattern that is being formed.

(Refer Slide Time: 21:03)



## Exception Flow

Module M37

Partha Pratim Das

Objectives & Outlines

Exceptions in C++

try-throw-catch

Exception Scope (try)

Exception Arguments (catch)

Exception Matching

Exception Raise (throw)

Advantages

std::exception

Module Summary

```
#include <iostream>
#include <exception>
using namespace std;
class MyException: public exception { };
class MyClass { public: ~MyClass() { } };
void h() { MyClass h_a;
    //throw 1;              // Line 1
    //throw 2.5;            // Line 2
    //throw MyException();  // Line 3
    //throw exception();    // Line 4
    //throw MyClass();      // Line 5
}   // Stack unwind, h_a.~MyClass() called
    // Passes on all exceptions
void g() { MyClass g_a;
    try { h();
        bool okay = true; // Not executed
    }
    // Catches exception from Line 1
    catch (int) { cout << "int\n"; }
    // Catches exception from Line 2
    catch (double) { cout << "double\n"; }
    // Catches exception from Line 3-5 & passes on
    catch (...) { throw; }
}   // Stack unwind, g_a.~MyClass() called
```

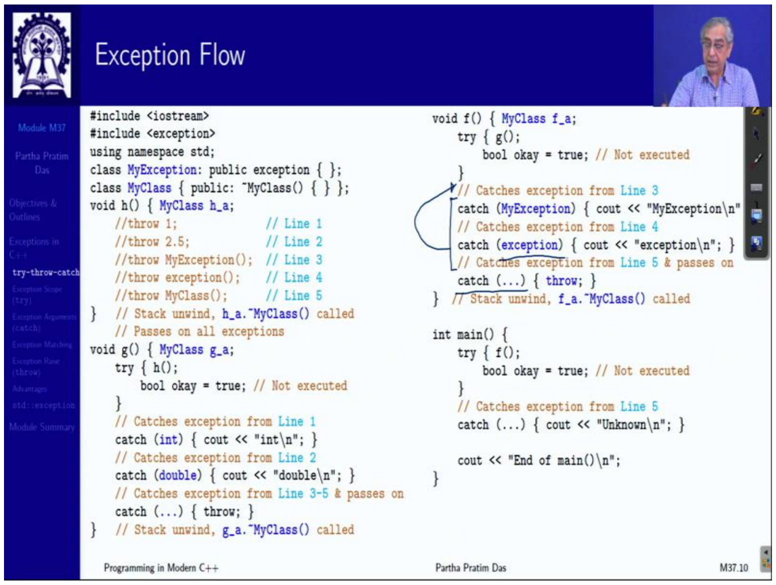
```
void f() { MyClass f_a;
    try { g();
        bool okay = true; // Not executed
    }
    // Catches exception from Line 3
    catch (MyException) { cout << "MyException\n"
    // Catches exception from Line 4
    catch (exception) { cout << "exception\n"; }
    // Catches exception from Line 5 & passes on
    catch (...) { throw; }
}  // Stack unwind, f_a.~MyClass() called

int main() {
    try { f();
        bool okay = true; // Not executed
    }
    // Catches exception from Line 5
    catch (...) { cout << "Unknown\n"; }

    cout << "End of main()\n";
}
```

$main \rightarrow f \rightarrow g \rightarrow h$

Programming in Modern C++ | Partha Pratim Das | M37.10



## Exception Flow

```
#include <iostream>
#include <exception>
using namespace std;
class MyException: public exception { };
class MyClass { public: ~MyClass() { } };
void h() { MyClass h_a;
    //throw 1;               // Line 1
    //throw 2.5;             // Line 2
    //throw MyException();   // Line 3
    //throw exception();     // Line 4
    //throw MyClass();       // Line 5
}   // Stack unwind, h_a.~MyClass() called
    // Passes on all exceptions
void g() { MyClass g_a;
    try { h();
        bool okay = true; // Not executed
    }
    // Catches exception from Line 1
    catch (int) { cout << "int\n"; }
    // Catches exception from Line 2
    catch (double) { cout << "double\n"; }
    // Catches exception from Line 3-5 & passes on
    catch (...) { throw; }
}   // Stack unwind, g_a.~MyClass() called
```

```
void f() { MyClass f_a;
    try { g();
        bool okay = true; // Not executed
    }
    // Catches exception from Line 3
    catch (MyException) { cout << "MyException\n"
    // Catches exception from Line 4
    catch (exception) { cout << "exception\n"; }
    // Catches exception from Line 5 & passes on
    catch (...) { throw; }
}   // Stack unwind, f_a.~MyClass() called

int main() {
    try { f();
        bool okay = true; // Not executed
    }
    // Catches exception from Line 5
    catch (...) { cout << "Unknown\n"; }

    cout << "End of main()\n";
}
```

Programming in Modern C++ Partha Pratim Das M37.10

Now, as g() is, as g() is thrown, so the actions on Line 3, 4, 5 are not still known. So, as g() is thrown, it comes back to f() which had called g() under the try block. f() has provided again three catch clauses in the order, but it has provided them by resolving the three types of exceptions that can actually happen. So, if I have Line 3, exceptions from Line 3 uncommented, I get a MyException object. And therefore, this is handled here. If I have Line 4, that is exception, general exception object, then it does not match here.

