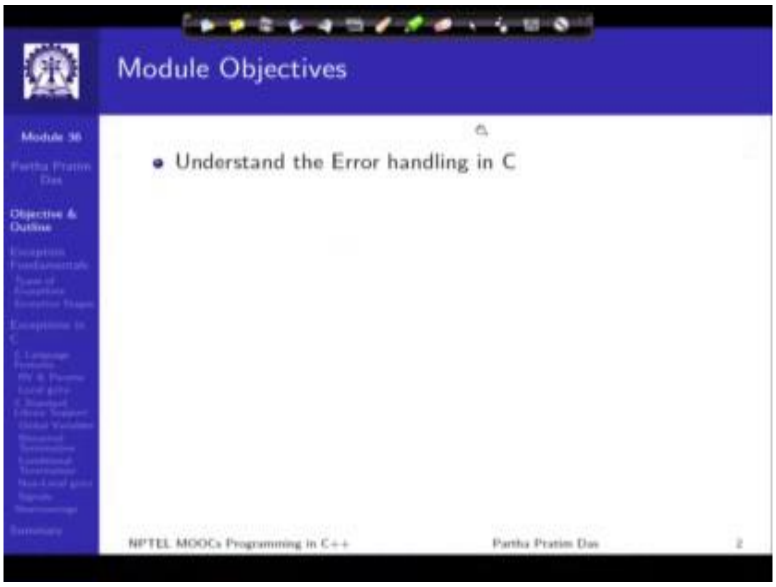
Programming in C++

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

Lecture – 52
Exceptions (Error Handling in C): Part – I

Welcome to module 36 of programming in C++. In this module and the next, we will try to take a look into exceptions, what is called exceptions in C++. It is basically handling errors, errors and extreme exception situations of a system, in general. And there are varieties of options that exist for it. Actual C++ does provide a whole lot of features, a very powerful exception handling feature. Before getting into that which we do in the next module.

(Refer Slide Time: 01:04)



Module Objectives

Module 36

Partha Pratim Das

Objective & Outline

Exception Fundamentals
Types of Exceptions
Exception Stages

Exceptions in C
C Language Features
RV & Params
Local goto
C Standard Library Support
Global Variables
Abnormal Termination
Conditional Termination
Non-Local goto
Signals
Shortcomings

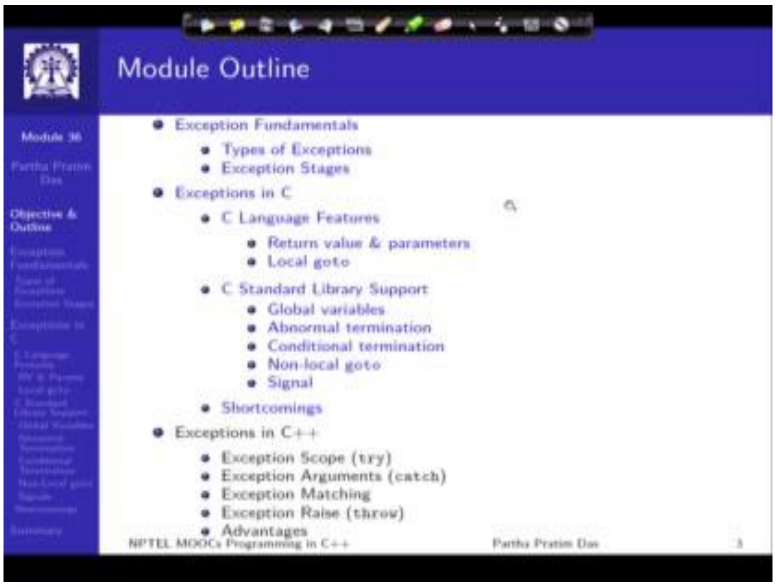
Summary

- Understand the Error handling in C

NPTEL MOOCs Programming in C++ Partha Pratim Das 2

We would first take a quick look into the error handling options that exist in C. I am sure you are aware of these already, but we just take a look, because they remain valid in C++ as well, and but they have certain short comings which the C++ exceptions would try to wave of.

(Refer Slide Time: 01:26)



Module Outline

Module 36

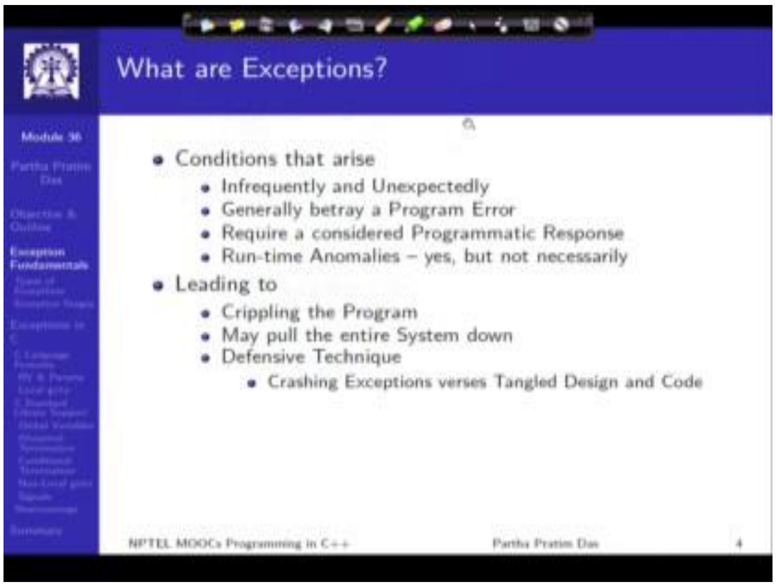
Partha Pratim Das

- 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

NPTEL MOOCs Programming in C++ Partha Pratim Das 3

So, this is the module out line for both the module 36 as well as module 37, certainly the blue part is what we do in this current module and that will be available on the left of your screen also the slides.

(Refer Slide Time: 01:44)



What are Exceptions?

Module 36

Partha Pratim Das

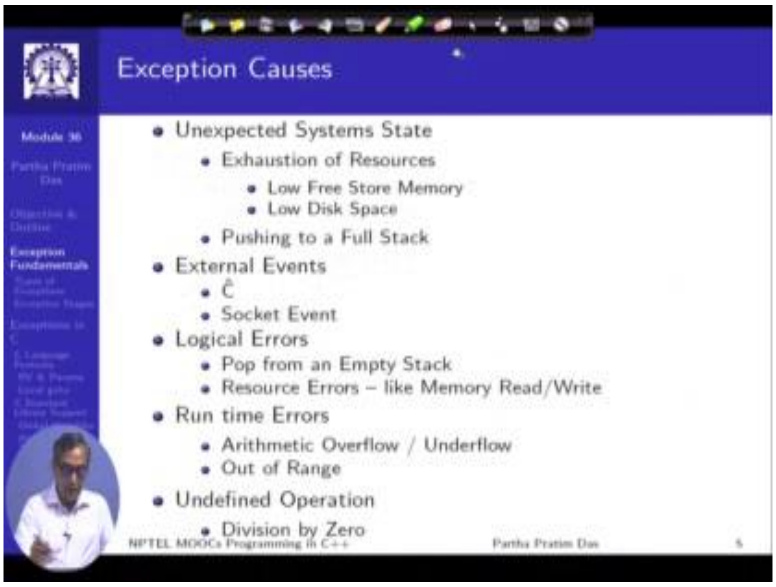
- Conditions that arise
  - Infrequently and Unexpectedly
  - Generally betray a Program Error
  - Require a considered Programmatic Response
  - Run-time Anomalies – yes, but not necessarily
- Leading to
  - Crippling the Program
  - May pull the entire System down
  - Defensive Technique
    - Crashing Exceptions verses Tangled Design and Code

NPTEL MOOCs Programming in C++ Partha Pratim Das 4

So, the question is what are exceptions? The exceptions are conditions that arise usually infrequently and certainly unexpectedly in many places. It generally betrays a program error and requires a certain amount of programmatic response. Usually, exceptions these conditions arise due to run time anomalies, but it may not necessarily be just run time anomalies they could occur even otherwise. And when these errors happen then that lead to crippling of the program. So, the program crashes or gets into some indeterminate state and things like that. And if it is really bad then it could not only impact the current program, but it could pull the system down all together.

So, certainly in C, what we have been doing is we have been following several defensive techniques that is try to conceive all different possible negative parts, all possible situations where things might go wrong and then try to take care of them. So that what happens is when we try to take care of the exceptions in this way then we get tangled into a whole lot of error code rather and the code that needs to take care of these error situations, you get tangled into a whole lot of a error paths, which clutter the basic design paths in the program. So that is a basic issue of exceptions situation.

