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 ofexceptions, 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
```

DISCUSSION:

Exception handling in C++ enables programs to manage errors that arise during execution. Instead of crashing, the program can detect these errors using the try, throw, and catch mechanisms. This approach allows the program to handle various types of problems and either continue functioning or terminate safely.

CONCLUSION:

Exception handling in C++ programs enhances their safety and reliability by enabling the detection and management of errors effectively. Instead of crashing unexpectedly, the program can respond to problems in a controlled manner, displaying messages or taking alternative actions. This approach improves the user experience and simplifies the program's maintenance and debugging. In conclusion, exception handling is a crucial tool for crafting robust and dependable C++ programs.