Why it does not match? Because to match, exception object to MyException object, I need a downcast, mind here. This is a ISA specialization. So, I need a downcast. A downcast is not permitted. So, it will not match here, but it will match at this point and this handling will be done. Finally, if I have uncommented Line 5, then I have MyClass, which obviously does not match here, it does not match here, and it will match the catch all clause again.

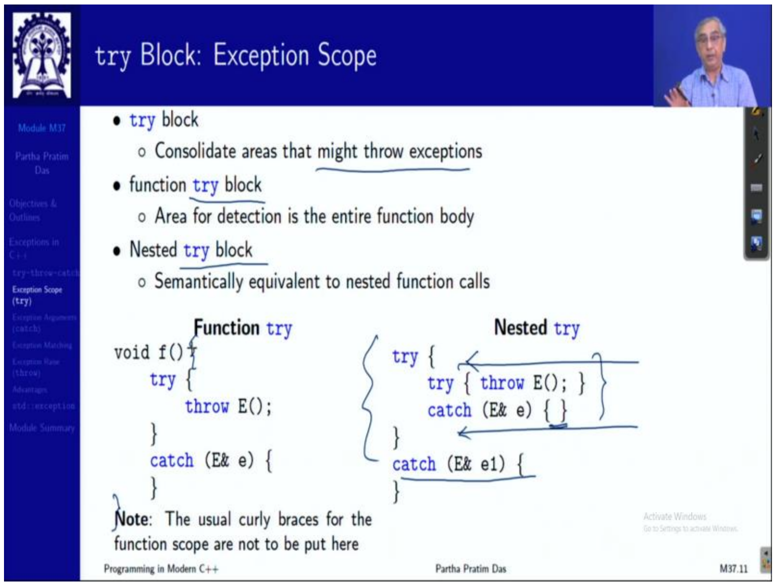
So, for Line 5, I still do not know what needs to be done in sitting in f(), because I am just catching it by, it was not kind of to me, so I am just catching it by catch all clause. And so what I decide to do is throw it again. As I throw it, the stack frame of f() goes out of context. So, the local object, automatic object of f() is destructed and the control comes back to main within the, within its try block, which has just one catch all clause, because main does not want to do anything, main just wants to say, if nobody could handle something, then this is what we will do.

So, possibly main will decide at this point that since it was not handled by the callee's f(), g() and h(), they could not decide that. main has no means of doing anything, it will probably just terminate the program, but it depends. So, this is the overall mechanism that is involved in terms of the exception propagation. So, you could see that how from a depth of call, you can propagate the exception nicely back to the topmost level which is main.

And in the process, unwind the stacks as needed and clean up the automatic objects. But in case, one of these functions could catch the right type of the exception, it will handle that and proceed from that point, it will not propagate it to its caller. Couple of points to note is if you consider the ordering of these two clauses, you must note that you cannot put this clause before the other one, that is catch exception cannot come before catch MyException, because if it does, then when a MyException object is thrown, exception will always catch that by upcast. And MyException clause will never be actually update.

So, you will have an upcast and it will just treat it as a generic exception, not, you will not get the specialized behavior. So, always the more general classes have to occur later. And by the extension of that same logic, your catch all clause must be the last one in the list of clauses, otherwise it will hide all other catch clauses that you have written. So, this is the overall exception mechanism, how it works in C++.

(Refer Slide Time: 25:10)



## try Block: Exception Scope

- try block
  - Consolidate areas that might throw exceptions
- function try block
  - Area for detection is the entire function body
- Nested try block
  - Semantically equivalent to nested function calls

**Function try**

```
void f()
    try {
        throw E();
    }
    catch (E& e) {
    }
```

**Nested try**

```
try {
    try { throw E(); }
    catch (E& e) { }
}
catch (E& e1) {
}
```

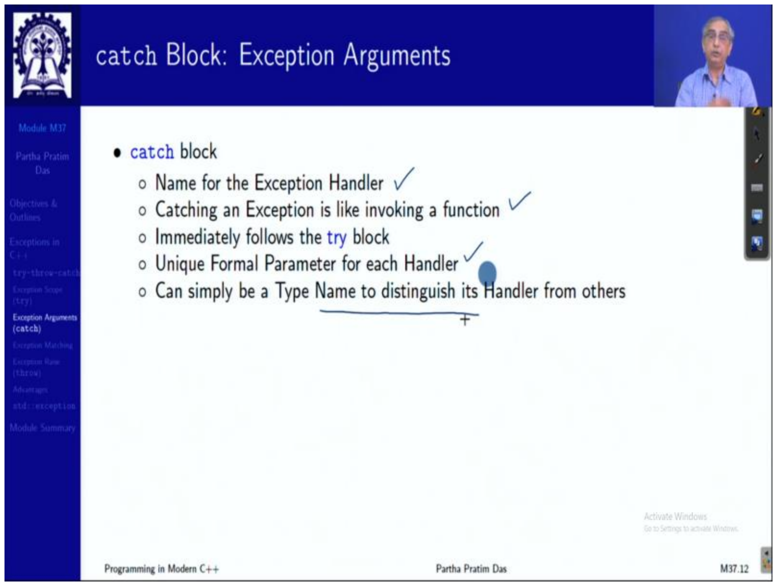
**Note**: The usual curly braces for the function scope are not to be put here

Now, quickly onto some syntax stuff and basic definitions just to consolidate. Now, try block is, consolidates the area that might throw exceptions. If you do not put codes in the try block, you will not have catch clauses. And therefore, if such a code will throw, the, it will automatically be thrown up to the caller, because certainly there is no catch clause associated.