(Refer Slide Time: 03:18)



Exception Causes

- Unexpected Systems State
    - Exhaustion of Resources
        - Low Free Store Memory
        - Low Disk Space
    - Pushing to a Full Stack
- External Events
    - Ĉ
    - 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

Module 36

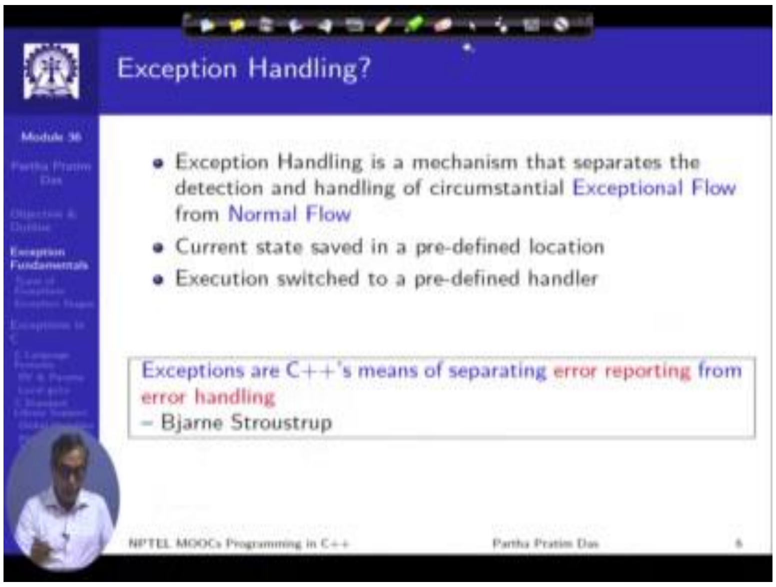
Partha Pratim Das

NPTEL MOOCs Programming in C++ Partha Pratim Das 5

And there are several causes I mean this is an indicative lecture to which covers most kinds of exceptions, but there could others. For example, primary being the unexpected systems state, for example, exhausting resources like memory, disk space and so on. Like trying to push into a stack whose int internal store is already become full. It could be due to external events like program termination given by the user control C or something like that. It could be due to a socket event, or it could also be due to different logical errors, for example, logical error is one like you are trying to pop from an empty stack.

Now, this is not related to system resources, but logically this is an error. There could be resource errors like memory read, write errors and so on. They could be other run time errors like arithmetic overflow underflow try to add two numbers and that overflows the number it could be out of range for this. And it could also be for undefined operations like division by 0 is something which will lead to exceptions. So, there are these are some of the typical reasons, but there could be more of those.

(Refer Slide Time: 04:35)



Exception Handling?

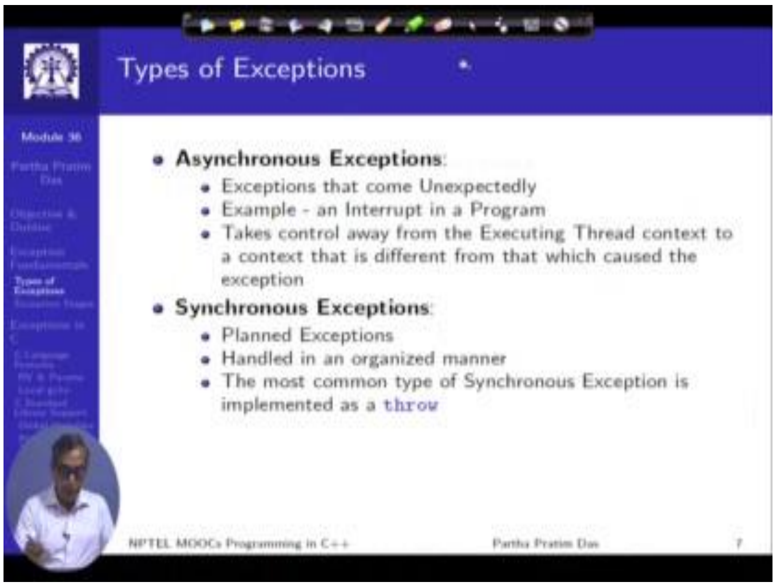
- Exception Handling is a mechanism that separates the detection and handling of circumstantial Exceptional Flow from Normal Flow
- Current state saved in a pre-defined location
- Execution switched to a pre-defined handler

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

NPTEL MOOCs Programming in C++ Partha Pratim Das 6

So, exception handling is what is very important in terms of C++ as Bjarne Stroustrup's comment shows, it is a mechanism that separates, this is the key idea that it tries to separate the circumstantial exceptional flow that is flow that arise due to these exceptional situation happening from the normal or the happy flow that we have actually designed for. So, that is basic thing that we want to do understand here. So, it the current state in this case should be saved in a pre defined location and the execution of the program should be handed over to a predefined exception handler or a function or a piece of code which is supposed to take care of the error situation.

(Refer Slide Time: 05:28)



Types of Exceptions

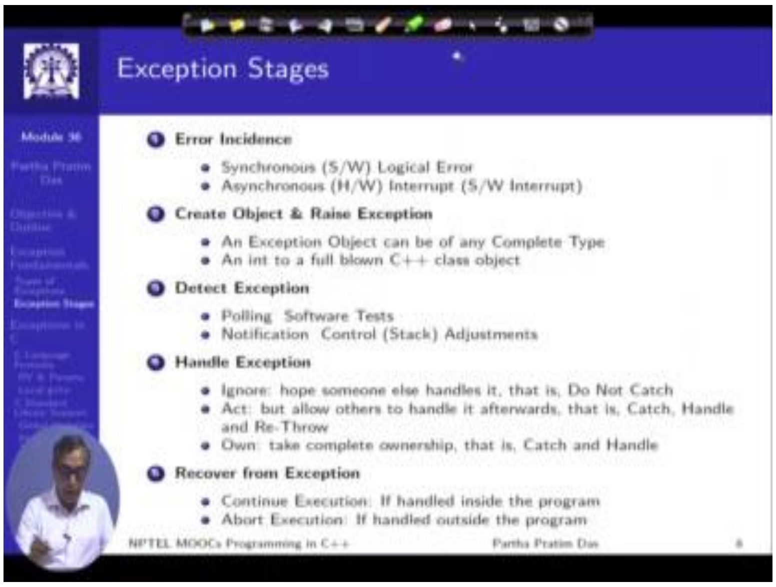
- **Asynchronous Exceptions**:
  - Exceptions that come Unexpectedly
  - Example - an Interrupt in a Program
  - Takes control away from the Executing Thread context to a context that is different from that which caused the exception
- **Synchronous Exceptions**:
  - Planned Exceptions
  - Handled in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

NPTEL MOOCs Programming in C++ Partha Pratim Das 7

So, if we also look at what are the types of exceptions? We can see that of the different causes that we have just listed. They can be broadly classified into two groups the asynchronous and the synchronous. Asynchronous is which is happening from a different thread usually. So, it is not in concurrence with the normal flow of the program that comes unexpectedly like the interrupt in a program and so on. Or it could be a planned exception planned in the sense; of course you do not plan for errors, but planned in the sense that it is something that happens along the flow of the program.

For example, you wanted to allocate memory and the resource was low, so an exception happens, you wanted to divide by a number and that number turns out to be zero. So, it is there will be little bit of different style to handle asynchronous exception, but for majority of the situations which are synchronous in nature. We would be like to that is the common situation. We will be implementing them in terms of certain feature in C++ called a through feature.

(Refer Slide Time: 06:36)



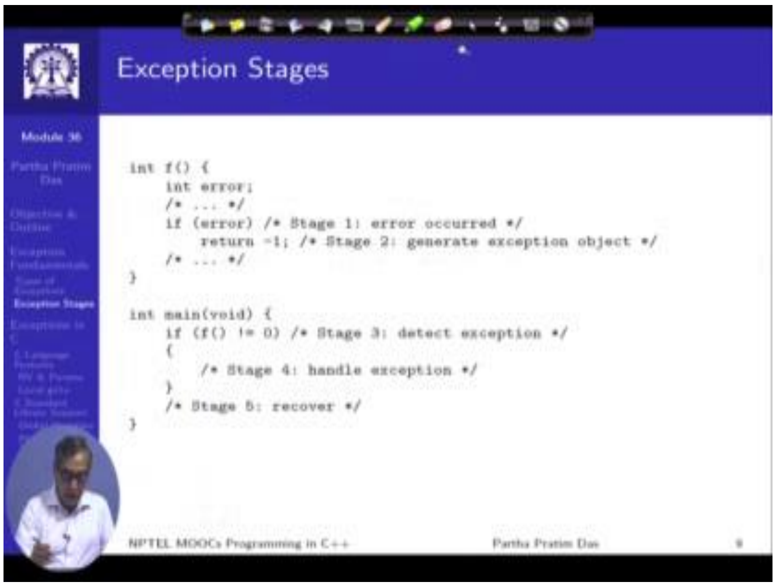
Exception Stages

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

NPTEL MOOCs Programming in C++ Partha Pratim Das 8

We will look into those. Now, before we get into the analysis, let us understand that exception actually happens in five different stages let me just first show you an example then we can come back here.

