

Object Oriented Programming C++

Chapter: Exception Handling

Exam-Oriented Questions and Answers

Section A: Short Questions and Answers

Q1. What is an exception in C++?

Answer:

An exception is an abnormal or unexpected condition that arises during program execution, such as division by zero or an invalid array index, which disrupts the normal flow of the program.

Q2. Name the three keywords used in C++ exception handling.

Answer:

try, throw, and catch.

Q3. What is the purpose of a try block?

Answer:

A try block encloses the code that is to be monitored for exceptions; if an exception occurs inside it, control is transferred to a matching catch block.

Q4. How is an exception raised in C++?

Answer:

An exception is raised using the throw statement, which can throw a value of any built-in type or an object of a user-defined class.

Q5. What does catch(...) mean?

Answer:

catch(...) is a generic catch block that can catch any type of exception, regardless of its data type, and is usually placed as the last handler.

Q6. What happens if no catch block matches a thrown exception?

Answer:

The special function terminate() is called, which by default invokes abort() and ends the program abnormally.

Q7. Can a try block have multiple catch blocks?

Answer:

Yes, a single try block can be followed by multiple catch blocks, each designed to handle a different type of exception.

Q8. In what order are catch blocks checked when an exception is thrown?

Answer:

They are checked in the order they are written, from top to bottom, and the first matching handler is executed.

Q9. What is meant by rethrowing an exception?

Answer:

Rethrowing means passing an already-caught exception further up the call stack using the statement throw; (without an operand), so an outer handler can process it.

Q10. What is stack unwinding?

Answer:

Stack unwinding is the process by which the runtime system destroys local objects and exits function calls one by one while searching for a matching catch handler after an exception is thrown.

Q11. Why should a catch block for a derived class be placed before the catch block for its base class?

Answer:

Because catch blocks are checked in order, a base class catch block placed first would match derived class objects too, so the more specific (derived class) handler would never be reached; hence derived class handlers must come first.

Q12. Give one limitation of exception handling in C++.

Answer:

Exception handling introduces some runtime overhead and, if overused for ordinary control flow instead of genuinely exceptional conditions, can make the program logic harder to follow.

Section B: Broad Questions and Answers

Q1. Explain the need for exception handling in C++ with a real-life type of example.

Answer:

In any program that performs operations such as division, array access, memory allocation, or file handling, certain runtime errors cannot always be prevented in advance. For instance, a program that divides two numbers entered by the user cannot know in advance whether the user will enter zero as the divisor. Without exception handling, such an error would either be checked manually with repeated if conditions scattered throughout the code, or it would cause the program to crash. Exception handling allows the programmer to place the risky operation inside a try block and write a separate catch block that specifically deals with the error, keeping the normal logic of the program clean and separate from the error-handling logic. This makes programs more robust, more readable, and easier to maintain, especially in large software systems where many different operations can fail in different ways.

Q2. Describe, with an example, how multiple catch blocks work together to handle different exception types thrown from the same try block.

Answer:

When a try block can potentially throw more than one type of exception, multiple catch blocks are written immediately after it, each specifying a different exception type in its parameter. For example, a try block might throw an int in one situation and a string in another; a catch(int e) block would then be written to handle the integer case, and a separate catch(const string& e) block would handle the string case. When an exception is thrown, the compiler examines each catch block in the order it appears and executes the first one whose type matches the thrown exception's type exactly (or, for class types, matches the base class of the thrown object). If none of the specific catch blocks match, a final catch(...) block, if present, handles the exception generically. This mechanism allows a single try block to safely manage several distinct failure scenarios without needing separate try blocks for each one.

Q3. What is stack unwinding? Explain its significance in resource management during exception handling.

Answer:

Stack unwinding is the process that takes place between the point where an exception is thrown and the point where it is finally caught. As control leaves the try block in search of a matching catch handler, every function call on the path is exited in turn, and for each of these function calls, the destructors of all local (automatic) objects that were successfully constructed are invoked before the function's stack frame is removed. This is significant because it ensures that resources such as dynamically allocated memory (when wrapped in objects with destructors), open file handles, or acquired locks are properly released even when an error interrupts the normal flow of execution. This behaviour is the basis of the RAII (Resource Acquisition Is Initialization) technique commonly used in C++ to write exception-safe code.

Q4. Discuss the limitations of exception handling in C++ and how a programmer should decide when to use it.

Answer:

While exception handling is a powerful mechanism, it is not free of drawbacks. It can introduce runtime overhead, particularly at the moment an exception is thrown, since the system must search the call stack for a suitable handler. It is also not intended for routine control flow; using exceptions to manage expected, everyday conditions (such as reaching the end of a loop) rather than genuinely exceptional situations tends to make code harder to read and reason about. In large programs, careless or excessive use of try-catch blocks and unclear exception class hierarchies can make the flow of error handling difficult to trace. Furthermore,

catching an exception only prevents abrupt termination at that point; it does not by itself guarantee that the underlying problem has been resolved, and if not paired with careful resource management, leaks can still occur. As a general guideline, exception handling should be reserved for truly exceptional, unpredictable runtime errors, while predictable and frequently occurring conditions are better handled with ordinary conditional logic.

Q5. Explain, with an example, why the order of catch blocks matters when exception classes are related by inheritance.

Answer:

When exception classes participate in an inheritance hierarchy, a catch block written for a base class type can also catch objects of any class derived from it, because a derived class reference or object can be treated as a base class reference. If such a base class catch block is placed before the catch blocks for its derived classes, it will intercept exceptions of the derived types as well, and the more specific catch blocks written afterward will never execute, since only the first matching handler is used. For example, if class `NetworkError` is a base class and `TimeoutError` is derived from it, placing `catch(NetworkError&)` before `catch(TimeoutError&)` means that a thrown `TimeoutError` object will always be handled by the `NetworkError` handler, hiding the specialised handling meant for timeouts. To avoid this, catch blocks for derived classes must always be placed before the catch block for their common base class.

