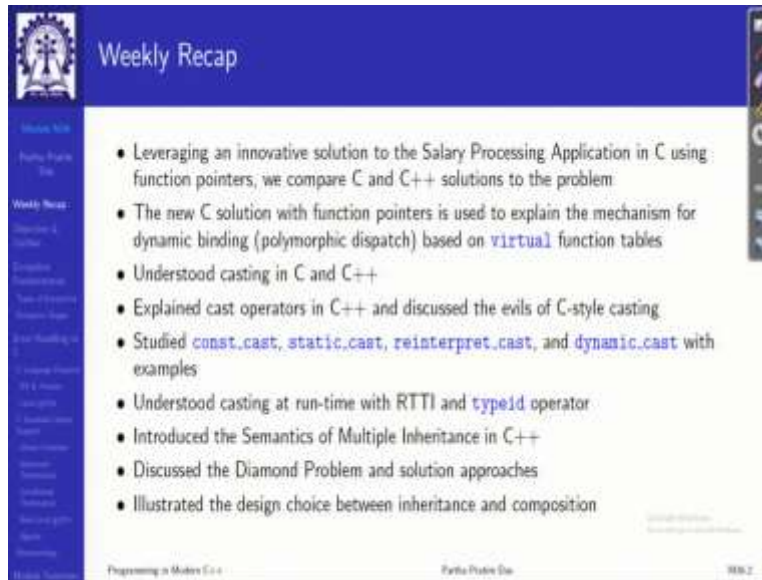


**Programming in Modern C++**  
**Professor Partha Pratim Das**  
**Department of Computer Science and Engineering**  
**Indian Institute of Technology, Kharagpur**  
**Lecture 36**  
**Exceptions (Error handling in C): Part 1**

(Refer Slide Time: 00:31)




The slide is titled "Weekly Recap" and features a blue header with the IIT Kharagpur logo on the left. A vertical navigation menu on the left side lists various topics. The main content area contains a bulleted list of topics covered in the week. At the bottom, it identifies the course as "Programming in Modern C++" and the lecturer as "Partha Pratim Das".

- Leveraging an innovative solution to the Salary Processing Application in C using function pointers, we compare C and C++ solutions to the problem
- The new C solution with function pointers is used to explain the mechanism for dynamic binding (polymorphic dispatch) based on `virtual` function tables
- Understood casting in C and C++
- Explained cast operators in C++ and discussed the evils of C-style casting
- Studied `const.cast`, `static.cast`, `reinterpret.cast`, and `dynamic.cast` with examples
- Understood casting at run-time with RTTI and `typeid` operator
- Introduced the Semantics of Multiple Inheritance in C++
- Discussed the Diamond Problem and solution approaches
- Illustrated the design choice between inheritance and composition

Welcome to Programming in Modern C++. We are in Week 8 and I am going to discuss Module 36. In the last week, we have talked about a variety of things. We have discussed about the virtual function pointed cable, how was it implemented, and for three modules, we have discussed different aspects of typecasting in C++ including polymorphic runtime casting. And in the last module, we have discussed about multiple inheritance.

(Refer Slide Time: 01:04)



The screenshot shows a presentation slide titled "Module Objectives". The slide content includes a single bullet point: "• Understand the Error handling in C". The slide is part of a larger presentation, as indicated by the navigation icons on the right and the footer text "Programming in Modern C++" and "Parth Pratap Das".

So, with this we kind of covered a major part of the primary features of C++ programming which you need to know. We will digress little bit, step aside. And in this module and the next, we are going to talk about how do you handle errors that happen in C or C++ program. In the present module, we will talk about error handling features available or styles available in C. And then in the next module, we will see how they are again refined significantly in C++.

(Refer Slide Time: 01:42)



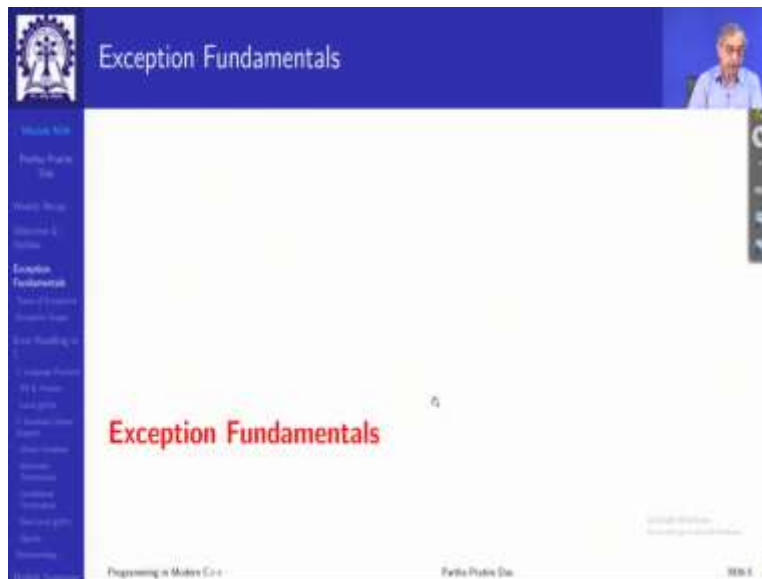
The screenshot shows a presentation slide titled "Module Outline". The slide content includes a list of topics:

- 1 Weekly Recap
- 2 Exception Fundamentals
  - Types of Exceptions
  - Exception Stages
- 3 Error Handling in C
  - C Language Features
    - Return Value & Parameters
    - Local goto
  - C Standard Library Support
    - Global Variables
    - Abnormal Termination
    - Conditional Termination
    - Non-Local goto
    - Signals
  - Shortcomings
- 4 Module Summary

The slide is part of a larger presentation, as indicated by the navigation icons on the right and the footer text "Programming in Modern C++" and "Parth Pratap Das".