(Refer Slide Time: 06:48)



Exception Stages

```
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 */
}
```

NPTEL MOOCs Programming in C++ Partha Pratim Das 9

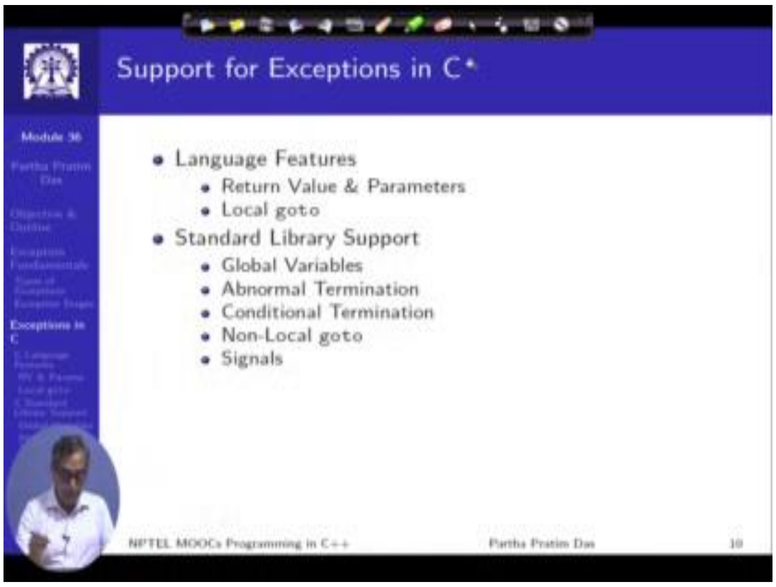
This is a very typical situation, this a function main which is calling a function f and in this case, it is assumed that as if, if error if function f works correctly then it will returns a 0; otherwise, it returns a non-zero value. So, you check if f as returned 0. And if it is not returned 0, then you know that something has gone wrong and you try taking care of that. So, if you look into what can happen is f is executing has been called is executing, let say is executing, and at some point it gets into a error situation this is I have just shown it as a as a kind of bool here, but this could arise due to many situations like low memory and so on. And this is called the stage one of an exception where you identify that an error situation has happened.

And then you report this error because f may not, for example, if it is low memory f does not know what to do in case of low memory. So, here f will have to report that which we say you generate an exception object, you will understand what does that mean in the context of C++ more, but basic thing is you want to report the error to the caller. So, when that happens the control comes back here, and control comes back with minus one which certainly fails this certainly satisfies this test. So, you know that an exception has happened that an error situation has happened.

So, the third stage is detect that the caller has been able to detect that an error has happened. And then this is what we say is a handler is a piece of code which should be executed only when this error situation as happened. For example, if this is due to low memory, then you made want to decide what you want to do here. For example, you could realize some of the memory that you have already allocated and you are not using at present you could release them so that more memory becomes available and you can go back and call f again. Whatever you decide to do is the basic handling of the exception. And once that has been handled then you have back to the normal flow which you say is a recover page that is you are recovered from that error.

So, create, report, detect, handle, and recover are the basic five stages through any exception can be handled. So, if you just go back to the; I am sorry, want to go back to the previous slide. So, here these are five stages error incidence, which is create, this is create object, and rise which is report, this is the detect, the handle and the recover which other different things that you can do in case of error situation. And how you do that depends on the mechanism that you are that is available to you mechanism that you are using.

(Refer Slide Time: 09:46)



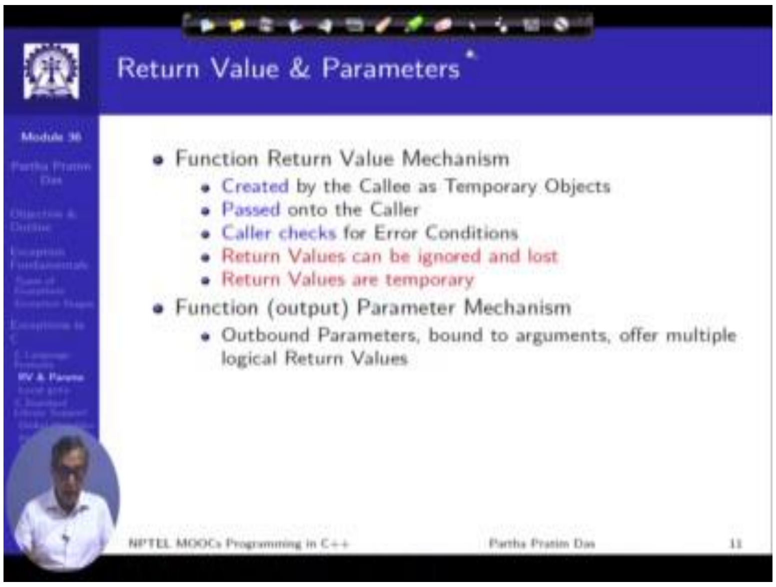
Support for Exceptions in C

- Language Features
  - Return Value & Parameters
  - Local goto
- Standard Library Support
  - Global Variables
  - Abnormal Termination
  - Conditional Termination
  - Non-Local goto
  - Signals

NPTEL MOOCs Programming in C++ Partha Pratim Das 10

So, in C, you have a variety of mechanisms and only very little is supported by the language actually the language does not support anything the C language does not support anything keeping error handling in mind. But you can use some of the language features to handle errors in a more structured way and instead, there are several libraries which is the part of the standard library which give you additional feature to be able to handle those errors. So, the first that you can do we will just take a quick look.

(Refer Slide Time: 10:18)



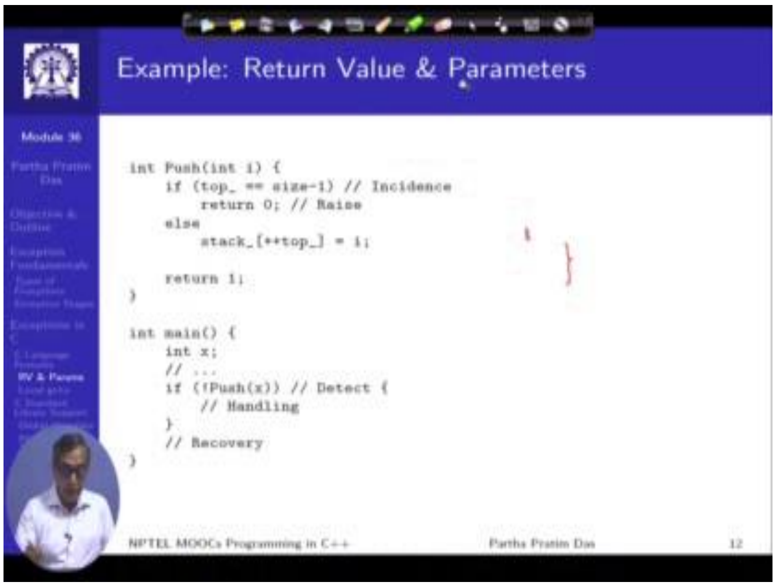
Return Value & Parameters

- 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

NPTEL MOOCs Programming in C++ Partha Pratim Das 11

This is something, which we are all familiar with that I call a function and I expect that function to return a value and that value will denote whether the function has successfully completed or there is an error, so this is by return value mechanism. Now if your function is actually suppose to return a value based on computation then you may want to return this error condition in terms of an additional parameter, possibly we will have some kind of a call by address parameter where you put the error and put it back. Now, the basic problem of these kind of error handling is the after the error has been reported it needs to be checked by the caller because if the error is a reported, but then the caller does not detect it then it carries on in some indeterminate stage. So, it a return values could be lost could be ignored and so that this is not a very effective mechanism for doing this.

(Refer Slide Time: 11:26)



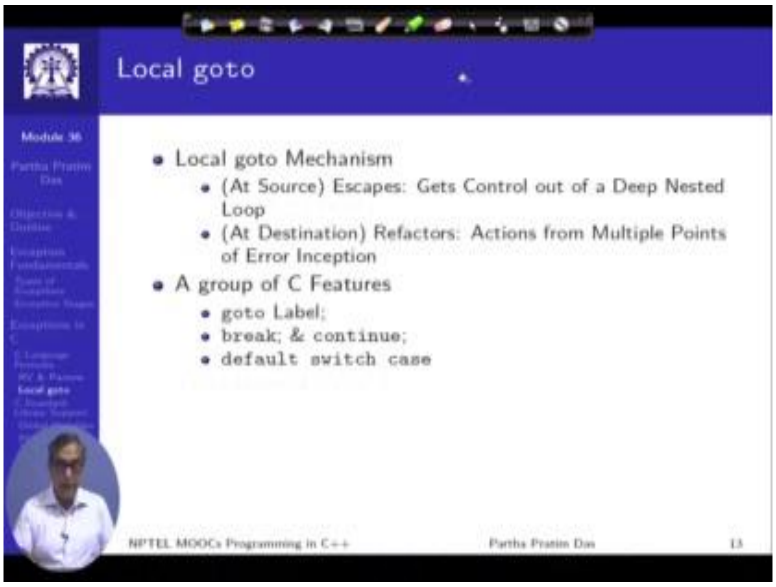
Example: Return Value & Parameters

```
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
}
```

NPTEL MOOCs Programming in C++ Partha Pratim Das 12

But several C programs use these mechanisms, for example, here I am looking at a push which returns an int. You will recall we have looked at push method for stack very often, and we typically have a void in the return. Because we do not really expect anything back from the push, but just to take care of the error you could have that it returns an int and so if it is got a say a the store has become full it returns 0 to designate that the push has felt. So, the caller will have to again detect that and handle that further. So, these are very simple mechanism very widely used a lot of C library is use this, but quite inefficient in that matter.

