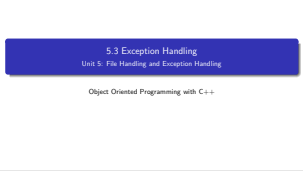
5.3 Exception Handling

Unit 5: File Handling and Exception Handling

Object Oriented Programming with C++



Lecture 43: Exception Handling in C++

- Course: Object Oriented Programming with C++
- Unit 5: File Handling and Exception Handling
- Topic 5.3: try, throw, catch; Multiple Throws and Catches

Lecture 43: Exception Handling in C++
- Course: Object Oriented Programming with C++
- Unit 5: File Handling and Exception Handling
- Topic 5.3: try, throw, catch; Multiple Throws and Catches

Welcome everyone to Lecture 43. Today we are transitioning from File Handling to the second major topic of Unit 5, which is Exception Handling. This is a crucial concept in modern C++ that allows us to build robust, fault-tolerant applications. Instead of programs crashing abruptly when something goes wrong, exception handling gives us a structured way to intercept these problems and recover gracefully.

What is an Exception?

- An exception is an abnormal condition or problem that arises during the runtime execution of a program.
- It is a synchronous event that disrupts the normal, expected flow of instructions.
- Common examples: Division by zero, out of memory, file not found, array index out of bounds.
- Exceptions provide a mechanism to transfer control from the point of error to an error-handler.

What is an Exception?
- An exception is an abnormal condition or problem that arises during the runtime execution of a program.
- It is a synchronous event that disrupts the normal, expected flow of instructions.
- Common examples: Division by zero, out of memory, file not found, array index out of bounds.
- Exceptions provide a mechanism to transfer control from the point of error to an error-handler.

Let's start by defining what an exception actually is. In simple terms, an exception is a runtime error. Unlike compile-time errors, which are usually syntax mistakes caught by the compiler, exceptions occur while the program is actually executing. For instance, if your program asks a user for a denominator and they enter 0, the CPU cannot perform division by zero. This mathematical impossibility causes an exception. If not handled, an exception will typically cause the operating system to forcefully terminate your program. Exception handling gives us a structured way to intercept these issues.

Traditional Error Handling vs. Exceptions

- Traditional Approaches:
- - Returning error codes (e.g., -1, NULL, or false).
- - Setting global flags (e.g., errno in C).
- Drawbacks of Traditional Approaches:
- - Error checking logic is tightly mixed with core business logic, making code hard to read.
- - Programmers can easily ignore or forget to check return codes.
- The Exception Handling Advantage:
- - Strictly separates error-handling code from regular code.
- - Enforces error handling: Unhandled exceptions automatically terminate the program, preventing silent failures.

Traditional Error Handling vs. Exceptions
- Traditional Approaches:
- - Returning error codes (e.g., -1, NULL, or false).
- - Setting global flags (e.g., errno in C).
- Drawbacks of Traditional Approaches:
- - Error checking logic is tightly mixed with core business logic, making code hard to read.
- - Programmers can easily ignore or forget to check return codes.
- The Exception Handling Advantage:
- - Strictly separates error-handling code from regular code.
- - Enforces error handling: Unhandled exceptions automatically terminate the program, preventing silent failures.

Before C++ introduced exceptions, developers relied heavily on traditional error handling. Functions would return special values like -1 or NULL to indicate failure. The problem with this is two-fold. First, your main logic gets cluttered with endless 'if error code == -1' statements. Second, a lazy developer might just ignore the return value, leading to unpredictable behavior later on—a silent failure. Exceptions solve this by completely separating the 'happy path' from the 'error path'. Furthermore, you cannot simply ignore an exception; if you don't handle it, the program will terminate. This forces developers to address potential errors.

The C++ Exception Handling Mechanism

- C++ exception handling is built upon three primary keywords:
- 1. try: Identifies a block of code in which exceptions might occur. It 'monitors' the code.
- 2. throw: Signals that an exception has occurred within a try block. It 'raises' the error.
- 3. catch: Defines a block of code that handles the exception. It 'intercepts' the error.

The C++ Exception Handling Mechanism
- C++ exception handling is built upon three primary keywords:
- 1. try: Identifies a block of code in which exceptions might occur. It 'monitors' the code.
- 2. throw: Signals that an exception has occurred within a try block. It 'raises' the error.
- 3. catch: Defines a block of code that handles the exception. It 'intercepts' the error.

