Unit 4: Exception Handling and Templates Programming

Syllabus :
Exception Handling: Review of traditional error handling, basics of exception handling, exception handling mechanism, throwing mechanism, catching mechanism, rethrowing an exception, specifying exceptions.
Templates and Generic Programming: Template concepts, Function templates, class templates, illustrative examples.

Section 1 : Exception Handling in C++

Exception handling in C++ is a structured way to handle errors and exceptional conditions that occur during program execution. It provides a way to separate error-handling code from regular code, making programs easier to read and maintain.

1.1 Review of Traditional Error Handling

Traditional error handling refers to the conventional methods used in programming languages to deal with errors or exceptional situations that may occur during program execution.

A. Methods:

1. Error Codes: Traditional error handling often involves the use of error codes or special return values to indicate the occurrence of an error. Functions or methods may return specific values (e.g., -1) to signal an error condition.
2. Global Variables: Some programming languages utilize global variables to store error information. These variables are checked after each operation to determine if an error occurred.
3. Conditional Statements: Developers typically use conditional statements such as if-else or switch-case to handle errors. These statements check for error conditions and execute appropriate error-handling code.

B. Example:

```
#include <iostream>
using namespace std;

int divide(int a, int b) {
    if (b == 0) {
        return -1; // Indicate error
    }
    return a / b;
}

int main() {
    int a = 10, b = 0;
    int result = divide(a, b);
    if (result == -1) {
        cout << "Error: Division by zero" << endl;
    } else {
        cout << "Result: " << result << endl;
    }
    return 0;
}
```

C. Limitations:

1. Error-Prone: Requires manual error checking, which can be easily overlooked.
2. Cluttered Code: Can lead to scattered error handling logic, making the code harder to read and maintain.
3. Limited Information: Return values and error codes provide limited information about the error context.
4. Control Flow Issues: Can lead to convoluted control flow, especially in deeply nested functions or complex logic.

1.2 Basics of Exception Handling

Exception

- An exception in programming refers to an abnormal condition or unexpected event that occurs during the execution of a program, disrupting the normal flow of control. Exceptions are typically caused by errors or exceptional conditions that arise at runtime and may prevent the program from continuing its execution as expected.
- Common reasons that cause exceptions include division by zero (run-time error), invalid input, null pointer dereference, out-of-bounds access, memory allocation failure, file not found, resource exhaustion, concurrency issues, hardware failures, and errors from system calls or library functions.

Exception Handling

- Exception handling is a mechanism in programming that deals with runtime errors or exceptional situations that may occur during the execution of a program. These exceptional situations could include division by zero, file not found, out-of-memory errors, and so on.
- Exception handling allows a program to respond to such situations in a controlled and graceful manner rather than abruptly terminating or producing undefined behavior.
- Exception handling in C++ involves the use of **try, catch, and throw** keywords to manage runtime errors and handle exceptional conditions gracefully.
    - **try Block:** The code that might generate an exception is placed inside a try block.

- **throw Statement:** When an error occurs, an exception is thrown using the throw statement.
- **catch Block:** The exception is caught by a catch block that handles the exception.

Example of Exception Handling

```
#include <iostream>
using namespace std;

int divide(int num, int den) {
    if (den == 0) {
        throw runtime_error("Division by zero"); // Throwing an exception
    }
    return num / den;
}

int main() {
    int num = 10, den = 0;

    try {
        int result = divide(num, den); // Code that may throw an exception
        cout << "Result: " << result << endl;
    } catch (const runtime_error& e) { // Catching the exception
        cout << "Error: " << e.what() << endl;
    }

    return 0;
}
```

Benefits of Exception Handling

1. Separating Error-Handling Code from Regular Code:
   - Exceptions allow you to separate the details of what to do when something out of the ordinary happens from the main logic of a program.
   - In traditional error management, error detection, reporting, and handling often lead to confusing spaghetti code. By using exceptions, you can keep the main flow of your code clean and deal with exceptional cases elsewhere.
2. Enhancing Robustness:
   - Exception handling ensures the continuity of your program even when unexpected errors occur.
   - Instead of crashing, your program gracefully handles exceptions, making it more robust and reliable.
3. Improving Readability and Maintainability:
   - Separating error-handling code from regular code improves the readability of your program.

- Developers can focus on the main logic without getting distracted by error-related clutter.

4. Accurate Error Reporting:
    - Exceptions provide meaningful error messages, making it easier to identify the cause of failures.
    - When an exception occurs, you can include relevant information (such as stack traces) to pinpoint the issue.
5. Facilitating Debugging and Troubleshooting:
    - Exception stack traces help you trace the sequence of method calls that led to the error.
    - Debugging becomes more efficient because you can quickly locate the problematic code.
6. Improved Security:
    - Proper exception handling prevents security vulnerabilities caused by unexpected behavior.
    - By handling exceptions gracefully, you reduce the risk of exposing sensitive information or allowing unauthorized access.
7. Better User Experience:
    - When exceptions are handled well, users experience fewer crashes or abrupt program terminations.
    - A smooth user experience contributes to overall satisfaction with your software.
8. Enabling Error Recovery Mechanisms:
    - Exceptions allow you to recover from errors gracefully.
    - You can catch exceptions, log relevant information, and take corrective actions without disrupting the program flow.

1.3 Exception Handling Mechanism

Exception handling in C++ provides a structured way to handle runtime errors or exceptional conditions that may occur during program execution, allowing the program to recover without crashing.
It involves three key components: try, throw, and catch.

- **try Block:** The try block encloses the code that might throw an exception. If an exception occurs within the try block, it is transferred to the appropriate catch block.
- **throw Statement:** The throw statement is used to explicitly throw an exception. It can be followed by an expression, which is typically an object of an exception class, to provide information about the error.
- **catch Block:** The catch block catches and handles exceptions thrown by the associated try block. It specifies the type of exception it can handle and provides code to deal with the exception.

A. Throwing Mechanism:

- The throwing mechanism is used to signal that an exceptional condition has occurred during program execution. It involves the throw statement.