(Refer Slide Time: 12:10)



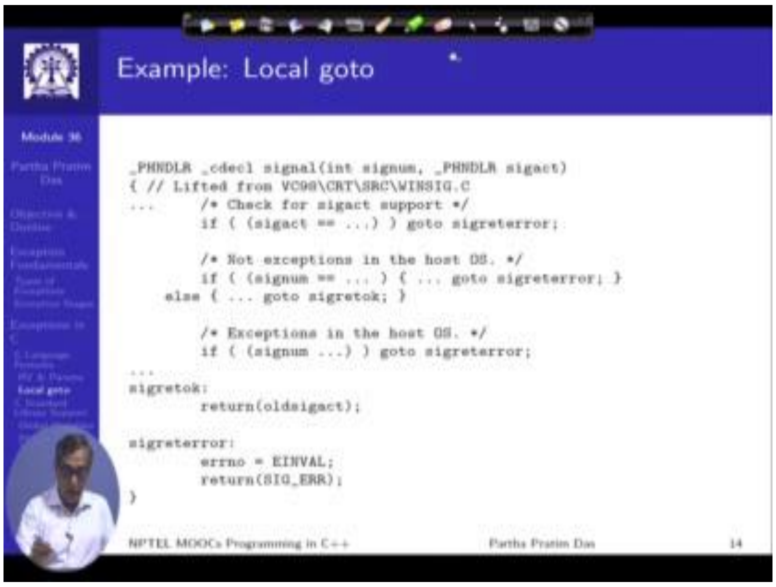
Local goto

- 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

NPTEL MOOCs Programming in C++ Partha Pratim Das 13

Another that we that is very frequently uses a local goto. The example will I mean there are there are several ways you can you can do local goto, in terms of actual goto, break continue, default switch case, these are all cases. For example, in switch we have a default. So, these are all cases where we are trying to take care of situations we have not been able to take care of in the happy path. So, local goto are a very typical way to handle.

(Refer Slide Time: 12:39)



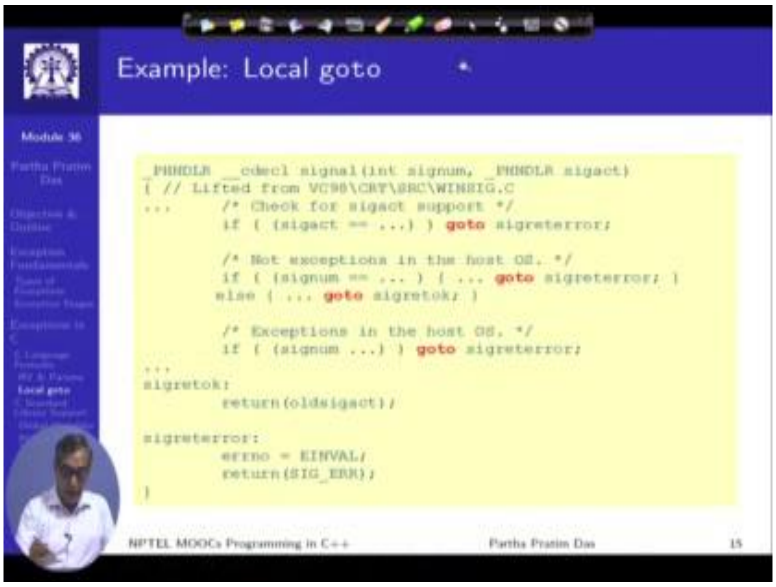
Example: Local goto

```
_PHNDLR _cdecl signal(int signum, _PHNDLR sigact)
{ // Lifted from VC98\CRT\SRC\WINSIG.C
...     /* Check for sigact support */
        if ( (sigact == ...) ) goto sigreterror;

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

        /* Exceptions in the host OS. */
        if ( (signum ...) ) goto sigreterror;
...
sigretok:
        return(oldsigact);

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

NPTEL MOOCs Programming in C++ Partha Pratim Das 14

For example, I have this code which, please do not bother about what the code does, but just look at fact that there are two labels used here. So, these two labels are designative of the successful functioning of the code and error functioning of the code. So, in different parts of the code, wherever you have a error situation happening say like this, you basically jump to this label; whereas, if you have at a successful completion then you jump to this. What helps you there is in case of error possibly you will have lot of code to take care of lot of things to may be certain objects which need to be deleted which need to be destructed and so on, all these you can take care of at one place you do not need to copy that code it multiple places.

(Refer Slide Time: 13:30)



Example: Local goto

Module 36

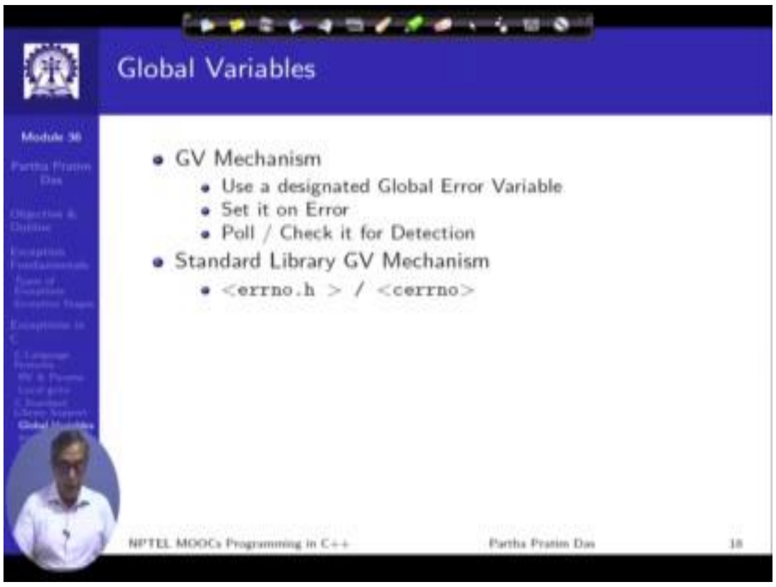
Partha Pratim Das

```
_PHNDLR __cdecl signal(int signum, _PHNDLR sigact)
{ // Lifted from VC98\CRT\SRC\WINSIG.C
...      /* Check for sigact support */
         if ( (sigact == ...) ) goto sigreterror;

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

         /* Exceptions in the host OS. */
         if ( (signum ...) ) goto sigreterror;
...
sigretok:
         return(oldsigact);

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

NPTEL MOOCs Programming in C++ Partha Pratim Das 15

So, this is how it would look like that you have this code and in error situations in all these it comes to this level; whereas if you have successful completion it comes to the sigretok label and it returns from there. So, this is just one convenient way to take care of errors.

(Refer Slide Time: 13:48)



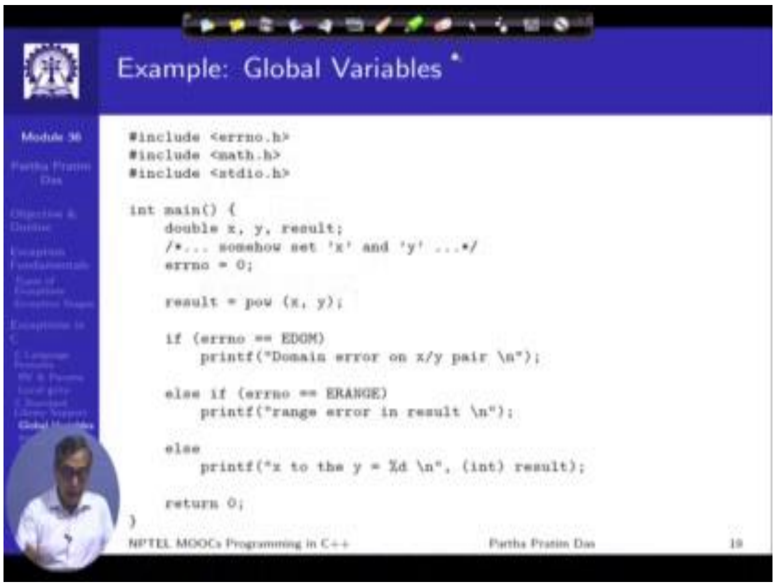
Global Variables

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

NPTEL MOOCs Programming in C++ Partha Pratim Das 18

Another which is a known as global variables. So, in many cases since a c has the functions as the global and there is no other scoping at that way. So, you may unrelated functions particularly the library functions may not have way to designate how the error will happen. So, what they do is they basically set a global variable if there is an error and this is given in the standard library errno dot h which in c will become c errno as you know.

