

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

Lecture – 53
Exceptions (Error Handling in C): Part – II

Welcome to Module 37 of Programming in C++. We have been discussing about Error Handling in C - C++ in the earlier module. We have taken a look into what are the different exception causes are, variety of them and their types asynchronous and synchronous.

We then took a detailed view in terms of the different mechanism that is available in C for handling errors, handling exception. And we saw that there is hardly any mechanism available, actually no mechanism was provided in C with the thought of handling errors, handling exception situations, but the return value and non local and local go to has been used for exception handling. But there are several standard library features that what added through multiple standard libraries to provide support for handling errors in C and that still leads to a lot of a short comings lot of difficulties and in view of this here we are interested to study about error handling in C++.

(Refer Slide Time: 01:20)

The slide displays a 'Module Outline' for 'Exception Fundamentals' and 'Exceptions in C++'. The left sidebar lists 'Module 37', 'Partha Pratim Das', 'Objective & Outline', 'Exceptions in C++', 'Exception Scope', 'Exception Arguments', 'Exception Matching', 'Exception Raise', 'Advantages', and 'Summary'. The main content area lists the following topics:

- Exception Fundamentals
 - Types of Exceptions
 - Exception Stages
- Exceptions in C
 - C Language Features
 - Return value & parameters
 - Local goto
 - C Standard Library Support
 - Global variables
 - Abnormal termination
 - Conditional termination
 - Non-local goto
 - Signal
 - Shortcomings
- Exceptions in C++
 - Exception Scope (try)
 - Exception Arguments (catch)
 - Exception Matching
 - Exception Raise (throw)
 - Advantages

At the bottom, it says 'NPTEL MOOCs Programming in C++' and 'Partha Pratim Das'.

This is a module outline as you know the lower part; the blue part is what you will be discussing in the module today.

(Refer Slide Time: 01:37).

The slide is titled "Expectations" and lists the following points:

- Separate Error-Handling code from Ordinary 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

As we move to C++, certainly the designers had been very sensitive and cautious about incorporating the error handling as the part of the language. The first expectation is we should be able to separate error handling code from ordinary code. Ordinary code means a normal flow. The normal flow and the error handling as we saw are often intermixed we would try to separate this out and that is a requirement of exception handling in C++.

Second it is a language mechanism rather than of the library, that is the language should at the core take (Refer Time: 02:19) of the fact that errors can happen exceptions will need to be managed and therefore provide a language feature for doing that.

Given as a language feature compilers would be able to track all automatic variables and take care of the lifetime destroying them when there is some situation to terminate or if not terminate a whole program, but to abnormally terminate the execution of a certain function the automatic variables will be taken care off. There are schemes for destruction of a dynamic memory and we will certainly aspect that the overhead for the designers which we saw in the C style would should be much less.

This is very very important which we look at is often the exception does not happen in the main, it will happen in function which is in a deep nested call; main has called one function; that has called another function; that has called yet another function and so on. Then in some 3, 4 levels, 5 levels of calls or 10 levels of calls some exception has happened so you need to come out of this deep nesting of calls in the exception situation so that should be possible. And variety of situations, we should be able to handle through a single handler.

(Refer Slide Time: 03:39)

```
void f() {
    A a;
    try {
        B b;
        g();
        h();
    }
    catch (UserExcp ex) {
        cout <<
        ex.what ();
    }
    return;
}

class UserExcp {
public:
    exceptions {}
    void g() {
        A a;
        UserExcp ex("From g()");
        throw ex;
    }
    return;
}
```

- g() called and exception raised
- Exception caught by catch clause

NPTEL MOOCs Programming in C++ Partha Pratim Das

These are the expectation with which the exception mechanism in C++ has been designed and this is a highest show the same illustration which I was showing earlier in the earlier module with the non local go to. So there is a function f which is calling a function g and this is function g. Now what it does it basically does something called a throw, we will see what does throw mean. And this is what some kind of what will say is a exception class.

Once g is called then certainly the first situation is a happy part, the normal flow; so g will come out through the return. If g comes our through the return it has a normal flow the control goes to the next statement h which is clear. But if g is called and because of

something happening in between because of some error situation arising in g it (Refer Time: 04:39) across an error situation that will throw an exception.

