**Exceptions**

Justin Starr

Department of STEM

CS 405 – Secure Coding

Professor Mike Alesso

July 28, 2024

**Exceptions**

**Screenshots**

**A screenshot of a computer

Description automatically generated**

**Process Summary**

The way I approached completing this activity was to first start out by having an understanding of the flow of the application and how it needed to work. I then began working on the TODOs from the beginning of the code in the function do\_even\_more\_custom\_apllication\_logic(), where it says to throw any standard exception. I chose to insert my throw after the initial std::cout statement because I wanted to show in the console that this function is reached before the exception is thrown. I continued to the next function where the TODO required was to wrap the call to the do\_even\_more\_custom\_application\_logic() function with an exception handler that catches the std::exception and displays a message to the user. This was done using a try/catch block where in the try portion the if statement is wrapped and a catch is used to catch the exception that is thrown. When completing the divide function, we know that no number can be divisible by zero, therefore I wrote an if statement where if the denominator is equal to zero, an exception would be thrown. Otherwise, the numerator divided by the denominator is returned to the do\_division() function. Here, the standard c++ defined exception I chose to use is the std::invalid\_argument exception since, we are dealing with one of the arguments and its validity. In the do\_division() function, all I had to do was wrap the code that was already there in the try portion of a try/catch block. I then wrote the catch portion which would catch the std::invalid\_argument exception when thrown by the divide function() and display the appropriate message to the user. Similarly, in the main function, to catch any unhandled exception, all I had to do was wrap the existing code within the main() function in the try portion of a try/catch block, and in the catch portion, any unhandled exceptions that are thrown would be caught here, with the appropriate message displayed to the user, to avoid unexpected program termination.

In the case of this program, if there were no catch-all handler in the main, the program would terminate due to an unhandled exception. This is because in the do\_custom\_application\_logic() function, the last TODO in this function required us to throw a custom exception derived from std::exception and catch it explicitly in main. The exception that is thrown is the last thing that happens before returning to the main function, and therefore the main function is expected to handle this exception. If the main function is not wrapped in a try/catch block, the exception goes unhandled, causing the application to abort termination unexpectedly. This could prove to cause issues, especially in more complex applications. Therefore, it is my conclusion that it is good practice to have an exception handler in the main, to catch any unhandled exceptions.

Overall, I did not experience any major issues with this activity. I have had some experience working with exception handling in prior coursework. This did give me some good practice with the various types of std::exceptions, such as std::runtime\_error and std::invalid\_argument, which I have never used before.