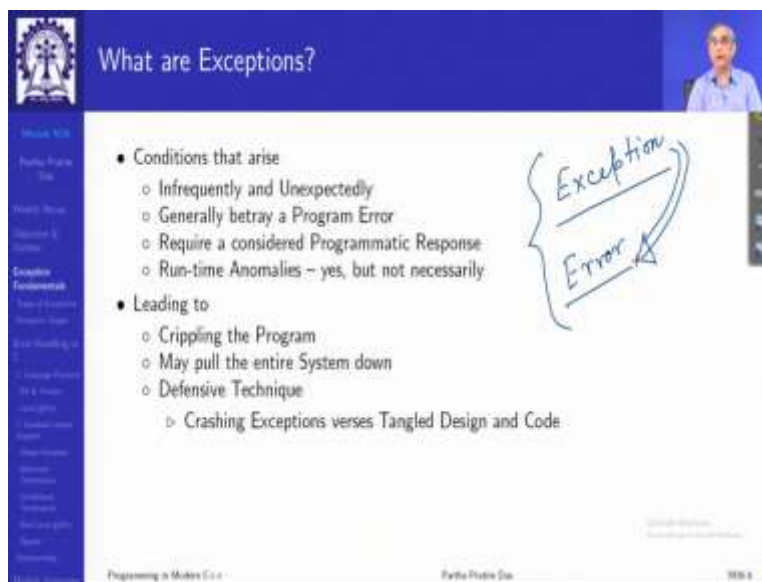
This is the outline and will be available on our left panel.

(Refer Slide Time: 01:48)



Now, let us, let me start by defining few fundamental notions.

(Refer Slide Time: 01:55)



The first thing that you must notice is I am using a word exception. We have in C programming talked about errors. Now, there is a subtle difference between errors and exceptions. Errors are normally what you know is a, it could be a programming error, it could be that the logic is not working correctly. It could be that you expect, you do not expect the stack to be empty, but it has become empty at a certain point, so the data is not appropriate.

Whereas an exception is often something which happens due to maybe external factors or factors in your program, but normally, exceptions should be infrequent, unexpected. And many a times, they are crippling, they could put the entire system into difficulty and so on. Now, this dichotomy of terms continue to remain, because as C evolved, the separate notion of exception or separating exceptions for the normal programming error was not there.

So, in C, you will find that all kinds, whether it is your programming logic error, whether your, whether it is a error due to a data being too large or too small, or it is an error because something has gone wrong in terms of your hardware, yeah, somebody has pressed control C or anything, all were treated as errors.

So, C talks about error handling, whereas C++ tried to build up on that and take out as much of these error situations, common error situations out in terms of what is defined as exception and still review with the scope of handling your program errors but others are handled in a very structured manner. We will talk about that in the next module. But that is why I say that you know at times you might feel confused with this use of word error and the exceptions, but they kind of are talking about the same thing, but errors are what we commit and exceptions often are what infrequently, unexpectedly happens.

(Refer Slide Time: 04:56)

**Exception Causes**

- Unexpected Systems State ✓
  - Exhaustion of Resources
    - ▷ Low Free Store Memory
    - ▷ Low Disk Space
  - Pushing to a Full Stack ←
- External Events ✓
  - Ctrl-C ✓
  - Socket Event ✓
- Logical Errors ✓
  - Pop from an Empty Stack ✓
  - Resource Errors – like Memory Read/Write
- Run-time Errors ✓
  - Arithmetic Overflow / Underflow
  - Out of Range
- Undefined Operation ✓
  - Division by Zero

Progressing to Modern C++ Part 10: Primitives 10/17

Now, what could be the cause of an exception? There could be several, and a majority of them have listed there. It could be due to unexpected system state, that is you have run out of free store, you have run out of disk area, you are trying to, you have an array to hold a stack and you are, you have filled up that array already. You are pushing to your full stack. So, these are kind of, you are running out of system state.

So, some of them at least is not under a programmer's control, while programming you would not conceive this. Some maybe if you are cautious or maybe not. There could be external events like control C to terminate a program that would be an event on the network socket and so on. There could be simple logical errors. This is what I was saying that more common errors, you are trying to pop from an empty stack, your resource errors like memory leak, memory read write errors and so on.

There could be run time errors like divide by 0, underflow, overflow, out of range. Undefined operation divide by 0 actually. This is undefined, not exactly run time error. So, there could be different cases of exceptions, root of exceptions. And the programmer has to handle all of that together.

(Refer Slide Time: 06:23)

**Exception Handling?**

- Exception Handling is a mechanism that separates the detection and handling of circumstantial Exceptional Flow from Normal Flow
- Current state saved in a pre-defined location
- Execution switched to a pre-defined handler

Exceptions are C++'s means of separating error reporting from error handling

– Bjarne Stroustrup

Programming in Modern C++ Partha Pratim Das 18/03

Now, in the world of C, this, in that since the programmer has to handle this, so there will be code detect such errors happening, to take care of that, to handle that, come back to continue the

program. So, this part is what is now called an exception flow. And normal flow refers to if nothing of this happened, then whatever the logic that you wanted to write. In C typically, these two are entangled twice, one into the other. In C++, the attempt has been to separate out the exception flow from the normal flow. We will see that, we will check that on the next module after we have seen the style of C.

(Refer Slide Time: 07:12)

**Types of Exceptions**

- **Asynchronous Exceptions:**
  - Exceptions that come Unexpectedly
  - Example - an Interrupt in a Program
  - Takes control away from the Executing Thread context to a context that is different from that which caused the exception
- **Synchronous Exceptions:**
  - Planned Exceptions
  - Handled in an organized manner
  - The most common type of Synchronous Exception is implemented as a throw

Programming in Modern C++ Partho Pratap Das 18/08/2018

There are exceptions are primarily of two types, asynchronous and synchronous. Asynchronous is which has no logic, no relationship with the current execution thread. So, that come in unexpectedly, like interrupting a program, taking a thread away from the context, and so on. Whereas synchronous exceptions are mostly planned exceptions. They are handled in an organized manner. And most common type, as we will see, is implemented in terms of a throw. This will come in details. I am just trying to give you a basic overview to the whole idea.

(Refer Slide Time: 007:55)

**Exception Stages**

- [1] Error Incidence**
  - Synchronous (S/W) Logical Error
  - Asynchronous (H/W) Interrupt (S/W Interrupt)
- [2] Create Object & Raise Exception**
  - An Exception Object can be of any Complete Type - an `int` to a full blown C++ class object
- [3] Detect Exception**
  - Polling - Software Tests
  - Notification - Control (Stack) Adjustments
- [4] Handle Exception**
  - Ignore: hope someone else handles it, that is, Do Not Catch
  - Act: but allow others to handle it afterwards, that is, Catch, Handle and Re-Throw
  - Own: take complete ownership, that is, Catch and Handle
- [5] Recover from Exception**
  - Continue Execution: If handled inside the program
  - Abort Execution: If handled outside the program

Progressing in Modern C++  
Part 10: Primitives

Now, typically exception is considered or error handling is considered to be in five stages. First, the incidence of the error, the error will have to happen. It can happen synchronously, logical error, or it can happen asynchronously, which is the hardware error, like run out of memory, run out of disk space, socket event, and so on. Once that has happened, then you create an object and raise that exception. You have to tell somebody either later part of your own function or the calling function that something has happened which was not desired.