Which means it will throw is a keyword, it will do throw and then it will put an object of the exception class; any class can be used as an exception class really. And if it throws then the control does not come back to h, does not come back to the statement immediately following g it comes back to what is known as a catch clause. So from here the control will come out here and then you can take care of the exception at this point. So it is called by exception clause.

If you look into what all have become different is now we have a try; we say it is a try block, try is a keyword, catch is another keyword, so there is a catch block and throw is another keyword. We will say try throw catch; kind of or try catch for simple terms. So what we do in the caller, we put the calls within a try block and a try block has an associated catch. And the call function may throw saying that thus exception as happened.

When it throws the basic behaviour is at this point the function was called if it throws the basic is the try does not continue, the control comes back to the try, but it does not continue in the immediately next statement, but it goes to the catch clause. In the catch clause it tries to find what kind of exception it as got and accordingly it will try to execute the catch clause. So it will execute this catch clause and then it will continue here.

Basically, there is some situation that has happened here which is given rise to the function g realising that exceptions needs to be reported, error needs to be reported. The error object is created here, then it is reported here I am talking about the exception stage this is a create stage if I cleanly draw it. This is a create stage where you create that this is a report stage.

And the interesting thing is we do not need to do anything special for the detect because the catch clause as is present will detect it immediately then it takes care of this in the code this is called the handler code the catch handler or the exception handler, handles it

and then you come to the next where you have the recover. This is how the exception will get handled in terms of a C++ mechanism.

(Refer Slide Time: 07:19)

The slide, titled "Exception Flow", displays C++ code and a call stack diagram. The code defines a class hierarchy and several functions. The call stack diagram shows the sequence of function calls: main, f, g, and h, with a question mark indicating the current point of execution.

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

class MyException : public exception {}
class MyClass {}

void h() { MyClass a;
//throw 1;
//throw 2.h;
//throw MyException();
//throw exception();
//throw MyClass();
}

void g() { MyClass a;
try {
h();
}
catch (int) { cout << "int"; }
catch (double) { cout << "double"; }
catch (...) { throw; }
}

void f() { MyClass a;
try {
g();
}
catch (MyException) { cout << "MyException"; }
catch (exception) { cout << "exception"; }
catch (...) { throw; }
}

int main() {
try {
f();
}
catch (...) { cout << "Unknown"; }
return 0;
}
```

Call stack diagram:

```
main
?
g
h
```

Let us look at a more detailed example and try to understand this. I have couple of slides following this where all these mechanisms are explained, but first I would explain it through an example. So, let us first look at what the example has, forgetting about the exception situation there is a function h, there is a function g, there is a function f and main function. And it is a simple nested call main calls f, f in turn calls g, g in turn calls h, h does something. These are basic structure. The difference that we are doing is when main calls f it puts it within a (Refer Time: 08:02) and has an associated catch.

When f calls g puts it in a try block, has multiple associated catch blocks. When g calls h it puts in a try block has multiple associated catch clause and h does the task, this is the situation. If we look into any of g, f, or main, we see that they are calling the function within the try block expecting or rather apprehending that this function might fail and if it fails it will throw something and whatever it throws I will catch in the associated catch clause.

Now let us see why it is so many catch clauses and what happens with that and so on. Let say I have commented here all of them because certainly if one throw happens in the function is gone. Suppose this is not commented, so what happens is it is doing something comes across an error and it throws one. If it throws one what does that mean? What is one? Basically it is throwing an int kind of object. If it throws then the controls goes, if we look into the calls stack then what do we have we have main, main has called f, f has called g, g has called h. This is a where we are at present.

Now if you do throw, then it has detected that it cannot proceed any further, so throw is pretty much like as if you want in terms of control flow you are trying to go out of this function. So you want to go out and go back to g. Now why do you go back in g? In g the call of h is guarded within the try block, so it does not go back to this point. Rather as you throw the control goes to the list of catch clauses that you have. And what have been thrown is; one which is an int.

Now once you get there what happens is something very very interesting it takes this object, object one and goes over this list of catch clauses and tries to match. If it finds a match then it immediately stops there and executes the corresponding handler. If it does not find a match it proceeds further. So what happens in case of if one has been thrown it is an int object, so catch is expecting an int since it is expecting an int it matches and it will execute this cout int. This is just saying that it if as if this is a handler.

Once this is done, then the control will jump the remaining of the catch clauses and some here and continue. Now, whatever g has to do; now the whole thing has already taken place, the error happened, it was reported, it has been detected here, it has been handled here, you recovered and you are proceeding to this part.

Let us see what happens if you, let us say again draw the stack that will be important in every case h, g, f main. Now let us say this is where something was happening and it reaches a point where it throws 2.5, just to show that that is a something different has happened.

Now what is 2.5? 2.5 is an object of the double type we know. As it throws the control comes back to g. Where does it come back? It does not come back here; it comes back to the beginning of the catch clause because something has been thrown. And you starts matching it has a double object it tries to match int it will fail it will not be able to match. Catch clause is do not match by implicit conversion.

If we look into this match strategy this looks pretty much like the overload matching. So as if you can think of then thrown as if is calling a function with a parameter one single parameter. And there are several functions as if here, three functions here which takes an int x a double and takes some dot dot dot and you are trying to resolve which particular function, which particular catch clause will be resolved.

But there are two major distinctions from the overload resolution; one is overload resolution allows you for implicit conversion. For example int will be matched with double double will be truncated and matched within and so on. This is not allowed in deciding on the catch clause. The second is overload resolution happens over the set of functions as a whole. Here it will happen according to this order. And as soon as you found a match you will not try to find if there is a better match, you will just end there and call that catch clause. So what will happen here? They have a double object; this will not match, but this will match. Now, it will do print double and then it will recovered continues here that is a sample thing.

Let us continue on this and try to see what happens next. Main, f, g, h, let us check this. Here what is my exception, there is a standard library exception which has a class called exception predefined which takes care of all different kinds of exception that can happen in C++. I have defined a class which is a specialization of exception class so some throwing an object of my exception. What happens again? As the control comes to g that is starts when this tries to match with int does not match have a my exception object, tries to match with catch the double does not match, then tries to match here, what is this? This is known as ellipses.

Ellipses are three dots they are supposed to match with anything. Whatever you have that will match with that. So, ellipses will match with one it will match with double, it will

match with my exception, it will match with exception anybody, but earlier cases when we are thrown 1 or we had throw 2.5 since int and double catch clauses where before this ellipses then it did not fall through to this point. But here with my exception these two did not match so it comes here and matches at this point this is also known as a catch all clause because it matches everything.

So it goes to handle that, and what is a handle do? Handle does something very simple it says throw which means that it is not going to do anything but as the h function as thrown a my exception object this will simply re throw that object to the caller. So which means it will re throw it will go back go upwards to its caller it will the stack will come down to its caller. The caller was f caller had called g and you have thrown, what is your exception object? Exception object is what you had received from h, my exception object.

So since you have come here through an exception being thrown you will continue on the catch clauses. So, now you have a my exception object and you have a my exception catch clause so it matches say my exception skip over the other catch clauses and then you go to the remaining of the f function and continue. So, it is not necessarily that the caller will be able to handle the exception that you throw, the caller might handle that, caller might propagate it higher up to for the caller to handle that and we say that is re throwing an exception this is what you have done specially here.

Let us move forward and see what if I throw an exception of the class exception of which my exception was a specialization. Now, what will happen is, I come again here it goes to this point because these two do not match I throw, I come, I come here, match the exception I can see the exception. Suppose I throw my class is some other class and I throw an exception of my class it will start looking here it will go through the catch clauses, finally it matches at this point it throws again goes through this, this none of them match my class so it thrown again, so it comes back to the try block in main goes to this catch clause where it matches again and you throw and you are saying that something unknown as happened. This is how the exceptions will get propagated from a deep function to the outer called function.

And there are some interesting things that you can note for example, suppose I did not have this catch clause. And say I have thrown my exception then what will happen, this does not match, this does not match. Now there is no other catch clause left so it falls here, now the default behaviour of the exception mechanism is if a caller has received an exception which it has not been able to handle because there was no catch clause, it by default will re throw it, and it will by default will re throw the same exception. So even without this we will get the same behaviour that it will go back and go to that my exception.

Now, if that is a case then why do we really have re throw, the reason we have re throw is in many cases I may have some catch condition and I may first want to handle something which I am capable off, and then I may want that my caller should also know it. So I may want to handle some and then again throw it because if I just handle it like I did here then the system understands that the exception is taken care of and it will proceed right here and it will not go to the caller it will not throw. But if I handle and then throw again, then it will take that a fresh exception has happened from the function g and we will go to its caller f. So this is the basic mechanism of exception that happens.

Now let me also show you what happens in terms of the stack, say main, f, g, h, let us say my exception has happened. Now it throws which means that the control has to come out of this function g, t comes out of it. So if it comes out of that which means that the stack frame which was associated with this function g will have to be removed. When this as to be removed what happens to this local object, this local object is been created. So, what exception does is normally what would have happened is at the normal exist point which is here the destructor of this function that is a colon tilde my class would have got invoked.

Now, you are existing from this point, now you are coming out from this point because you have done a throw. But even at this point the destructor of this local object will get called. So all this destructors of all object which are live at this point then destructors will get called. And only after that the stack frame will be squashed and the control will go back to g.

Now if we have thrown my exception then certainly it does not match here, does not match here, it matches here, and it re throws. So the control goes back from g to f which means that the stack frame of g will also have to be unwinded and that will be done at this point, that will be done at this point, and this local object of g will get destroyed. So the basic advantage of exception base flow is that it not only takes care of your automatically going up in the stack based on the call sequence and the try blocks and matches the corresponding catch clause in the try block. Whenever it goes out of a function because the function has thrown it destroys all the local objects that were present.

At any point of time, if you have a thrown from here it has been passed on throw this and now you are you are working on f you know that for g and h the two functions that have thrown all local objects have been properly cleaned up. The stack has been as it is called this process of this process of removing destroying the local objects automatic objects and removing the stack frames as you throw objects from one function to its parent is known as stack unwinding the stack is collapsing and till you get to the function where you are supposed to continue. This is the basic exception mechanism which is based on the try block.

Certainly, let me also mentioned another point let say suppose this was there and let say in a function g instead of this you have just written g. Suppose you have called a function which is not within the try block and then this my exception has been raised. If my exception as been raised does not match here, does not match here, from this re throw will happen, so my exception happens here. After this g may be there is some other statement.

Now what happens is if you this is not within the try block also, then if you get an exception in g then once the control returns because of that exception you do not continue at the next statement, you immediately re throw and raise the same exception object to the parent. These are default mechanism. This default mechanism is important because if you forget to put try block around some functions calls which may have some exceptions raised then you will have a test of a difficulty of not being able to handle the

error, so by default if you have not raised if you have not put a function within a try block then you simply pass it on to a parent, your parent keeps on doing that.

Finally, if you have some exception in the main which is not guarded by a try block or there is a catch clause. Suppose, here I did not have this catch clause suppose here I had some catch clause for some other class and so on. But I did not have a catch or clause then it is quite possible that an exception like thrown like my class object will not be caught at this point because it does not match here. In that case the main cannot certainly throw, because it does not have a caller. So in that case a termination handler will get called. So, that is a basic mechanism of exceptions that happen.

(Refer Slide Time: 26:11)

The slide is titled "try Block: Exception Scope" and contains the following content:

- 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) {
}
```

