CS2810 OOAIA

Exceptions

Example Code Snippets

1. Basic try-catch block

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

int main() {
    try {
        int age;
        std::cout << "Enter your age: ";
        std::cin >> age;

        if (age < 0) {
            throw std::invalid_argument("Age cannot be negative!");
        }

        std::cout << "Your age is: " << age << std::endl;
    }
    catch (const std::invalid_argument& e) {
        std::cerr << "Error: " << e.what() << std::endl;
    }
    catch (...) {
        std::cerr << "An unknown error occurred!" << std::endl;
    }
    return 0;
}
```

2. Stack unwinding demonstration

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

void functionC() {
    throw std::runtime_error("Error from functionC");
}

void functionB() {
    std::cout << "Starting functionB\n";
    functionC();
    std::cout << "This won't be executed\n";
}

void functionA() {
    try {
        functionB();
    }
    catch (const std::exception& e) {
        std::cerr << "Caught in functionA: " << e.what() <<
std::endl;
        throw; // Re-throw the exception
    }
}

int main() {
    try {
        functionA();
    }
    catch (const std::exception& e) {
        std::cerr << "Caught in main: " << e.what() << std::endl;
    }
    return 0;
}
```

3. Custom exception class

```
#include <iostream>
#include <string>

class NetworkException : public std::exception {
private:
    std::string message;
public:
    NetworkException(const std::string& msg) : message(msg) {}
    const char* what() const noexcept override {
        return message.c_str();
    }
};

void connectToServer() {
    bool connectionFailed = true; // Simulate failure
    if (connectionFailed) {
        throw NetworkException("Failed to connect to server:
timeout");
    }
}

int main() {
    try {
        connectToServer();
    }
    catch (const NetworkException& e) {
        std::cerr << "Network error: " << e.what() << std::endl;
    }
    catch (const std::exception& e) {
        std::cerr << "Standard exception: " << e.what() << std::endl;
    }
    return 0;
}
```

4. Exception specification with different types

```
#include <iostream>
#include <string>
#include <vector>

double divideNumbers(double a, double b) {
    if (b == 0) {
        throw "Division by zero!"; // Throwing a C-string
    }
    return a / b;
}

int main() {
    std::vector<std::pair<double, double>> tests = {{10, 2}, {5, 0},
{8, 4}};

    for (const auto& test : tests) {
        try {
            double result = divideNumbers(test.first, test.second);
            std::cout << test.first << " / " << test.second
                      << " = " << result << std::endl;
        }
        catch (const char* msg) {
            std::cerr << "Error: " << msg << std::endl;
        }
        catch (...) {
            std::cerr << "Unknown error occurred" << std::endl;
        }
    }

    return 0;
}
```

5. Nested try-catch blocks

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

void processInput(int value) {
    try {
        if (value < 0) {
            throw std::out_of_range("Negative value not allowed");
        }
        if (value > 100) {
            throw std::overflow_error("Value too large");
        }
        std::cout << "Processed value: " << value << std::endl;
    }
    catch (const std::out_of_range& e) {
        std::cerr << "Out of range in processInput: " << e.what() <<
std::endl;
        throw; // Re-throw to outer handler
    }
}

int main() {
    int inputs[] = {50, -5, 150};

    for (int val : inputs) {
        try {
            try {
                processInput(val);
            }
            catch (const std::overflow_error& e) {
                std::cerr << "Overflow in main: " << e.what() <<
std::endl;
            }
        }
        catch (const std::exception& e) {
            std::cerr << "General exception in main: " << e.what() <<
std::endl;
        }
```

```
    }

    return 0;
}
```

Concept	Description	Example
`try` block	Code that might throw exceptions	`try { riskyOperation(); }`
`catch` block	Handles specific exceptions	`catch (const std::exception& e) {...}`
Standard exceptions	Derived from `std::exception`	`std::runtime_error`, `std::logic_error`
Custom exceptions	User-defined exception classes	`class MyError : public std::exception {...}`
Throwing	Raising an exception	`throw std::invalid_argument("msg");`
Catch by reference	Prevents slicing, more efficient	`catch (const std::exception& e)`
Catch-all	Handles any exception	`catch (...) {...}`
Stack unwinding	Automatic cleanup when exception propagates	Destructors called as stack unwinds
Exception safety	Guarantees about program state	Basic, strong, no-throw guarantees

Practice Problems

1. Problem: Create a program that asks the user to input two numbers and divides them. Handle the following exceptions:

- Division by zero
- Invalid input (non-numeric values)

Examples:

```
Enter two numbers: 10 2
Result: 5

Enter two numbers: 5 0
Error: Division by zero!

Enter two numbers: abc 5
Error: Invalid input!
```

2. Problem: Implement a `StudentGrade` class that stores a student's grade (0-100). Throw custom exceptions for:

- `GradeTooLowException` if grade < 0
- `GradeTooHighException` if grade > 100
- `InvalidGradeException` if input isn't numeric

Examples:

```
Enter grade: 85
Grade recorded: 85

Enter grade: -5
Error: Grade too low!
```

```
Enter grade: 105
Error: Grade too high!

Enter grade: abc
Error: Invalid grade format!
```

3. Problem: Write a program that attempts to:

1. Open a file ("data.txt")
2. Read integers from it
3. Calculate their average

Handle these exceptions:

- File not found
- Empty file
- Non-integer data in file
- Division errors

Example:

```
File Input:
10
20
30
Abc

Output:
Error: Non-integer data found in file!
```

4. Problem: Create a program with 3 nested function calls (main → funcA → funcB → funcC). Have funcC throw an exception that should be caught in main. Add print statements in each function to demonstrate stack unwinding.

Example:

```
Entering funcA
Entering funcB
Entering funcC
Exception caught in main: Error from funcC
```

5. Problem: Implement a `DynamicArray` class that:

- Dynamically allocates memory
- Provides bounds-checked access
- Throws `std::out_of_range` for invalid indices
- Guarantees no memory leaks even if exceptions occur

Write test cases that:

1. Access valid indices
2. Access invalid indices
3. Cause memory allocation failures

6. Problem: Create a template function `safeGet` that:

- Takes a vector and index as input
- Returns the element if index is valid
- Throws different exceptions for:
    - Negative indices
    - Indices >= vector size
    - Empty vector

Write test cases for:

```
vector<int> v = {10, 20, 30};
safeGet(v, 1);   // Should return 20
safeGet(v, -1);  // Should throw
safeGet(v, 5);   // Should throw
vector<int> empty;
safeGet(empty, 0); // Should throw
```