So, you create an object which is indicative of that and you raise that exception which this object could be a simple integer value that, I mean, we often say that from main return 0, it means health. If I return minus 1, if I return minus 2, it means that something is wrong. So, that is the notion of the object.

If I give, return an int, which is -1, it says that, well, not everything is fine. Then if you are raising that, then it has to be detected. That somebody has to respond to that, somebody has to check that I have returned -1. So, that is polling, it could happen through notification also, we will see more of that later.

And then you have to handle that exception. There could be several strategies which are specific to a particular situation. You could ignore, it may be okay. I know that, that is fine, I do not care.

It could act that do something and also let others do more things. So, you do something and maybe call other function to do more or you can take total ownership, is okay. I have understood what the error was, I have taken care of that and now it is all handled, we can proceed. And finally, to recover, that is, if it is possible, that is what is desirable that you continue with the execution, or if you cannot, then you abort. So, these are the basic five stages of an exception or error handling that one has to deal with.

(Refer Slide Time: 10:23)

```
int f() {
    int error;
    /* ... */
    if (error) /* Stage 1: error occurred */
        return -1; /* Stage 2: generate exception object */
    /* ... */
}

int main(void) {
    if (f() != 0) /* Stage 3: detect exception */
    {
        /* Stage 4: handle exception */
    }
    /* Stage 5: recover */
}
```

And I will show you a simple example. So, function main calls a function f here, f has a variable internally which it checks to see if an error has happened or it sets if an error has happened. So, these are different function codes that go in that. And if the error has happened, then it returns -1. So, when you check this value error and you find that error has occurred, that is the instance, incidence of the error, stage 1. Then when you return, you have generated the exception object, which is an integer value -1 and you are raising it. The control comes back here.

Then you are checking if it is not 0. If it is 0, then everything is healthy. If it is not 0, then an exception has happened, so you detect. Then you write a piece of code to handle that whatever needs to be done, and then you proceed again. Maybe we will call that function again or do some other function, and so on. So, this is, this structure simply tells you in principle what are these five stages that they do.

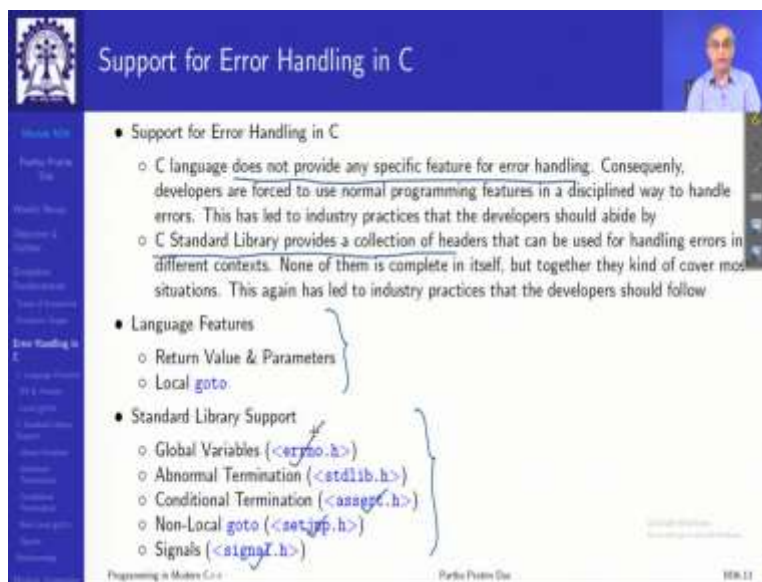


(Refer Slide Time: 11:56)



So, this was the overall general idea about exceptions and errors. And let me talk specifically about what you have in C.

(Refer Slide Time: 11:55)



Now, as a language, C has not provided any specific feature for error handling. It was not considered at that time, and henceforth, the language has no feature for error handling. The developers have to use the normal program code development styles, like the one we just saw to handle errors. So, over time, what has happened, there are common industry practices which

have emerged which the developers should abide by, several companies have coding styles where they say that you will have to handle this error like this, that error like that, and so on and so forth.

When it was realized that C language has no support for error handling, several C standard library headers started coming up to deal with different types of errors in different contexts. But there was no comprehensive support, because the language is already there fixed and you are providing library to support. So, each header library, standard library deals with certain aspects of certain types of errors, but not together complete in itself.

So, the onus still lies with the programmer to decide which particular library to use in which context. And if you do that properly, then most of the error situations can be handled pretty well in C as well. But the best that I would advise is follow the industry practices, because that is a most important, then you will not, you will be consistent with your fellow developers and you will not need to reinvent the wheels.

Now, if we look at the language features that the only thing C has is it can return a value, single value or multiple parameters, and it has something called a local goto, local goto, which we normally say do not use. So, this is probably the only situation where we will advise that you may use gotos locally. And then there are different standard, you can see that there are five types of support I have talked of Global Variable, Abnormal Termination, Conditional Termination, Non-Local gotos which is a new concept and signals, each one come from a different library component, different header.

Some are like the setjmp, assert, signal, errno number, these are specifically created for error handling standard library of course and several other things as well. So, now we will go over and look at each one of them, what do they do and how they can be used for error handling?

(Refer Slide Time: 14:46)

**Return Value & Parameters**

- **Function Return Value Mechanism**
  - Created by the Callee as Temporary Objects
  - Passed onto the Caller
  - Caller checks for Error Conditions
  - Return Values can be ignored and lost
  - Return Values are temporary
- **Function (output) Parameter Mechanism**
  - Outbound Parameters
  - Bound to arguments
  - Offer multiple logical Return Values

Handwritten annotations on the slide:  
- `int add(int, int)` (circled)  
- `int add(int, int, int *)` (circled)  
- `c = add(a, b)`  
- `c = add(a, b, &err)`

