

Object Oriented Programming C++

Chapter: Exception Handling

Exam-Oriented Coding Problems and Solutions (Set 5)

Coding Problems and Solutions

Problem 1. Write a C++ program to calculate the area of a triangle from its three sides, throwing an exception if the sides do not form a valid triangle.

Solution:

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

double area(double a, double b, double c) {
    if (a <= 0 || b <= 0 || c <= 0)
        throw invalid_argument("Sides must be positive");
    if (a + b <= c || b + c <= a || a + c <= b)
        throw invalid_argument("Triangle inequality violated");
    double s = (a + b + c) / 2;
    return sqrt(s * (s - a) * (s - b) * (s - c));
}

int main() {
    double t[3][3] = { {3, 4, 5}, {1, 2, 10}, {-2, 3, 4} };
    for (auto& s : t) {
        try {
            double ar = area(s[0], s[1], s[2]);
            cout << "Area = " << ar << endl;
        }
        catch (const invalid_argument& e) {
            cout << "Invalid triangle: " << e.what() << endl;
        }
    }
    return 0;
}
```

Output:

```
Area = 6
Invalid triangle: Triangle inequality violated
Invalid triangle: Sides must be positive
```

Explanation:

The function `area()` performs two checks before using Heron's formula: all sides must be positive, and the sum of any two sides must be greater than the third. Each failed check throws `invalid_argument` with its own message. The first test case (3, 4, 5) is valid and gives area 6. The second (1, 2, 10) violates the triangle inequality, and the third contains a negative side. Validating input by throwing, instead of printing and continuing, makes sure that an invalid area is never used.

Problem 2. Write a C++ program to find the average of the elements of a vector, throwing an exception if the vector is empty.

Solution:

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

```

#include <stdexcept>
using namespace std;

double average(const vector<int>& v) {
    if (v.empty())
        throw domain_error("Cannot find average of an empty list");
    int sum = 0;
    for (int x : v)
        sum += x;
    return (double)sum / v.size();
}

int main() {
    vector<int> a = {10, 20, 30, 45};
    vector<int> b;           // empty list
    try {
        double r1 = average(a);
        cout << "Average of a = " << r1 << endl;
        double r2 = average(b);
        cout << "Average of b = " << r2 << endl;
    }
    catch (const domain_error& e) {
        cout << "Error: " << e.what() << endl;
    }
    return 0;
}

```

Output:

```

Average of a = 26.25
Error: Cannot find average of an empty list

```

Explanation:

Dividing the sum by the size of an empty vector would be a division by zero, so the function throws `domain_error` before attempting it. The first call, with four elements, returns 26.25. The second call, with an empty vector, throws, so the statement printing the second average is skipped and control moves to the catch block. The standard class `domain_error` is intended for cases where the input is outside the domain for which an operation is defined.

Problem 3. Write a C++ program with a `Matrix` class whose `add()` function throws an exception when the dimensions of the two matrices are different.

Solution:

```

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

class Matrix {
    int rows, cols;
    int data[3][3];
public:
    Matrix(int r, int c, int fill) : rows(r), cols(c) {
        for (int i = 0; i < r; i++)
            for (int j = 0; j < c; j++)
                data[i][j] = fill;
    }
    Matrix add(const Matrix& m) const {
        if (rows != m.rows || cols != m.cols)
            throw invalid_argument("Matrix dimensions do not match");
        Matrix res(rows, cols, 0);
        for (int i = 0; i < rows; i++)

```

```

        for (int j = 0; j < cols; j++)
            res.data[i][j] = data[i][j] + m.data[i][j];
        return res;
    }
    void show() const {
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++)
                cout << data[i][j] << " ";
            cout << endl;
        }
    }
};

int main() {
    Matrix a(2, 2, 1), b(2, 2, 5), c(3, 3, 2);
    try {
        Matrix s = a.add(b);
        cout << "A + B:" << endl;
        s.show();
        Matrix t = a.add(c);          // 2x2 + 3x3 : not possible
        t.show();
    }
    catch (const invalid_argument& e) {
        cout << "Error: " << e.what() << endl;
    }
    return 0;
}

```

Output:

```

A + B:
6 6
6 6
Error: Matrix dimensions do not match

```

Explanation:

Matrix addition is defined only for matrices of the same size. The `add()` function compares the row and column counts and throws `invalid_argument` if they differ. Adding two 2 x 2 matrices works and shows a matrix of 6s. Adding a 2 x 2 matrix and a 3 x 3 matrix throws, so the line that would display the result is never executed. Reporting the error with an exception is better than returning a partly filled matrix, which a caller might mistake for a correct result.

Problem 4. Write a C++ program to simulate an ATM PIN check that allows three attempts, using one exception for a wrong PIN and another for a blocked card.

Solution:

```

#include <iostream>
using namespace std;

class WrongPin {
public:
    int attemptsLeft;
    WrongPin(int n) : attemptsLeft(n) {}
};

class CardBlocked {};

void checkPin(int entered, int attempt) {
    const int correct = 4321;
    if (entered != correct) {
        if (attempt == 3)

```

```

        throw CardBlocked();
        throw WrongPin(3 - attempt);
    }
    cout << "PIN accepted. Welcome!" << endl;
}

int main() {
    int tries[] = {1111, 2222, 3333};
    try {
        for (int i = 0; i < 3; i++) {
            try {
                checkPin(tries[i], i + 1);
                break;
            }
            catch (const WrongPin& w) {
                cout << "Wrong PIN. Attempts left: "
                    << w.attemptsLeft << endl;
            }
        }
    }
    catch (const CardBlocked&) {
        cout << "Card blocked after 3 wrong attempts" << endl;
    }
    return 0;
}

```

Output:

```

Wrong PIN. Attempts left: 2
Wrong PIN. Attempts left: 1
Card blocked after 3 wrong attempts

```

Explanation:

The inner try-catch handles only WrongPin, which is thrown for the first two failed attempts, and the loop continues. On the third failure checkPin() throws CardBlocked. The inner catch does not match this type, so the exception leaves the loop and is caught by the outer catch block. This example shows nested try blocks, the use of exception objects to carry data (attempts left), and an exception being used to leave a loop.

Problem 5. Write a C++ program that throws a pointer to a dynamically created exception object and catches it by pointer. Who is responsible for destroying the object?

Solution:

```

#include <iostream>
using namespace std;

class MyError {
public:
    int code;
    MyError(int c) : code(c) {}
    ~MyError() {
        cout << "MyError object destroyed" << endl;
    }
};

int main() {
    try {
        throw new MyError(404);           // exception object created on the heap
    }
    catch (MyError* e) {
        cout << "Caught pointer, error code = " << e->code << endl;
        delete e;                          // programmer must free the object
    }
}

```

```
    }  
    return 0;  
}
```

Output:

```
Caught pointer, error code = 404  
MyError object destroyed
```

Explanation:

The statement `throw new MyError(404)` creates the exception object on the heap and throws a pointer to it. The handler must therefore have a pointer parameter, `catch(MyError*)`, and must call `delete` itself; the destructor message proves that the object is destroyed only when `delete` is executed. If the catch block forgot to delete it, the memory would leak. For this reason, it is better practice to throw objects by value and catch them by reference, so that the runtime manages the object automatically.

Problem 6. Write a C++ program in which a recursive function throws an exception at a certain depth. Show the order of construction and destruction of the local objects.

Solution:

```
#include <iostream>  
using namespace std;  
  
class Frame {  
    int level;  
public:  
    Frame(int l) : level(l) {  
        cout << "Enter level " << level << endl;  
    }  
    ~Frame() {  
        cout << "Leave level " << level << endl;  
    }  
};  
  
void recurse(int n) {  
    Frame f(n);  
    if (n == 3)  
        throw "Maximum depth reached";  
    recurse(n + 1);  
}  
  
int main() {  
    try {  
        recurse(1);  
    }  
    catch (const char* msg) {  
        cout << "Caught: " << msg << endl;  
    }  
    return 0;  
}
```

Output:

```
Enter level 1  
Enter level 2  
Enter level 3  
Leave level 3  
Leave level 2  
Leave level 1  
Caught: Maximum depth reached
```

Explanation:

Each recursive call creates a local Frame object, so "Enter" messages are printed for levels 1, 2 and 3. At level 3 the exception is thrown. During stack unwinding, the three active function calls are exited one by one, and the destructors of their local Frame objects run in the reverse order of creation, so the "Leave" messages appear for levels 3, 2 and 1. Only after that does the catch block in main() execute. This is a useful program for questions on the order of stack unwinding.

Problem 7. Write a C++ program using the noexcept operator to show which expressions are declared as non-throwing.

Solution:

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

void normalFunc() {}
void safeFunc() noexcept {}
void riskyFunc() { throw 1; }

int main() {
    cout << boolalpha;
    cout << "noexcept(normalFunc()) = " << noexcept(normalFunc()) << endl;
    cout << "noexcept(safeFunc())   = " << noexcept(safeFunc()) << endl;
    cout << "noexcept(riskyFunc())  = " << noexcept(riskyFunc()) << endl;
    cout << "noexcept(5 + 3)        = " << noexcept(5 + 3) << endl;
    return 0;
}
```

Output:

```
noexcept(normalFunc()) = false
noexcept(safeFunc())   = true
noexcept(riskyFunc())  = false
noexcept(5 + 3)        = true
```

Explanation:

The noexcept operator takes an expression and returns true at compile time if the expression is declared not to throw, without actually executing it. A function with no noexcept specifier, such as normalFunc(), is assumed to be potentially throwing, so the result is false even though its body is empty. safeFunc() is declared noexcept, so the result is true, while riskyFunc() is false. The simple expression 5 + 3 cannot throw, so it is true. Note that none of the functions are called.