So, it is good to put that. There are two kinds of try blocks. One is what we have seen, which can be nested also. You can have one tri block within another try block that is do few things, then do something under try if there is a problem, handle that and proceed. And in the, in the overall, either here or here, or if you want to re-throw from the inner try block, you can always handle in the outer one, so it can be nested.

And there is something which is specially called a function try block, where the entire function body is put in the try block. In many cases, that is done, just it is, it has the int x, except that in case of a function try block, the typic, the actual function block curly braces must not be provided. If you provide them, they become a normal try block. If you do not, then the compiler treats the entire try block as the function body. You can have returns from within that and so on, all, everything else is the same, but this is what is a different type of try block you can have.

(Refer Slide Time: 26:58)



## catch Block: Exception Arguments

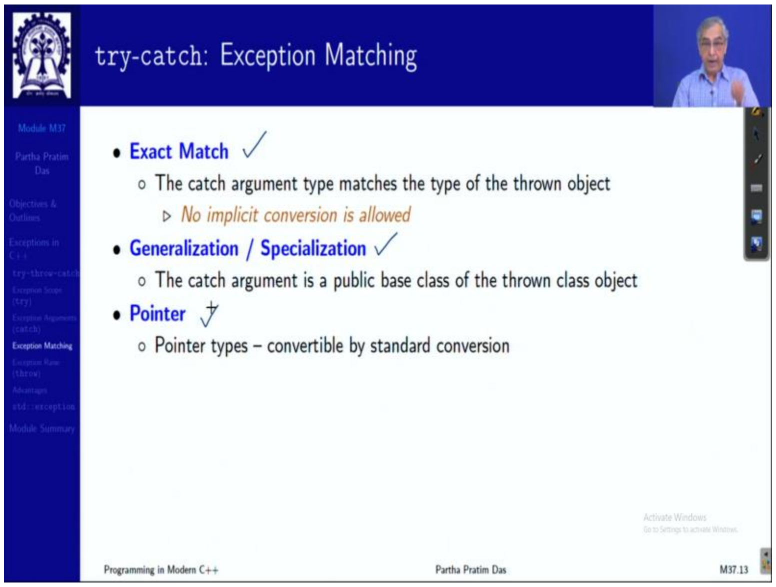
- catch block
  - Name for the Exception Handler ✓
  - Catching an Exception is like invoking a function ✓
  - Immediately follows the try block
  - Unique Formal Parameter for each Handler ✓
  - Can simply be a Type Name to distinguish its Handler from others

Programming in Modern C++ Partha Pratim Das M37.12

Similarly, catch block catches the exception. It, so it has an exception argument. Normally the exception argument is taken as a, as a reference parameter, certainly, because you do not want to unnecessarily keep copying the exception object. They are finally temporary. So, you take that as a, as a reference. And also, if you, if you want, they can make it a constant reference if the context demands. So, it has, it is basically the name of the exception handler you can say.

So, catching an exception is more like, throwing an exception is like preparing the argument and catching an exception is like calling a function. So, the body of the catch clause is kind of conceptually the handler function that is getting called. And it is unique with formal parameters for each handler and it can simply have a type name if you just do not want any details from the object runtime instance.

(Refer Slide Time: 28:03)



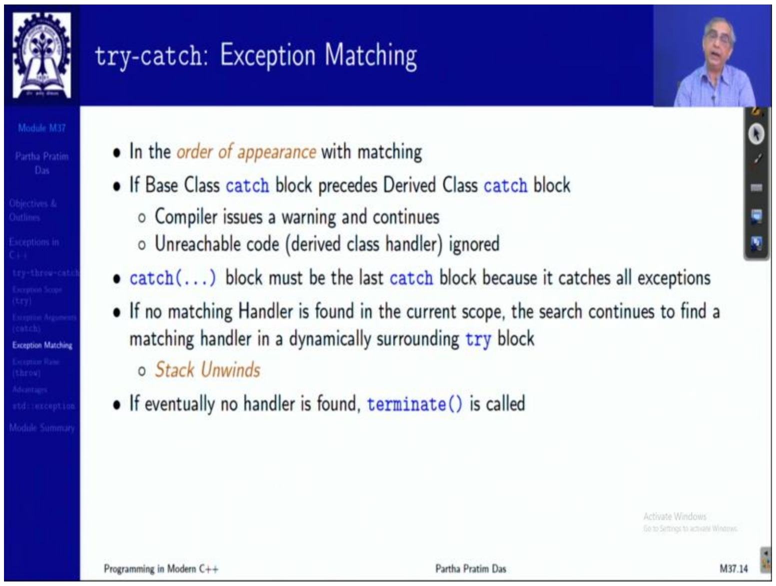
## try-catch: Exception Matching

- **Exact Match** ✓
  - The catch argument type matches the type of the thrown object
    - ▷ *No implicit conversion is allowed*
- **Generalization / Specialization** ✓
  - The catch argument is a public base class of the thrown class object
- **Pointer**
  - Pointer types – convertible by standard conversion

Programming in Modern C++ Partha Pratim Das M37.13

Try-catch matching, I have already mentioned that either most of them are exact matches, there is no, there is no implicit conversion that is allowed, but you can use upcast in terms of matching a general, a specialized object for a generalized exception class. Of course, in terms of pointer type, the standard conversions will apply, if you throw a point pointed type of exception. You might need to do that, because if through the exception, you need to really pass a lot of information, which has to be persistent and so on.

