C++ Exception Handling

Introduction to Exception Handling

Exception handling in C++ provides a structured and controlled way to handle runtime errors.
It separates error detection from error handling and allows code to recover from unexpected situations.

An exception is an unexpected event (like division by zero, file not found, etc.) that disrupts the normal flow of the program. Exception handling allows the programmer to catch and manage such errors gracefully.

Basic Try-Catch Mechanism

The fundamental mechanism consists of:

- A `try` block where exceptions might be thrown
- One or more `catch` blocks to handle specific exceptions
- The `throw` statement to raise an exception

Basic Syntax

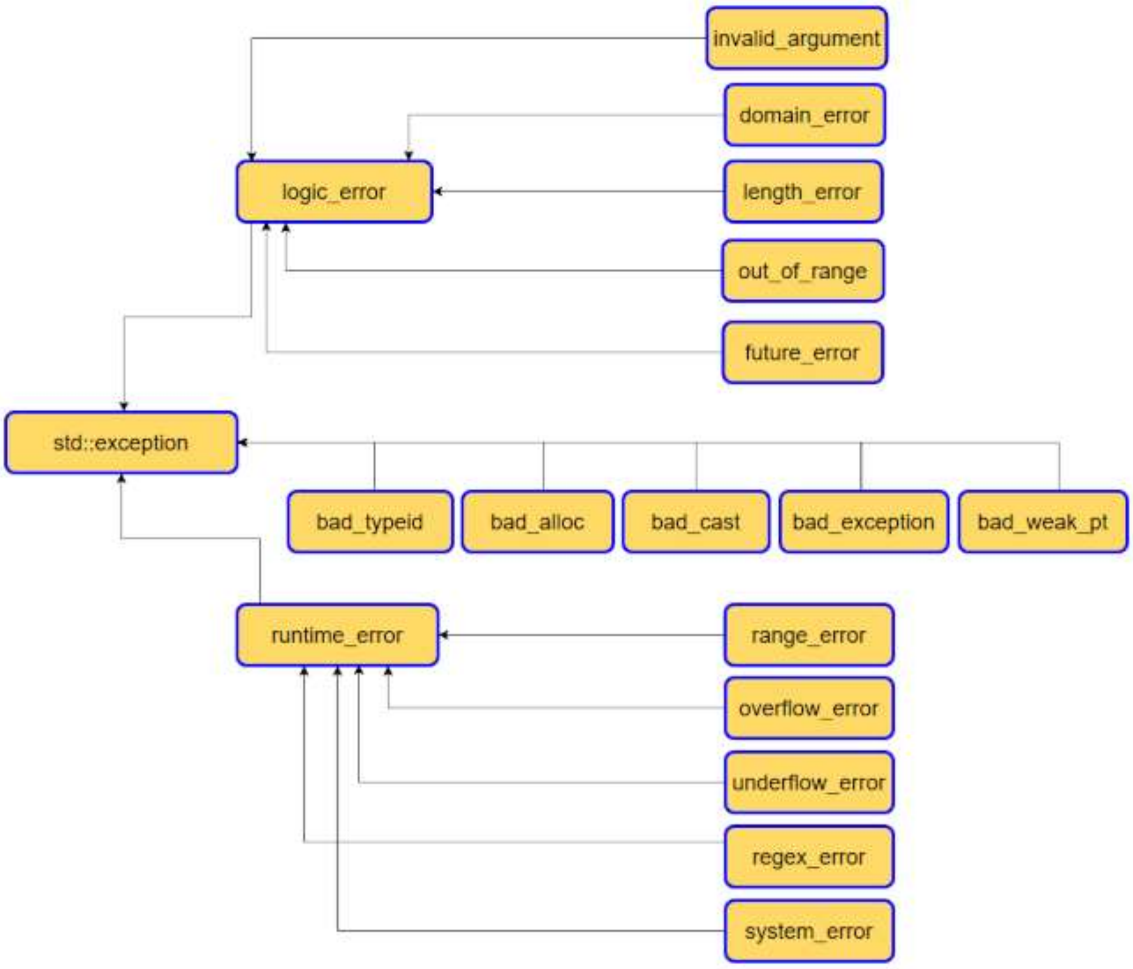
```
 try {
// Code that might throw an exception if (some_error_condition)
throw exception_value;

 }

 catch (data_type variable_name) {

   // Code to handle the exception }
```

