5.3 Exception Handling

Unit 5: File Handling and Exception Handling

Object Oriented Programming with C++

5.3 Exception Handling
Unit 5: File Handling and Exception Handling
Object Oriented Programming with C++

Lecture 39: Exception Handling in C++

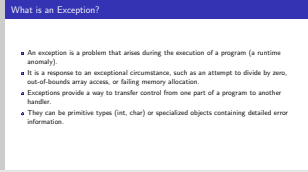
- Unit 5: File Handling and Exception Handling
- Topic 5.3: Exception Handling Mechanism
- Keywords: try, throw, catch, Multiple Catches

Lecture 39: Exception Handling in C++
- Unit 5: File Handling and Exception Handling
- Topic 5.3: Exception Handling Mechanism
- Keywords: try, throw, catch, Multiple Catches

Welcome to Lecture 39. Today, we are transitioning from file handling into another critical aspect of robust programming: Exception Handling. When dealing with files, you likely noticed that things can go wrong—files might not exist, or permissions might be denied. How do we handle such unexpected events gracefully without crashing the program? That's what C++ exception handling is all about.

What is an Exception?

- An exception is a problem that arises during the execution of a program (a runtime anomaly).
- It is a response to an exceptional circumstance, such as an attempt to divide by zero, out-of-bounds array access, or failing memory allocation.
- Exceptions provide a way to transfer control from one part of a program to another handler.
- They can be primitive types (int, char) or specialized objects containing detailed error information.



Let's define what we mean by an 'exception'. In everyday language, an exception is something out of the ordinary. In C++, an exception is an anomalous situation or a runtime error. It interrupts the normal, sequential flow of the program. For example, if your program tries to allocate memory but the system is out of RAM, an exception is triggered. Unlike syntax errors which are caught at compile-time by the compiler, exceptions are runtime phenomena that must be handled dynamically.

Why Exception Handling?

- Traditional error handling uses return codes (e.g., returning -1) or global flags (like `errno`).
- Disadvantages of traditional methods:
- - Error checking logic heavily clutters the main business logic.
- - Return codes can be silently ignored by the programmer, leading to downstream failures.
- - Difficult to return both valid data and error codes from the same function signature.
- Advantages of C++ Exceptions:
- - Strictly separates error-handling code from the main execution path.
- - Unhandled exceptions forcefully terminate the program, preventing silent, hard-to-debug failures.

Why Exception Handling?
- Traditional error handling uses return codes (e.g., returning -1) or global flags (like errno).
- Disadvantages of traditional methods:
- - Error checking logic heavily clutters the main business logic.
- - Return codes can be silently ignored by the programmer, leading to downstream failures.
- - Difficult to return both valid data and error codes from the same function signature.
- Advantages of C++ Exceptions:
- - Strictly separates error-handling code from the main execution path.
- - Unhandled exceptions forcefully terminate the program, preventing silent, hard-to-debug failures.

Before the advent of structured exception handling, programmers relied on returning error codes. But this approach is flawed. Your code becomes a nested mess of 'if' statements checking return values. Worse, a lazy programmer might ignore the return value, leading to silent, catastrophic failures later on. Exceptions solve this beautifully. If an exception is thrown and you don't handle it, the program crashes immediately, forcing you to address the bug. It also keeps your 'happy path' logic clean and readable.

The C++ Exception Handling Mechanism

- C++ exception handling is built entirely upon three reserved keywords:
- 1. try: Identifies a block of code for which particular exceptions will be activated and monitored.
- 2. throw: A program throws an exception when a problem is detected. This transfers control to the handler.
- 3. catch: Defines a block of code (an exception handler) that receives the thrown exception and dictates how to recover.

The C++ Exception Handling Mechanism
- C++ exception handling is built entirely upon three reserved keywords:
- 1. try: Identifies a block of code for which particular exceptions will be activated and monitored.
- 2. throw: A program throws an exception when a problem is detected. This transfers control to the handler.
- 3. catch: Defines a block of code (an exception handler) that receives the thrown exception and dictates how to recover.

