

Object Oriented Programming C++

Chapter: Exception Handling

Exam-Oriented Coding Problems and Solutions (Set 4)

Coding Problems and Solutions

Problem 1. Write a C++ program that processes an array of numbers, throws an exception for each negative value, and still continues to add up the valid values.

Solution:

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

int checkedValue(int x) {
    if (x < 0)
        throw x;
    return x;
}

int main() {
    int data[] = {10, -5, 20, -8, 30};
    int sum = 0;
    for (int v : data) {
        try {
            sum += checkedValue(v);
        }
        catch (int bad) {
            cout << "Skipping invalid value: " << bad << endl;
        }
    }
    cout << "Sum of valid values = " << sum << endl;
    return 0;
}
```

Output:

```
Skipping invalid value: -5
Skipping invalid value: -8
Sum of valid values = 60
```

Explanation:

The function `checkedValue()` throws the value itself if it is negative. Because the try-catch is placed inside the loop, each exception is handled within the same iteration, the message is printed, and the loop moves on to the next element. Only the valid values 10, 20 and 30 are added, giving 60. Placing the try block inside a loop is the standard way to continue after an error; placing it outside the loop would end the loop at the first exception.

Problem 2. Write a C++ program with a base exception class `NetworkError` and two derived classes `TimeoutError` and `DNSError`. Handle all of them with a single catch block using the base class reference.

Solution:

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

class NetworkError {
public:
```

```

    virtual void show() const {
        cout << "General network error" << endl;
    }
    virtual ~NetworkError() {}
};

class TimeoutError : public NetworkError {
public:
    void show() const override {
        cout << "Connection timed out" << endl;
    }
};

class DNSError : public NetworkError {
public:
    void show() const override {
        cout << "Domain name could not be resolved" << endl;
    }
};

void connectTo(int code) {
    if (code == 1) throw TimeoutError();
    if (code == 2) throw DNSError();
    if (code == 3) throw NetworkError();
    cout << "Connected successfully" << endl;
}

int main() {
    for (int code = 0; code <= 3; code++) {
        try {
            connectTo(code);
        }
        catch (const NetworkError& e) {
            e.show();        // virtual call keeps the derived behaviour
        }
    }
    return 0;
}

```

Output:

```

Connected successfully
Connection timed out
Domain name could not be resolved
General network error

```

Explanation:

Each derived class overrides the virtual function `show()`. One handler, `catch(const NetworkError&)`, catches objects of the base class and of both derived classes. Because the exception is caught by reference and `show()` is virtual, the correct derived version runs for each object, so a timeout and a DNS failure print different messages. Catching by value instead would slice the object and always call the base version.

Problem 3. Write a C++ program with three nested function calls `level1()`, `level2()` and `level3()`, where an exception thrown in `level3()` is caught in `main()`. Show the output and explain which statements are skipped.

Solution:

```

#include <iostream>
using namespace std;

void level3() {

```

```

        cout << "level3: about to throw" << endl;
        throw "Fatal error in level3";
        cout << "level3: end" << endl;
    }

    void level2() {
        cout << "level2: calling level3" << endl;
        level3();
        cout << "level2: end" << endl;
    }

    void level1() {
        cout << "level1: calling level2" << endl;
        level2();
        cout << "level1: end" << endl;
    }

    int main() {
        try {
            level1();
        }
        catch (const char* msg) {
            cout << "main: caught -> " << msg << endl;
        }
        cout << "main: program continues" << endl;
        return 0;
    }
}

```

Output:

```

level1: calling level2
level2: calling level3
level3: about to throw
main: caught -> Fatal error in level3
main: program continues

```

Explanation:

Execution goes down through level1() and level2() into level3(), where the throw statement is executed. None of the remaining statements in level3(), level2() and level1() run (the three "end" lines are missing from the output), because the exception unwinds all three function calls and goes directly to the catch block in main(). After the handler finishes, the program continues normally with the statement following the try-catch group. This shows that an exception can be handled far from the place where it was thrown.

Problem 4. Write a C++ program that tries to open a file that does not exist and reports the problem by throwing and catching a runtime_error.