(Refer Slide Time: 28:43)



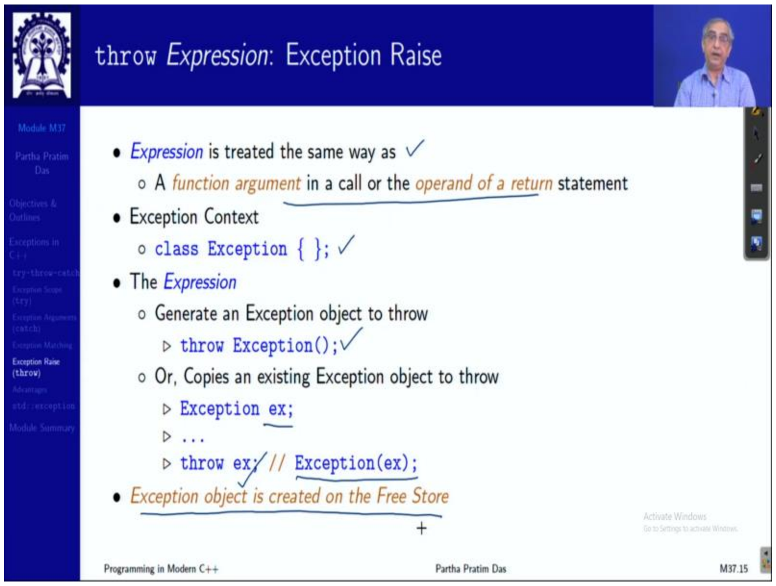
## try-catch: Exception Matching

- In the *order of appearance* with matching
- If Base Class `catch` block precedes Derived Class `catch` block
  - Compiler issues a warning and continues
  - Unreachable code (derived class handler) ignored
- `catch(...)` block must be the last `catch` block because it catches all exceptions
- If no matching Handler is found in the current scope, the search continues to find a matching handler in a dynamically surrounding `try` block
  - *Stack Unwinds*
- If eventually no handler is found, `terminate()` is called

Programming in Modern C++ Partha Pratim Das M37.14

The order of, they are matched, according to the order of appearance. We have seen that base class catch clause precedes the derived class catch clause. We have seen catch all clause has to be at the end, all these we have seen. And finally, if no handler can be found, even in the main, then the terminate is called. Terminate is a function which terminates the program.

(Refer Slide Time: 29:08)



## throw *Expression*: Exception Raise

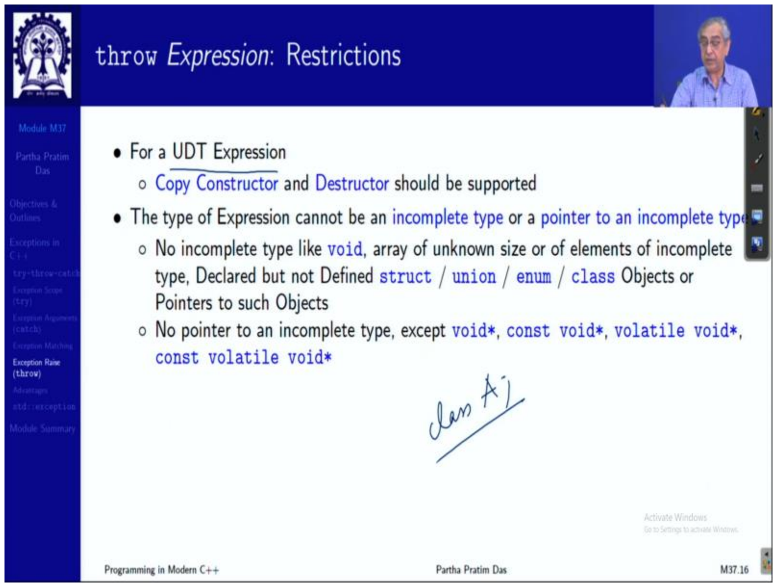
- *Expression* is treated the same way as ✓
  - A *function argument* in a call or the *operand of a return* statement
- Exception Context
  - `class Exception { };` ✓
- The *Expression*
  - Generate an Exception object to throw
    - ▷ `throw Exception();` ✓
  - Or, Copies an existing Exception object to throw
    - ▷ `Exception ex;`
    - ▷ `...`
    - ▷ `throw ex; // Exception(ex);`
- *Exception object is created on the Free Store*

Programming in Modern C++ Partha Pratim Das M37.15

So, in terms of raising, throwing, the exception is treated in the same way as a function argument. So, you are preparing the argument and then throwing it for the catch clause to call the handler. So, this is the exception context, throw exceptions as we have seen. You can create an exception object and then throw it, you can do that, you can create the exception object explicitly. So, these are different options that you have for this.

