EXCEPTION C++

Exception Handling in C++

Exception handling is a crucial mechanism in C++ that allows you to deal with runtime errors and exceptional circumstances in a structured way. It helps in writing robust and fault-tolerant code by separating the error-handling logic from the normal program flow.

1. Basic Concepts

- **Exception:** An exception is an event that occurs during the execution of a program, disrupting the normal flow of instructions. It's often a signal that an error or an unusual condition has occurred.
- **Error vs. Exception:** While often used interchangeably, an error is a more general term. An exception is a specific way of handling certain types of errors in C++.
- **The `try`, `catch`, and `throw` Keywords:** These are the core elements of C++ exception handling.
  - **`try`:** A block of code that might throw an exception is enclosed in a `try` block.
  - **`catch`:** A `catch` block is used to handle a specific type of exception. It follows a `try` block.
  - **`throw`:** The `throw` keyword is used to raise (or throw) an exception when an error condition is detected.

2. How Exception Handling Works

1. **The `try` Block:**

```
* Code that might generate an exception is placed within a `try` block.

* A `try` block is always followed by one or more `catch` blocks.

* If no exception occurs within the `try` block, the code executes normally,
and the `catch` blocks are ignored.
```

2. **The `throw` Keyword:**

```
* When an error condition is detected within the `try` block, the `throw`
keyword is used to throw an exception.

* The `throw` statement interrupts the normal execution of the `try` block.

* The operand of the `throw` keyword is an object representing the
```

```
exception. This object can be of any data type, but it's common to use
objects from the standard exception classes or custom exception classes.
```

3. **The `catch` Block:**

```
* `catch` blocks are defined to handle specific types of exceptions.

* Each `catch` block specifies the type of exception it can handle in its
parameter list (the exception handler).

* When an exception is thrown, the C++ runtime looks for a `catch` block
that can handle the exception.

* If a matching `catch` block is found, the code within that block is
executed. This code is responsible for taking appropriate actions to handle
the error (e.g., logging the error, displaying an error message, cleaning up
resources, or attempting to recover).

* If no matching `catch` block is found, the exception propagates up the
call stack. If it's not caught at any level, the program terminates (usually
by calling `std::terminate`).
```

4. Syntax

```
try {
    // Code that might throw an exception
    if (/* error condition */) {
        throw /* exception object */; // Throw an exception
    }
    // ... normal code execution ...
}
catch (/* exception type 1 */ /* exception object name */) {
    // Code to handle exception type 1
}
catch (/* exception type 2 */ /* exception object name */) {
    // Code to handle exception type 2
}
// ... more catch blocks ...
catch (...) {
    // Code to handle any type of exception (catch-all)
}
```

5. Exception Handling Process

1. The program enters a `try` block.
2. Code within the `try` block executes.
3. If an exception is thrown:

```
* The remaining code in the `try` block is skipped.

* The program searches for a matching `catch` block.

* A `catch` block matches if its exception type is the same as the type of
the thrown exception, or if it's a base class of the thrown exception (when
catching by reference or pointer).

* If a matching `catch` block is found, the code within that `catch` block
is executed.

* If no matching `catch` block is found, the exception propagates to the
calling function. This process is called "stack unwinding."
```

4. If no exception is thrown, the `catch` blocks are skipped, and the program continues execution after the `try`-`catch` sequence.

6. Catching Exceptions

- **Catch by Type:**

```
catch (std::runtime_error e) { // Catch by value
    // Handle runtime_error exceptions
    std::cerr << "Caught exception: " << e.what() << std::endl;
}

catch (std::runtime_error& e) { // Catch by reference
    // Handle runtime_error exceptions
    std::cerr << "Caught exception: " << e.what() << std::endl;
}

catch (const std::runtime_error& e) { // Catch by const reference
    // Handle runtime_error exceptions
    std::cout << "Caught exception: " << e.what() << std::endl; //
Access exception information
}
```

- Catching by reference (`&`) is generally preferred to avoid unnecessary copying of the exception object.
- Catching by `const` reference is recommended to prevent accidental modification of the exception object.

- **Catch All:**

```
catch (...) {
    // Handle any type of exception
    std::cerr << "An unknown exception occurred." << std::endl;
}
```

- The `catch (...)` syntax can catch any type of exception.
- It should be used as the last `catch` block in a sequence, as it will catch everything.
- You don't have access to the exception object itself in a `catch (...)` block.

