Object Oriented Programming C++

Chapter: Exception Handling

1. Concepts and Uses of Exception Handler

An exception is an abnormal condition or unexpected event that arises during the execution of a program, such as division by zero, an invalid array index, running out of memory, or an attempt to open a file that does not exist. In traditional programming, such errors were handled using return codes, global error flags, or conditional checks scattered throughout the code, which made programs harder to read and maintain. C++ provides a structured mechanism called exception handling to detect and respond to such runtime errors in a clean and organised manner, separating the normal logic of a program from the error-handling logic.

An exception handler is a block of code specifically designed to deal with a particular type of exception. When an error condition occurs, the normal flow of the program is interrupted, and control is transferred to the nearest matching exception handler. This allows the programmer to detect an error at the point where it occurs and handle it at a point where enough information is available to deal with it meaningfully, even if that point is in a completely different function.

Uses of exception handling include:

- Separating error-detection code from error-handling code, which improves readability and maintainability.
- Allowing errors to be handled at an appropriate level in the program, rather than forcing every function to check for every possible error.
- Preventing abrupt and uncontrolled termination of a program by providing a way to recover from, or gracefully exit after, an error.
- Enabling robust handling of runtime errors in large software systems, such as invalid user input, resource allocation failures, and file I/O errors.
- Allowing a single handler to manage exceptions raised from multiple points in a program or even from library code.

2. The try / throw / catch Construct

C++ implements exception handling using three keywords: try, throw, and catch. Together, these keywords form the basic construct used to detect, signal, and handle exceptions.

2.1 The try block

The try block encloses the section of code that is being monitored for exceptions. Any statement inside a try block that may cause an error is watched during execution; if an exception occurs, control immediately leaves the try block and is transferred to a matching catch block. A try block must be followed by one or more catch blocks.

2.2 The throw statement

The throw statement is used to signal that an exception has occurred. It can throw an object of any type, including built-in types such as int and char, string literals, or objects of user-defined classes. Throwing an exception immediately transfers control out of the current block and begins the search for a matching catch handler.

2.3 The catch block

A catch block defines the handler for a specific type of exception. It is written immediately after a try block and specifies the type of exception it can handle, along with the code to execute when that exception is caught. A catch block that uses the ellipsis catch(...) can catch any type of exception, and is often used as a default or final handler.

2.4 General syntax and example

```
try {
    // code that may throw an exception
    if (b == 0)
        throw "Division by zero!";
    result = a / b;
}
catch (const char* msg) {
    cout << "Error: " << msg << endl;
}
```

In this example, the division inside the try block is monitored. If the divisor b is zero, a string literal is thrown as an exception. Control immediately jumps out of the try block, skipping the remaining statements, and is passed to the catch block whose parameter type (const char*) matches the type of the thrown value.

3. Uses and Implementation of Multiple Exceptions

A single try block can be associated with more than one catch block, allowing a program to handle several different types of exceptions that may arise from the same section of code. When an exception is thrown, the compiler checks the catch blocks in the order in which they are written, from top to bottom, and executes the first one whose parameter type matches the type of the thrown exception. If no match is found among the specific catch blocks, a catch(...) block, if present, will handle it as a default case.

This is particularly useful in programs where different operations within the same try block can fail for different reasons, and each type of failure needs to be reported or handled differently.

```
try {
    int choice;
    cin >> choice;
    if (choice == 1) throw 10;              // int exception
    if (choice == 2) throw 'A';              // char exception
    if (choice == 3) throw 3.14;             // double exception
    if (choice == 4) throw string("error"); // string exception
}
catch (int e) {
    cout << "Caught an int exception: " << e << endl;
}
catch (char e) {
    cout << "Caught a char exception: " << e << endl;
}
catch (double e) {
    cout << "Caught a double exception: " << e << endl;
}
catch (...) {
    cout << "Caught an unknown exception!" << endl;
}
```

Points to remember while implementing multiple exceptions:

- The order of catch blocks matters: a catch block for a base class must be placed after catch blocks for its derived classes, otherwise the base class handler will catch objects meant for the derived handlers.
- A catch(...) block, if used, should generally be placed last, since it matches every exception type.
- An exception can be rethrown from within a catch block using the statement throw; (with no operand), which passes the same exception further up the call stack to be handled by an outer handler.
- It is also possible to nest try blocks, where an inner try block is placed within an outer try block, allowing localized handling of certain exceptions while letting others propagate outward.
- As control passes out of a try block in search of a handler, C++ performs stack unwinding, during which the destructors of all local objects that go out of scope are automatically called, ensuring proper cleanup of resources.

