

Object Oriented Programming C++

Chapter: Exception Handling

Exam-Oriented Questions and Answers (Set 2)

Section A: Short Questions and Answers

Q1. Define exception handling in C++.

Answer:

Exception handling is a mechanism provided by C++ to detect runtime errors at the point where they occur and transfer control to a special block of code (the handler) that deals with them, using the keywords try, throw, and catch.

Q2. Differentiate between an error and an exception.

Answer:

An error (such as a syntax error) is usually detected at compile time and must be fixed before the program can run, whereas an exception is an abnormal runtime condition, such as division by zero or failure to allocate memory, which can be detected and handled while the program is executing.

Q3. What happens to the statements that follow a throw statement inside a try block?

Answer:

They are skipped. As soon as an exception is thrown, control leaves the try block immediately and is transferred to the matching catch block.

Q4. Can an object of a user-defined class be thrown as an exception?

Answer:

Yes. C++ allows any type to be thrown, including objects of user-defined classes, which is useful for carrying detailed error information to the handler.

Q5. What is a nested try block?

Answer:

A nested try block is a try block placed inside another try block. If the inner block cannot handle a thrown exception, the exception is passed to the handlers of the outer try block.

Q6. What is the purpose of the what() function?

Answer:

what() is a virtual member function of the standard exception class that returns a C-style string describing the exception. Derived classes override it to provide a meaningful error message.

Q7. Which exception is thrown when the new operator fails to allocate memory?

Answer:

The new operator throws an exception of type std::bad_alloc (declared in the <new> header) when the requested memory cannot be allocated.

Q8. Why is it recommended to catch exceptions by reference?

Answer:

Catching by reference avoids making an unnecessary copy of the exception object and prevents object slicing, so that a derived class object caught through a base class reference retains its derived behaviour (for example, a virtual what() call).

Q9. Does C++ provide a finally block like Java?

Answer:

No. C++ does not have a finally block. Cleanup is instead done through destructors of local objects, which are automatically called during stack unwinding (the RAII technique).

Q10. Is it compulsory to give a name to the parameter of a catch block?

Answer:

No. If the handler does not need to use the thrown value, the parameter name can be omitted, for example `catch(int)` or `catch(MyException&)`.

Q11. What is the use of the noexcept specifier?

Answer:

The noexcept specifier declares that a function is not expected to throw any exception. If such a function does throw one, the program calls `std::terminate()`.

Q12. Write the difference between throw and return.

Answer:

`return` gives control back to the immediate caller of a function with a value, while `throw` transfers control to the nearest matching catch handler, which may be many function calls up the call stack, and unwinds the stack on the way.

Section B: Broad Questions and Answers

Q1. Explain the flow of control in the exception handling mechanism of C++ step by step.

Answer:

The exception handling mechanism works in a well-defined sequence. First, the program enters a try block and executes its statements normally. If no exception occurs, the try block completes and all the catch blocks following it are skipped. If an exception is detected, a throw statement is executed, which creates an exception object of the thrown type. Control then immediately leaves the try block, and any local objects created inside it are destroyed. The runtime system next examines the catch blocks associated with that try block, one by one in the order written, and selects the first one whose parameter type matches the type of the thrown object. The statements inside that catch block are executed, and after the catch block finishes, execution continues with the first statement following the last catch block of that try-catch group. If no catch block matches, the search continues in the enclosing try blocks and then in the calling functions through stack unwinding; if still no handler is found, `std::terminate()` is called and the program ends abnormally.

Q2. What are nested try blocks? Explain how an exception is propagated from an inner try block to an outer one.

Answer:

A try block can contain another try block within it, and such an arrangement is called a nested try block. This is useful when different parts of a code section can fail in different ways and some failures should be handled locally while others should be handled at a higher level. When an exception is thrown inside the inner try block, the runtime first looks for a matching catch block attached to the inner try block. If one is found, the exception is handled there and execution continues after the inner try-catch group. If none matches, the exception is propagated outward, and the catch blocks of the enclosing outer try block are examined in the same manner. Local objects of the inner block are destroyed during this propagation. If the outer handlers also fail to match, the exception continues up the call stack until a handler is found or the program terminates. A catch block can also deliberately pass an exception to an outer handler by rethrowing it with the statement `throw;`.

