

Experiment No- 11

Date-

Aim – To study Exception Handling in C++**Theory –****Exception Handling in C++**

Exception handling is a critical feature in C++ that allows a program to manage and respond to unexpected events, known as exceptions. It provides a structured way to handle errors without crashing the program. By using exceptions, developers can write code that is cleaner, easier to read, and more robust.

1. What is an Exception?

An exception is an event that occurs during the execution of a program that disrupts its normal flow. This could be due to various reasons, such as:

- Invalid user input
- File not found errors
- Memory allocation failures
- Division by zero
- Array index out of bounds

When an exception occurs, the program can throw an exception object, which contains information about the error.

2. Components of Exception Handling

C++ uses three main keywords for exception handling: try, catch, and throw.

- **try Block:** This is where you place the code that might throw an exception. It allows you to test a block of code for errors.
- **catch Block:** This follows the try block and contains code that is executed if an exception is thrown in the try block. It handles the exception.
- **throw Statement:** This is used to signal that an exception has occurred. You can throw exceptions explicitly by using the throw keyword followed by an exception object.

3. Basic Syntax

Here is a simple structure of exception handling in C++:

```
try {
```

```
// Code that may throw an exception  
  
throw someException; // Throw an exception  
  
{ catch (const ExceptionType& e) {  
  
    // Code to handle the exception  
  
}
```

4. How it Works

When an exception is thrown, the normal flow of control is disrupted. The C++ runtime system looks for a matching catch block. If it finds one, it transfers control to that block, allowing the program to handle the exception gracefully. If no matching catch block is found, the program terminates

5. Best Practices

- **Use Exceptions for Error Handling:** Exceptions should be used for error handling instead of return codes, as they provide a clearer and more manageable way to handle errors.
- **Catch by Reference:** Always catch exceptions by reference to avoid slicing and unnecessary object copies.
- **Avoid Using Exceptions for Control Flow:** Exceptions should be used for exceptional conditions, not for regular control flow.
- **Always Provide a what() Method:** If you create custom exceptions, ensure that they override the what() method to provide meaningful error messages.

6. Performance Considerations

While exceptions provide a powerful mechanism for error handling, they can have performance implications. Throwing and catching exceptions can be costly, so it is best to use them judiciously.

In summary, exception handling in C++ is a powerful tool that allows developers to write more reliable and maintainable code. It provides a clear and effective way to manage errors, ensuring that programs can respond gracefully to unexpected situations. By understanding and properly implementing exception handling, programmers can create robust applications that are easier to debug and maintain.

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

Program-

```
#include <iostream>
#include <stdexcept>

double divide(double
numerator, double
denominator) { if
(denominator == 0) {
throw
std::invalid_argument("Division by
zero is not allowed.");
}
return numerator / denominator;
}

void performDivision(double a, double
b) {
try {
double result = divide(a,
b);
std::cout << "Result: " <<
result << std::endl; } catch
(const
std::invalid_argument& e) {
std::cout << "Caught an exception: "
<< e.what() << std::endl;
throw; // Rethrow the exception
for further handling
}
}

int main() {
try {
```

OUTPUT –

```
Caught an exception: Division by zero is not allowed.
Main caught an exception: Division by zero is not allowed.
```

```
double num1 = 10.0;
double num2 = 0.0; // Change
this to a non-zero value to avoid
the exception
performDivision(num1, num2);
    } catch (const
std::invalid_argument& e) {
std::cout << "Main caught an
exception: " << e.what() <<
std::endl;
    }

    return 0;
}
```

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

```

#include <iostream>
#include <stdexcept>
#include <vector>

void performOperations(int index) {
    std::vector<int> numbers = {1, 2, 3};

    // Throwing an exception if the index is invalid
    if (index < 0) {
        throw std::invalid_argument("Index cannot be negative.");
    }
    if (index >= numbers.size()) {
        throw std::out_of_range("Index is out of range.");
    }

    // Access the element at the specified index
    std::cout << "Value at index " << index << ": " <<
        numbers[index] << std::endl;
}

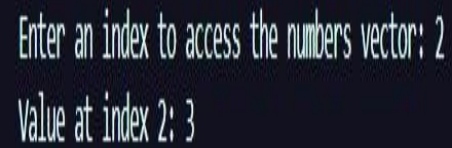
int main() {
    int index;

    // Ask the user for an index
    std::cout << "Enter an index to access the numbers vector: ";
    std::cin >> index;

    try {
        performOperations(index);
    } catch (const std::invalid_argument& e) {
        std::cout << "Caught invalid_argument exception: " <<
            e.what() << std::endl;
    } catch (const std::out_of_range& e) {
        std::cout << "Caught out_of_range exception: " <<
            e.what() << std::endl;
    } catch (...) { // Catch-all for any other exceptions
        std::cout << "Caught an unknown exception." << std::endl;
    }

    return 0;
}

```

Output –


```

Enter an index to access the numbers vector: 2
Value at index 2: 3

```


[C] Write a C++ program to implement a queue class that supports various data types and allows users to insert and remove items. The queue should maintain two and handle exceptions for exceeding capacity and removing from an empty queue. Include an interactive `main()` function that lets users test the queue functionality and recover from exceptions without corrupting the queue's contents.