The core of C++ exception handling revolves around three keywords: try, throw, and catch. Think of it conceptually like a baseball game. You 'try' to hit the ball. If the pitcher makes a mistake (an error), he 'throws' a wild pitch. The catcher then has to 'catch' it to stabilize the play. In our code, the 'try' block monitors for runtime errors. If a fatal error occurs, we 'throw' an exception object. Finally, a matching 'catch' block receives that object and executes recovery logic.

The 'try' Block

- Syntax:
- try {
- // Protected code that might generate an exception
- // Functions called from here are also monitored
- }
- Purpose: To enclose and monitor statements that may cause exceptions.
- Crucial Rule: A try block cannot stand alone. It must be immediately followed by at least one catch block.

The 'try' Block
- Syntax:
- try {
- // Protected code that might generate an exception
- // Functions called from here are also monitored
- }
- Purpose: To enclose and monitor statements that may cause exceptions.
- Crucial Rule: A try block cannot stand alone. It must be immediately followed by at least one catch block.

Let's examine the 'try' block. You wrap your risky code—code that interacts with hardware, allocates memory, or does complex math like division—inside a try block. If everything executes smoothly, the program simply skips the subsequent catch blocks and continues. However, if something goes wrong inside this block, and a 'throw' occurs, the execution instantly jumps out of the try block, abandoning the rest of the code inside it, and begins searching for a catch block.

The 'throw' Statement

- Syntax: throw exception_value;
- The exception_value can be of any valid C++ data type (int, string literal, or a user-defined object).
- When 'throw' is executed, the current function's control flow stops immediately.
- Example 1: throw 404; // Throwing an integer code
- Example 2: throw "Database connection failed!"; // Throwing a string

The 'throw' Statement
- Syntax: throw exception_value;
- The exception_value can be of any valid C++ data type (int, string literal, or a user-defined object).
- When 'throw' is executed, the current function's control flow stops immediately.
- Example 1: throw 404; // Throwing an integer code
- Example 2: throw "Database connection failed!"; // Throwing a string

When an error condition is verified within a try block, or deeply nested within a function called by a try block, you use the 'throw' keyword. You can throw essentially anything: an integer error code, a descriptive string, or ideally, a specialized exception object. The exact moment 'throw' executes, the current function halts. Control is immediately surrendered to the C++ runtime environment, which begins the search for an appropriate 'catch' mechanism.

The 'catch' Block

- Syntax:
- catch (datatype parameter_name) {
- // Code to handle and recover from the exception
- }
- The datatype specified must exactly match the type of the value that was thrown.
- The parameter_name is optional if you only care about the type, but necessary if you want to inspect the thrown value (e.g., read the error message).
- Catch blocks must be placed immediately after the try block.

The 'catch' Block
- Syntax:
- catch (datatype parameter_name) {
- // Code to handle and recover from the exception
- }
- The datatype specified must exactly match the type of the value that was thrown.
- The parameter_name is optional if you only care about the type, but necessary if you want to inspect the thrown value (e.g., read the error message).
- Catch blocks must be placed immediately after the try block.

The 'catch' block acts as your safety net. It looks somewhat similar to a standard function definition. It explicitly specifies what type of exception it is designed to handle. If you threw a double, you need a catch block that accepts a double. Inside the catch block, you write the recovery logic—this could be logging the error to a file, prompting the user for new input, or assigning safe default values to variables.

Example: Basic Division by Zero

```
double divide(double a, double b) {
if (b == 0) {
throw "Division by zero error!";
}
return a / b;
}
int main() {
try {
cout ¡¡ divide(10, 0); // Triggers the exception
} catch (const char* msg) {
cerr ¡¡ "Caught exception: " ¡¡ msg ¡¡ endl;
}
return 0;
}
```


Example: Basic Division by Zero
double divide(double a, double b) {
if (b == 0) {
throw "Division by zero error!";
}
return a / b;
}
int main() {
try {
cout ¡¡ divide(10, 0); // Triggers the exception
} catch (const char* msg) {
cerr ¡¡ "Caught exception: " ¡¡ msg ¡¡ endl;
}
return 0;
}

Here is a classic, practical example. We have a divide function. Before attempting division, we proactively check if the denominator is zero. If it is, we throw a string literal. In main(), we wrap the call to divide() inside a try block. Because we pass 0, the function throws the string. Execution leaps out of the try block, skipping the cout statement, and lands in the catch block that expects a 'const char*'. The error message is printed gracefully, and the program exits normally.

