

Object Oriented Programming C++

Chapter: Exception Handling

Exam-Oriented Coding Problems and Solutions (Set 6)

Coding Problems and Solutions

Problem 1. Write a C++ program to implement a circular queue of size 3 that throws user-defined exceptions when the queue is full or empty.

Solution:

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

class QueueFull {};
class QueueEmpty {};

class Queue {
    int arr[3];
    int front, rear, count;
public:
    Queue() : front(0), rear(0), count(0) {}
    void enqueue(int x) {
        if (count == 3)
            throw QueueFull();
        arr[rear] = x;
        rear = (rear + 1) % 3;
        count++;
    }
    int dequeue() {
        if (count == 0)
            throw QueueEmpty();
        int x = arr[front];
        front = (front + 1) % 3;
        count--;
        return x;
    }
};

int main() {
    Queue q;
    try {
        for (int i = 1; i <= 4; i++) {
            q.enqueue(i * 10);
            cout << "Enqueued " << i * 10 << endl;
        }
    }
    catch (const QueueFull&) {
        cout << "Queue is full" << endl;
    }
    try {
        for (int i = 0; i < 4; i++) {
            int x = q.dequeue();
            cout << "Dequeued " << x << endl;
        }
    }
    catch (const QueueEmpty&) {
        cout << "Queue is empty" << endl;
    }
}
```

```
    return 0;
}
```

Output:

```
Enqueued 10
Enqueued 20
Enqueued 30
Queue is full
Dequeued 10
Dequeued 20
Dequeued 30
Queue is empty
```

Explanation:

The Queue class uses front and rear indices that wrap around using the modulus operator, and a counter that tracks the number of elements. The fourth enqueue() finds the queue full and throws QueueFull, so its "Enqueued" message is never printed. After the three stored values are removed in first-in, first-out order, the fourth dequeue() finds the queue empty and throws QueueEmpty. The two exception classes carry no data because their type alone describes the error.

Problem 2. Write a C++ program to implement a class template Stack<T> that throws an exception on overflow and underflow, and use it with int and string types.

Solution:

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

template <typename T>
class Stack {
    T arr[3];
    int top;
public:
    Stack() : top(-1) {}
    void push(const T& x) {
        if (top == 2)
            throw runtime_error("Stack is full");
        arr[++top] = x;
    }
    T pop() {
        if (top == -1)
            throw runtime_error("Stack is empty");
        return arr[top--];
    }
};

int main() {
    Stack<int> si;
    Stack<string> ss;
    try {
        si.push(1);
        int a = si.pop();
        cout << "int stack popped: " << a << endl;
        int b = si.pop();           // stack is empty now
        cout << "int stack popped: " << b << endl;
    }
    catch (const runtime_error& e) {
        cout << "int stack error: " << e.what() << endl;
    }
}
```

```

    try {
        ss.push("A");
        ss.push("B");
        ss.push("C");
        ss.push("D");           // stack is full
    }
    catch (const runtime_error& e) {
        cout << "string stack error: " << e.what() << endl;
    }
    return 0;
}

```

Output:

```

int stack popped: 1
int stack error: Stack is empty
string stack error: Stack is full

```

Explanation:

A class template lets the same exception-throwing logic work for any data type, so the full and empty checks are written only once. The int stack pushes one value and pops it successfully; the second pop finds it empty, so runtime_error is thrown and the statement printing b is skipped. The string stack accepts three values, and the fourth push throws "Stack is full". The exception type is independent of the template parameter T.

Problem 3. Write a C++ program that defines a user-defined exception class derived from runtime_error and carrying an extra data member (the line number).

Solution:

```

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

class FileFormatError : public runtime_error {
    int line;
public:
    FileFormatError(const string& msg, int ln)
        : runtime_error(msg), line(ln) {}
    int getLine() const { return line; }
};

void parse(int lineNo, const string& text) {
    if (text.empty())
        throw FileFormatError("Empty line found", lineNo);
    cout << "Line " << lineNo << " OK: " << text << endl;
}

int main() {
    string lines[] = {"name=Amit", "", "age=20"};
    for (int i = 0; i < 3; i++) {
        try {
            parse(i + 1, lines[i]);
        }
        catch (const FileFormatError& e) {
            cout << "Format error: " << e.what()
                << " (line " << e.getLine() << ")" << endl;
        }
    }
    return 0;
}

```

Output:

```
Line 1 OK: name=Amit
Format error: Empty line found (line 2)
Line 3 OK: age=20
```

Explanation:

FileFormatError passes its message to the base class constructor `runtime_error(msg)`, so `what()` works without being overridden, and adds its own member line with the getter `getLine()`. The function `parse()` throws it for an empty line. The handler prints both the message and the line number. Deriving from a standard exception class is the recommended way to define custom exceptions, because the class can then also be caught by a handler for `runtime_error` or `exception`.