The entire exception handling architecture in C++ revolves around three simple keywords: try, throw, and catch. Think of it like a baseball game. The 'try' block is the pitcher's mound where the action happens. If something goes wrong, the pitcher 'throws' the ball (the exception). The 'catch' block is the catcher standing behind home plate, ready to catch the specific type of ball thrown and decide what to do next.

The 'try' Block

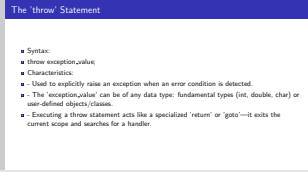
- Syntax:
- try {
- // Code that might generate an exception
- // Normal program logic
- }
- Characteristics:
- - A try block contains the logic of your program that you want to monitor for errors.
- - If an exception occurs within it, normal execution is instantly halted.
- - Control is immediately transferred to the nearest enclosing catch block.
- - A try block MUST be followed by at least one catch block.

The 'try' Block
- Syntax:
- try {
- // Code that might generate an exception
- // Normal program logic
- }
- Characteristics:
- - A try block contains the logic of your program that you want to monitor for errors.
- - If an exception occurs within it, normal execution is instantly halted.
- - Control is immediately transferred to the nearest enclosing catch block.
- - A try block MUST be followed by at least one catch block.

Let's look at the 'try' block in detail. You wrap the risky parts of your code inside a try block. This tells the compiler, 'Hey, monitor this section. If anything throws an error, be ready to catch it.' The moment an error is thrown inside a try block, the rest of the code inside that block is skipped. It's an immediate exit. Also, structurally, a try block cannot stand alone. It is a syntax error to have a try block without a subsequent catch block to handle the potential fallout.

The 'throw' Statement

- Syntax:
- throw exception_value;
- Characteristics:
- - Used to explicitly raise an exception when an error condition is detected.
- - The 'exception_value' can be of any data type: fundamental types (int, double, char) or user-defined objects/classes.
- - Executing a throw statement acts like a specialized 'return' or 'goto'—it exits the current scope and searches for a handler.



When your code detects that an impossible situation has arisen—like receiving a zero for a denominator, or failing to allocate memory—you use the 'throw' statement. You can throw almost anything in C++: an integer, a string, or a custom error object. Modern C++ heavily favors throwing objects, particularly those derived from the standard library's exception class, but syntactically, you can throw a simple integer like 'throw 404;'. Once 'throw' is executed, no further code in that try block runs.

The 'catch' Block

- Syntax:
- catch (type exception_variable) {
- // Code to handle the exception
- }
- Characteristics:
- - Acts as the exception handler. It catches the exception thrown by the try block.
- - The 'type' specified in the catch block must match the data type of the value thrown.
- - If a type match is found, the code inside the catch block executes to resolve the error or log it.

The 'catch' Block
- Syntax:
- catch (type exception_variable) {
- // Code to handle the exception
- }
- Characteristics:
- - Acts as the exception handler. It catches the exception thrown by the try block.
- - The 'type' specified in the catch block must match the data type of the value thrown.
- - If a type match is found, the code inside the catch block executes to resolve the error or log it.

The catch block is where you fix the problem, log the error, or alert the user. It immediately follows the try block. The parameter inside the catch block's parentheses dictates what kind of exception it is willing to catch. If the try block throws an 'int', a catch block expecting a 'double' or a 'string' will ignore it. The types must match. The exception_variable allows you to access the value that was thrown, so you can read the error message or error code.

Example: Division by Zero

- int a = 10, b = 0, result;
- try {
- if (b == 0) {
- throw "Division by zero error!"; // Throwing a string literal (const char*)
- }
- result = a / b;
- cout ¡¡ "Result: " ¡¡ result ¡¡ endl;
- }
- catch (const char* msg) {
- cout ¡¡ "Exception Caught: " ¡¡ msg ¡¡ endl;
- }

Example: Division by Zero
int a = 10, b = 0, result;
try {
if (b == 0) {
throw "Division by zero error!"; // Throwing a string literal (const char*)
}
result = a / b;
cout ¡¡ "Result: " ¡¡ result ¡¡ endl;
}
catch (const char* msg) {
cout ¡¡ "Exception Caught: " ¡¡ msg ¡¡ endl;
}

Here is a classic example. We are trying to divide a by b. Inside the try block, we first check if b is zero. If it is, we use the throw keyword to throw a string literal. Because we threw an exception, the actual division 'result = a / b;' never executes. The program jumps straight to the catch block that accepts a 'const char*'. It prints our error message, and the program continues running safely after the catch block, rather than crashing.