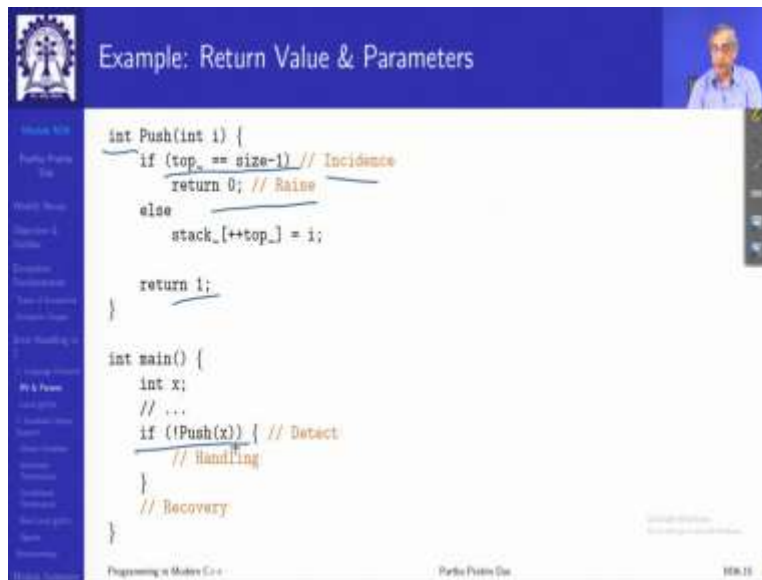
The first one, return value and mechanism is very simple. The caller checks for error condition and when you have an error situation, you return the error object as we just saw. The return value can be ignored and lost, that is a problem. And return values are temporary, so you will not, if you do not catch it or if you do not take action on it, as soon as the control from the function comes back, you will lose that information. If you now once you have that, if you want to use return value to give you the status of what has happened in the function, then you lose the basic value of the function. So, what I am meaning is I have add, say int, int, int.

Now, how do I use return value to check whether add has happened properly or it had a overflow? It is adding two integers, it might have an overflow. So, how do I check that? If I want to return a status value here, then I lose the basic property of the function which is to give the added value of the two numbers passed to it. So, what something unnatural that we can do is we can have more parameters. This is an explicitly outbound argument where I can pass an integer value back to my caller to show whether the function is done successfully or not. But it is a huge problem, because he will use the function as `C add(a, b)`. Now, you have to use it as `C is add(a, b, &error)`.

Even that does not work, because you are not being able to check this error. So, if you have to check that, you have to do if, you have to first do `C add` all this, then you have to check the error

that has come up. Take actions, it is a big mess. The whole beauty of functions get lost, but that is the best that you have if you are using function returning values.

(Refer Slide Time: 17:47)



The slide displays the following C code:

```
int Push(int i) {  
    if (top_ == size-1) // Incidence  
        return 0; // Raise  
    else  
        stack_[++top_] = i;  
  
    return 1;  
}  
  
int main() {  
    int x;  
    // ...  
    if (!Push(x)) { // Detect  
        // Handling  
    }  
    // Recovery  
}
```

Like here, I have changed the signature of stack to return an int, so that if the stack is full, then you return a 0, otherwise you return 1. So, always after push you have to check that to detect and to be able to handle.

(Refer Slide Time: 18:12)



The slide lists the following points:

- Local goto Mechanism
  - (At Source) *Escapes*: Gets Control out of a Deep Nested Loop
  - (At Destination) *Refactors*: Actions from Multiple Points of Error Inception
- A group of C Features
  - goto Label;
  - break continue;
  - default switch case

Local gotos or simply gotos for now has always happened C programmers, has helped C programmers to kind of get out from a very deeply nested loop or to refactor the same code, same error handling code from multiple places together. So, under this, I will put all different kind of control statements like goto, break and continue which are things that you do for error handling, that if some error conditions happen, you break from a loop or you continue in the next iteration of the loop without doing the rest, you can, have default in case of switch to take care of situations that are not expected and so on and so forth.

(Refer Slide Time: 19:05)

```
_PHELA _cdecl signal(int signum, _PHELA sigact)
{ // Lifted from VC98\CRT\SRC\WINSIG.C
... /* Check for sigact support */ ✓
    if ( (sigact == ...) ) goto sigacterror;
    /* Not exceptions in the host OS. */ ✓
    if ( (signum == ...) ) { ... goto sigacterror; }
    else { ... goto sigactok; }
    /* Exceptions in the host OS. */ ✓
    if ( (signum ...) ) goto sigacterror;
}

sigactok:
    return(oldsigact);

sigacterror:
    errno = EINVAL;
    return(SIG_ERR);
}
```

So, for example, this is just a sample code to show you this I lifted from the WINSIG.C, source of certain version of Visual Studio and a lot of the code I have skipped, because it is a huge code. I just wanted to show that, well, in different points. The programmer is checking for different error conditions like sigact support, not exceptions in the hostways, exceptions in the hostways and so on. And doing a goto to a common level, gotos are bad we know. But this kind of a discipline uses code, because otherwise we would need to write the handling code of what happens on that error right at that point, which will be lot of code repetition. And as you know, any code repetition is bad.

(Refer Slide Time: 20:10)

The slide, titled "Example: Local goto", displays a C code snippet for a signal handler. The code uses `goto` statements to branch to different error handling labels. Red arrows illustrate the control flow from various `goto` statements to the `sigretok` and `sigreterror` labels. The `sigretok` label returns the original signal, while the `sigreterror` label sets `errno = EINVAL` and returns `EINVAL`.

```
_FHANDLER _cdecl signalist signal, _FHANDLER sigact)
{ // Lifted from VC96\CR\MSVC\WINSIG.C
... /* Check for sigact support */
  if ( !sigact == ...) goto sigreterror;

  /* Not exceptions in the host OS. */
  if ( !signal == ...) { ... goto sigreterror; }
  else { ... goto sigretok; }

  /* Exceptions in the host OS. */
  if ( !signal ...) goto sigreterror;
...
sigretok:
  return(oldsigact);
sigreterror:
  errno = EINVAL;
  return(EINVAL);
}
```

So, this is another style which is often used. So, this is just to show you in terms of visualization that this is the program code, and if an error happens and all of these are gathered into one point, so which gives you a power to refactor. And if it is okay, then it gathers to a different point. So, you have only two program points where you may need to fix what you do. But from everywhere else, the success and failure fall into these two buckets. So, that is industry recommended practice of using gotos for handling errors.