(Refer Slide Time: 14:25)



Example: Global Variables *

Module 36

Partha Pratim Das

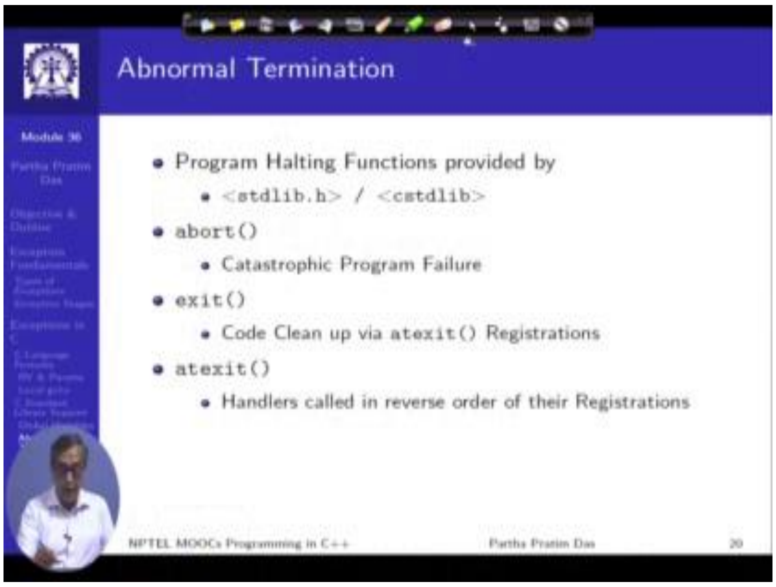
```
#include <errno.h>
#include <math.h>
#include <stdio.h>

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);

    return 0;
}
```

NPTEL MOOCs Programming in C++ Partha Pratim Das 19

So, in this for example, here we are trying to do a power, this is a pow is a library function which rises x to the power y and it might become just too large. So, if it does become too large, then it sets a variable error no. This error no you can see that it is declare here because this is declared as a global. So, you start by setting it to 0, which is basically clearing it out saying that there is no error. And then if this function gets into some range error x to the power y may be just too large to represent then error no is set to erange which is a manifest constant defined again in the library which tells you that this condition as happened.

If it is a domain error that is if is something on which you cannot do a cannot rise the power x to the power y then it errorno will become edom. So, again you are responsibility, the basic create and report is within this library function pow, and as an application programmer, it is your responsibility to detect that and you can see that ah code which otherwise would have been just this much has become so much has become, so that you can actually take care of the error situation. But using global variables like this, this header is a very common way to handle errors in C.

(Refer Slide Time: 15:43)



Abnormal Termination

- 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

NPTEL MOOCs Programming in C++ Partha Pratim Das 20

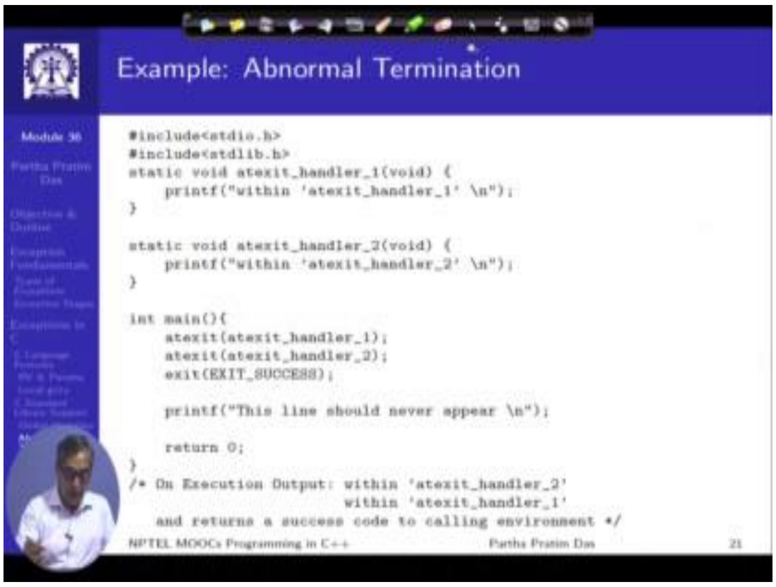
Several errors particularly mean that we cannot proceed further than that. So, if that error has happened then we need to actually terminate the program. Now, there are two kinds of termination that typically C supports; the major part is abnormal termination, which is for this several functions have provided in the C standard library. One is abort where which you can call at any point in a C program and hence in a C++ program. So, this is known as a catastrophic program failure. So, what abort does is that from that point exactly from that point where abort is called, it simply makes and exit from the function not only that function from the whole program all together. However, deeply it may be nested and just returns it to the environment.

In contrast, you could also use exit function, the abort and exit in terms of termination are both abnormal terminations, they have similar effects in that way. But there is a significant difference in terms of the fact that if there are say global objects which have been created initialized, there are local objects which have been initialized, and you are at a point where you want to basically terminate an exit. If you abort then it is catastrophic, so you do not call the destructors of these objects.

So, these objects you basically exit without destructing these objects which could otherwise be very dangerous. For example, a one of the global objects are created could be a single ton which is holding a say a lock to a global database. And if you do not call the destructor which is suppose to release that lock it means that the when your program terminates the lock is still held by your program, and it is very difficult to unlock that in future.

In contrast, exit also has a similar behavior, but it does a code clean up, it basically calls all the destructors which have been registered so far in the reverse order of their construction and calls them one after the other, so it does clean up. And for doing this clean up it uses another library function called atexit; atexit is basically a handler common handler for all these errors. So, what atexit does it, it basically registers different destructors so as function pointers and when you exit from the program it will call them in the reverse order of their actual creation.

(Refer Slide Time: 18:18)



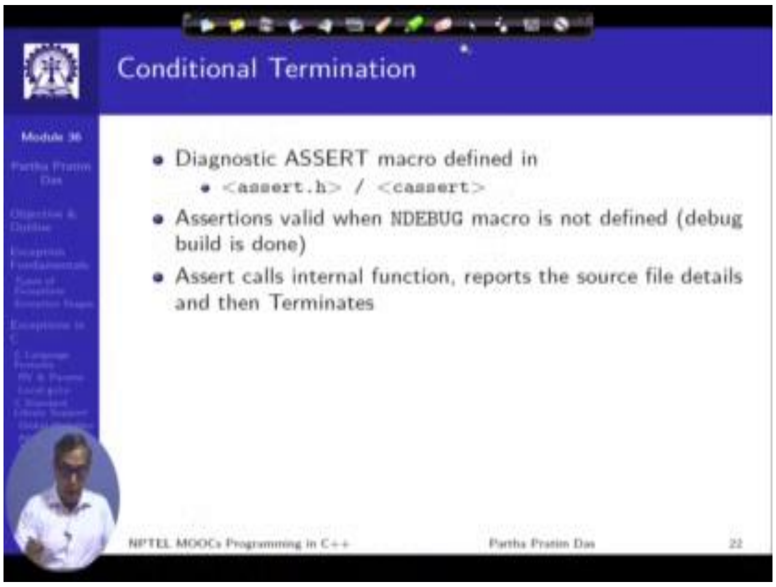
Example: Abnormal Termination

```
#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;
}
/* On Execution Output: within 'atexit_handler_2'
                        within 'atexit_handler_1'
   and returns a success code to calling environment */
```

NPTEL MOOCs Programming in C++ Partha Pratim Das 21

So, you could try out this code, which show you that you have two handlers defined here. Do not worry about what this handlers do, they practically do nothing and in the main you could register them by with the atexit, so that if you exit from the program here you are doing you are basically doing successful exit. If you do any exit from the program then it does not immediately exit, it will first call the last register function, which is atexit handler 2 then it will call the earlier register function atexit handler 1 and only after that it will exit. So, this mechanism can be used not only to clean up the just a clean up the objects by calling their destructor, but it can also be used to perform any clean up task that you may want and you can register appropriate a handler for that with atexit and then exit with success or with failure whatever values one. So, abnormal termination is a major feature which at least allows you to stop a program crash of course, if you are not using abort, abort is basically equivalent to a crash in the program.

(Refer Slide Time: 19:33)



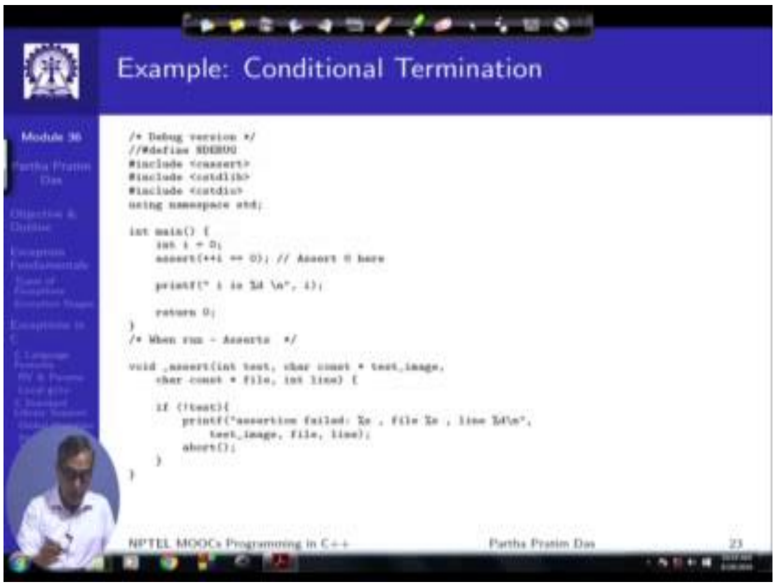
Conditional Termination

- 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

NPTEL MOOCs Programming in C++ Partha Pratim Das 22

So, another that you can use which is called a conditional termination, this is basically a debug time feature, there is a library called assert which has a assert macro, which you can provide to assert a certain condition in terms of the program. So, I will just show you an example.