Tracing the Control Flow

- Step-by-step Execution during an Exception:
- 1. Program enters the try block and executes sequentially.
- 2. An error condition is met, and the throw statement executes.
- 3. Control immediately leaves the try block. Subsequent lines in the try block are ignored.
- 4. The runtime system searches for a catch block matching the thrown type.
- 5. The matching catch block executes.
- 6. Program execution resumes immediately AFTER the catch block (it does not return to the try block).

Tracing the Control Flow
- Step-by-step Execution during an Exception:
- 1. Program enters the try block and executes sequentially.
- 2. An error condition is met, and the throw statement executes.
- 3. Control immediately leaves the try block. Subsequent lines in the try block are ignored.
- 4. The runtime system searches for a catch block matching the thrown type.
- 5. The matching catch block executes.
- 6. Program execution resumes immediately AFTER the catch block (it does not return to the try block).

Understanding the control flow is critical. The most common misconception beginners have is that after the catch block finishes executing, the program goes back into the try block to try again. This is false. Once you are thrown out of a try block, you cannot go back in. The program resumes execution at the first line of code that appears below the very last catch block. The try block is considered permanently aborted for that specific execution pass.

Delegation: Throwing from Functions

- Exceptions do not have to be thrown directly inside the try block's local scope.
- They can be thrown from within a function that is called from inside a try block.
- If a function throws an exception but doesn't catch it locally, the exception propagates up the 'call stack'.
- This allows centralized error handling: low-level functions detect/throw errors, and high-level functions (like main) catch/handle them.

Delegation: Throwing from Functions
- Exceptions do not have to be thrown directly inside the try block's local scope.
- They can be thrown from within a function that is called from inside a try block.
- If a function throws an exception but doesn't catch it locally, the exception propagates up the 'call stack'.
- This allows centralized error handling: low-level functions detect/throw errors, and high-level functions (like main) catch/handle them.

You don't have to put your throw statements directly inside the try block. In professional software, you usually call utility functions from within a try block. If one of those utility functions throws an exception, the compiler looks for a try/catch inside that function. If it doesn't find one, it travels back up the call stack to the function that called it, looking for a try/catch there. This allows you to write one main try/catch block that handles errors originating from deep within your application's architecture.

Introduction to Multiple Catches

- A single try block can execute code that might throw several different types of exceptions.
- To handle these different scenarios appropriately, a try block can be followed by multiple catch blocks.
- Syntax:
- try { ... }
- catch (int e) { ... }
- catch (double e) { ... }
- catch (MyCustomException e) { ... }

Introduction to Multiple Catches
- A single try block can execute code that might throw several different types of exceptions.
- To handle these different scenarios appropriately, a try block can be followed by multiple catch blocks.
- Syntax:
- try { ... }
- catch (int e) { ... }
- catch (double e) { ... }
- catch (MyCustomException e) { ... }

Real-world try blocks are usually doing more than just one thing. A try block might attempt to open a file, parse an integer from it, and allocate memory. Opening a file might throw a string exception, parsing might throw an int exception, and memory allocation might throw a standard library exception. To handle all these varied possibilities, C++ allows you to chain multiple catch blocks after a single try block. The compiler will check them one by one until it finds a type match.

Example: Multiple Throws and Catches

- try {
- int choice = 2;
- if (choice == 1) throw 10; // Throws an int
- else if (choice == 2) throw 3.14; // Throws a double
- else throw 'E'; // Throws a char
- }
- catch (int x) { cout ¡¡ "Caught int: " ¡¡ x; }
- catch (double x) { cout ¡¡ "Caught double: " ¡¡ x; }
- catch (char x) { cout ¡¡ "Caught char: " ¡¡ x; }

Example: Multiple Throws and Catches
- try {
- int choice = 2;
- if (choice == 1) throw 10; // Throws an int
- else if (choice == 2) throw 3.14; // Throws a double
- else throw 'E'; // Throws a char
- }
- catch (int x) { cout ¡¡ "Caught int: " ¡¡ x; }
- catch (double x) { cout ¡¡ "Caught double: " ¡¡ x; }
- catch (char x) { cout ¡¡ "Caught char: " ¡¡ x; }

In this example, our logic simulates different error conditions based on a 'choice' variable. If choice is 1, we throw an integer. The runtime skips the rest of the try block, skips over the int and double catches, and directly executes the 'catch(double x)' block because 3.14 is a double. Only one catch block will ever execute for a single thrown exception. Once that catch block finishes, the rest of the catch blocks are skipped.