Note that exception object is always created on the free store. The reason is simple, because the stack is being unwinded, so there is no, you know, available stack frame, or there is no stable stack frame in which you can create the exception object. Naturally, you do not want to create it in the global space and clutter that. So, you create this as, in the free store, it is dynamically created and will get destroyed when the, the exception has been handled or propagated upwards.

(Refer Slide Time: 30:13)



## throw *Expression*: Restrictions

Module M37

Partha Pratim Das

Objectives & Outlines

Exceptions in C++

try-throw-catch

Exception Scope (try)

Exception Arguments (catch)

Exception Matching

Exception Raise (throw)

Advantages

std::exception

Module Summary

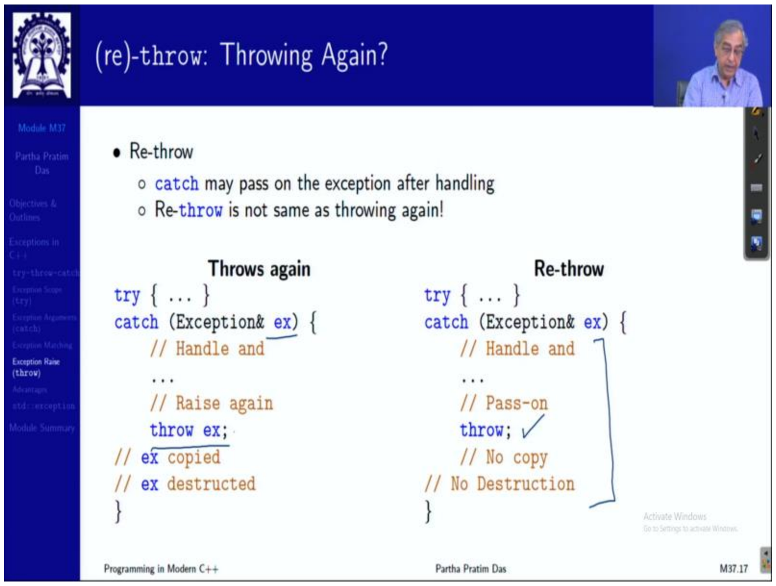
- For a UDT Expression
  - Copy Constructor and Destructor should be supported
- The type of Expression cannot be an incomplete type or a pointer to an incomplete type
  - No incomplete type like `void`, array of unknown size or of elements of incomplete type, Declared but not Defined `struct` / `union` / `enum` / `class` Objects or Pointers to such Objects
  - No pointer to an incomplete type, except `void*`, `const void*`, `volatile void*`, `const volatile void*`

class A;

Programming in Modern C++ Partha Pratim Das M37.16

So, there are certain restrictions that if you are using a user-defined type for, as an exception object, then you must provide the copy constructor and the destructor. Obviously, destructor is required so that at the end, you can clean up. Copy constructor is required in case you want to re-throw, right? And any type can be used, but it has to be a complete type. You cannot have an incomplete type or pointed to an incomplete type usually. So, like I have just written class A, a forward declaration. I cannot use this as an exception type for the simple reason that I need constructor, destructor, copy constructor, all of these four the exception types.

(Refer Slide Time: 31:04)



## (re)-throw: Throwing Again?

- Re-throw
  - catch may pass on the exception after handling
  - Re-throw is not same as throwing again!

**Throws again**

```
try { ... }
catch (Exception& ex) {
    // Handle and
    ...
    // Raise again
    throw ex;
// ex copied
// ex destructed
}
```

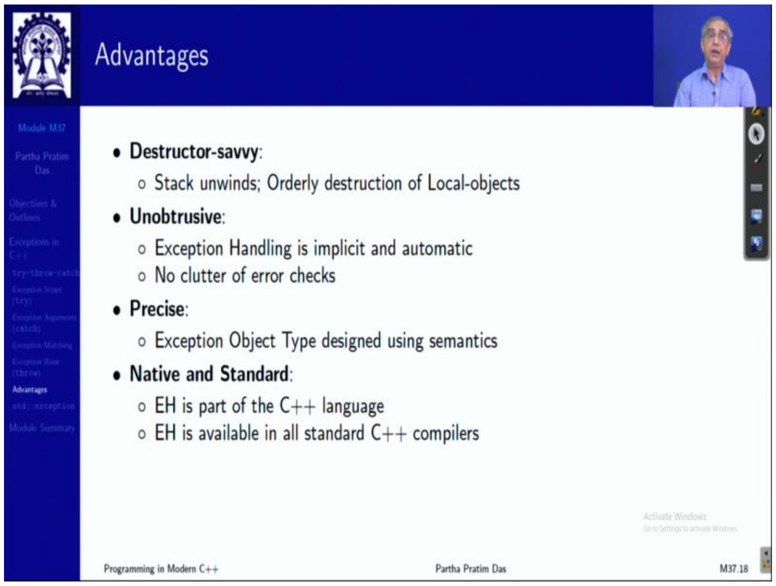
**Re-throw**

```
try { ... }
catch (Exception& ex) {
    // Handle and
    ...
    // Pass-on
    throw; ✓
    // No copy
// No Destruction
}
```

Programming in Modern C++ Partha Pratim Das M37.17

As we have seen that you can re-throw, you can re-throw in two ways. One is, this is what we saw. You can just write throw, and/or you have caught the exception and then you throw it again. There is a subtle difference in this. If you just do this, then no new exception object is created. Whatever was created is just passed on. Whatever was received is just passed on. But here, what was received will be copied, made a new copy of and the original one will be destructed, that is where you need the copy constructor. So, these are the two differences you have to decide which one will suit your purpose better.