Q6. Compare exception handling with traditional error-handling techniques such as return codes and error flags.

Answer:

Traditional error handling typically relies on a function returning a special value (such as -1 or NULL) or setting a global error flag to indicate that something went wrong, requiring the calling code to check this value after every single call. This approach tends to clutter the program with repetitive checks, and it is easy for a programmer to forget to check a return value, allowing an error to pass unnoticed. Exception handling, in contrast, allows the point where an error is detected to be separated from the point where it is handled; if an error is not caught locally, it automatically propagates up the call stack until a suitable handler is found, without requiring every intermediate function to explicitly check for it. This generally leads to cleaner code for the normal case, along with a guarantee that an unhandled error cannot be silently ignored, since it will eventually cause the program to terminate if no handler exists. The trade-off is the runtime overhead and added complexity that exception handling can introduce if used carelessly.

Section C: Coding Questions and Answers

Q1. Write a C++ program that uses a try-catch block to handle a division-by-zero condition.

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

int main() {
    int a, b;
    cout << "Enter two integers: ";
    cin >> a >> b;
    try {
        if (b == 0)
            throw "Division by zero is not allowed!";
        cout << "Result: " << (a / b) << endl;
    }
    catch (const char* msg) {
        cout << "Exception caught: " << msg << endl;
    }
    return 0;
}
```

Answer:

The program reads two integers and attempts their division inside a try block. Before performing the division, it checks whether the divisor is zero; if so, it throws a string literal describing the error. The catch(const char* msg) block matches this thrown type and prints the error message, preventing the program from crashing due to an invalid division.

Q2. Write a C++ program with multiple catch blocks that handles int, char, and double exceptions thrown from the same try block, based on user input.

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

int main() {
    int choice;
    cout << "Enter choice (1-int, 2-char, 3-double): ";
    cin >> choice;
    try {
        if (choice == 1) throw 100;
        else if (choice == 2) throw 'X';
        else if (choice == 3) throw 3.14;
        else throw choice;
    }
    catch (int e) {
        cout << "Caught int exception: " << e << endl;
    }
    catch (char e) {
        cout << "Caught char exception: " << e << endl;
    }
    catch (double e) {
        cout << "Caught double exception: " << e << endl;
    }
    return 0;
}
```

Answer:

Depending on the value entered by the user, the try block throws an exception of a different type: int, char, or double. Each catch block is written to match one specific type, and only the block whose parameter type

matches the thrown value's type is executed. This demonstrates how a single try block can be linked to several catch blocks to deal with different kinds of errors.

Q3. Write a C++ program that defines a custom exception class derived from `std::exception` and uses it to report an invalid age value.

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

class InvalidAgeException : public exception {
public:
    const char* what() const noexcept override {
        return "Invalid age: age must be between 0 and 120.";
    }
};

void checkAge(int age) {
    if (age < 0 || age > 120)
        throw InvalidAgeException();
    cout << "Age " << age << " is valid." << endl;
}

int main() {
    int age;
    cout << "Enter age: ";
    cin >> age;
    try {
        checkAge(age);
    }
    catch (const exception& e) {
        cout << "Exception: " << e.what() << endl;
    }
    return 0;
}
```

Answer:

InvalidAgeException is a user-defined class derived from the standard exception class, and it overrides the `what()` function to return a descriptive error message. The `checkAge()` function throws an object of this class whenever an invalid age is passed. In `main()`, the catch block accepts a reference to the base class exception, which is able to catch the derived `InvalidAgeException` object as well, and calls `what()` to display the appropriate message.

Q4. Write a C++ program that demonstrates rethrowing an exception from an inner catch block to an outer catch block.

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

void innerFunction() {
    try {
        throw runtime_error("Error occurred in innerFunction");
    }
    catch (const runtime_error& e) {
        cout << "Logged in innerFunction: " << e.what() << endl;
        throw; // rethrow the same exception
    }
}

int main() {
    try {
```

```

        innerFunction();
    }
    catch (const runtime_error& e) {
        cout << "Handled in main: " << e.what() << endl;
    }
    return 0;
}

```

Answer:

The innerFunction() throws a runtime_error, which is first caught within its own try-catch block, allowing it to log the error locally. The statement throw; (with no operand) then rethrows the same exception object, letting it propagate to the caller. In main(), this rethrown exception is caught again by an outer catch block, which performs the final handling. This pattern is often used when a function needs to perform local cleanup or logging before letting the error be handled at a higher level.

Q5. Write a C++ program that demonstrates stack unwinding by showing that destructors of local objects are called when an exception propagates.

```

#include <iostream>
using namespace std;

class Demo {
    string name;
public:
    Demo(string n) : name(n) {
        cout << "Constructing " << name << endl;
    }
    ~Demo() {
        cout << "Destructing " << name << endl;
    }
};

void functionB() {
    Demo obj2("obj2 in functionB");
    throw runtime_error("Error thrown in functionB");
}

void functionA() {
    Demo obj1("obj1 in functionA");
    functionB();
}

int main() {
    try {
        functionA();
    }
    catch (const runtime_error& e) {
        cout << "Caught in main: " << e.what() << endl;
    }
    return 0;
}

```

Answer:

When functionB() throws an exception, control must leave both functionB() and functionA() before reaching the catch block in main(). During this process, the destructors of obj2 and obj1 are automatically called in reverse order of their construction, even though neither function contains a catch block of its own. This output demonstrates stack unwinding in action: local objects are always properly destroyed as the stack is unwound, regardless of where the exception is ultimately caught.