Problem 8. Write a C++ program that adds two integers safely, throwing an exception if the result would exceed the range of int.

Solution:

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

int safeAdd(int a, int b) {
    if (b > 0 && a > INT_MAX - b)
        throw overflow_error("Result is above INT_MAX");
    if (b < 0 && a < INT_MIN - b)
        throw overflow_error("Result is below INT_MIN");
    return a + b;
}

int main() {
```

```

int pairs[3][2] = { {100, 200}, {INT_MAX, 1}, {INT_MIN, -1} };
for (auto& p : pairs) {
    try {
        int r = safeAdd(p[0], p[1]);
        cout << p[0] << " + " << p[1] << " = " << r << endl;
    }
    catch (const overflow_error& e) {
        cout << "Overflow detected: " << e.what() << endl;
    }
}
return 0;
}

```

Output:

```

100 + 200 = 300
Overflow detected: Result is above INT_MAX
Overflow detected: Result is below INT_MIN

```

Explanation:

Signed integer overflow is undefined behaviour in C++, so the check must be done before the addition. For positive b, overflow occurs if a is greater than INT_MAX - b; for negative b, it occurs if a is less than INT_MIN - b. In these cases overflow_error is thrown with a message, and the addition is never performed. The first pair is within range and prints 300; the other two are the extreme cases at the top and bottom of the range.

Problem 9. Write a C++ program for a simple library that issues books, using one exception class for a missing book and another for a book that is already issued.

Solution:

```

#include <iostream>
#include <map>
#include <string>
using namespace std;

class BookNotFound {
public:
    string title;
    BookNotFound(const string& t) : title(t) {}
};

class AlreadyIssued {
public:
    string title;
    AlreadyIssued(const string& t) : title(t) {}
};

class Library {
    map<string, bool> issued;          // title -> is it issued?
public:
    Library() {
        issued["C++ Primer"] = false;
        issued["Data Structures"] = false;
    }
    void issueBook(const string& title) {
        auto it = issued.find(title);
        if (it == issued.end())
            throw BookNotFound(title);
        if (it->second)
            throw AlreadyIssued(title);
        it->second = true;
    }
};

```

```

        cout << "Issued: " << title << endl;
    }
};

int main() {
    Library lib;
    string requests[] = {"C++ Primer", "C++ Primer", "Operating Systems"};
    for (const string& r : requests) {
        try {
            lib.issueBook(r);
        }
        catch (const AlreadyIssued& e) {
            cout << "Sorry, already issued: " << e.title << endl;
        }
        catch (const BookNotFound& e) {
            cout << "Book not found: " << e.title << endl;
        }
    }
    return 0;
}

```

Output:

```

Issued: C++ Primer
Sorry, already issued: C++ Primer
Book not found: Operating Systems

```

Explanation:

The Library class keeps the issue status of each title in a map. The issueBook() function throws BookNotFound if the title is not in the map, and AlreadyIssued if the title is present but already marked as issued. The first request succeeds. The second request for the same title throws AlreadyIssued, and the third request, for a title that is not in the library, throws BookNotFound. Each exception object carries the title, so the handlers can print a specific message.

Problem 10. Write a C++ program that throws integer error codes from a function and uses a switch statement inside the catch block to display a suitable message for each code.

Solution:

```

#include <iostream>
using namespace std;

void operation(int id) {
    if (id == 1) throw 101;
    if (id == 2) throw 202;
    if (id == 3) throw 303;
    cout << "Operation " << id << " completed" << endl;
}

int main() {
    for (int id = 0; id <= 4; id++) {
        try {
            operation(id);
        }
        catch (int code) {
            switch (code) {
                case 101: cout << "Error 101: File not found" << endl; break;
                case 202: cout << "Error 202: Access denied" << endl; break;
                default: cout << "Error " << code << ": Unknown" << endl;
            }
        }
    }
}

```

```
    return 0;  
}
```

Output:

```
Operation 0 completed  
Error 101: File not found  
Error 202: Access denied  
Error 303: Unknown  
Operation 4 completed
```

Explanation:

The function throws integer values as error codes, and a single `catch(int)` block receives each of them. A switch statement on the caught value selects the message to display, with a default case for codes that have no specific message (303 here). Operations 0 and 4 throw nothing and complete normally, so the catch block is not executed for them. This approach is useful when a program uses numeric error codes, but user-defined exception classes are generally clearer for larger programs.