Objective:

To understand the concept and role of exception handling in C++ for managing runtime errors effectively; to learn the advantages of exceptions over conventional error handling, the structure and mechanism of try-catch blocks, the use of multiple catch handlers, catching all types of exceptions, rethrowing exceptions, passing arguments with exceptions, specifying exceptions in function declarations, handling exceptions during construction and destruction of objects, and managing uncaught and unexpected exceptions.

Theory:

# Basics of Exception Handling

Exception handling is a mechanism to handle runtime errors in a structured way. It separates error-handling code from regular code using `try`, `throw`, and `catch` blocks.

Syntax:

try {

// Code that may throw an exception throw exception\_value;

} catch (type arg) {

// Code to handle the exception

}

Example:

#include <iostream>

using namespace std;

int main() { try { throw 10; // throwing an integer exception

} catch (int e) { cout << "Caught exception: " << e << endl;

} return 0;

}

**Advantage Over Conventional Error Handling** - Avoids use of error codes.

* Makes code cleaner and easier to maintain.
* Ensures error-handling code is separate from business logic.
* Supports object-oriented features.

Example Comparison:

Conventional error handling:

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

}

Exception handling:

int divide(int a, int b) { if (b == 0) throw "Divide by zero!";

return a / b;

}

# Exception Handling Mechanism

Three key components:

* try: encloses code that might throw an exception.
* throw: raises an exception.
* catch: handles the exception.

Example:

|  |
| --- |
| try { throw "Something went wrong!"; } catch (const char\* msg) {  cout << "Caught: " << msg << endl;  } |

# Multiple Handlers

Multiple catch blocks can be used to handle different types of exceptions.

Example:

|  |
| --- |
| try { throw 3.14; } catch (int e) {  cout << "Caught int: " << e << endl;  } catch (double e) {  cout << "Caught double: " << e << endl; } |

# Catching All Exceptions

To catch any type of exception regardless of its data type, use `catch(...)`.

Example:

|  |
| --- |
| try { throw "error"; } catch (...) {  cout << "Caught unknown exception" << endl;  } |

# Rethrowing Exception

An exception can be rethrown using `throw;` to be handled at a higher level.

Example:

|  |
| --- |
| void test() { try { throw "error inside test";  } catch (...) {  cout << "Caught inside test, rethrowing..." << endl; throw; // rethrowing  }  }    int main() { try {  test();  } catch (const char\* msg) {  cout << "Caught in main: " << msg << endl;  }  } |

# Exception with Arguments

Exceptions can be thrown as objects with data members to carry more context.

Example:

|  |
| --- |
| class MyException { string message; public:  MyException(string msg) : message(msg) {} string getMessage() { return message; }  };    int main() { try { |

throw MyException("Custom error occurred");

} catch (MyException& e) { cout << e.getMessage() << endl;

}

}

# Exception Specification for Function

Deprecated in modern C++, previously used to specify what exceptions a function might throw.

Old Syntax:

void func() throw(int, double); // func may throw int or double

Modern C++ uses noexcept:

void func() noexcept; // guaranteed not to throw exceptions

Example:

void safe() noexcept {

cout << "This won't throw." << endl;

}

# Exceptions in Constructors and Destructors

* If an exception is thrown in a constructor, object construction fails.
* Avoid throwing exceptions in destructors — it can cause terminate() to be called.

Example:

|  |
| --- |
| class Demo { public: Demo() {  throw "Constructor failed!"; }  };    int main() { try {  Demo d;  } catch (const char\* e) {  cout << e << endl;  }  } |

# Handling Uncaught Exceptions

Use std::set\_terminate() to handle uncaught exceptions gracefully.

Example:

|  |
| --- |
| #include <iostream> #include <exception>  using namespace std;    void myTerminate() { cout << "Uncaught exception detected!" << endl; exit(1);  }    int main() { set\_terminate(myTerminate); throw 42; // not caught  } |

# Handling Unexpected Exception

Use std::set\_unexpected() to define a handler for unexpected exceptions thrown outside of function's specification.

Example:

|  |
| --- |
| #include <iostream> #include <exception>  using namespace std;    void unexpectedHandler() { cout << "Unexpected exception occurred!" << endl; exit(1);  }    void test() throw(int) { throw 'x'; // violates specification  }    int main() {  set\_unexpected(unexpectedHandler); test();  } |

set\_unexpected() is deprecated in modern C++ and replaced by noexcept.

Q1) Write a C++ program to handle divide-by-zero exception using try-catch block. Input two numbers. If denominator is zero, throw and catch an exception.

Code:

|  |
| --- |
| #include <iostream> #include <stdexcept>  using namespace std;    int main() { double numerator, denominator;    cout << "Enter numerator: "; cin >> numerator; cout << "Enter denominator: "; cin >> denominator;    try {    if (denominator == 0) { throw "Error: Division by zero is not allowed."; }    double result = numerator / denominator; cout << "Result: " << result << endl;  } catch (const char\* msg) {  cout << msg << endl;  }    return 0;  } |

Output:

Enter numerator: 12

Enter denominator: 0

Error: Division by zero is not allowed.

Q2) Write a C++ program to demonstrate multiple catch blocks handling different data types. Throw and handle int, char, and string type exceptions in separate catch blocks.

Code:

|  |
| --- |
| #include <iostream>  #include <stdexcept> #include <string>  using namespace std;    int main() { try {  throw 42; } catch (int e) {  cout << "Caught an integer exception: " << e << endl;  } try {  throw 'A'; } catch (char e) {  cout << "Caught a character exception: " << e << endl;  } try { throw string("This is a string exception");    } catch (string e) {  cout << "Caught a string exception: " << e << endl;  } catch(...){ cout<< "An unknown exception occurred." << endl;  }  return 0;  } |

Output:

Caught an integer exception: 42

Caught a character exception: A

Caught a string exception: This is a string exception

Q3) Write a program using catch-all handler (catch(...)) to handle any kind of exception. Illustrate a case where an unexpected data type is thrown and caught generically.

