

Object Oriented Programming C++

Chapter: Exception Handling

Exam-Oriented Coding Problems and Solutions (Set 3)

Coding Problems and Solutions

Problem 1. Write a C++ program with a function that calculates the factorial of a number and throws an exception if the number is negative. Handle the exception in the calling function.

Solution:

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

long long factorial(int n) {
    if (n < 0)
        throw n;           // throw the invalid value itself
    long long f = 1;
    for (int i = 2; i <= n; i++)
        f *= i;
    return f;
}

int main() {
    int values[] = {5, -3};
    for (int n : values) {
        try {
            long long result = factorial(n);
            cout << "Factorial of " << n << " = " << result << endl;
        }
        catch (int x) {
            cout << "Error: negative number " << x << endl;
        }
    }
    return 0;
}
```

Output:

```
Factorial of 5 = 120
Error: negative number -3
```

Explanation:

The factorial() function throws the offending integer when n is negative. The exception is not handled inside the function; it propagates to the caller, where catch(int) receives the value. Because the result is stored in a variable before printing, no half-printed line appears when the exception is thrown. This shows that an error can be detected in one function and handled in another.

Problem 2. Write a C++ program to find the roots of a quadratic equation. Throw an exception if the coefficient a is zero, and a different exception if the roots are complex. Use separate catch blocks.

Solution:

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

void solveQuadratic(double a, double b, double c) {
    if (a == 0)
```

```

        throw invalid_argument("Coefficient 'a' cannot be zero.");
double d = b * b - 4 * a * c;
if (d < 0)
    throw runtime_error("Roots are complex (negative discriminant).");
double r1 = (-b + sqrt(d)) / (2 * a);
double r2 = (-b - sqrt(d)) / (2 * a);
cout << "Roots: " << r1 << " and " << r2 << endl;
}

int main() {
    double tests[3][3] = { {1, -5, 6}, {0, 2, 3}, {1, 2, 5} };
    for (auto& t : tests) {
        try {
            solveQuadratic(t[0], t[1], t[2]);
        }
        catch (const invalid_argument& e) {
            cout << "Invalid input: " << e.what() << endl;
        }
        catch (const runtime_error& e) {
            cout << "Math error: " << e.what() << endl;
        }
    }
    return 0;
}

```

Output:

```

Roots: 3 and 2
Invalid input: Coefficient 'a' cannot be zero.
Math error: Roots are complex (negative discriminant).

```

Explanation:

The function throws `invalid_argument` when `a` is zero (the equation is not quadratic) and `runtime_error` when the discriminant is negative. Two separate catch blocks handle these two types differently. The three test cases show a normal result, the first exception, and the second exception. Both classes come from the standard library, so `e.what()` returns the message supplied at the time of the throw.

Problem 3. Write a C++ program to implement a bank account class with a `withdraw()` function that throws a user-defined exception, carrying the balance and requested amount, when funds are insufficient.

Solution:

```

#include <iostream>
using namespace std;

class InsufficientFunds {
public:
    double balance, amount;
    InsufficientFunds(double b, double a) : balance(b), amount(a) {}
};

class Account {
    double balance;
public:
    Account(double b) : balance(b) {}
    void withdraw(double amt) {
        if (amt > balance)
            throw InsufficientFunds(balance, amt);
        balance -= amt;
        cout << "Withdrawn " << amt << ", balance left = "
              << balance << endl;
    }
}

```

```

    }
};

int main() {
    Account acc(5000);
    try {
        acc.withdraw(2000);
        acc.withdraw(4000);
    }
    catch (const InsufficientFunds& e) {
        cout << "Insufficient funds! Balance = " << e.balance
            << ", requested = " << e.amount << endl;
    }
    return 0;
}

```

Output:

```

Withdrawn 2000, balance left = 3000
Insufficient funds! Balance = 3000, requested = 4000

```

Explanation:

InsufficientFunds is a user-defined class whose objects carry extra information (the current balance and the requested amount) from the point of the error to the handler. The first withdrawal succeeds and leaves 3000. The second asks for 4000, so the function throws, and the catch block reads the data members of the exception object to build a detailed message. This is a major advantage of throwing objects rather than plain values.