(Refer Slide Time: 19:51)



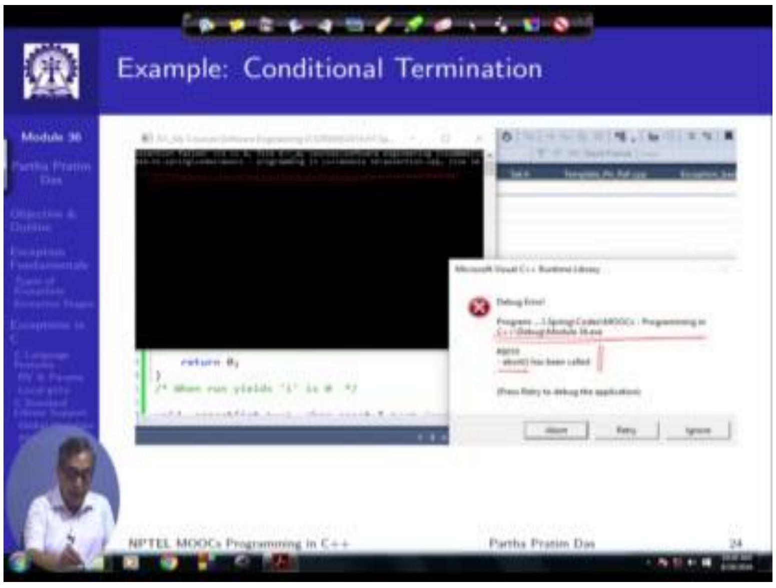
Example: Conditional Termination

```
/* Debug version */
//#define NDEBUG
#include <cassert>
#include <cstdlib>
#include <cstdio>
using namespace std;

int main() {
    int i = 0;
    assert(++i == 0); // Assert 0 here

    printf(" i is %d \n", i);

    return 0;
}
/* When run - Asserts  */

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();
    }
}
```

NPTEL MOOCs Programming in C++ Partha Pratim Das 23

So, what you are doing here is int is 0; and then you are asserting that plus plus i is equal to 0. We are saying that i must be asserting this means this condition must be true. So, if this condition is true then it does not do anything; if this condition is false then it raises an exception then it will the throw and a certain kind of window out and tell you that this has happened. So, if you actually run this program then I will show what happens.

(Refer Slide Time: 20:24)



Example: Conditional Termination

```
    return 0;
}
/* When run yields 'i' is 0  */
```

Microsoft Visual C++ Runtime Library

Debug Error!

Program: ...\Spring\Codes\MOOCs - Programming in C++\Debug\Module 36.exe

abort() has been called

(Press Retry to debug the application)

Abort Retry Ignore

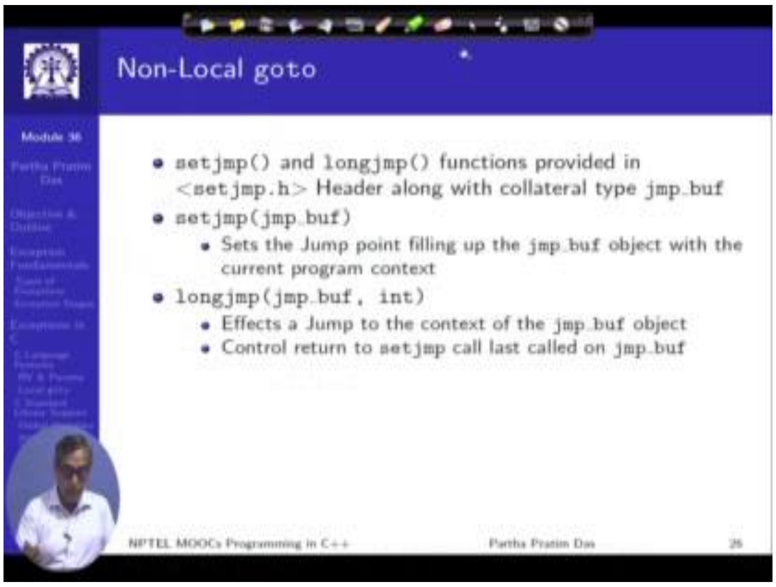
NPTEL MOOCs Programming in C++ Partha Pratim Das 24

If we run this program it will show you something like this that it says that at this while running this program here, it is not very visible, but you can actually see what is the program source and in which line it has happened. It says that your assert has happened and result of assert is actually calling abort, but this helps you to check different conditions at run time in a program, and check out, if you let me just go back to this. So, it helps you to check out if this condition where satisfied, here obliviously this is a stupid code. So, this condition is not supposed to be satisfied.

Now, the advantage of assert is certainly if you put to many of them in the program then at the run time there will be lot of places where your checks will keep on happening and that will become a detrimental to the performance of the program. So, what assert gives you is there is a manifest constant and debug. In some compiler, it is called a in debug; in some other compiler, it will be called something else. So, if that is this means that you are not debugging. So, if you put that on then this at assert feature would not be there. So, if you build a debug version which I did, I have actually commented out this. So, I am saying that it is indeed debug version. So, assert must assert.

But if I put this on, I will just show you, if I put this on, I have now put this on I making a release version where I do not want this assert would be there which will fail and all that, but it is just that it will not assert and report me anything. Or in other words, basically this assert at the compile time will get removed in the build code the build code will actually not have that assert because this is done based on the n debug. So, this is another way that you can do a conditional termination and particularly very useful while you are debugging your program.

(Refer Slide Time: 22:28)



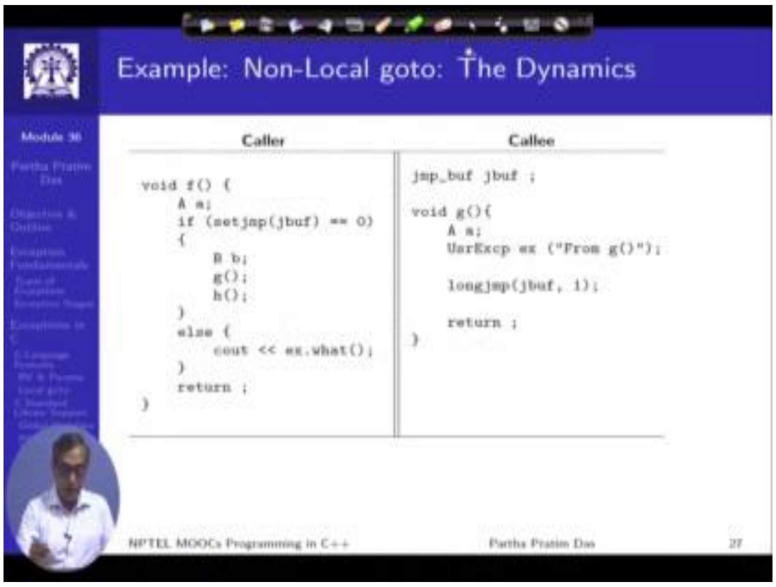
Non-Local goto

- 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

NPTEL MOOCs Programming in C++ Partha Pratim Das 26

Last feature which may not many of you may be very familiar with this is called a non-local goto. For which you have two functions provided in a standard library header called s e t j m p set jump dot h, which has two functions set jump and long jump. And it takes a jump buffer as a parameter. So, what you basically do is ah certainly the local goto’s are are a good mechanism, but certainly you cannot have vocal goto's across functions because as you know that goto’s are always limited to one function scope. So, non-local goto's are set jump, long jump give you a mechanism by which you could have jumps across functions.

(Refer Slide Time: 23:16)



Example: Non-Local goto: The Dynamics

Caller	Callee

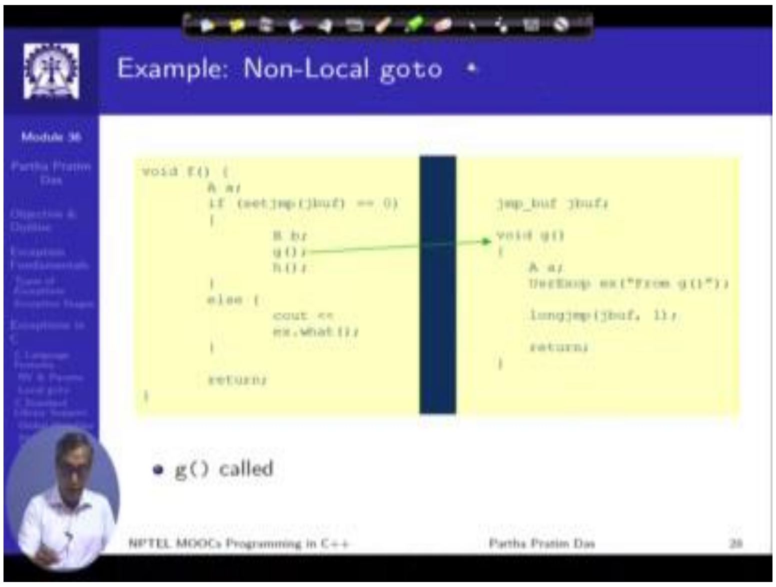
`void f() {`
`    A a;`
`    if (setjmp(jbuf) == 0)`
`    {`
`        B b;`
`        g();`
`        h();`
`    }`
`    else {`
`        cout << ex.what();`
`    }`
`    return ;`
`}`
	`jmp_buf jbuf ;`
	
	`void g(){`
	`    A a;`
	`    UsrExcp ex ("From g()");`
	
	`    longjmp(jbuf, 1);`
	
	`    return ;`
	`}`

NPTEL MOOCs Programming in C++ Partha Pratim Das 27

So, this is how you write it suppose f is a function, and g is another function, and f basically calls g. So, what you do is you define a buffer jump buffer say j buff, and in the program in the function being called you could give a long jump with the buffer with certain integer code. So, what will happen is in the caller, if you have called set jump then when the function control returns, it will return either with this value if you have executed this long jump which is an error condition possibly or it will return with 0. So, if returns with a 0, you know that this is a gone in without an error; if it returns with a 1, you know that at this point you had the error.