Code:

|  |
| --- |
| #include <iostream>  #include<stdexcept>    using namespace std;    int main(){    int choice; cout << "enter 1 for double, 2 for bool and 3 for any other :"; cin >> choice;    try{ if(choice==1){ throw 3.1415;  } else if(choice==2){ throw true;  } else{ throw nullptr;  }    }  catch (double e) { cout << "Caught double exception: " << e << endl;  }  catch (bool e) { cout << "Caught boolean exception: " << (e ? "true" : "false") << endl;  }  catch (...) {  cout << "Caught unknown exception using catch-all handler" << endl; }    return 0;    } |

Output:

enter 1 for double, 2 for bool and 3 for any other :2

Caught boolean exception: true

Q4) Write a C++ program that rethrows an exception after catching it once. Use a nested try catch where the inner catch block rethrows the exception to be handled by the outer block.

Code:

|  |
| --- |
| #include<iostream>  #include<stdexcept>    using namespace std;    int main() { try { try { throw runtime\_error("Inner exception");  }  catch (const exception& e) { cerr << "Caught inner exception: " << e.what() << endl; throw;  }  }  catch (const exception& e) { cerr << "Caught outer exception: " << e.what() << endl;  }    return 0;  } |

Output:

Caught inner exception: Inner exception

Caught outer exception: Inner exception

Q5) Write a program to demonstrate throwing and catching a user-defined exception class with message argument.Define a custom class MyException and pass an error message to its constructor.

Code:

|  |
| --- |
| #include <iostream>  #include <string> #include <stdexcept> using namespace std; class MyException{ string message; public:  MyException(const string& msg) : message(msg) {} string getMessage() const { return message;  }  };    int main() { int choice; cout << "Enter 1 to throw MyException: "; cin >> choice;    try {  if (choice == 1) { throw MyException("This is a user-defined exception message.");  }  cout << "No exception thrown." << endl;  }    catch (const MyException& e) { cout << "Caught MyException: " << e.getMessage() << endl; }      catch (...) { cout << "Caught an unknown exception." << endl; }    return 0;  } |

Output:

Enter 1 to throw MyException: 1

Caught MyException: This is a user-defined exception message.

Write a program that sets a custom terminate handler using set\_terminate() and demonstrates uncaught exception handling.Throw an exception with no matching catch block.

Code:

|  |
| --- |
| #include <iostream>  #include <exception>  #include <string>    using namespace std;    void customTerminateHandler() {  cerr << "Custom terminate handler called. Exiting program." << endl; exit(1);  }    int main() { set\_terminate(customTerminateHandler);  try { throw runtime\_error("Uncaught exception");  } catch (const exception& e) {  cout << "Caught exception: " << e.what() << endl;  }    return 0;  } |

Output:

Caught exception: Uncaught exception

Write a C++ program with a function that violates its exception specification and handles it using set\_unexpected().Use throw(double) in the function declaration but throw an int.

Code:

|  |
| --- |
| #include <iostream> #include <exception>  using namespace std;    void unexpectedHandler() { cerr << "Unexpected exception caught. Exiting program." << endl; exit(1);  }    void testFunction() throw (double) { throw 42;  }    int main() { set\_unexpected(unexpectedHandler);  try {  testFunction();  } catch (const exception& e) {  cout << "Caught exception: " << e.what() << endl;  }    return 0;  } |

Output:

Unexpected exception caught. Exiting program.

Write a program to show exception handling inside a class constructor and destructor. Handle constructor exceptions properly; avoid throwing from destructor.

Code:

|  |
| --- |
| #include <iostream> #include <exception>  using namespace std;    class MyClass { public:  MyClass() {  cout << "Constructor called." << endl;  throw runtime\_error("Exception in constructor"); }    ~MyClass() {  cout << "Destructor called." << endl;  }  };    int main() { try {  MyClass obj;  } catch (const exception& e) {  cerr << "Caught exception: " << e.what() << endl;  }    return 0;  } |

Output:

Constructor called.

Caught exception: Exception in constructor

Write a function that accepts user input and throws an exception if input is invalid (e.g., non-integer for age).Use exception handling to validate data entry.

Code:

|  |
| --- |
| #include <iostream>  #include <stdexcept>  #include <limits>    using namespace std;    void validateAgeInput(int age) { if (age < 0 || age > 120) { throw invalid\_argument("Invalid age. Age must be between 0 and 120.");  }  }    int main() {  int age;    cout << "Enter your age: "; cin >> age;  try {  validateAgeInput(age); cout << "Valid age entered: " << age << endl;  } catch (const invalid\_argument& e) { cerr << "Error: " << e.what() << endl;  }    return 0;  } |

Output:

Enter your age: -5

Error: Invalid age. Age must be between 0 and 120.

Write a C++ program to demonstrate exception propagation across multiple function calls.Function A calls B, which calls C. C throws an exception. Handle it in A.

Code:

|  |
| --- |
| #include <iostream> #include <stdexcept> using namespace std; void functionC() { throw runtime\_error("Exception thrown in function C"); }    void functionB() { functionC();  }    void functionA() { try {  functionB();  } catch (const runtime\_error& e) {  cerr << "Caught exception in function A: " << e.what() << endl;  }  }    int main() { functionA(); return 0;  } |

Output:

Caught exception in function A: Exception thrown in function C