NPTEL MOOCs Programming in C++ Partha Pratim Das

What we have done in the next couple of slides, whatever I have discussed those are documented here for example with some more details. For example, a function as a whole may have a try block. So, all that you do is instead of putting multiple try block you say that a function and the (Refer Time: 26:31) of function header and this is the function body. So, in between that you write try so which means that the whole functions is a part of try. Naturally the try blocks can be nested within try block you could have other blocks and so on, but try block must have a catch block with that.

(Refer Slide Time: 26:47)

The slide is titled "try-throw-catch" and is part of Module 37. It displays two code snippets side-by-side. The left snippet shows a function `f()` that declares a variable `A a;`, enters a `try` block where it calls `g()` and `h()`, and then has a `catch (UserExp4 ex)` block that prints `ex.what()` before returning. The `try` block is highlighted with an orange oval. The right snippet shows a class `UserExp1` with a public exception type `public exception1 {}` and a `void g()` method that declares `A a;`, creates a `UserExp1 ex("From g()")`, throws it, and returns. A small circular inset image of the presenter is visible in the bottom left corner.

```
void f() {
    A a;
    try {
        B b;
        g();
        h();
    }
    catch (UserExp4 ex) {
        cout <<
            ex.what();
    }
    return;
}

class UserExp1 {
public:
    exception1 {}
};

void g() {
    A a;
    UserExp1 ex("From g()");
    throw ex;
    return;
}
```

• try Block

NPTEL MOOCs Programming in C++ Partha Pratim Das 10

This is from the earlier example just highlighting what is that.