Solution:

```

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

void readFile(const string& name) {
    ifstream file(name);
    if (!file.is_open())
        throw runtime_error("Cannot open file: " + name);
    cout << "File opened successfully" << endl;
}

int main() {

```

```

    try {
        readFile("nonexistent_file.txt");
    }
    catch (const runtime_error& e) {
        cout << "Error: " << e.what() << endl;
    }
    return 0;
}

```

Output:

```
Error: Cannot open file: nonexistent_file.txt
```

Explanation:

File streams do not throw exceptions on failure by default; they only set an error state. The function therefore checks `is_open()` itself and throws `runtime_error` with a descriptive message that includes the file name. The caller handles it in a catch block and prints `e.what()`. Because the file `nonexistent_file.txt` is not present, the exception is always thrown here. This pattern converts a silent failure into an explicit error that cannot be ignored.

Problem 5. Write a C++ program that defines a custom exception class derived from `std::exception`, stores an error message in a string member, and overrides `what()`.

Solution:

```

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

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

void process(int level) {
    if (level < 1)
        throw AppException("Level is too low: " + to_string(level));
    if (level > 10)
        throw AppException("Level is too high: " + to_string(level));
    cout << "Level " << level << " accepted" << endl;
}

int main() {
    int levels[] = {5, 0, 15};
    for (int l : levels) {
        try {
            process(l);
        }
        catch (const exception& e) {
            cout << "Exception: " << e.what() << endl;
        }
    }
    return 0;
}

```

Output:

```
Level 5 accepted
Exception: Level is too low: 0
Exception: Level is too high: 15
```

Explanation:

AppException stores the message in a `std::string` member and returns it from `what()` using `c_str()`. The `process()` function builds a different message for each kind of failure, using `to_string()` to include the offending value. In `main()`, the handler catches by reference to the base class `std::exception`, so any exception derived from it can be handled by the same block, and the virtual `what()` returns the message of the actual thrown object. Level 5 is within the range, so no exception occurs.

Problem 6. Write a C++ program with a Fraction class whose constructor throws an exception when the denominator is zero. Demonstrate valid and invalid use.

Solution:

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

class Fraction {
    int num, den;
public:
    Fraction(int n, int d) : num(n), den(d) {
        if (d == 0)
            throw invalid_argument("Denominator cannot be zero");
    }
    Fraction add(const Fraction& f) const {
        return Fraction(num * f.den + f.num * den, den * f.den);
    }
    void show() const {
        cout << num << "/" << den << endl;
    }
};

int main() {
    try {
        Fraction a(1, 2), b(1, 3);
        Fraction c = a.add(b);
        cout << "Sum = ";
        c.show();
        Fraction d(3, 0);           // invalid fraction
        d.show();
    }
    catch (const invalid_argument& e) {
        cout << "Error: " << e.what() << endl;
    }
    return 0;
}
```

Output:

```
Sum = 5/6
Error: Denominator cannot be zero
```

Explanation:

Throwing from the constructor is the correct way to report that an object cannot be created in a valid state, because a constructor has no return value. The first two fractions are valid, and their sum 5/6 is computed

and displayed. The construction of Fraction d(3, 0) throws `invalid_argument`, so the object is never created, the next statement `d.show()` is skipped, and control moves to the catch block.

Problem 7. Write a C++ program in which a function catches a low-level `out_of_range` exception and throws a different, application-level exception instead (exception translation).

Solution:

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

class DataError {
public:
    string info;
    DataError(const string& s) : info(s) {}
};

int getItem(const vector<int>& v, size_t index) {
    try {
        return v.at(index);
    }
    catch (const out_of_range&) {
        // translate the low-level exception into an application-level one
        throw DataError("Item " + to_string(index) + " does not exist");
    }
}

int main() {
    vector<int> v = {5, 10, 15};
    try {
        int a = getItem(v, 2);
        cout << "Item 2 = " << a << endl;
        int b = getItem(v, 9);
        cout << "Item 9 = " << b << endl;
    }
    catch (const DataError& e) {
        cout << "DataError: " << e.info << endl;
    }
    return 0;
}
```

Output:

```
Item 2 = 15
DataError: Item 9 does not exist
```

Explanation:

The function `getItem()` catches the standard `out_of_range` exception thrown by `vector::at()` and throws a user-defined `DataError` object in its place. The caller then deals only with `DataError` and does not need to know how the data is stored internally. This is called exception translation, and it keeps low-level details hidden from higher levels of a program. The first call succeeds and returns 15; the second call (index 9) triggers the translated exception.