Problem 4. Write a C++ program to show what happens when an exception is thrown from a constructor. Which destructors are called?

Solution:

```

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

class Student {
    string name;
    int marks;
public:
    Student(string n, int m) : name(n), marks(m) {
        if (m < 0 || m > 100)
            throw m;
        cout << "Student " << name << " created, marks " << marks << endl;
    }
    ~Student() {
        cout << "Student " << name << " destroyed" << endl;
    }
};

int main() {
    try {
        Student s1("Amit", 85);
        Student s2("Riya", 150);    // constructor throws here
        cout << "This line is not reached" << endl;
    }
    catch (int m) {
        cout << "Invalid marks: " << m << endl;
    }
    return 0;
}

```

Output:

```
Student Amit created, marks 85
Student Amit destroyed
Invalid marks: 150
```

Explanation:

When the constructor of s2 throws, the object s2 is never fully constructed, so its destructor is not called. However, s1 was fully constructed, so during stack unwinding its destructor runs before the catch block executes, which is why "Student Amit destroyed" appears before the error message. The rule to remember: only fully constructed objects are destroyed during unwinding.

Problem 5. Write a C++ program that throws exceptions of types int, string, and double from the same function and handles them using a specific catch block and a catch-all block.

Solution:

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

void test(int choice) {
    try {
        if (choice == 1) throw 42;
        else if (choice == 2) throw string("text exception");
        else if (choice == 3) throw 2.5;
        cout << "No exception for choice " << choice << endl;
    }
    catch (int e) {
        cout << "int caught: " << e << endl;
    }
    catch (...) {
        cout << "Unknown exception caught by catch(...)" << endl;
    }
}

int main() {
    for (int i = 0; i <= 3; i++)
        test(i);
    return 0;
}
```

Output:

```
No exception for choice 0
int caught: 42
Unknown exception caught by catch(...)
Unknown exception caught by catch(...)
```

Explanation:

Choice 0 throws nothing, so the try block completes normally and no catch block runs. Choice 1 throws an int, which matches catch(int). Choices 2 and 3 throw a string and a double; neither matches catch(int), so both are handled by the catch(...) block, which accepts any type. Note that catch(...) is placed last, because it would otherwise intercept every exception before the specific handlers.

Problem 6. Write a C++ program to create a SafeArray class that overloads the subscript operator [] and throws an out_of_range exception for an invalid index.

Solution:

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

```
using namespace std;

class SafeArray {
    int data[5];
public:
    SafeArray() {
        for (int i = 0; i < 5; i++)
            data[i] = (i + 1) * 10;
    }
    int& operator[](int index) {
        if (index < 0 || index >= 5)
            throw out_of_range("Index out of bounds");
        return data[index];
    }
};

int main() {
    SafeArray arr;
    try {
        cout << "arr[2] = " << arr[2] << endl;
        arr[4] = 99;
        cout << "arr[4] = " << arr[4] << endl;
        int x = arr[7]; // invalid index
        cout << "arr[7] = " << x << endl;
    }
    catch (const out_of_range& e) {
        cout << "Exception: " << e.what() << endl;
    }
    return 0;
}
```

Output:

```
arr[2] = 30
arr[4] = 99
Exception: Index out of bounds
```

Explanation:

The overloaded operator[] checks the index before returning a reference to the element. Returning a reference lets the operator be used on the left side of an assignment, as in arr[4] = 99. The access arr[7] is invalid, so out_of_range is thrown, and the statements after it in the try block are skipped. This technique gives C++ arrays the bounds checking that built-in arrays lack.

Problem 7. Write a C++ program that converts strings to integers using stoi() and handles the invalid_argument and out_of_range exceptions.

Solution:

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

int main() {
    string inputs[] = {"123", "abc", "999999999999"};
    for (const string& s : inputs) {
        try {
            int n = stoi(s);
            cout << "\"" << s << "\" converted to " << n << endl;
        }
        catch (const invalid_argument&) {
            cout << "\"" << s << "\" is not a valid number" << endl;
        }
    }
}
```

```

    }
    catch (const out_of_range&) {
        cout << "\"" << s << "\" is too large for an int" << endl;
    }
}
return 0;
}