(Refer Slide Time: 31:47)



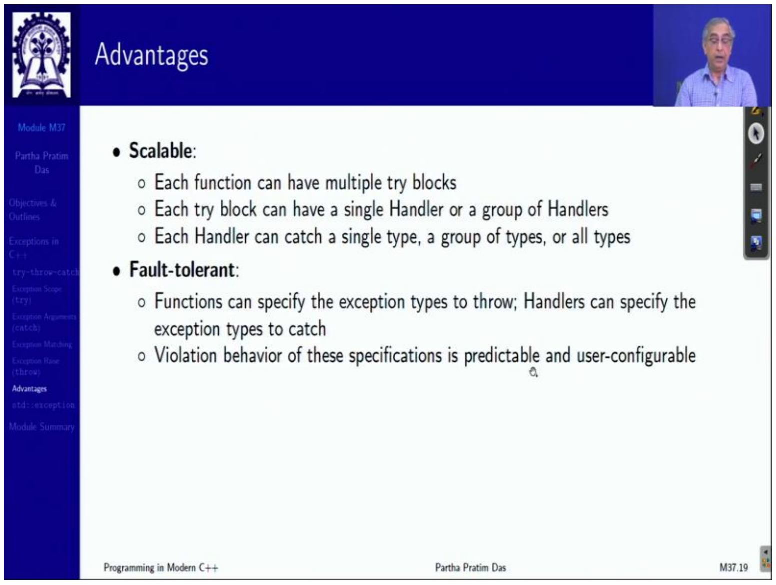
## Advantages

- **Destructor-savvy**:
  - Stack unwinds; Orderly destruction of Local-objects
- **Unobtrusive**:
  - Exception Handling is implicit and automatic
  - No clutter of error checks
- **Precise**:
  - Exception Object Type designed using semantics
- **Native and Standard**:
  - EH is part of the C++ language
  - EH is available in all standard C++ compilers

Programming in Modern C++ Partha Pratim Das M37.18

Certainly, law of advantages of exceptions, destructor-savvy, unobtrusive, precise, they are native and standard, because they are part of the language. Now, these are the, these are the shortcomings that we had noted for C now in C++ they are really the strengths.

(Refer Slide Time: 32:07)



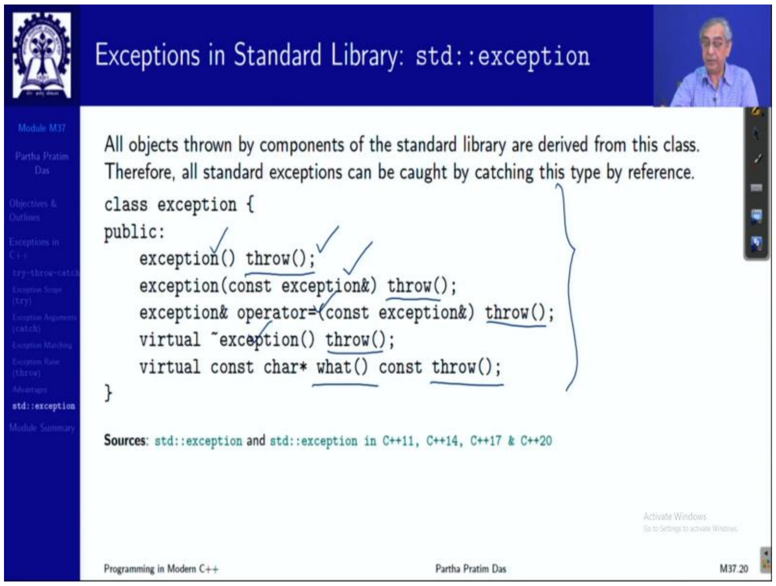
## Advantages

- **Scalable**:
  - Each function can have multiple try blocks
  - Each try block can have a single Handler or a group of Handlers
  - Each Handler can catch a single type, a group of types, or all types
- **Fault-tolerant**:
  - Functions can specify the exception types to throw; Handlers can specify the exception types to catch
  - Violation behavior of these specifications is predictable and user-configurable

Programming in Modern C++ Partha Pratim Das M37.19

Besides that, there is a scalable, because each, function can have multiple try blocks, each try block can have multiple handlers, each handler can have a single or group of types, all types, it is fall tolerant, because it makes functions specify what should be their error behavior and so on. So, exceptions are kind of a solution for all kinds of error handling.