Problem 8. Write a C++ program that retries an operation up to five times if it throws an exception, and stops retrying once it succeeds.

Solution:

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

int attempt = 0;

void unreliableOperation() {
    attempt++;
    if (attempt < 3)
        throw runtime_error("Operation failed");
    cout << "Operation succeeded on attempt " << attempt << endl;
}

int main() {
    const int maxTries = 5;
    for (int i = 1; i <= maxTries; i++) {
        try {
            unreliableOperation();
            break;          // success: leave the loop
        }
        catch (const runtime_error& e) {
            cout << "Attempt " << i << ": " << e.what()
                << ", retrying..." << endl;
        }
    }
    return 0;
}
```

Output:

```
Attempt 1: Operation failed, retrying...
Attempt 2: Operation failed, retrying...
Operation succeeded on attempt 3
```

Explanation:

The function `unreliableOperation()` is made to fail on its first two calls by throwing `runtime_error`, and to succeed on the third. The try-catch group sits inside a loop, so each failure is reported and the loop retries. On the third attempt the function returns normally, the `break` statement is executed, and the loop ends without using up all five tries. This pattern is commonly used for temporary failures such as network or file access problems.

Problem 9. Write a C++ program to validate passwords using three user-defined exception classes: `TooShort`, `NoDigit` and `NoUpper`, with a separate catch block for each.

Solution:

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

class TooShort {};
class NoDigit {};
class NoUpper {};

void validate(const string& pwd) {
    if (pwd.length() < 6)
        throw TooShort();
}
```

```

    bool digit = false, upper = false;
    for (char ch : pwd) {
        if (isdigit(ch)) digit = true;
        if (isupper(ch)) upper = true;
    }
    if (!digit) throw NoDigit();
    if (!upper) throw NoUpper();
    cout << "\"" << pwd << "\": valid password" << endl;
}

int main() {
    string tests[] = {"ab1", "abcdefg", "abcdef1", "Abcdef1"};
    for (const string& p : tests) {
        try {
            validate(p);
        }
        catch (TooShort&) {
            cout << "\"" << p << "\": too short (min 6)" << endl;
        }
        catch (NoDigit&) {
            cout << "\"" << p << "\": needs a digit" << endl;
        }
        catch (NoUpper&) {
            cout << "\"" << p << "\": needs an uppercase letter" << endl;
        }
    }
    return 0;
}

```

Output:

```

"ab1": too short (min 6)
"abcdefg": needs a digit
"abcdef1": needs an uppercase letter
"Abcdef1": valid password

```

Explanation:

The three empty classes are used only as exception types, which is enough because the type itself tells the handler what went wrong. The validate() function checks the rules in order and throws the first exception that applies. Each of the four test passwords fails a different rule, or passes all of them, so every catch block is exercised once. Using separate types lets each error be handled with a different message.

Problem 10. Write a C++ program using a Lock class to demonstrate how RAII ensures that a resource is released even when an exception is thrown.

Solution:

```

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

class Lock {
    const char* name;
public:
    Lock(const char* n) : name(n) {
        cout << name << " locked" << endl;
    }
    ~Lock() {
        cout << name << " unlocked" << endl;
    }
};

```

```

void criticalSection(bool fail) {
    Lock lk("Resource");           // acquired in the constructor
    cout << "Working inside critical section" << endl;
    if (fail)
        throw runtime_error("Failure inside critical section");
    cout << "Work finished normally" << endl;
}

int main() {
    bool cases[] = {false, true};
    for (bool f : cases) {
        try {
            criticalSection(f);
        }
        catch (const exception& e) {
            cout << "Caught: " << e.what() << endl;
        }
    }
    return 0;
}

```

Output:

```

Resource locked
Working inside critical section
Work finished normally
Resource unlocked
Resource locked
Working inside critical section
Resource unlocked
Caught: Failure inside critical section

```

Explanation:

The Lock object acquires the resource in its constructor and releases it in its destructor. In the first call no exception occurs, and the destructor runs when the function ends normally. In the second call, an exception is thrown while the lock is held; during stack unwinding the destructor of the local object lk is automatically called, so "Resource unlocked" is printed before the catch block runs. This shows why C++ needs no finally block: cleanup code placed in destructors runs in both cases.