Control Flow in Exception Handling

- 1. Execution enters the try block.
- 2. If no exception occurs, the try block finishes normally, and all associated catch blocks are skipped entirely.
- 3. If an exception IS thrown:
- - The remaining code within the try block is bypassed.
- - The runtime sequentially searches the catch blocks for a type match.
- - The matching catch block executes.
- - Normal execution resumes directly after the LAST catch block.
- 4. Critical: If no matching catch block is found anywhere in the call stack, the program invokes std::terminate().

Control Flow in Exception Handling
- 1. Execution enters the try block.
- 2. If no exception occurs, the try block finishes normally, and all associated catch blocks are skipped entirely.
- 3. If an exception IS thrown:
- - The remaining code within the try block is bypassed.
- - The runtime sequentially searches the catch blocks for a type match.
- - The matching catch block executes.
- - Normal execution resumes directly after the LAST catch block.
- 4. Critical: If no matching catch block is found anywhere in the call stack, the program invokes std::terminate().

Understanding the exact control flow is paramount. Throwing an exception behaves like a non-local goto statement. If an exception triggers on line 2 of a 10-line try block, lines 3 through 10 are completely skipped. The system immediately jumps to evaluate the catch blocks. If it finds a match, it runs the handler and then continues execution below the whole try-catch construct. If it searches all the way up through every calling function and finds no handler, the C++ runtime forcefully aborts the application.

Multiple 'catch' Blocks

- A single try block can be protected by multiple catch blocks sequentially.
- This allows you to handle various different failure scenarios in highly specific ways.
- Syntax:
- try { / *risky code* / }
- catch (int e) { / *handle int errors* / }
- catch (char e) { / *handle char errors* / }
- catch (double e) { / *handle double errors* / }
- The compiler checks the catch blocks strictly in the top-to-bottom order they appear.
- Only the FIRST matching catch block is executed.

Multiple 'catch' Blocks
- A single try block can be protected by multiple catch blocks sequentially.
- This allows you to handle various different failure scenarios in highly specific ways.
- Syntax:
- try { / *risky code* / }
- catch (int e) { / *handle int errors* / }
- catch (char e) { / *handle char errors* / }
- catch (double e) { / *handle double errors* / }
- The compiler checks the catch blocks strictly in the top-to-bottom order they appear.
- Only the FIRST matching catch block is executed.

In real-world applications, a complex block of code might fail for several reasons. A function might throw an integer to indicate a file missing error, and a string to indicate a network timeout. You can chain multiple catch blocks together after a single try block. When an exception occurs, the C++ runtime evaluates these catch blocks from top to bottom. The first one that precisely matches the type of the thrown exception executes. Once that single block finishes, the rest are bypassed.

Example: Multiple Catches in Action

- try {
- int errorCode = 2;
- if (errorCode == 1) throw 404; // throws int
- if (errorCode == 2) throw 'E'; // throws char
- if (errorCode == 3) throw 3.14; // throws double
- }
- catch (int i) { cout ¡¡ "Caught integer: " ¡¡ i; }
- catch (char c) { cout ¡¡ "Caught character: " ¡¡ c; }
- catch (double d) { cout ¡¡ "Caught double: " ¡¡ d; }

Example: Multiple Catches in Action
try {
int errorCode = 2;
if (errorCode == 1) throw 404; // throws int
if (errorCode == 2) throw 'E'; // throws char
if (errorCode == 3) throw 3.14; // throws double
}
catch (int i) { cout ¡¡ "Caught integer: " ¡¡ i; }
catch (char c) { cout ¡¡ "Caught character: " ¡¡ c; }
catch (double d) { cout ¡¡ "Caught double: " ¡¡ d; }

Look at this illustrative example. Depending on the value of the 'errorCode' variable, we throw entirely different data types. Because errorCode is set to 2, the program executes `throw 'E';`. The runtime evaluates the catch blocks. It bypasses the 'int' catch block because the types don't match, and hits the 'char' catch block. It prints 'Caught character: E'. The 'double' catch block is ignored.