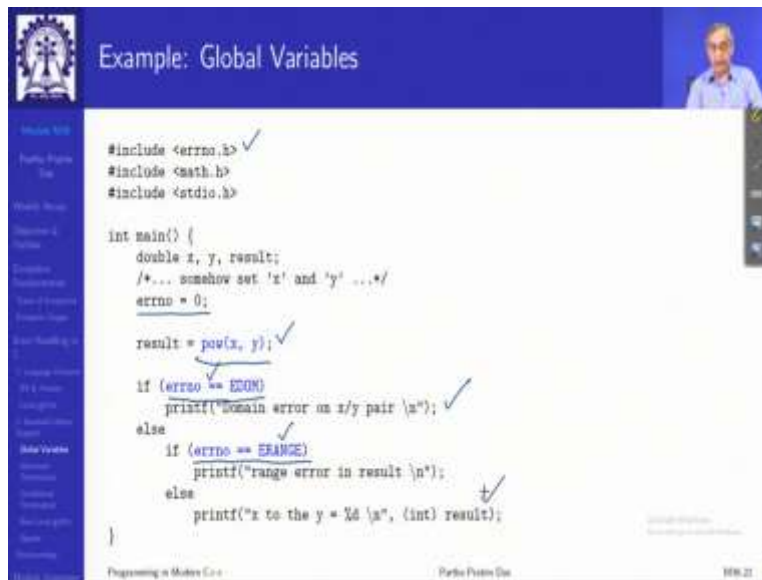
(Refer Slide Time: 20:47)

The slide, titled "Global Variables", lists two mechanisms for handling global variables (GVs) in the context of error handling:

- GV Mechanism
  - Use a designated Global Error Variable
  - Set it on Error
  - Poll / Check it for Detection
- Standard Library GV Mechanism
  - `<errno.h>/<errno>`

Now, let us go into the different features that the libraries provide. The GV Mechanism or Global Variable Mechanism is several library functions practice this that if they come across an error, if they happen to have an error, then they set a global variable which is provided in the errno.h. So, an example will make it clear.

(Refer Slide Time: 21:14)



```
#include <errno.h> ✓
#include <math.h>
#include <stdio.h>

int main() {
    double x, y, result;
    /*... somehow set 'x' and 'y' ...*/
    errno = 0;

    result = pow(x, y); ✓

    if (errno == EDOM) ✓
        printf("domain error on x/y pair \n"); ✓
    else
        if (errno == ERANGE) ✓
            printf("range error in result \n");
        else ✓
            printf("x to the y = %d \n", (int) result); ✓
}
```

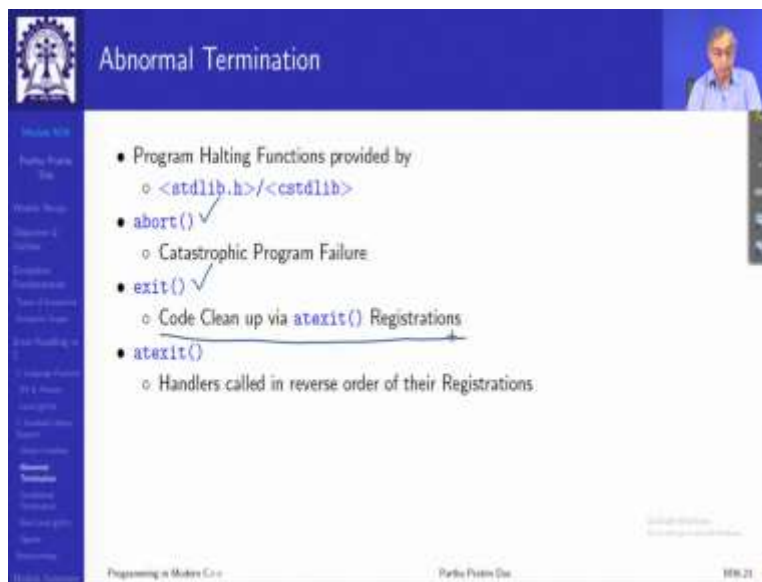
Here, I am trying to do a power function, x to the power y. So, x and y are some values. Now, when you do that, before that you set errno to 0 and you have included this header file. errno is defined in that header file, is you are not declaring that variable, you are just setting it to 0, it is a global variable. So, if pow has some kind of a problem, say it is not of the proper domain where pow can be used. It will set errno to enumerated value EDOM error DOM.

So, if after pow, you find that errno is EDOM, then you know that this error has happened in pow and you can decide what to do. pow is taking power, so it can potentially make the numbers very big, so it can easily go out of range. So, this is another which tells you that it has gone out of range and you can check that and do this. Otherwise, you know that everything is correct and you can proceed with the result.

So, this is a standard practice that several C standard library functions follow and you can get if the function has worked properly or not using this GV Mechanism. And naturally after you have

found an error, naturally what you will have to do is to reset the error no again to 0 so that you can find the next error.

(Refer Slide Time: 22:47)



The slide is titled "Abnormal Termination" and features a blue header with a logo on the left and a small video inset of a speaker on the right. The main content is a list of program halting functions provided by `<stdlib.h>/<cstdlib>`. The list includes:

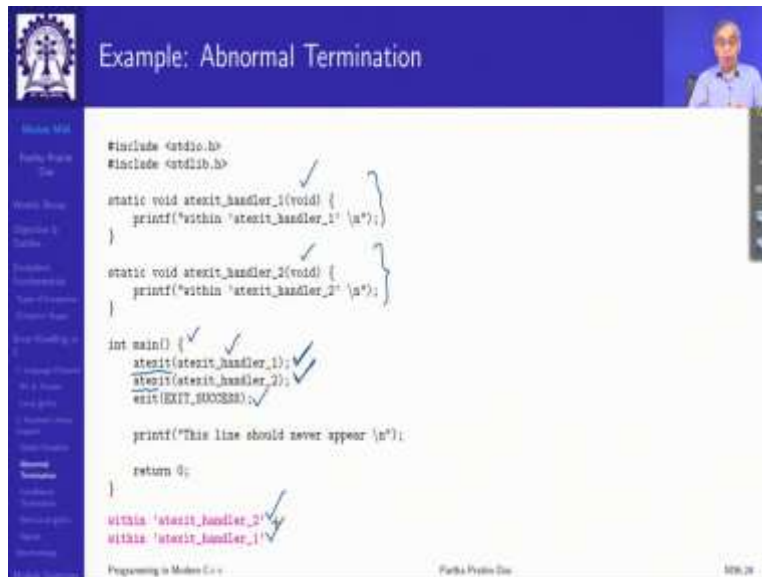
- Program Halting Functions provided by
  - `<stdlib.h>/<cstdlib>`
- `abort()` ✓
  - Catastrophic Program Failure
- `exit()` ✓
  - Code Clean up via `atexit()` Registrations
- `atexit()`
  - Handlers called in reverse order of their Registrations

At the bottom of the slide, there is a footer with the text "Programming in Modern C++", "Part 10: Program Exit", and "10/21".