Q3. Describe the standard exception classes provided by C++ and their hierarchy.

Answer:

The C++ standard library provides a set of predefined exception classes, all of which are derived from the base class `std::exception`, declared in the `<exception>` header. This base class provides the virtual function `what()`, which returns a description of the error. Two important direct subclasses declared in `<stdexcept>` are `logic_error`, representing errors that could in principle be avoided by correct program logic (for example `invalid_argument`, `out_of_range`, and `length_error`), and `runtime_error`, representing errors that can only be detected while the program runs (for example `overflow_error` and `underflow_error`). Other standard exceptions derived directly from `exception` include `bad_alloc` (thrown by `new` on allocation failure), `bad_cast` (thrown by `dynamic_cast` on an invalid reference cast), and `bad_typeid`. Because these classes share a common base, a single catch block written for `std::exception` can handle any of them, while more specific catch blocks can be placed before it for finer control.

Q4. Compare catching an exception by value, by pointer, and by reference. Which is preferred and why?

Answer:

When a handler catches an exception by value, a copy of the exception object is made; this is costly for large objects and, if a derived class object is caught through a base class parameter, the derived part is cut off (object slicing), so overridden virtual functions such as `what()` will not behave as expected. When caught by

pointer, the thrown expression must itself be a pointer, which raises the question of who is responsible for deleting the object and risks memory leaks or dangling pointers if the object was a local variable. Catching by reference avoids copying, preserves the true dynamic type of the object so that virtual functions work correctly, and requires no manual memory management. For these reasons, the widely recommended practice is to throw objects by value and catch them by (const) reference, as in `catch(const exception& e)`.

Q5. Explain the role of `terminate()`, `set_terminate()`, and the `noexcept` specifier in C++ exception handling.

Answer:

The function `std::terminate()` is called by the runtime when exception handling cannot proceed, for example when a thrown exception has no matching handler, or when an exception escapes from a function declared `noexcept`. By default, `terminate()` calls `abort()`, which ends the program immediately without completing normal cleanup. The function `set_terminate()` allows the programmer to register a custom handler function that is called instead of the default behaviour, usually to log diagnostic information or perform minimal cleanup before the program ends; however, such a handler must still end the program and cannot resume normal execution. The `noexcept` specifier is placed after a function's parameter list to promise that the function will not throw exceptions, which lets the compiler optimise and lets callers rely on that guarantee; if the promise is broken at runtime, `terminate()` is invoked.

Q6. What is exception safety? Explain how RAII helps in writing exception-safe programs.

Answer:

A piece of code is called exception-safe if it continues to behave correctly, without leaking resources or leaving data in an inconsistent state, even when an exception is thrown while it is executing. The main danger arises when a function acquires a resource such as dynamically allocated memory, a file handle, or a lock, and an exception occurs before the code that releases it is reached. RAII (Resource Acquisition Is Initialization) addresses this by tying the lifetime of a resource to the lifetime of an object: the resource is acquired in the object's constructor and released in its destructor. Because destructors of fully constructed local objects are guaranteed to be called during stack unwinding, the resource is released automatically no matter how the scope is exited, whether normally or by an exception. Standard library classes such as `unique_ptr`, `shared_ptr`, `vector`, and `fstream` are built on this principle, which is why they are preferred over raw pointers and manual cleanup in exception-prone code.

Section C: Coding Questions and Answers

Q1. Write a C++ program to demonstrate nested try blocks where an exception thrown in the inner block is handled by the outer block.