Now the significance of this number 1 is certainly you could have multiple of them in your called function. I can have a j but at a different point with 2. So, if I fail from that point then when I come back to the caller with this set jump check, I will actually find this value 2. So, this is this some point is called 1, some point is being called 2, some point may be called 3 and default is 0. So, if I have a normal termination, set jump will come back with 0. So, I will be able to just continue with normal execution, but if I come out with any of these error points then set jump will have an appropriate value of whatever you have long jumped with, and based on that you could write the else conditions. For example, here I just show with 1, so I have a 1, so if set jump comes here, then the normal condition it will have a 0; if it is executed long jump, it will come with 1, so the else will get satisfied where you can find out what situation is.

(Refer Slide Time: 25:19)



Example: Non-Local goto

```
void f() {
        A a;
        if (setjmp(jbuf) == 0)
        {
                B b;
                g();
                h();
        }
        else {
                cout <<
                ex.what();
        }

        return;
}
```

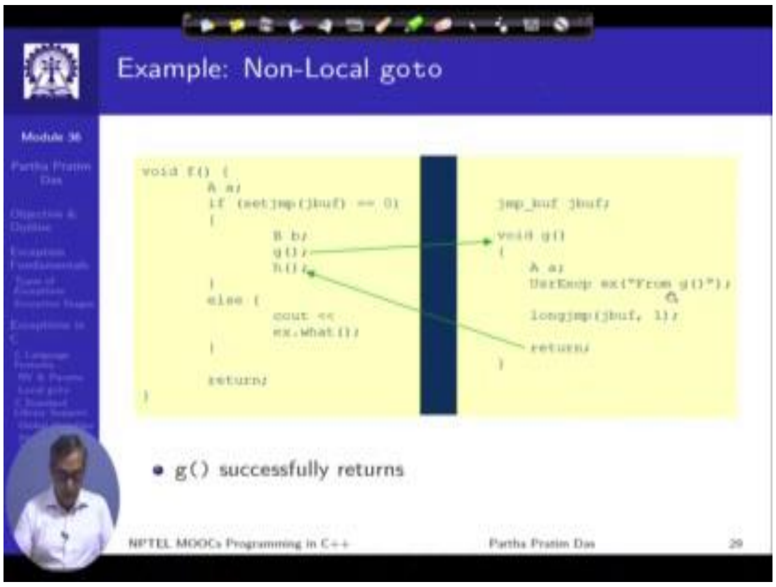
```
jmp_buf jbuf;

void g()
{
        A a;
        UsrExcp ex("From g()");

        longjmp(jbuf, 1);

        return;
}
```

- g() called

NPTEL MOOCs Programming in C++ Partha Pratim Das 29

So, if you just look it in terms of dynamics then this is what the g is getting called.

(Refer Slide Time: 25:25)



Example: Non-Local goto

```
void f() {
        A a;
        if (setjmp(jbuf) == 0)
        {
                B b;
                g();
                h();
        }
        else {
                cout <<
                ex.what();
        }

        return;
}
```

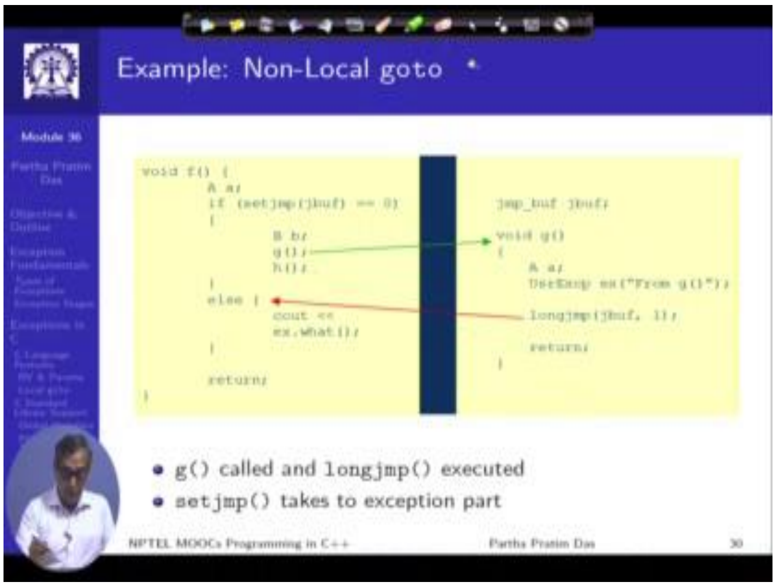
```
jmp_buf jbuf;

void g()
{
        A a;
        UsrExcp ex("From g()");

        longjmp(jbuf, 1);

        return;
}
```

- g() successfully returns

NPTEL MOOCs Programming in C++ Partha Pratim Das 29

So, in normal situation g successfully completed, this did not get executed this was not executed. So, you return you come back here, and continue you come back to h because that is a statement immediately following g.

(Refer Slide Time: 25:41)



Example: Non-Local goto

```
void f() {
        A a;
        if (setjmp(jbuf) == 0)
        {
                B b;
                g();
                h();
        }
        else {
                cout <<
                ex.what();
        }

        return;
}
```

```
jmp_buf jbuf;

void g()
{
    A a;
    UsrExcp ex("From g()");

    longjmp(jbuf, 1);

    return;
}
```

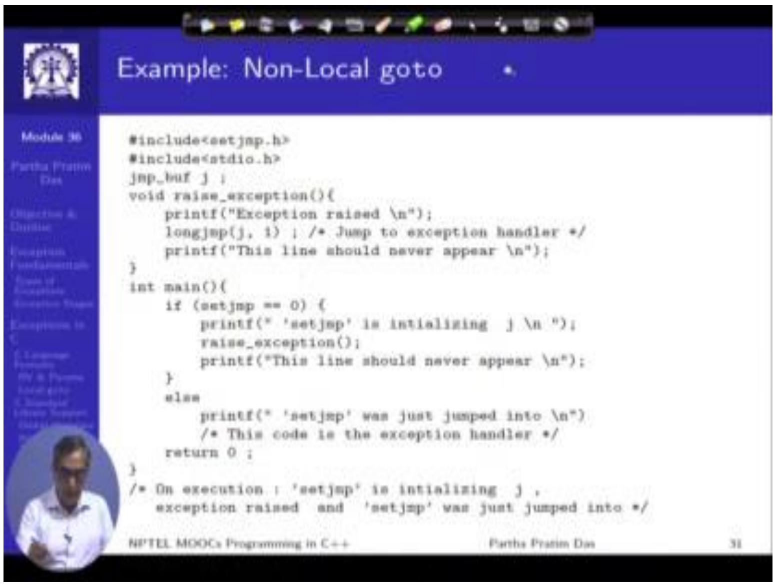
- g() called and longjmp() executed
- setjmp() takes to exception part

NPTEL MOOCs Programming in C++ Partha Pratim Das 30

But instead if you called g, and you come across this that is this under some condition that this error situation has happened in g, you do a long jump 1, then also you come back. As soon as you do long jump, you actually control actually comes back it does not wait to go up to the return, because this is some kind of error that you have come across. It does come back, but it does not come back to h because you have not completed g. It comes back to check what the set jump value is set jump value is 1 here. So, this condition has failed, so it comes back to else.

Here, you could have if set jump j buf equal to equal to 1, else something. So, it depending on that you can have a switch in the calling function f to decide which particular point g has failed from. So, this is the less known mechanism and we will see that of course, this is not very clean mechanism this is we will need remember how many points of function can come back from and obviously, everything will have to be pre decided pre designed and so on.