Catching All Exceptions: catch(...)

- Sometimes, you need a default fallback handler that catches absolutely ANY exception, regardless of its type.
- Achieved using the ellipsis (`...`) in the catch parameter list.
- Syntax:
- catch (...) {
- cout ¡¡ "An unknown exception occurred!";
- }
- Crucial Rule: The `catch(...)` block must ALWAYS be placed as the LAST catch block in a sequence.
- If placed earlier, it acts as a black hole, intercepting exceptions meant for specific handlers below it.

Catching All Exceptions: catch(...)
- Sometimes, you need a default fallback handler that catches absolutely ANY exception, regardless of its type.
- Achieved using the ellipsis (...) in the catch parameter list.
- Syntax:
- catch (...) {
- cout ¡¡ "An unknown exception occurred!";
- }
- Crucial Rule: The catch(...) block must ALWAYS be placed as the LAST catch block in a sequence.
- If placed earlier, it acts as a black hole, intercepting exceptions meant for specific handlers below it.

What if you are utilizing a third-party library that throws exceptions, but you aren't certain what types it might throw? Or what if you want a final, absolute safety net in your main loop to prevent a hard crash? C++ provides the catch-all handler using three dots. This intercepts any exception type. However, because it's a wildcard match, it must reside at the very bottom of your catch block list. If you place it at the top, the compiler will likely issue a warning, as specific handlers below it become unreachable code.

The Magic of Stack Unwinding

- When an exception is thrown, the runtime system searches up the call stack for a valid handler.
- As it abruptly exits blocks and functions looking for a catch block, it destroys all local, automatic objects created in those scopes.
- This automated cleanup process is called 'Stack Unwinding'.
- Destructors of local objects are guaranteed to be called automatically.
- This is the foundation of RAII (Resource Acquisition Is Initialization), preventing memory leaks when errors occur.

The Magic of Stack Unwinding
- When an exception is thrown, the runtime system searches up the call stack for a valid handler.
- As it abruptly exits blocks and functions looking for a catch block, it destroys all local, automatic objects created in those scopes.
- This automated cleanup process is called 'Stack Unwinding'.
- Destructors of local objects are guaranteed to be called automatically.
- This is the foundation of RAII (Resource Acquisition Is Initialization), preventing memory leaks when errors occur.

This is arguably the most powerful feature of C++ exceptions, known as Stack Unwinding. When an exception is thrown deep inside a chain of nested function calls, C++ doesn't just blindly jump to the catch block. It meticulously walks backward up the call stack. As it violently exits each function, it automatically invokes the destructors for any local objects that were created on the stack. This is vital. It means if you opened a file, acquired a mutex, or allocated smart pointers, throwing an exception won't cause a resource leak. The destructors clean things up on the way out automatically.

Rethrowing an Exception

- A particular catch block might not have enough context to completely resolve an exception.
- It can perform partial handling (such as logging the error) and then pass the exception further up the call stack.
- Use the `throw;` keyword (with no operand) inside a catch block to rethrow.
- Syntax:
- catch (int e) {
- logErrorToFile(e); // Partial handling
- throw; // Rethrow the exact same exception to the caller
- }

Rethrowing an Exception
- A particular catch block might not have enough context to completely resolve an exception.
- It can perform partial handling (such as logging the error) and then pass the exception further up the call stack.
- Use the throw; keyword (with no operand) inside a catch block to rethrow.
- Syntax:
- catch (int e) {
- logErrorToFile(e); // Partial handling
- throw; // Rethrow the exact same exception to the caller
- }

Sometimes a function catches an exception, logs it to a debugging file, but realizes it lacks the authority or context to actually fix the program state. In this scenario, it can 'rethrow' the exception to a higher-level module. To accomplish this, you simply write the keyword 'throw;' followed by a semicolon, inside a catch block. This takes the exact exception object you just caught and throws it again, allowing an outer try-catch block to handle the final recovery or user notification.