Program –

```
#include <iostream>
#include <stdexcept>

using namespace std;

// Template class for Queue template
<typename T>
class Queue { private:
    static const int capacity = 10; // Fixed size for simplicity
    T data[capacity]; // Array to store queue elements
    int front; // Index of the front of the queue
    int rear; // Index of the rear of the queue
    int count; // Current number of elements in the queue

public:
    // Constructor to initialize the queue
    Queue() : front(0), rear(0), count(0) {} // Simple constructor

    // Enqueue operation
    void enqueue(T item) {
        // Check if the queue is full
        if (count >= capacity) {
            // Throw an exception if capacity is exceeded
            throw overflow_error("Queue capacity exceeded!");
        }
        // Insert the item at the rear
        data[rear] = item;
        // Update the rear index, wrapping around if necessary
        rear = (rear + 1) % capacity; // Increment the count of items
        count++;
    }

    // Dequeue operation
    T dequeue() {
        // Check if the queue is empty
        if (count <= 0) {
            // Throw an exception if trying to dequeue from an empty queue
            throw underflow_error("Queue is empty!");
        }
        // Retrieve the item from the front
        T item = data[front];
        // Update the front index, wrapping around if necessary
        front = (front + 1) % capacity; // Decrement the count of items
        count--;
    }
};
```

Output –

```
1. Enqueue
2. Dequeue
3. Display
4. Exit
Choose an option: 2
Queue is empty!

1. Enqueue
2. Dequeue
3. Display
4. Exit
Choose an option: 1
Enter number of values to enqueue (max 10): 11
Cannot enqueue more than queue capacity.

1. Enqueue
2. Dequeue
3. Display
4. Exit
Choose an option: 1
Enter number of values to enqueue (max 10): 10
Enter value to enqueue: 5
Enter value to enqueue: 6
Enter value to enqueue: 7
Enter value to enqueue: 8
Enter value to enqueue: 9
Enter value to enqueue: 0
Enter value to enqueue: 1
Enter value to enqueue: 2
Enter value to enqueue: 3
Enter value to enqueue: 4
Queue elements: 5 6 7 8 9 0 1 2 3 4

1. Enqueue
2. Dequeue
3. Display
4. Exit
Choose an option: 1
Enter number of values to enqueue (max 10): 5
Enter value to enqueue: 1
Queue capacity exceeded!

1. Enqueue
2. Dequeue
3. Display
4. Exit
Choose an option: 4
Exiting...
```

<pre> // Return the dequeued item return item; } // Get the number of items in the queue int size() const { return count; // Return the current count of items } // Check if the queue is empty bool isEmpty() const { return count == 0; // Return true if count is zero } // Function to get the maximum capacity of the queue int getCapacity() const { return capacity; // Return the maximum capacity } // Display the elements in the queue void display() const { if (isEmpty()) { cout << "Queue is empty.\n"; } return; cout << "Queue elements: "; for (int i = 0; i < count; ++i) { cout << data[(front + i) % capacity] << " "; } cout << "\n"; } }; // Interactive main function int main() { int n; // Number of values to enqueue Queue<int> queue; // Create a queue for integers int choice, value; // Variables for user choice and value input do { // Menu for user choices cout << "\n1. Enqueue\n2. Dequeue\n3. Display\n4. Exit\nChoose an option: "; cin >> choice; </pre>	<pre> try { switch (choice) { case 1: // Enqueue cout << "Enter number of values to enqueue (max " << queue.getCapacity() << "): "; cin >> n; if (n > queue.getCapacity()) { throw overflow_error("Cannot enqueue more than queue capacity."); } for (int i = 0; i < n; i++) { cout << "Enter value to enqueue: "; cin >> value; // Use 'value' for input queue.enqueue(value); // Call enqueue method } queue.display(); // Display the queue break; case 2: // Dequeue value = queue.dequeue(); // Call dequeue method cout << value << " dequeued.\n"; queue.display(); // Display the queue after dequeue break; case 3: // Display queue.display(); // Display current queue break; case 4: // Exit cout << "Exiting...\n"; break; default: cout << "Invalid option. Please choose again.\n"; } } catch (const overflow_error& e) { // Catch overflow error if the queue is full cerr << e.what() << "\n"; // Output error message } catch (const underflow_error& e) { // Catch underflow error if the queue is empty cerr << e.what() << "\n"; // Output error message } } while (choice != 4); // Continue until user chooses to exit return 0; // End of the program } </pre>
--	--

Conclusion – All the codes were successfully executed using the concepts of Exception Handling.