Experiment No: 10 Date:

Aim: To study Exception handling in C++

Theory:

1. Try-Catch Blocks

Exception handling in C++ involves the use of try-catch blocks. Code within the "try" block is monitored for exceptions, and if any occur, the corresponding "catch" block is executed, allowing for controlled error handling.

2. Throwing Exceptions

Developers can use the `throw` keyword to generate exceptions explicitly. This is useful for signaling errors or exceptional conditions within the program. Custom exception classes can be created to provide detailed information about the error.

3. Standard Exception Classes

C++ standard library provides a set of exception classes, such as `std::exception`, which can be used as base classes for custom exceptions. This hierarchy helps in organizing and handling different types of exceptions effectively.

4. Catching Specific Exceptions

Catch blocks can be designed to catch specific types of exceptions. This allows for targeted error handling, where different catch blocks handle different exception types, providing more granular control over the response to specific errors.

5. Catch-All Blocks

A catch-all block, specified as `catch (...)`, can be used to catch any exception that is not caught by preceding catch blocks. While it provides a safety net, it should be used judiciously, as it may make debugging more challenging.

6. Rethrowing Exceptions

In some cases, it's beneficial to catch an exception, perform some actions, and then rethrow the same exception using the `throw` statement. This allows for additional processing while maintaining the original exception's characteristics.

7. Nested Try-Catch Blocks

Exception handling can be nested, with inner try-catch blocks within the scope of outer ones. This hierarchical structure allows for handling exceptions at different levels of the program, promoting a more organized and modular approach.

8. Exception Specifications

C++ supports exception specifications, which allow developers to declare the types of exceptions that a function may throw. However, their use is discouraged in modern C++, and the `noexcept` specifier is preferred for indicating that a function does not throw exceptions.

9. RAII (Resource Acquisition Is Initialization)

RAII is a design principle that promotes resource management through object lifetimes. Using smart pointers and resource-managing classes, such as `std::unique_ptr` and `std::shared_ptr`, helps automate resource cleanup, reducing the need for manual exception handling.

10. Exception Safety Guarantees

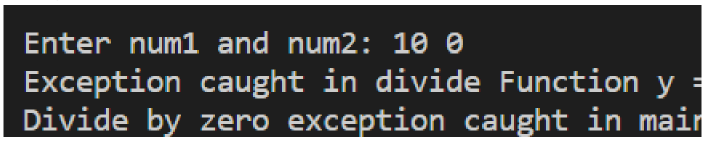
Functions in C++ may provide different levels of exception safety guarantees: no-throw, basic, and strong. Understanding and documenting these guarantees are crucial for writing robust code that can gracefully recover from exceptional conditions.

[A] Write a C++ program to implement exceptional handling concept (Divide by zero) using exception rethrow mechanism

```
#include <iostream>
using namespace std;
void divide(float x, float y) {
  try {
    if (y != 0) {
      cout << "Division = " << x / y;
    } else {
      throw y;
    }
  } catch (int q) {
    cout << "Exception caught in divide Function y = "
<< q << endl;
    throw;
  }
}
int main() {
  float num1, num2;
  cout << "\nEnter num1 and num2: ";
  cin >> num1 >> num2;

  try {
    divide(num1, num2);
  } catch (int p) {
    cout << "Divide by zero exception caught in main,
" << "num2 = " << p << endl;
  }
  return 0;
}
```

Output:

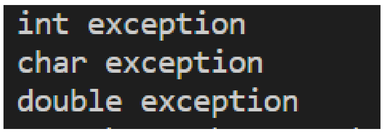


```
Enter num1 and num2: 10 0
Exception caught in divide Function y =
Divide by zero exception caught in mair
```

[B] Write a C++ program to implement a multi catch exception handling mechanism

```
#include <iostream>
using namespace std;
void testFunction(int param);
int main() {
  testFunction(1);
  testFunction(0);
  testFunction(-1);
}
void testFunction(int param) {
  try {
    if (param == 1)
      throw param;
    else if (param == 0)
      throw 'c';
    else if (param == -1)
      throw -1.0;
  }
  catch (int m) {
    cout << "int exception" << endl;
  }
  catch (char n) {
    cout << "char exception" << endl;
  }
  catch (double q) {
    cout << "double exception" << endl;
  }
}
```

Output:



```
int exception
char exception
double exception
```

Conclusion: All the concepts were understood and implemented in the above codes.

Nitesh Naik

(Subject Faculty)