Exception Types

1. **Standard Exceptions**: C++ provides standard exception classes in `<exception>`
   - `std::exception`: Base class for all standard exceptions
   - `std::runtime_error`   : Runtime logic errors
   - `std::logic_error`: Logic errors detectable before the program runs

2. **Custom Exceptions**: Create by inheriting from `std::exception`.

```
class MyException : public std::exception {
    public:
        const char* what() const noexcept override {
            return "My custom exception";
        }
};
```



Advanced Features

3. **Multiple Catch Blocks**: Can catch different types of exceptions.

```cpp
try {
    // code
        } catch (CustomException& e) {
            // Handle custom exception
} catch (std::runtime_error& e) {
    // Handle runtime error
} catch (...) {
    // Catch all other exceptions
}
```

4. **Catch-All Handler**: Use `catch(...)` to catch any exception.
5. **Exception Propagation**: If not caught, exceptions propagate up the call stack.
6. Function Exception Specification:

```
void myFunction() noexcept; // Cannot throw exceptions
```

Best Practices

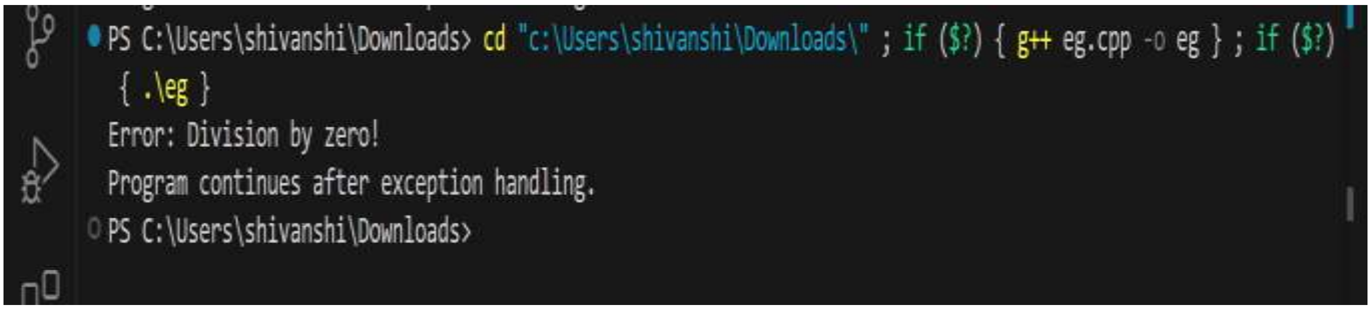
7. **Resource Management**: Use RAII (Resource Acquisition Is Initialization) with smart pointers to prevent resource leaks.
8. Best Practices:

```
• Only use exceptions for exceptional conditions •
Catch exceptions by reference
• Don't let destructors throw exceptions •
Keep exception classes simple
```

9. Common Exception Types:
    - `std::bad_alloc`: Memory allocation failure
    - `std::out_of_range`: Index out of bounds
    - `std::invalid_argument`   : Invalid argument passed
10. **Performance Impact**: Exception handling has minimal overhead when no exceptions occur.