(Refer Slide Time: 26:43)



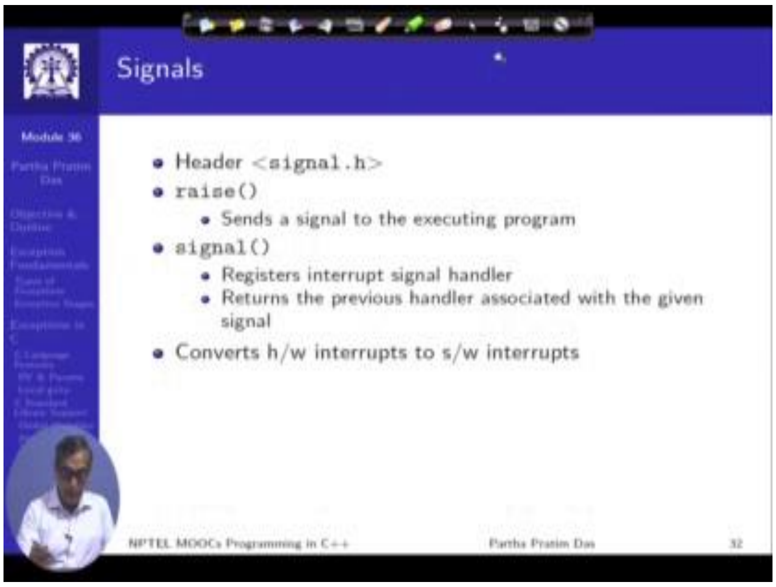
Example: Non-Local goto

```
#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 == 0) {
        printf(" 'setjmp' is intializing  j \n ");
        raise_exception();
        printf("This line should never appear \n");
    }
    else
        printf(" 'setjmp' was just jumped into \n")
        /* This code is the exception handler */
    return 0 ;
}
/* On execution : 'setjmp' is intializing  j ,
   exception raised  and  'setjmp' was just jumped into */
```

NPTEL MOOCs Programming in C++ Partha Pratim Das 31

This here I have given a code which you can just copy paste and try out this behavior of set jump long jump.

(Refer Slide Time: 26:52)



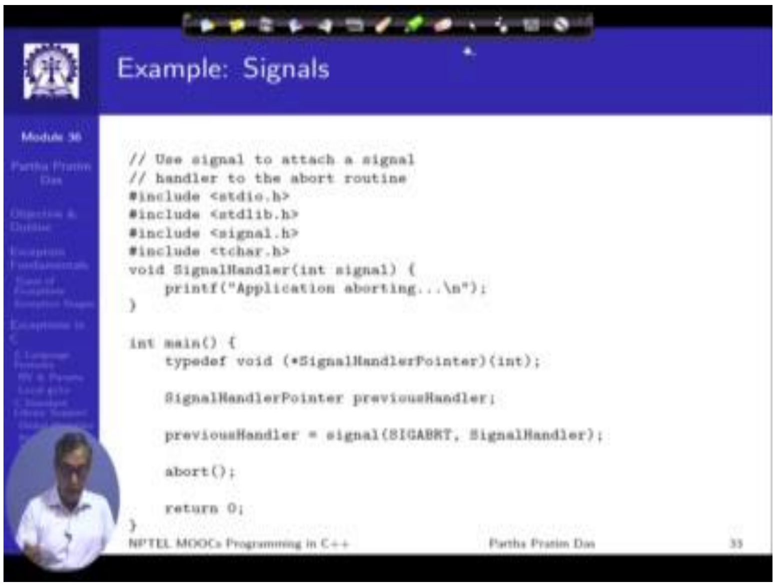
Signals

- 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

NPTEL MOOCs Programming in C++ Partha Pratim Das 32

Besides that, you have signals which people who have done some programming in terms of operating system will know there is mechanism to send signal. You can send a signal this is in signal dot h and there is a associated handler, that is your function pointer which you can put.

(Refer Slide Time: 27:09)



Example: Signals

Module 36

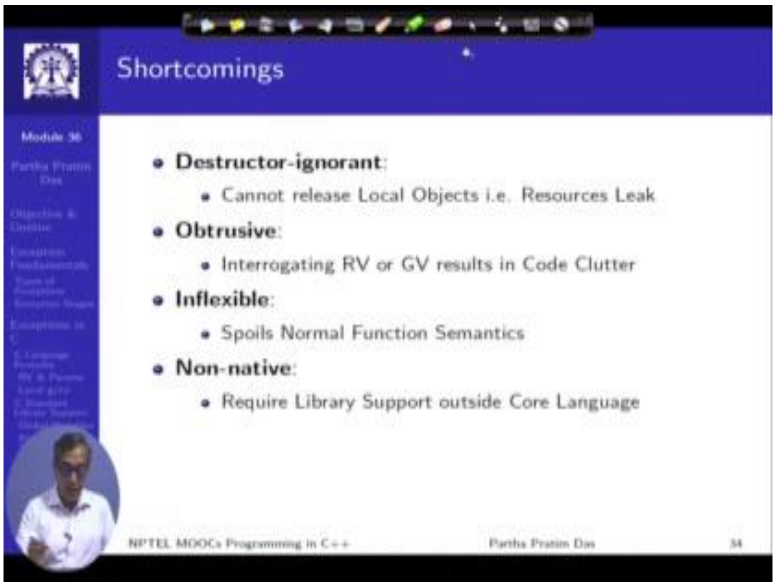
Partha Pratim Das

```
// Use signal to attach a signal
// handler to the abort routine
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <tchar.h>
void SignalHandler(int signal) {
    printf("Application aborting...\n");
}

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

    SignalHandlerPointer previousHandler;

    previousHandler = signal(SIGABRT, SignalHandler);

    abort();

    return 0;
}
```

NPTEL MOOCs Programming in C++ Partha Pratim Das 33

So, signal handlers are of this kind. So, you can define a handler and you can give different signals. For with different signals you can associate different handler, so which says that. If I have SIGABRT then this particular signal handler which is this one will get execute, will get invoked so that is a basic signaling mechanism and after that you may decide to abort. So, that is another mechanism available in C.

(Refer Slide Time: 27:40)



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

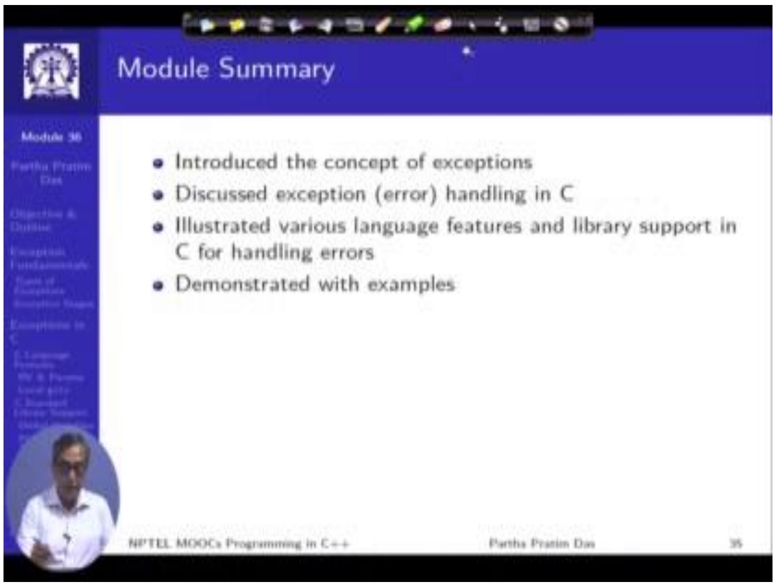
NPTEL MOOCs Programming in C++ Partha Pratim Das 34

So, we have seen that C has provided I mean because C did not take care of the errors situation the exception situation in the language designed, it come mostly as a post after thought. So, there is several different kinds of mechanism was plotted then through the standard libraries and variety of them a mixture of that, but none of them is gives you a clean solution, and these are some of the short comings of error handling in C.

It is destructor ignorant that is most of these will not when you exist, when you come out of a function already scope based on exit on a terminate the program, the destructor of the currently residing objects in different local scopes will not get destructed, so that is a that is a major resource leak issue that we have. It is obtrusive in interrogating the return value or global result results in a certainly a lot of code clutter is inflexible, because it is spoils the normal function semantics as we saw in push the normal semantics is not return anything. But just to take care of the error, we had to put something and this situation will get really, for example, how do you handle error in top, for example, you remember the top.

So, top is top is supposed to say it is an integer stack then top is suppose to return you an int. So, if this is the function signature, how would you return the error because you do not have error value. So, either use a global value which is a clutter which cannot happen if you are in a recursive call. Otherwise, you pass in a parameter through which you will return that error value which is really inflexible, it changes the basic signature of the methods that we have. And it is non-native in the way that the language did not take cognizance of the error situations and is not a part of the core. So, these are some of the major difficulties of error handling in C. I just wanted to take you through this in spite of the fact that I am sure all of you have used some or all of these methods of error handling at different stages, but I just wanted to highlight that all of these really does not solves the core problem of having extreme situations having error situations.

(Refer Slide Time: 29:54)



Module Summary

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

NPTEL MOOCs Programming in C++ Partha Pratim Das 35

To summarize, we have introduced the concept of basic concept of exceptions, the types and stages and discuss the error handling in C. We have illustrated various language features and not really many of them, and the library support that exist in C for this, and we demonstrate it with examples. In the next module, we will get into the core of exceptions in C++.