```

Output:

```

"123" converted to 123
"abc" is not a valid number
"999999999999" is too large for an int

```

Explanation:

The standard function `stoi()` reports errors by throwing exceptions. For the string "abc", no conversion is possible, so `invalid_argument` is thrown. For "999999999999", the number is valid but too large to fit in an `int`, so `out_of_range` is thrown. Since the try-catch is inside the loop, an error for one string does not stop the program from processing the remaining strings.

Problem 8. Write a C++ program to demonstrate the `bad_cast` exception using `dynamic_cast` on a reference.

Solution:

```

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

class Animal {
public:
    virtual ~Animal() {}
};
class Dog : public Animal {};
class Cat : public Animal {};

int main() {
    Dog d;
    Animal& ref = d;
    try {
        dynamic_cast<Dog&>(ref);
        cout << "Cast to Dog succeeded" << endl;
        dynamic_cast<Cat&>(ref);
        cout << "Cast to Cat succeeded" << endl;
    }
    catch (const bad_cast&) {
        cout << "bad_cast caught: object is not a Cat" << endl;
    }
    return 0;
}

```

Output:

```

Cast to Dog succeeded
bad_cast caught: object is not a Cat

```

Explanation:

`dynamic_cast` on a reference cannot return a null value to signal failure, so it throws `std::bad_cast` when the cast is invalid. The object is really a `Dog`, so the first cast succeeds. The second cast to `Cat` fails, and the `bad_cast` handler runs. For this to work, the base class must be polymorphic, which is why `Animal` has a virtual destructor, and the header `<typeinfo>` is required.

Problem 9. Write a C++ program to show what happens when an exception is not caught anywhere, and how `set_terminate()` can be used to customise the behaviour.

Solution:

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

void myTerminate() {
    cout << "Custom terminate handler: unhandled exception." << endl;
    exit(1);
}

int main() {
    set_terminate(myTerminate);
    cout << "Program started" << endl;
    throw 100;                // no catch block anywhere
    cout << "This line is not reached" << endl;
    return 0;
}
```

Output:

```
Program started
Custom terminate handler: unhandled exception.
```

Explanation:

The exception thrown in `main()` has no matching catch block, so the runtime calls `std::terminate()`. Because `set_terminate()` registered `myTerminate()` beforehand, that function is called instead of the default one. A terminate handler must end the program, so it calls `exit(1)` here. Without `set_terminate()`, the default behaviour would call `abort()` and the program would end abnormally. The statement after `throw` is never executed.

Problem 10. Write a C++ program with a function template `safeDivide()` that throws an exception when the divisor is zero, and use it with both `int` and `double` values.

Solution:

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

template <typename T>
T safeDivide(T a, T b) {
    if (b == 0)
        throw "Division by zero attempted!";
    return a / b;
}

int main() {
    try {
        int r1 = safeDivide(10, 4);
        cout << "10 / 4 (int) = " << r1 << endl;
        double r2 = safeDivide(10.0, 4.0);
        cout << "10.0 / 4.0 (double) = " << r2 << endl;
        double r3 = safeDivide(7.5, 0.0);
        cout << "7.5 / 0.0 (double) = " << r3 << endl;
    }
    catch (const char* msg) {
        cout << "Exception: " << msg << endl;
    }
}
```

```
    return 0;  
}
```

Output:

```
10 / 4 (int) = 2  
10.0 / 4.0 (double) = 2.5  
Exception: Division by zero attempted!
```

Explanation:

A single template function handles both int and double types, and the throw statement inside it works for every type. The first two calls succeed and print 2 and 2.5. The third call has a zero divisor, so the function throws a string literal, which is caught by catch(const char*). Note that integer division by zero would be undefined behaviour in C++, and floating-point division by zero gives infinity, so an explicit check followed by throw is the safe way to report this error.