Standard Exception Classes

- While throwing primitives (ints, chars) works, professional C++ relies on the standard exception hierarchy in `<stdexcept>`.
- All standard exceptions derive from a common base class: `std::exception`.
- Common standard exceptions:
- - `std::bad_alloc`: Thrown by the `new` operator when RAM is exhausted.
- - `std::out_of_range`: Thrown by container methods like `vector::at()`.
- - `std::runtime_error`: Generic class for errors detectable only at runtime.
- Using `catch(const std::exception& e)` can polymorphic-ally catch any standard exception.

Standard Exception Classes
- While throwing primitives (ints, chars) works, professional C++ relies on the standard exception hierarchy in <stdexcept>.
- All standard exceptions derive from a common base class: std::exception.
- Common standard exceptions:
- - std::bad_alloc: Thrown by the new operator when RAM is exhausted.
- - std::out_of_range: Thrown by container methods like vector::at().
- - std::runtime_error: Generic class for errors detectable only at runtime.
- Using catch(const std::exception& e) can polymorphic-ally catch any standard exception.

While throwing integers or strings is fine for small scripts, professional C++ codebases utilize structured exception classes. The standard library provides a rich hierarchy of exception classes in the header ¡stdexcept¿. The root of this entire hierarchy is std::exception. It features a virtual function called what() that returns a C-style string describing the error. You should strongly prefer throwing these standard exceptions, as it standardizes error handling across different libraries.

Creating Custom Exception Classes

- For complex applications, you can define proprietary exception classes by inheriting from `std::exception`.
- Override the virtual `what()` method to provide a customized error message.
- class NetworkTimeoutException : public std::exception {
- public:
- const char* what() const noexcept override {
- return "Network request timed out after 30s.";
- }
- };
- Benefits: Custom exceptions can store additional state data, like error codes, timestamps, or failed IP addresses.

Creating Custom Exception Classes
- For complex applications, you can define proprietary exception classes by inheriting from std::exception.
- Override the virtual what() method to provide a customized error message.
- class NetworkTimeoutException : public std::exception {
- public:
- const char* what() const noexcept override {
- return "Network request timed out after 30s.";
- }
- };
- Benefits: Custom exceptions can store additional state data, like error codes, timestamps, or failed IP addresses.

For large, domain-specific applications, the standard exceptions might not be descriptive enough. You can define your own exception classes by inheriting from std::exception. The primary requirement is to override the what() method to return a helpful message. The real power of custom exceptions is that they are full-fledged objects. You can add custom data members to them. For example, a FileReadException class could encapsulate the name of the file that failed and the specific OS-level error code.

Lecture Summary

- Exceptions provide a clean, structural, and robust methodology to handle runtime anomalies.
- `try` blocks enclose code that is vulnerable to generating errors.
- `throw` statements trigger exceptions, abruptly halting normal sequential execution.
- `catch` blocks act as specialized handlers, selected based on the exception's data type.
- Multiple `catch` blocks can follow a single `try`, acting as a switch statement for types.
- Stack unwinding ensures safe resource cleanup during exception propagation.
- Always order catch blocks from most specific to least specific (`catch(...)`).

Lecture Summary
- Exceptions provide a clean, structural, and robust methodology to handle runtime anomalies.
- try blocks enclose code that is vulnerable to generating errors.
- throw statements trigger exceptions, abruptly halting normal sequential execution.
- catch blocks act as specialized handlers, selected based on the exception's data type.
- Multiple catch blocks can follow a single try, acting as a switch statement for types.
- Stack unwinding ensures safe resource cleanup during exception propagation.
- Always order catch blocks from most specific to least specific (catch(...)).

To summarize today's lecture, exception handling completely separates the 'what the program should do' from the 'what to do if things fail miserably'. It forces developers to acknowledge errors, practically eliminating silent failures. Remember the try, throw, catch triad. Be keenly aware of how multiple catch blocks are evaluated sequentially from top-to-bottom, and always remember that C++ will dutifully clean up your local objects via stack unwinding when an exception flies past.