(Refer Slide Time: 32:31)



## Exceptions in Standard Library: std::exception

All objects thrown by components of the standard library are derived from this class. Therefore, all standard exceptions can be caught by catching this type by reference.

```
class exception {
public:
    exception() throw();
    exception(const exception&) throw();
    exception& operator=(const exception&) throw();
    virtual ~exception() throw();
    virtual const char* what() const throw();
}
```

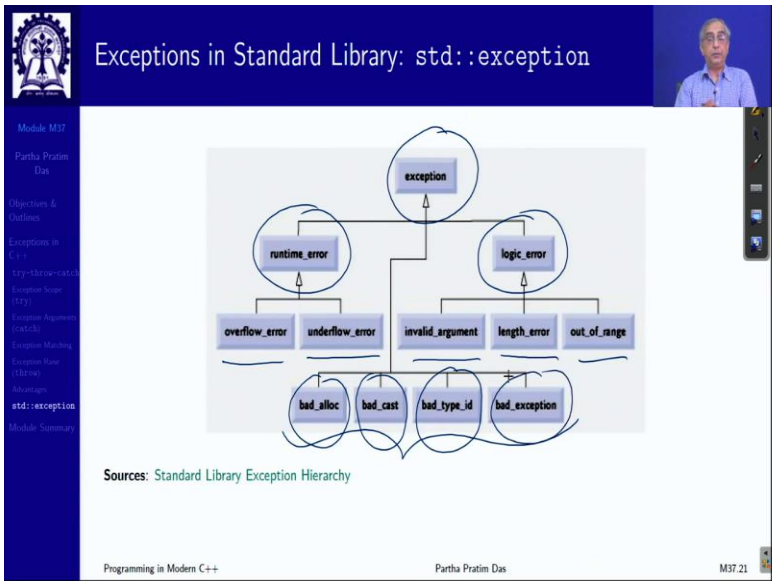
**Sources**: std::exception and std::exception in C++11, C++14, C++17 & C++20

Programming in Modern C++ Partha Pratim Das M37.20

Now, coming to the standard library, what the standard library gives you, standard library gives you this class, which has a constructor, the writing throw with the parenthesis here says this is called no throw guarantee, that is, all these functions themselves guarantee that they will not throw anything. So, that otherwise you are trying to build that exception object and that throws and, again you are trying to build an exception object, so this will be an infinite loop. So, these have exception guarantees.

So, you have the constructor, the copy, constructor, the copy assignment operator, the destructor and a special what() function where you can put a message that you can extract later on. So, it is good to derive your exceptions specialize your exceptions from this so that the general interface remains the same.

(Refer Slide Time: 33:30)

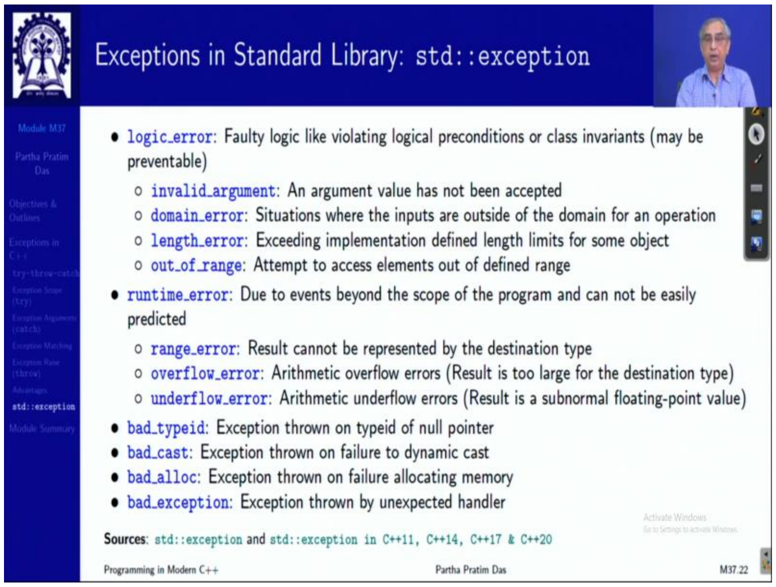

## Exceptions in Standard Library: std::exception

Sources: Standard Library Exception Hierarchy

Programming in Modern C++ Partha Pratim Das M37.21

There are various specializations of this class exception, broadly are runtime and logical errors like runtime is overflow, underflow and arithmetic, logical is invalid argument, length error, out of range and so on. Besides that, there are some, some of the special exception cases like bad_cast we have seen for dynamic cast, bad_type_id we have seen for type ID. bad_alloc will happen when you are trying to do new and there is not enough memory. If you have thrown a wrong exceptions, then you will get a bad_exception and so on.