Problem 4. Write a C++ program to show how `unique_ptr` releases resources automatically when an exception is thrown.

Solution:

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

class Resource {
    int id;
public:
    Resource(int i) : id(i) {
        cout << "Resource " << id << " acquired" << endl;
    }
    ~Resource() {
        cout << "Resource " << id << " released" << endl;
    }
};

void work() {
    unique_ptr<Resource> r1 = make_unique<Resource>(1);
    unique_ptr<Resource> r2 = make_unique<Resource>(2);
    throw runtime_error("Something went wrong");
    cout << "Normal end of work()" << endl;
}

int main() {
    try {
        work();
    }
    catch (const exception& e) {
        cout << "Caught: " << e.what() << endl;
    }
    return 0;
}
```

Output:

```
Resource 1 acquired
Resource 2 acquired
Resource 2 released
Resource 1 released
Caught: Something went wrong
```

Explanation:

Each `unique_ptr` owns one `Resource` object and deletes it in its own destructor. When `work()` throws, stack unwinding destroys the local `unique_ptr` objects in reverse order of creation, so `Resource 2` is released first

and then Resource 1, before the catch block runs. The "Normal end" statement is never reached. No explicit delete or try-catch is needed inside work(). This is the RAII idea applied with smart pointers, and it makes leaks due to exceptions impossible for such resources.

Problem 5. Write a C++ program to compare the new operator with the nothrow form of new when memory allocation fails.

Solution:

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

int main() {
    const long long size = 1000000000000LL;    // about 4 TB of ints

    int* p = new (nothrow) int[size];
    if (p == nullptr) {
        cout << "nothrow new: returned nullptr" << endl;
    } else {
        cout << "nothrow new: allocated" << endl;
        delete[] p;
    }

    try {
        int* q = new int[size];
        cout << "normal new: allocated" << endl;
        delete[] q;
    }
    catch (const bad_alloc&) {
        cout << "normal new: bad_alloc thrown" << endl;
    }
    return 0;
}
```

Output:

```
nothrow new: returned nullptr
normal new: bad_alloc thrown
```

Explanation:

By default, new reports failure by throwing bad_alloc. The form new (nothrow) is the older style: it does not throw, but returns a null pointer, which the programmer must check explicitly. Both requests here ask for about 4 terabytes, which cannot be satisfied on a normal machine, so the first form gives a null pointer and the second throws bad_alloc, which is caught by reference. The header <new> is required for both bad_alloc and nothrow.

Problem 6. Write a C++ program for a simple calculator that throws a string for division by zero and a char for an unknown operator, with two different catch blocks.

Solution:

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

double calculate(double a, char op, double b) {
    switch (op) {
        case '+': return a + b;
        case '-': return a - b;
        case '*': return a * b;
        case '/':
            if (b == 0)
                throw "Division by zero";
            return a / b;
        default:
            throw op;
    }
}
```

```

        throw "Division by zero";
        return a / b;
    default:
        throw op;                // unknown operator thrown as a char
    }
}

int main() {
    struct Test { double a; char op; double b; };
    Test tests[] = { {8, '+', 2}, {8, '/', 0}, {8, '%', 2} };
    for (const Test& t : tests) {
        try {
            double r = calculate(t.a, t.op, t.b);
            cout << t.a << " " << t.op << " " << t.b << " = " << r << endl;
        }
        catch (const char* msg) {
            cout << "Math error: " << msg << endl;
        }
        catch (char op) {
            cout << "Unknown operator: " << op << endl;
        }
    }
    return 0;
}

```

Output:

```

8 + 2 = 10
Math error: Division by zero
Unknown operator: %

```

Explanation:

The calculate() function throws two different types of exception: a string literal (const char*) for division by zero, and the operator character itself (char) for an unsupported operator. Two catch blocks match these types separately. The first test case succeeds and prints $8 + 2 = 10$. The second raises the division error, and the third raises the unknown operator error for %. Throwing the offending character allows the handler to display exactly what was wrong.

Problem 7. Write a C++ program with a Date class whose constructor throws an exception for an invalid year, month or day, taking leap years into account.

Solution:

```

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

class Date {
    int d, m, y;
public:
    Date(int day, int month, int year) : d(day), m(month), y(year) {
        if (year < 1)
            throw invalid_argument("Invalid year");
        if (month < 1 || month > 12)
            throw invalid_argument("Invalid month");
        int days[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
        bool leap = (year % 4 == 0 && year % 100 != 0) || year % 400 == 0;
        int maxDay = days[month - 1];
        if (month == 2 && leap)
            maxDay = 29;
        if (day < 1 || day > maxDay)
            throw invalid_argument("Invalid day for the given month");
    }
}

```

