A screenshot of a computer

Description automatically generated

**Identifying Exception Scenarios**

The process began by identifying exception scenarios within the code. In the do\_even\_more\_custom\_application\_logic() function, a standard C++ exception, such as std::runtime\_error, was thrown to simulate an error occurring during the execution of custom application logic. In the divide() function was managed by throwing a std::invalid\_argument exception when the denominator equals zero. CustomException, was created by deriving it from std::exception to represent user-defined errors in the program.

**Throwing Exceptions**

The next step was throwing the exceptions in appropriate places. In do\_even\_more\_custom\_application\_logic(), the std::runtime\_error exception was thrown to simulate a problem occurring during custom logic processing. In divide(), a check was added to determine if the denominator is zero. If true, a std::invalid\_argument exception was thrown to prevent division by zero. Lastly, in do\_custom\_application\_logic(), the custom exception, CustomException, was explicitly thrown to demonstrate the process of defining and triggering user-specific errors.

**Catching Exceptions**

Exception handling was implemented using try-catch blocks to manage errors. In do\_custom\_application\_logic(), a try-catch block was used to catch std::exception and manage any standard errors thrown by do\_even\_more\_custom\_application\_logic(). The exception’s what() message was displayed to provide insight into the error. In the do\_division() function, a try-catch block was employed to specifically catch the std::invalid\_argument exception thrown during division by zero. This prevents the program from crashing and ensures errors are managed in a controlled manner.

**Managing Multiple Exception Types in main()**

The main() function was designed with a series of try-catch blocks to catch several types of exceptions in a structured order. The custom exception (CustomException) was caught first to manage any specific errors. After that, any standard exceptions (std::exception) were caught, followed by a generic catch block to manage any uncaught or unknown exceptions. This approach ensured that the program did not crash unexpectedly, even if an unhandled exception was thrown.

**Key Techniques and Concepts**

Throughout this process, key techniques were used, including exception throwing and handling. The use of throw allowed for triggering errors when specific conditions, such as division by zero, were met. Exception handling with try-catch blocks enabled the program to catch and manage specific exceptions. Custom exceptions were defined by inheriting from std::exception, which allowed for categorizing and handling errors in a more structured way.

**Result**

By the end of the process, the program was able to robustly manage several types of exceptions without crashing. It provided clear and informative error messages for the user and ensured that execution could continue smoothly in case of errors. This approach improved the program’s overall reliability by managing errors such as invalid operations and other unexpected conditions, leading to a more fault-tolerant application.

This process of exception handling ensures that errors are caught and managed effectively, which improves the program's stability, readability, and robustness. By using try-catch blocks and throwing appropriate exceptions, the application can continue running even in the presence of errors, creating a more resilient program.