(Refer Slide Time: 34:10)


## Exceptions in Standard Library: std::exception

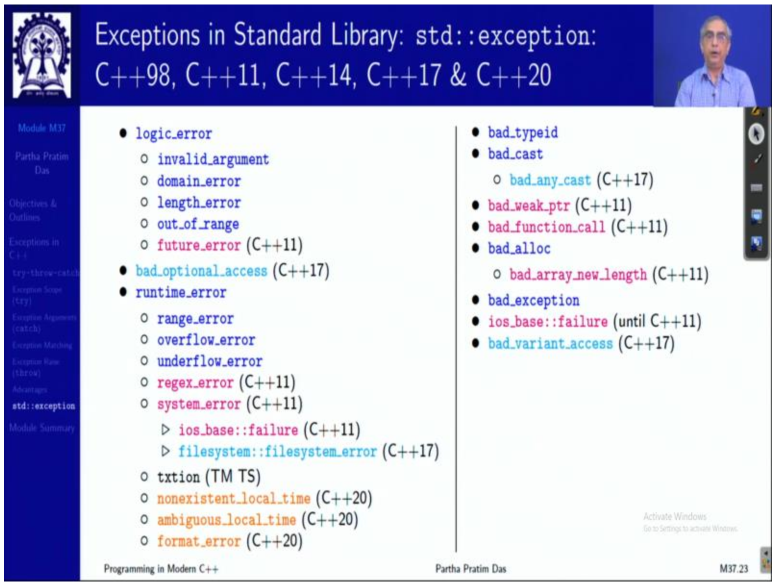
- logic_error: Faulty logic like violating logical preconditions or class invariants (may be preventable)
  - invalid_argument: An argument value has not been accepted
  - domain_error: Situations where the inputs are outside of the domain for an operation
  - length_error: Exceeding implementation defined length limits for some object
  - out_of_range: Attempt to access elements out of defined range
- runtime_error: Due to events beyond the scope of the program and can not be easily predicted
  - range_error: Result cannot be represented by the destination type
  - overflow_error: Arithmetic overflow errors (Result is too large for the destination type)
  - underflow_error: Arithmetic underflow errors (Result is a subnormal floating-point value)
- bad_typeid: Exception thrown on typeid of null pointer
- bad_cast: Exception thrown on failure to dynamic cast
- bad_alloc: Exception thrown on failure allocating memory
- bad_exception: Exception thrown by unexpected handler

Sources: std::exception and std::exception in C++11, C++14, C++17 & C++20

Programming in Modern C++ Partha Pratim Das M37.22

So, it is a huge provision. So, this is what C++03 had of which you just saw the simplified diagram. You have to really understand what these are and make use of them, because not in every case, you need to throw your own exception. The exceptions will be generated by the system and you just need to pass it on.

(Refer Slide Time: 34:33)



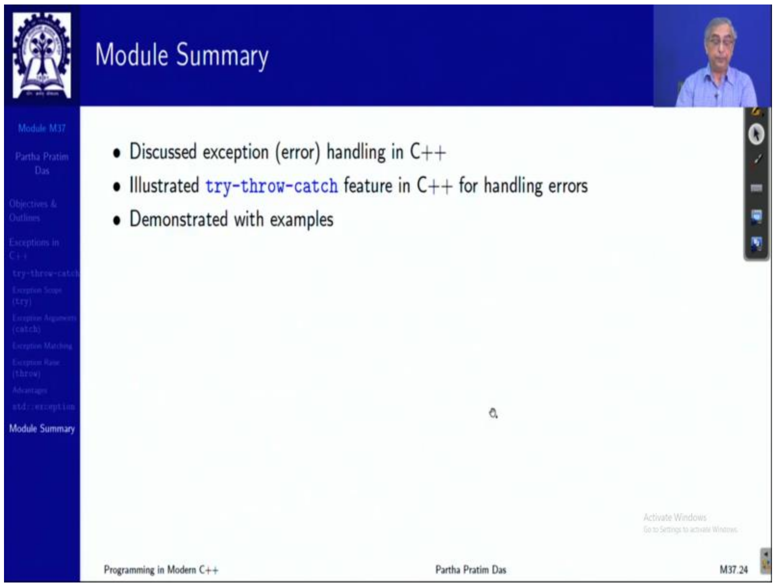
Exceptions in Standard Library: std::exception: C++98, C++11, C++14, C++17 & C++20

- logic_error
  - invalid_argument
  - domain_error
  - length_error
  - out_of_range
  - future_error (C++11)
- bad_optional_access (C++17)
- runtime_error
  - range_error
  - overflow_error
  - underflow_error
  - regex_error (C++11)
  - system_error (C++11)
    - ios_base::failure (C++11)
    - filesystem::filesystem_error (C++17)
  - txtion (TM TS)
  - nonexistent_local_time (C++20)
  - ambiguous_local_time (C++20)
  - format_error (C++20)
- bad_typeid
- bad_cast
  - bad_any_cast (C++17)
- bad_weak_ptr (C++11)
- bad_function_call (C++11)
- bad_alloc
  - bad_array_new_length (C++11)
- bad_exception
- ios_base::failure (until C++11)
- bad_variant_access (C++17)

Programming in Modern C++ Partha Pratim Das M37.23

Going to the future, later versions of C++, several, there has been several extensions to this standard library, so which I have shown by different colors. This is just for your information. We are not going to discuss this, or you know, have it in, in the assignments immediately when you do the modern part of the C++, we will talk a little bit about some of them.

(Refer Slide Time: 34:59)



Module Summary

- Discussed exception (error) handling in C++
- Illustrated try-throw-catch feature in C++ for handling errors
- Demonstrated with examples

Programming in Modern C++ Partha Pratim Das M37.24

So, to conclude, we have discussed about exception handling in C++, and illustrated try-catch-throw features for handling errors with examples. Thank you very much for your attention. See you in the next module.