```

    }
    void show() const {
        cout << d << "/" << m << "/" << y << endl;
    }
};

int main() {
    int tests[4][3] = { {29, 2, 2024}, {29, 2, 2023},
                        {15, 13, 2022}, {10, 5, 2021} };
    for (auto& t : tests) {
        try {
            Date dt(t[0], t[1], t[2]);
            cout << "Valid date: ";
            dt.show();
        }
        catch (const invalid_argument& e) {
            cout << t[0] << "/" << t[1] << "/" << t[2]
                << " -> " << e.what() << endl;
        }
    }
    return 0;
}

```

Output:

```

Valid date: 29/2/2024
29/2/2023 -> Invalid day for the given month
15/13/2022 -> Invalid month
Valid date: 10/5/2021

```

Explanation:

The constructor validates the year, then the month, then the day against the number of days in that month, using 29 days for February in leap years. A year is a leap year if it is divisible by 4 and not by 100, or if it is divisible by 400. Hence 29/2/2024 is valid but 29/2/2023 is not, and month 13 is rejected. Throwing from the constructor means that an invalid Date object can never exist.

Problem 8. Write a C++ program that uses catch(...) to release dynamically allocated memory and then rethrows the same exception to the caller.

Solution:

```

#include <iostream>
using namespace std;

void process() {
    int* data = new int[100];
    cout << "Memory allocated" << endl;
    try {
        throw "Processing failed";
    }
    catch (...) {
        delete[] data;           // cleanup for ANY exception
        cout << "Memory released in catch(...)" << endl;
        throw;                   // pass the same exception on
    }
}

int main() {
    try {
        process();
    }
    catch (const char* msg) {

```

```

        cout << "main handled: " << msg << endl;
    }
    return 0;
}

```

Output:

```

Memory allocated
Memory released in catch(...)
main handled: Processing failed

```

Explanation:

The function `process()` allocates memory and then enters a try block in which an exception is thrown. The handler `catch(...)` matches any exception type, deletes the array, and then executes `throw;` (with no operand) to pass the original exception on unchanged. The caller `main()` receives the original `const char*` exception and handles it. This pattern, cleanup in a catch-all followed by a rethrow, was widely used before RAII classes became the preferred approach.

Problem 9. Write a C++ program to perform a linear search on an array and throw an exception, carrying the searched value, when the element is not found.

Solution:

```

#include <iostream>
using namespace std;

class NotFound {
public:
    int value;
    NotFound(int v) : value(v) {}
};

int linearSearch(const int arr[], int n, int key) {
    for (int i = 0; i < n; i++)
        if (arr[i] == key)
            return i;
    throw NotFound(key);
}

int main() {
    int arr[] = {15, 8, 23, 4, 42};
    int keys[] = {23, 99};
    for (int k : keys) {
        try {
            int pos = linearSearch(arr, 5, k);
            cout << k << " found at index " << pos << endl;
        }
        catch (const NotFound& e) {
            cout << e.value << " not found in the array" << endl;
        }
    }
    return 0;
}

```

Output:

```

23 found at index 2
99 not found in the array

```

Explanation:

Instead of returning a special value such as -1 for "not found", the function throws a `NotFound` object that contains the key. This forces the caller to deal with the failure, and the error can no longer be mistaken for a

valid index. The key 23 is found at index 2, while 99 causes the exception, and the handler reads the value from the exception object to print a specific message.

Problem 10. Write a C++ program to show how handlers for `logic_error`, `exception` and `catch(...)` respond to different standard exception types, and explain the order of the handlers.

Solution:

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

void raise(int n) {
    if (n == 1) throw out_of_range("out_of_range thrown");
    if (n == 2) throw invalid_argument("invalid_argument thrown");
    if (n == 3) throw overflow_error("overflow_error thrown");
    if (n == 4) throw 99;
}

int main() {
    for (int n = 1; n <= 4; n++) {
        try {
            raise(n);
        }
        catch (const logic_error& e) {
            cout << "logic_error handler: " << e.what() << endl;
        }
        catch (const exception& e) {
            cout << "exception handler: " << e.what() << endl;
        }
        catch (...) {
            cout << "catch-all handler: unknown type" << endl;
        }
    }
    return 0;
}
```

Output:

```
logic_error handler: out_of_range thrown
logic_error handler: invalid_argument thrown
exception handler: overflow_error thrown
catch-all handler: unknown type
```

Explanation:

Both `out_of_range` and `invalid_argument` are derived from `logic_error`, so the first handler catches them. `overflow_error` is derived from `runtime_error`, not from `logic_error`, so it falls through to the handler for the common base class `exception`. The integer 99 is not derived from anything, so only `catch(...)` can handle it. The handlers are written from the most specific to the most general; reversing the first two would make the `logic_error` handler unreachable, since `exception` would catch everything first.