Let us move to the next one which is abnormal termination. That is this you take care of course of if you have come to a point where there is no other way than to terminate. There are two ways to terminate, two functions to terminate a function, terminate a program. One is called abort, which is called catastrophic program failure. There is nothing that else that you can do. So, you just want to go out of this. You call abort, it does nothing. But there is another which mostly you should use, which is known as exit. Exit can take care of a few things using at exit registration. So, let us see what is at exit.



(Refer Slide Time: 23:36)



```
#include <stdio.h>
#include <stdlib.h>

static void atexit_handler_1(void) {
    printf("within 'atexit_handler_1' \n");
}

static void atexit_handler_2(void) {
    printf("within 'atexit_handler_2' \n");
}

int main() {
    atexit(atexit_handler_1);
    atexit(atexit_handler_2);
    exit(EXIT_SUCCESS);

    printf("This line should never appear \n");

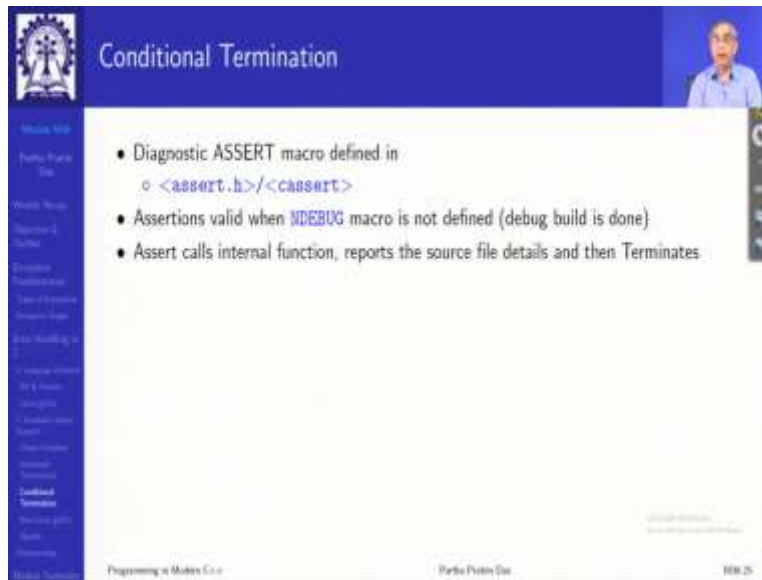
    return 0;
}

within 'atexit_handler_2'
within 'atexit_handler_1'
```

Here is a main function. Here are two functions that I have defined as handlers. What is important to note is I have called the library function at exit and passed the pointer to the first function. What it does atexit internally maintains a stack of function pointers, so it will put it inside the stack. Then I have done atexit once more. A second handler is also pushed onto the atexit stack.

Then I do exit. If I do exit what it does, it goes to that stack, pops up the first function pointer, calls it. So, you will get atexit\_handler 2. Similarly it takes us next, calls it. So, if you can see with this atexit mechanism, I can, before I go out I can try to do some wrap up, maybe some memory was allocated which now need to be released and all that. Those can be done using these handlers. And for moving on to C++, please understand that this atexit mechanism is which makes the invisible call to the destructors possible when you go out of the function scope. But if you instead of exit, if you call abort, this atexit registered handlers will not be called.

(Refer Slide Time: 25:27)



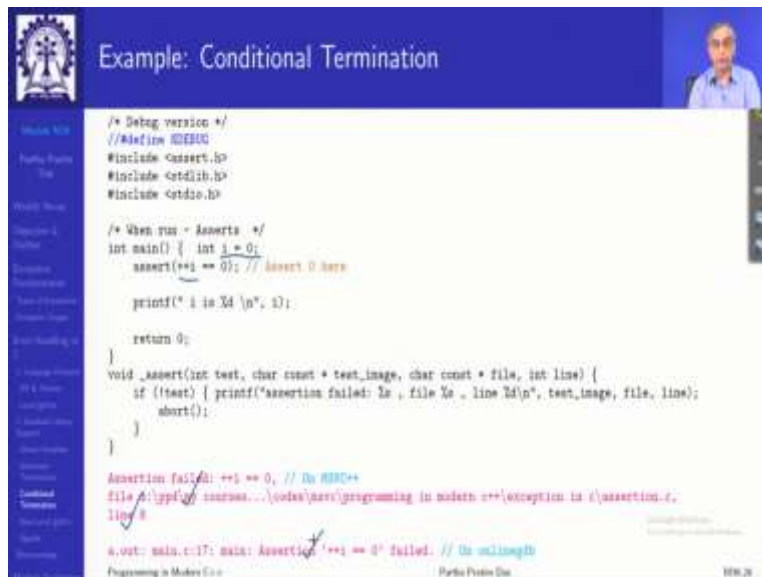
### Conditional Termination

- Diagnostic ASSERT macro defined in
  - `<assert.h>/<cassert>`
- Assertions valid when `NDEBUG` macro is not defined (debug build is done)
- Assert calls internal function, reports the source file details and then Terminates

Programming in Modern C++ | Parth Patel | 25:27

There is a conditional termination also. You can, you have a feature in assert, where you can assert a certain property under the debug build.

(Refer Slide Time: 25:36)



### Example: Conditional Termination

```
/* Debug version */
// #define NDEBUG
#include <assert.h>
#include <stdlib.h>
#include <stdio.h>

/* When run - Asserts */
int main() { int i = 0;
    assert(++i == 0); // assert 0 here
    printf("i is %d \n", i);
    return 0;
}

void _assert(int test, char const * test_image, char const * file, int line) {
    if (!test) { printf("assertion failed: %s, file %s, line %d\n", test_image, file, line);
                abort();
    }
}

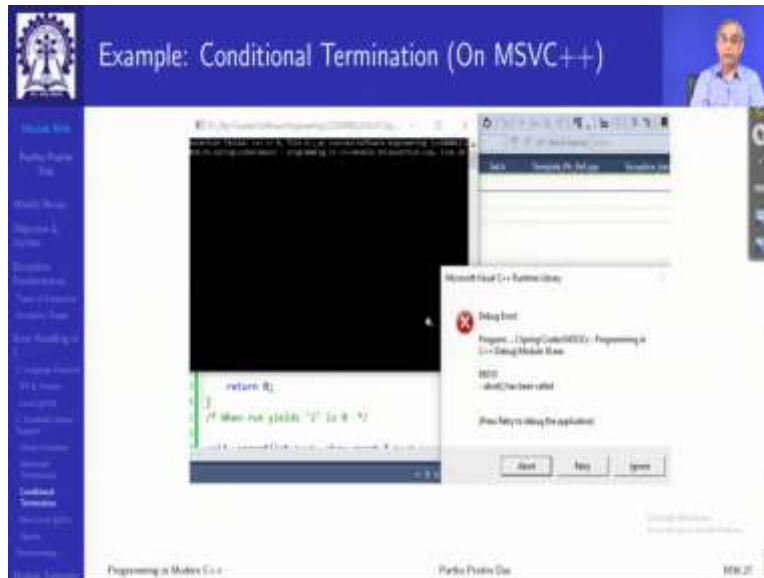
Assertion failed: ++i == 0, // Do NOT++
file ./pp1.cpp, source.../codes/ncv/programming in modern c++/exception in c/assertion.cpp,
line 8
a.out: main.c:17: main: Assertion '++i == 0' failed. // Do nothing!!
```