4. Limitations of Exception Handling

Although exception handling is a powerful and structured mechanism for dealing with runtime errors, it has certain limitations that programmers should be aware of:

- Runtime overhead: Exception handling can add a small amount of overhead to program execution, particularly when exceptions are thrown, since the runtime system must search the call stack for a matching handler.
- Not suited for ordinary control flow: Exceptions are meant for exceptional, unexpected conditions; using them for routine control flow (such as loop termination) is considered poor practice and can make code harder to follow.
- Complexity in large programs: Overuse of exceptions, deeply nested try-catch blocks, or unclear exception hierarchies can make a program's error-handling logic difficult to trace and maintain.
- No guarantee of recovery: Catching an exception only means that abnormal termination is prevented at that point; it does not automatically fix the underlying problem or guarantee that the program can continue meaningfully.
- Resource management concerns: If exceptions are not handled carefully alongside proper resource management techniques (such as RAII), resource leaks can still occur despite the use of exception handling.
- Unhandled exceptions terminate the program: If a thrown exception does not match any catch block in the entire call stack, the special function terminate() is called, which by default calls abort() and ends the program abruptly.
- Cannot handle synchronous hardware-level errors uniformly: Certain low-level errors (such as some hardware interrupts) are not naturally expressed as C++ exceptions and may require separate handling mechanisms.

Multiple Choice Questions (with Answer Key)

1. Which keyword is used to raise an exception explicitly in C++?

(a) catch
(b) throw
(c) try
(d) raise

2. The block of code that monitors for exceptions is written inside:

(a) a catch block
(b) a throw block
(c) a try block
(d) a handler block

3. A catch block that can handle any type of exception is written as:

(a) catch(all)
(b) catch(...)
(c) catch(void)
(d) catch(any)

4. If no matching catch block is found for a thrown exception, the program calls:

(a) main()
(b) terminate()
(c) exit(0)
(d) abort() only

5. In C++, exception handling separates:

(a) data from functions
(b) error-detection code from error-handling code
(c) classes from objects
(d) declaration from definition

6. Multiple catch blocks following a single try block are called:

(a) catch chain
(b) catch sequence
(c) catch stack
(d) overloaded catch

7. When an exception is thrown, the search for a matching catch handler proceeds:

(a) from bottom to top
(b) in the order the catch blocks are written, top to bottom
(c) in random order
(d) only in the nearest function

8. A catch block that takes a base class reference can also catch:

(a) only base class objects
(b) only derived class objects
(c) objects of the base class and any of its derived classes
(d) no objects at all

9. Which of the following can be thrown as an exception in C++?

(a) only int and char
(b) only objects of classes
(c) any data type, including built-in types and objects

(d) only pointers

10. What happens to a thrown exception if the current function does not catch it?

(a) The program terminates immediately with no further action
(b) It is propagated up the call stack to the caller
(c) It is silently ignored
(d) It is automatically converted to a warning

11. The standard exception class hierarchy in C++ is rooted at:

(a) std::error
(b) std::exception
(c) std::runtime
(d) std::throwable

12. Stack unwinding during exception handling refers to:

(a) compressing the stack memory
(b) destroying local objects as the stack frames are exited while searching for a handler
(c) increasing stack size dynamically
(d) converting the stack into a heap

13. Rethrowing a caught exception inside a catch block is done using:

(a) throw;
(b) rethrow;
(c) throw again;
(d) catch();

14. One limitation of exception handling in C++ is that it:

(a) makes programs run faster in all cases
(b) can add runtime overhead and, if overused, may make code harder to read
(c) removes the need for a try block
(d) guarantees recovery from every error

15. If an exception type does not match any catch block and no catch(...) is present, the behaviour is:

(a) the exception is caught by main() automatically
(b) the program continues normally
(c) std::terminate() is invoked and the program aborts
(d) the compiler reports an error at compile time

Answer Key

1-(b) 2-(c) 3-(b) 4-(b) 5-(b) 6-(a) 7-(b) 8-(c) 9-(c) 10-(b) 11-(b) 12-(b) 13-(a) 14-(b) 15-(c)

Broad / Descriptive Questions

1. Explain the concept of exception handling in C++. Why is it needed in large software systems?
2. Describe the try, throw, and catch construct in C++ with the help of a suitable program example.
3. What is the general syntax of exception handling in C++? Explain the role of each keyword with a diagram or flow of control.
4. Discuss how multiple catch blocks are used to handle different types of exceptions. Illustrate with an example program.
5. Explain the rules that govern the order of catch blocks when multiple catch statements follow a single try block.
6. What is meant by rethrowing an exception? Explain with an example why a program might need to rethrow an exception.
7. Describe the concept of stack unwinding during exception propagation, with a suitable example.
8. Write a C++ program that throws different types of exceptions (e.g., int, char*, and a custom class object) and handles each using multiple catch blocks.
9. What are the limitations of exception handling in C++? Discuss the performance and design trade-offs involved.
10. Compare traditional error handling (using return codes/error flags) with exception handling. What advantages does exception handling offer, and what are its drawbacks?