(Refer Slide Time: 26:50)

The slide is titled "catch Block: Exception Arguments" and is part of Module 37. It lists five bullet points describing the characteristics of a catch block. A small circular inset image of the presenter is visible in the bottom left corner.

- 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 be simply a Type Name to distinguish its Handler from others

NPTEL MOOCs Programming in C++ Partha Pratim Das 11

Then catch block has a different arguments that we have seen every catch block will have one argument and that has to be unique between the catch blocks.

(Refer Slide Time: 27:03)

```
void f() {  
    A a;  
    try {  
        B b;  
        g();  
        h();  
    }  
    catch (UserError& ex) {  
        cout <<  
            ex.what();  
    }  
    return;  
}
```

```
class UserError {  
public: exception();  
};  
void g() {  
    A a;  
    UserError ex("From g()");  
    throw ex;  
    return;  
}
```

● throw Expression

NPTEL MOOCs Programming in C++ Partha Pratim Das 17

So, the arguments here have to be unique. And there are two interesting things to note one, is how you should pass this argument, should you pass it as a value or as a reference. Usually it is passed as a reference. Particularly if it is a user defined object type the reason you would like to pass it as a reference because then you do not need to copy that exception object you can just keep on propagating.

And whenever it the handler ends, whenever the handler terminate if it has not re thrown whenever the handler terminates it will destroy that exception object. It is also interesting that the exceptions object are kind of so exception objects becomes kind of automatic objects because they are constructed at the point when they are thrown or before that in the function that it is throwing them and they will get destructed in some other function scope in the catch handler. There is kind of a very interesting automatic object which has a lifetime scope which is dependent on the runtime not necessarily on the compile time which is typical for all other automatic objects.

And therefore, the exception objects will always have to be created in the free store in the stack not in the stack, but in the, what you say as a heap. Because you cannot it in the stack because you do not know for how long you will need to maintain this object. So, you do not know whether the stack frame of a function in which you create that how long

that stack frame will exist. So you will typically create these objects are created in the free store.

(Refer Slide Time: 28:46)

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

NPTEL MOOCs Programming in C++ Partha Pratim Das 13

We have talked about exception matching.

(Refer Slide Time: 28:51)

try-throw-catch

```
void f() {
    A ar
    try {
        B b;
        g();
        h();
    } catch (DerExcp& ex) {
        ex.what();
    }
    return;
}
```

```
class DerExcp {
public:
    DerExcp() {}
};

void g() {
    A ar
    DerExcp ex("from g()");
    throw ex;
}
```

- **Expression Matching**

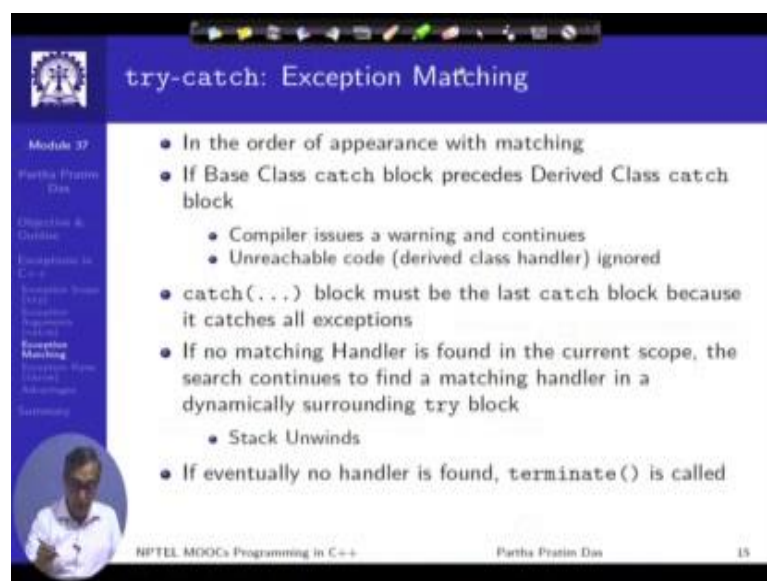
NPTEL MOOCs Programming in C++ Partha Pratim Das 14