Programming in Modern C++ | Parth Patel | 25:36

So, you do that using the NDEBUG definition of the compiler. So, debug or NDEBUG, NDEBUG means do not debug, debug means debug. So, I have kept it commented. So, I am in the debugging build. And I say assert and I say some condition. So, if that condition fails then it will assert, that it will terminate the program at that point. So, assertion failed because here, if

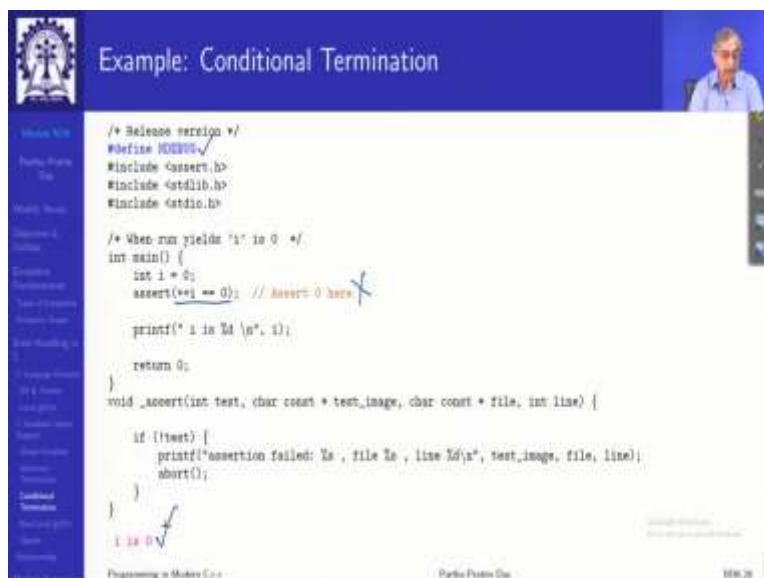
you looked at then i as 0 and I do ++i, so I became one, so it is not equal to 0. So, it has failed and it gives a file name, the line number. All of these I can easily print. This is from Microsoft Visual C++. This is a simple output from online GDB. You can check what kind of output you get on your compiler. Now, the advantage of this assert is we are using this, at compile time, you can see conditions at different places.

(Refer Slide Time: 26:45)



But well, this is how the visual studio screen looks after the assertion has failed.

(Refer Slide Time: 26:53)



Now, the advantage of assert is that if you have checked everything and you are okay, then for the release build, you can just erase the assert silently. All that you need to do is to set NDEBUG, define NDEBUG. If you define NDEBUG, then this part is under ifdef condition, and therefore this part will not come in the compilation, it is not in the source. So, in the same case, now you do not have any assertion happening. Now, you have the actual output that is going on. So, this is very good for conditional compilation.

(Refer Slide Time: 27:31)

Non-Local goto

- `setjmp()` and `longjmp()` functions provided in `<setjmp.h>` Header along with collateral type `jmp_buf`
- `setjmp(jmp_buf)`
  - Sets the Jump point filling up the `jmp_buf` object with the current program context
- `longjmp(jmp_buf, int)`
  - Effects a Jump to the context of the `jmp_buf` object.
  - Control return to `setjmp` call last called on `jmp_buf`

Programming in Modern C++ | Part 10: Primitives | 10.21

The most important and closest to C++ is what is called a non-local goto that is a local, with local goto, you can take care of error which is within a function. But what you have called a function and that function has an error, how does that information come back to the caller? That is the basic question. So, it uses two calls, two functions, one is called `setjmp`, one is called `longjmp`.

`setjmp` sets a point in the caller, where, in case of error, you would like to come back and `longjmp` is in the called function where you say that, well, I have an error, so I want to go back to the caller where the `setjmp` was done. Now, suddenly you have two different environments, two functions. One is a calling function, one is a called function. So, when you are doing `longjmp` you are in the call function. So, how do you know the environment of the calling function? So, you use a `jmp_buf`, which stores this environment information. That is the simple idea and we will see how this has got, has been imbibed in C++ in terms of a beautiful exception mechanism.

(Refer Slide Time: 28:50)

The slide illustrates the dynamics of a non-local goto. It is divided into two main sections: 'Caller' and 'Callee'.

**Caller Code:**

```
int main() {
    if (setjmp(buf) == 0) {
        printf("g() called\n");
        g();
        printf("g() resumed\n");
    }
    else
        printf("g() failed\n");
    return 0;
}
```

**Callee Code:**

```
jmp_buf buf;

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

**Execution Flow:**

- (1) g() called: An arrow points from the `g();` line in the caller to the `void g() {` line in the callee.
- (2) g() successfully returned: An arrow points from the `return;` line in the callee back to the `printf("g() resumed\n");` line in the caller.

**Checklist:**

- g() called ✓
- g() started ✓
- g() ended ✓
- g() returned ✓

The slide also includes a navigation sidebar on the left and a small video inset of the presenter in the top right corner.

So, this is how you do it. This is the header, this is the jmp buffer that I have defined and you do setjmp, jump buffer. So, at this point, the buffer gets the environment of the main. And I am doing it for the first time, so it is 0, which means that everything is good. So, I say that g is being called, call g, the control comes here and the execution starts. If I assume that there is no error, then the execution will go till the end.

If there is an error, then this long jump will happen. So, let us see the subsequent progress. So, here I have assumed that there is no error. So, I have set the return point, I have called g naturally the control comes up to this point and goes back to this point. So, you say g called, then g started, g ended and g returned. This is the no error path.

(Refer Slide Time: 30:10)

The slide illustrates the dynamics of a non-local goto using `setjmp` and `longjmp`. It is divided into two columns: 'Caller' and 'Callee'.