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

int main() {
    try {
        cout << "Outer try block" << endl;
        try {
            cout << "Inner try block" << endl;
            throw 'E';          // char exception
        }
        catch (int e) {          // does not match char
            cout << "Inner catch (int): " << e << endl;
        }
        cout << "This line is not executed" << endl;
    }
    catch (char e) {
        cout << "Outer catch (char): " << e << endl;
    }
    return 0;
}
```

Output:

```
Outer try block
Inner try block
Outer catch (char): E
```

Answer:

The inner try block throws a char exception. The only inner handler accepts an int, so it does not match, and the exception is passed to the outer try block. The outer catch(char) block matches and handles it. Note that the statement after the inner try-catch group is skipped, because control left that region while the exception was propagating.

Q2. Write a C++ program to handle memory allocation failure using the bad_alloc exception.

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

int main() {
    try {
        // request an unreasonably large amount of memory
        long long* p = new long long[1000000000000LL];
        cout << "Memory allocated successfully" << endl;
        delete[] p;
    }
    catch (const bad_alloc& e) {
        cout << "Allocation failed: " << e.what() << endl;
    }
    return 0;
}
```

Output:

```
Allocation failed: std::bad_alloc (exact text depends on the compiler)
```

Answer:

The program requests a very large block of memory (about 8 terabytes) using the new operator. On a typical system this request cannot be satisfied, so new throws a std::bad_alloc exception instead of returning a null pointer. The catch block receives it by reference and prints the message returned by what(), so the program ends cleanly instead of crashing.

Q3. Write a C++ program that uses the at() function of a vector and handles the out_of_range exception.

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

int main() {
    vector<int> v = {10, 20, 30};
    try {
        cout << "v[1] = " << v.at(1) << endl;
        cout << "v[5] = " << v.at(5) << endl;    // invalid index
    }
    catch (const out_of_range& e) {
        cout << "Exception caught: " << e.what() << endl;
    }
    return 0;
}
```

Output:

```
v[1] = 20
Exception caught: (message describing the invalid index; wording depends on the
compiler)
```

Answer:

Unlike the subscript operator [], the at() function checks the index and throws a std::out_of_range exception if it is invalid. The first access is valid and prints 20. The second access uses index 5, which is beyond the size of the vector, so at() throws, control leaves the try block, and the catch block prints the error description. This is a convenient way to make array-style access safe.

Q4. Write a C++ program to implement a stack of size 3 that throws user-defined exceptions for overflow and underflow conditions.

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

class StackOverflow {};
class StackUnderflow {};

class Stack {
    int arr[3];
    int top;
public:
    Stack() : top(-1) {}
    void push(int x) {
        if (top == 2) throw StackOverflow();
        arr[++top] = x;
    }
    int pop() {
        if (top == -1) throw StackUnderflow();
        return arr[top--];
    }
};
```

```

int main() {
    Stack s;
    try {
        s.push(10);
        s.push(20);
        s.push(30);
        s.push(40);           // stack is full
    }
    catch (StackOverflow&) {
        cout << "Stack Overflow!" << endl;
    }

    try {
        for (int i = 0; i < 4; i++)
            cout << "Popped: " << s.pop() << endl;
    }
    catch (StackUnderflow&) {
        cout << "Stack Underflow!" << endl;
    }
    return 0;
}

```

Output:

```

Stack Overflow!
Popped: 30
Popped: 20
Popped: 10
Stack Underflow!

```

Answer:

Two empty classes, StackOverflow and StackUnderflow, are used purely as exception types. The fourth push() call finds the stack full and throws StackOverflow, which is caught in the first try-catch group. In the second group, three pop() calls succeed, and the fourth finds the stack empty and throws StackUnderflow. This shows how user-defined exception classes make error conditions in a data structure explicit and easy to handle.

Q5. Write a C++ program to show why a derived class handler must be placed before its base class handler.

```

#include <iostream>
using namespace std;

class Base {};
class Derived : public Base {};

int main() {
    try {
        throw Derived();
    }
    catch (Derived&) {           // specific handler first
        cout << "Caught Derived" << endl;
    }
    catch (Base&) {             // general handler afterwards
        cout << "Caught Base" << endl;
    }
    return 0;
}

```

Output:

```

Caught Derived

```

Answer:

A Derived object is thrown, and the catch blocks are tested in the order written, so the catch(Derived&) handler matches first and is executed. If the two catch blocks were written in the opposite order, catch(Base&) would match the Derived object first (since a derived object can be treated as a base object), and the output would be Caught Base; the Derived handler would then never run, and most compilers would warn about this unreachable handler.