This is just detailed here is to what the exception that you throw, how that is matched here. You will have to remember that if you have an exception and its specialization then certainly when you write the clauses you should first write the catch clause for the base class and then the catch clause for the specialised class. Because you remember the exceptions are catch is matched is from one end to the other. If you have a specialised class exception thrown then it will then I saying the right thing, I am sorry I am saying the wrong thing. So, if I have exception here, and my exception here, then I have two catch clauses; one for my exception and another for the exception.

Now, what I am trying to point out is if you have an exception of the base class then certainly it will not match here, because this means a down casting but it will go up and match there. So, instead of this if you had say it has my exception here then you will have a problem because when you have an exception of the derived class then it will match here because you are doing a up cast.

So, you will have to remember that whenever you have a hierarchy of a classes being used for the exception then the specialised one has to come higher earlier and the generalised one have to come later. So that mechanism will have to be followed.

(Refer Slide Time: 30:46)



The slide is titled "try-catch: Exception Matching" and is part of Module 37, "Partha Prasin Das". It contains a list of bullet points explaining exception matching rules:

- 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

The slide footer includes "NPTEL MOOCs Programming In C++", "Partha Prasin Das", and the number "15".

This is all described in this point so you can go throw that.

(Refer Slide Time: 30:50)

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

NPTEL MOOCs Programming in C++ Partha Pratim Das 18

We have talked about raising the exception, what happens in throw.

(Refer Slide Time: 30:57)

throw Expression: Restrictions

- For a UDT Expression
 - Copy Constructor and Destructor should be supported
- The type of Expression cannot be
 - An 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)
 - A pointer to an Incomplete type, except `void*`, `const void*`, `volatile void*`, `const volatile void*`

NPTEL MOOCs Programming in C++ Partha Pratim Das 18

This is the exception showing that.

(Refer Slide Time: 31:00)

(re)-throw: Throwing Again?

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

Throws again	Re-throw
<pre>try { ... } catch (Exception& ex) { // Handle and ... // Raise again throw ex; // ex copied // ex destructed }</pre>	<pre>try { ... } catch (Exception& ex) { // Handle and ... // Pass-on throw; // No copy // No Destruction }</pre>

NPTEL MOOCs Programming in C++ Partha Pratim Das 18

All these we have discussed now, the re throw what happens in terms of re throw that we have discussed.

(Refer Slide Time: 31:10)

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

NPTEL MOOCs Programming in C++ Partha Pratim Das 20

Finally, before we conclude you would like to note that there are several advantages of this try catch for mechanism. It is a destructor savvy because it take and on mind the

stack and clean up the local object is a non obtrusive because it separates out the whole code clutter into separate normal flow and the exception flow it is precise. It is native and standard, in the sense is the part of the language not part of a third party or stand standard library like that.

(Refer Slide Time: 31:42)

The slide is titled "Advantages" and is part of a presentation on "Programming in C++". It lists two main advantages of exceptions:

- **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

The slide also includes a navigation bar at the top, a sidebar on the left with a table of contents, and a small video inset of the presenter in the bottom left corner. The footer contains the text "NPTEL MOOCs Programming in C++", "Partha Pratim Das", and the slide number "21".

In that process this is quite scalable and quite fault tolerant. Scalable in it does not matter as to how many try blocks you have, how much of nesting you have, how much catch clauses you have, this will work in every case in a right manner.

(Refer Slide Time: 31:59)

Module Summary

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

NPTEL MOOCs Programming in C++ Partha Pratim Das 25

With this to summarise we have discussed the exception or error handling in C++ particularly we have try to understand the try catch throw mechanism of C++ which takes care of all possible ways the exceptions can happen and all possible ways you can handle that.

Actually if you can continue to design properly with try throw catch in C++ you would not need any of the exception mechanism in C. Of course you will still need to use them because some of the system calls do use those error mechanisms like, putting signals or putting error number. So you will still need to use for those cases, but for your own code you will never need to use any of the C error handling or C standard library functions.