SAMPLE CODE:

```cpp
#include <iostream>
using namespace std;
int main() {
    try {
        // Code that might throw an exception
        int dividend = 10;
        int divisor = 0;

        if (divisor == 0) {
            throw runtime_error("Division by zero!");
        }

        int result = dividend / divisor;
        cout << "Result: " << result << endl;
    }
    catch (const runtime_error& e) {
        // Code to handle the exception
        cerr << "Error: " << e.what() << endl;
    }

    cout << "Program continues after exception handling." << endl;
    return 0;
}
```



```
PS C:\Users\shivanshi\Downloads> cd "c:\Users\shivanshi\Downloads\" ; if ($?) { g++ eg.cpp -o eg } ; if ($?)
 { .\eg }
Error: Division by zero!
Program continues after exception handling.
PS C:\Users\shivanshi\Downloads>
```

Multiple Catch Blocks

You can handle different exceptions with multiple catch blocks. The catch blocks are evaluated in order.

```
#include <iostream>
#include <string>
using namespace std;
int main() {
    try {
        int option = 2;

        if (option == 1) {
            throw runtime_error("Runtime error occurred");
        }
        else if (option == 2) {
            throw out_of_range("Index out of range");
        }
        else if (option == 3) {
            throw "C-style string exception";
        }
    }
    catch (const runtime_error& e) {
        cerr << "Runtime error: " << e.what() << endl;
    }
    catch (const out_of_range& e) {
        cerr << "Out of range: " << e.what() << endl;
    }
    catch (const char* msg) {
        cerr << "C-style exception: " << msg << endl;
    }
    return 0;}
```

Another eg (code snippet):

```
try {

  throw 4.5;

}

catch (int e) {

  cout << "Caught integer exception: " << e << endl;

}

catch (double e) {

  cout << "Caught double exception: " << e << endl;

}
```

Catching All Exceptions

You can catch any exception using catch (...)
This should generally be used as the last catch block.
Code snippet:

```
try {
  throw "Some error";
}
catch (...) {
  cout << "Caught an exception of unknown type." << endl;
}
```

SAMPLE CODE

```
#include <iostream>
using namespace std;
int main() {
    try {
        throw 42;  // Throwing an integer
    }
    catch (const exception& e) {
        cerr << "Standard exception: " << e.what() << endl;
    }
    catch (...) {
        cerr << "Unknown exception caught" << endl;
    }
    return 0;}
```

Standard Exceptions

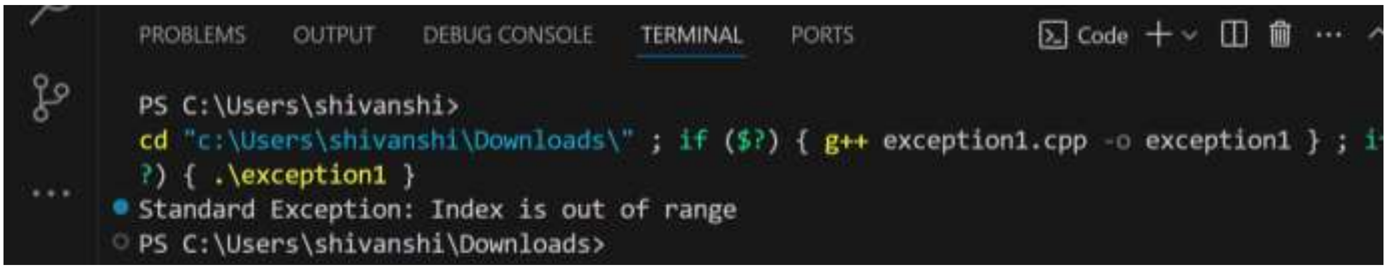
Common classes from **<stdexcept> header file:**

- logic_error
- runtime_error
- out_of_range
- overflow_error

Example:

SAMPLE CODE 1

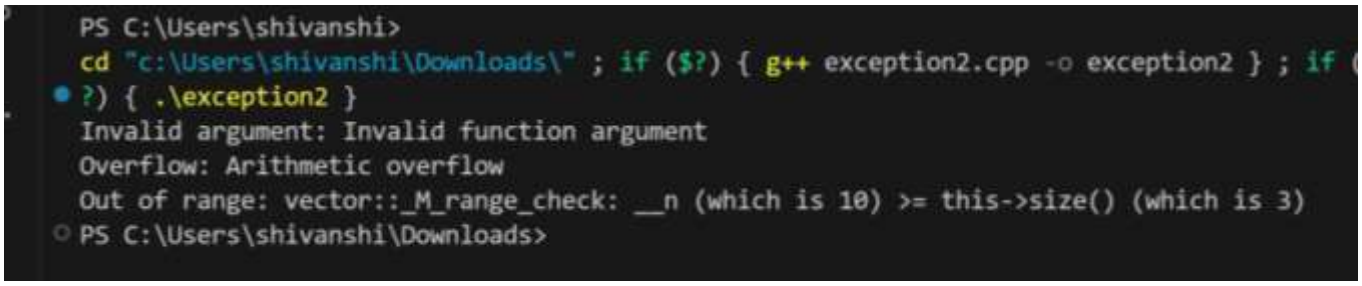
```
#include <iostream>

#include <stdexcept>

using namespace std;

int main() {

  try {

    throw out_of_range("Index is out of range");

  }

  catch (exception &e) {

    cout << "Standard Exception: " << e.what() << endl;

  }

}
```



```
PS C:\Users\shivanshi>
cd "c:\Users\shivanshi\Downloads\" ; if ($?) { g++ exception1.cpp -o exception1 } ; i
?) { .\exception1 }
Standard Exception: Index is out of range
PS C:\Users\shivanshi\Downloads>
```

SAMPLE CODE 2

```cpp
#include <iostream>
#include <stdexcept>
#include <vector>

void demonstrateStandardExceptions() {
    try {
        // logic_error family
        throw std::invalid_argument("Invalid function argument");
    }
    catch (const std::invalid_argument& e) {
        std::cerr << "Invalid argument: " << e.what() << std::endl;
    }

    try {
        // runtime_error family
        throw std::overflow_error("Arithmetic overflow");
    }
    catch (const std::overflow_error& e) {
        std::cerr << "Overflow: " << e.what() << std::endl;
    }

    try {
        // out_of_range example with vector
        std::vector<int> vec = {1, 2, 3};
        std::cout << vec.at(10) << std::endl;  // will throw out_of_range
    }
    catch (const std::out_of_range& e) {
        std::cerr << "Out of range: " << e.what() << std::endl;
    }
}

int main() {
    demonstrateStandardExceptions();
    return 0;
}
```



```
PS C:\Users\shivanshi>
cd "c:\Users\shivanshi\Downloads\" ; if ($?) { g++ exception2.cpp -o exception2 } ; if (
?) { .\exception2 }
Invalid argument: Invalid function argument
Overflow: Arithmetic overflow
Out of range: vector::_M_range_check: __n (which is 10) >= this->size() (which is 3)
PS C:\Users\shivanshi\Downloads>
```

Rethrowing Exceptions

You can catch an exception, perform some actions, and then rethrow it for handling at a higher level.

Code Snippet:

```
try {
  throw 20;
}
catch (int e) {
  cout << "Caught " << e << ", rethrowing..." <<
  endl; throw;
}
```

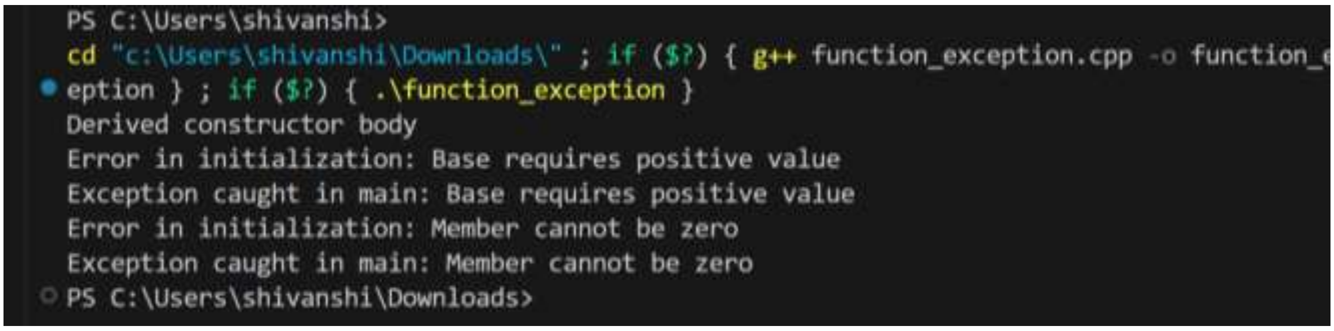
SAMPLE CODE:

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

void process(){
    try {
        throw runtime_error("Original error");
    }
    catch (const exception& e) {
        cerr << "Logging in process(): " << e.what() << endl;
        // Do some cleanup

        // Rethrow the same exception
        throw;
    }
}
int main() {
    try {
        process();
    }
    catch (const exception& e) {
        cerr << "Caught in main(): " << e.what() << endl;
    }

    return 0;
}
```

Function Try Blocks

Function try blocks are useful for catching exceptions thrown by member initializers in constructors or base class construction.

```cpp
#include <iostream>
#include <stdexcept>

class Base {
public:
    Base(int value) {
        if (value < 0) {
            throw std::invalid_argument("Base requires positive value");
        }
    }
};

class Member {
public:
    Member(int value) {
        if (value == 0) {
            throw std::invalid_argument("Member cannot be zero");
        }
    }
};

class Derived : public Base {
private:
    Member m_member;
    int m_value;

public:
    // Function try block for constructor
    Derived(int baseValue, int memberValue)
    try : Base(baseValue),      // May throw
          m_member(memberValue),  // May throw
          m_value(100)
    {
        std::cout << "Derived constructor body" << std::endl;
        // Constructor body
    }
    catch (const std::invalid_argument& e) {
        std::cerr << "Error in initialization: " << e.what() << std::endl;
        // Note: The exception is automatically rethrown after this catch block
    }
};

int main() {
    try {
        // This will succeed
        Derived d1(10, 5);

        // This will fail in the Base constructor
        Derived d2(-5, 5);
    }
    catch (const std::exception& e) {
        std::cerr << "Exception caught in main: " << e.what() << std::endl;
    }

    try {
        // This will fail in the Member constructor
        Derived d3(10, 0);
    }
    catch (const std::exception& e) {
        std::cerr << "Exception caught in main: " << e.what() << std::endl;
    }

    return 0;
}
```



```
PS C:\Users\shivanshi>
cd "c:\Users\shivanshi\Downloads\" ; if ($?) { g++ function_exception.cpp -o function_e
eption } ; if ($?) { .\function_exception }
Derived constructor body
Error in initialization: Base requires positive value
Exception caught in main: Base requires positive value
Error in initialization: Member cannot be zero
Exception caught in main: Member cannot be zero
PS C:\Users\shivanshi\Downloads>
```

Custom exception code:

This example shows:

1. A simple custom exception class inheriting from std::exception
2. Basic implementation of the required what() method
3. How to throw and catch the custom exception

```cpp
#include <iostream>

#include <exception>

#include <string>


// Simple custom exception

class MyException : public std::exception {

    std::string msg;

public:

    MyException(const std::string& s) : msg(s) {}

    const char* what() const noexcept override {

        return msg.c_str();

    }

};


// Function that uses our custom exception

void divide(int a, int b) {

    if (b == 0) {

        throw MyException("Division by zero");

    }

    std::cout << "Result: " << a / b << std::endl;

}


int main() {

    try {

        divide(10, 2);  // Works fine

        divide(10, 0);  // Throws exception

    } catch (const MyException& e) {

        std::cout << "Error: " << e.what() << std::endl;

    }

    return 0;

}
```

Exception Propagation

If an exception is thrown and not caught within a function, it propagates up the call stack until a matching catch block is found.

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

void level3() {
    throw runtime_error("Error in level 3");
}

void level2() {
    level3();  // Exception will propagate upward
}
void level1() {
    try {
        level2();
    }
    catch (const exception& e) {
        cerr << "Caught in level1: " << e.what() << endl;
    }
}

int main() {
    level1();
    cout << "Program continues after exception handling." << endl;
    return 0;
}
```