```
Caller
int main() {
    if (setjmp(buf) == 0) {
        printf("g() called\n");
        g();
        printf("g() returned\n");
    }
    else {
        printf("g() failed\n");
        return 0;
    }
}

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

Execution flow is shown with arrows: from `main()` to `g()`, and from `g()` back to the `else` block in `main()`.

Execution steps:

- (1) `g()` called
- (2) `longjmp` executed
- (3) `setjmp` takes to handler

Log output:

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

Footer: Programming in Modern C++, Pariksha Pradhan Das, slide 22

Now, suppose I have set this error to true. I am just using a variable to simulate an error situation. So, again the `setjmp` is setting the buffer with the environment to 0 it is called. So, the control comes here. We are in a new environment. `g` started. So, `g` called, `g` started. Now, the error is true. So, you do a `longjmp`. What it does? In the `longjmp`, it uses the environment from the buffer and goes back to that same point, but with a non-zero value. So, it goes back to this context of `setjmp` with 1, which means that the else will now get executed. So, this part of the function does not continue and we say that function `g` has thrown and only the failed will happen now. So, `g` called `long jump` executed and `setjump` text to the handler. So, this is the basic mechanism of what is called nonlocal go to or `setjmp`, `longjmp`.

(Refer Slide Time: 31:31)

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

jmp_buf j;

void raise_exception() {
    printf("Exception raised. \n");
    longjmp(j, 1); /* Jump to exception handler */
    printf("This line should never appear \n");
}

int main() {
    if (setjmp(j) == 0) {
        printf("'setjmp' is initializing j. \n");
        raise_exception();
        printf("This line should never appear \n");
    }
    else
        printf("'setjmp' was just jumped into. \n");
    /* The exception handler code here */
    return 0;
}
```

'setjmp' is initializing j.  
Exception raised.  
'setjmp' was just jumped into.

Progressing in Modern C++ Parth Patel Dec 16th 21

It is somewhat counterintuitive. So, practice this example. So, I have given another example for you to practice and understand what is going on, but it is a very powerful mechanism and we will see how it seamlessly gets into C++ where you do not have to remember all this buffer and all these values and so on.

(Refer Slide Time: 31:42)

- Header <signal.h>
- `raise()`
  - Sends a signal to the executing program
- `signal()`
  - Registers interrupt signal handler
  - Returns the previous handler associated with the given signal
- Converts h/w interrupts to s/w interrupts

Progressing in Modern C++ Parth Patel Dec 16th 21

```
// Use signal to attach a signal
// handler to the abort routine
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

void SignalHandler(int signal) {
    printf("Application aborting...\n");
}

int main() {
    typedef void (*SignalHandlerPointer)(int);
    SignalHandlerPointer previousHandler;

    previousHandler = signal(SIGABRT, SignalHandler);

    abort();
    return 0;
}

Application aborting...
```

The last is signals which you can send a signal to an executing program and you can have a handler for that signal. Signals are particularly used if you have studied operating system, you must have talked about signals quite a lot. Signals are particularly used to convert a hardware interrupt into a software one. So, here I show how to do that, include signal define and handler which should, what it should do if that hardware event happens, then this is a function pointer for the SignalHandler and put a previous handler and then do signal.

So, I have actually forcibly sending signal abort to myself with this SignalHandler and this signal function returns me whatever was registered as a last function. Because it can, at a time, it can have only one registered function for every type of signal that you have. So, what will happen at this point? This has got registered. So, when I go to, when I get this signal abort, which I have sent myself, I will have this code executed which is say that application is aborting, and then you come and abort, you leave everything and go ahead. So, these are the different mechanisms that are possible.



(Refer Slide Time: 33:29)

**Shortcomings**

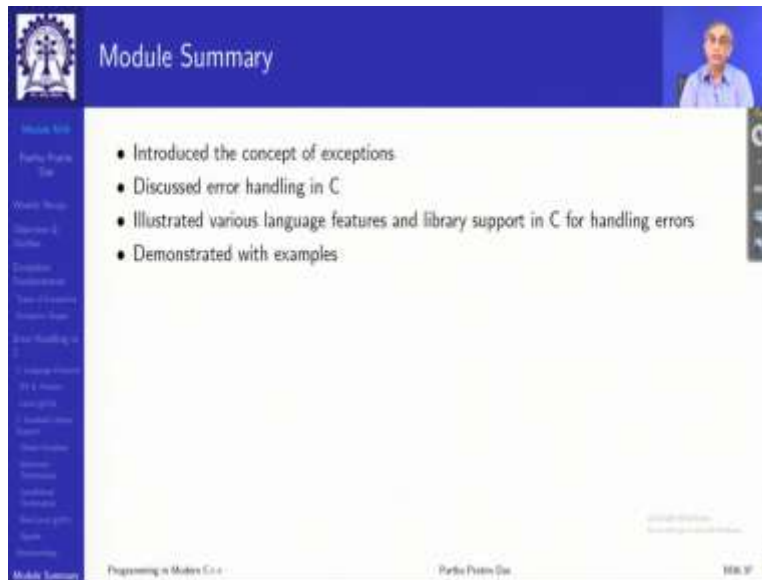
- **Destructor-ignorant:**
  - Cannot release Local Objects i.e. Resources Leak
- **Obtrusive:**
  - Interrogating RV or GV results in Code Clutter
- **Inflexible:**
  - Spoils Normal Function Semantics
- **Non-native:**
  - Require Library Support outside Core Language

Parth Patel 33:29

So, we have seen a number of C mechanisms, but they help you in error handling and that is how the C programmers survive. But certainly, they have a lot of shortcomings like they are destructor ignorant, they, it does not, they will not destroy the objects when long jump is going out of scope, they are obtrusive, that is, they are mixing up with your normal logic as we have saw in, seen in return value mechanism or in global variable mechanism.

They are not flexible, because normal function semantics is getting lost, because you have to have value parameters and so on. They are non-native, because they are not part of the language and they are coming as library support and so on. So, these shortcomings will be solved when we actually go on to C++ error handling exceptions in the next module.

(Refer Slide Time: 34:13)



**Module Summary**

- 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

Programming in Modern C++ Parth Patel

So, to summarize in this module, we have introduced the concept of exceptions and particularly discussed error handling in C, and illustrated various language features, but primarily library support in C for handling errors with examples. Thank you very much for your attention and see you in the next module.