Rule: Order of 'catch' Blocks Matters

- When an exception is thrown, the catch blocks are evaluated sequentially from top to bottom.
- The FIRST catch block with a matching type is executed.
- Critical Inheritance Rule: If you are catching both base class and derived class exception objects, the derived class catch block MUST appear BEFORE the base class catch block.
- Why? Because a base class reference can catch a derived class object, leading to intercepted errors.

Rule: Order of 'catch' Blocks Matters
- When an exception is thrown, the catch blocks are evaluated sequentially from top to bottom.
- The FIRST catch block with a matching type is executed.
- Critical Inheritance Rule: If you are catching both base class and derived class exception objects, the derived class catch block MUST appear BEFORE the base class catch block.
- Why? Because a base class reference can catch a derived class object, leading to intercepted errors.

Ordering is highly important when you start using Object-Oriented principles with exceptions. Let's say you have an exception class 'MathError', and a derived class 'DivideByZeroError'. If your first catch block catches 'MathError', and your second catches 'DivideByZeroError', the second block will never run! When a DivideByZeroError is thrown, the compiler sees the 'MathError' block first, realizes that a DivideByZeroError *is a* MathError due to inheritance, and executes the first block. Always order your catches from most specific to most generic.

The Catch-All Handler: catch(...)

- Sometimes you want to guarantee that your program catches an exception, even if you didn't anticipate its exact data type.
- C++ provides a universal catch-all handler:
- catch (...) {
- cout ¡¡ "An unknown exception occurred.";
- }
- The three dots (ellipsis) signify that this block matches absolutely ANY exception type.
- It MUST be placed as the LAST catch block in a sequence.

The Catch-All Handler: catch(...)
- Sometimes you want to guarantee that your program catches an exception, even if you didn't anticipate its exact data type.
- C++ provides a universal catch-all handler:
- catch (...) {
- cout ¡¡ "An unknown exception occurred.";
- }
- The three dots (ellipsis) signify that this block matches absolutely ANY exception type.
- It MUST be placed as the LAST catch block in a sequence.

The catch-all handler, represented by three dots inside the parentheses, is your ultimate safety net. It catches literally any exception that wasn't caught by the specific handlers above it. It's excellent for ensuring your program absolutely doesn't crash from an unhandled exception. However, because you don't know the type, you don't get a variable to inspect, meaning you won't know exactly what went wrong. For this reason, it should always be placed at the very bottom of your catch chain.

Exceptions and Stack Unwinding

- What happens to local variables when an exception is thrown and the scope is abruptly exited?
- Stack Unwinding: When an exception is thrown, the C++ runtime safely destroys all local objects created within the try block before the throw occurred.
- The destructors for these objects are automatically called in reverse order of their creation.
- This crucial feature prevents resource leaks (like open files or locked mutexes) during error conditions.

Exceptions and Stack Unwinding
- What happens to local variables when an exception is thrown and the scope is abruptly exited?
- Stack Unwinding: When an exception is thrown, the C++ runtime safely destroys all local objects created within the try block before the throw occurred.
- The destructors for these objects are automatically called in reverse order of their creation.
- This crucial feature prevents resource leaks (like open files or locked mutexes) during error conditions.

Before we wrap up, I want to mention a sophisticated C++ feature called Stack Unwinding. If you instantiate objects inside a try block, and an exception causes the program to abruptly leave that block, what happens to those objects? In some languages, they might leak memory. In C++, the runtime automatically calls the destructors for all fully constructed local objects before transferring control to the catch block. This guarantees that files are closed and memory is freed, maintaining system stability.

Summary and Best Practices

- Exceptions provide a clean, enforced way to handle runtime errors.
- Always match try blocks with at least one catch block.
- Use multiple catches to handle different error conditions specifically.
- Best Practice: Throw objects by value, but catch them by constant reference (e.g., catch(const MyException& e)) to avoid unnecessary copying.
- Use catch(...) sparingly, primarily as a final safety net at the highest level of your program.

Summary and Best Practices
- Exceptions provide a clean, enforced way to handle runtime errors.
- Always match try blocks with at least one catch block.
- Use multiple catches to handle different error conditions specifically.
- Best Practice: Throw objects by value, but catch them by constant reference (e.g., catch(const MyException& e)) to avoid unnecessary copying.
- Use catch(...) sparingly, primarily as a final safety net at the highest level of your program.

To summarize, exception handling using try, throw, and catch is the standard way to handle runtime anomalies in C++. It separates error handling from business logic. As a best practice, when throwing objects, you should catch them by constant reference. This prevents the compiler from making a copy of the exception object, which is faster and prevents an issue called object slicing. We will look closer at custom exception objects in our next class.