7. Throwing Exceptions

- You can throw any type of object as an exception, but it's good practice to use objects derived from the `std::exception` class or its derived classes.
- Example:

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

int main() {
    int x = -5;
    try {
        if (x < 0) {
            throw std::invalid_argument("x must be non-negative");
        }
        std::cout << "x is: " << x << std::endl;
    }
    catch (const std::invalid_argument& e) {
        std::cerr << "Error: " << e.what() << std::endl;
        return 1;
    }
    std::cout << "Program continues after the catch block." <<
std::endl;
    return 0;
}
```

In this example, if `x` is negative, an `std::invalid_argument` exception is thrown. The `catch` block handles the exception, prints an error message, and the program continues execution.

8. Standard Exception Classes

- C++ provides a hierarchy of standard exception classes in the `<stdexcept>` header.
- These classes are derived from the base class `std::exception`.
- The `std::exception` class has a virtual member function `what()` that returns a description of the exception.
- **Common Standard Exceptions:**
  - `std::logic_error`: Represents errors in the program's logic.
    - `std::invalid_argument`: Invalid argument passed to a function.
    - `std::domain_error`: Argument is outside the domain of the function.
    - `std::out_of_range`: Index is out of the valid range.
  - `std::runtime_error`: Represents errors that can only be detected at runtime.
    - `std::runtime_error`: A general runtime error.
    - `std::overflow_error`: Arithmetic overflow occurred.
    - `std::underflow_error`: Arithmetic underflow occurred.
- Using the standard exception classes promotes consistency and makes your code more understandable.

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

int main() {
    try {
        std::vector<int> myVector = { 1, 2, 3 };
        std::cout << myVector.at(5) << std::endl; // throws
std::out_of_range
    }
    catch (const std::out_of_range& e) {
        std::cerr << "Error: " << e.what() << std::endl;
        return 1;
    }
    return 0;
}
```

This example demonstrates the use of `std::out_of_range`, which is thrown when trying to access an element that is out of bounds.

9. Custom Exception Classes

- You can define your own exception classes by inheriting from `std::exception` or one of its derived classes.
- This allows you to create exception types that are specific to your application and can carry more detailed error information.

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

class MyException : public std::exception {
public:
    MyException(const std::string& message) : message_(message) {}
    const char* what() const noexcept override { return
message_.c_str(); }
private:
    std::string message_;
};

int main() {
    try {
        // ...
        if (/* my error condition */) {
            throw MyException("Something bad happened in my code.");
        }
        // ...
    }
    catch (const MyException& e) {
        std::cerr << "MyException caught: " << e.what() << std::endl;
        return 1;
    }
    catch (const std::exception& e) {
        std::cerr << "Standard exception caught: " << e.what() <<
std::endl;
        return 1;
    }
    return 0;
}
```

Here, a custom exception class `MyException` is derived from `std::exception`. This allows you to throw exceptions specific to your program's needs.

10. **Exception Specifications (`noexcept`)**

- **Deprecated:** Exception specifications (e.g., `throw(int, std::runtime_error)`) were deprecated in C++11 and removed in C++17.
- `noexcept`:
  - The `noexcept` specifier is used to indicate whether a function can throw an exception.
  - `noexcept` or `noexcept(true)`: The function is guaranteed not to throw an exception.
  - `noexcept(false)`: The function may throw an exception (this is the default).
- Using `noexcept` can help the compiler optimize code.
- If a function declared `noexcept` throws an exception, `std::terminate()` is called.

```
#include <iostream>

void noThrowFunction() noexcept {
    std::cout << "This function will not throw." << std::endl;
}

void mayThrowFunction() {
    std::cout << "This function might throw." << std::endl;
    // throw std::runtime_error("An exception occurred!"); // Uncomment
to see what happens
}

int main() {
    try {
        noThrowFunction();
        mayThrowFunction();
    }
    catch (const std::exception& e) {
        std::cerr << "Caught an exception: " << e.what() << std::endl;
        return 1;
    }
    std::cout << "Program continues." << std::endl;
    return 0;
}
```

The `noThrowFunction` is declared with `noexcept`, guaranteeing it won't throw. If you uncomment the throw line in `mayThrowFunction`, you'll see how exceptions are handled.

11. Stack Unwinding

- When an exception is thrown, the program searches for a matching `catch` block.

- If no matching `catch` block is found in the current function, the function's execution is terminated.
- The process of unwinding the function call stack involves destroying local objects in each terminated function.
- This is important for resource management (e.g., ensuring that destructors are called to release memory or close files).

```
#include <iostream>

class MyClass {
public:
    MyClass(const std::string& name) : name_(name) {
        std::cout << "Constructor for " << name_ << std::endl;
    }
    ~MyClass() {
        std::cout << "Destructor for " << name_ << std::endl;
    }
private:
    std::string name_;
};

void func1() {
    MyClass obj1("obj1");
    std::cout << "func1() called" << std::endl;
    throw std::runtime_error("Exception from func1");
}

void func2() {
    MyClass obj2("obj2");
    std::cout << "func2() called" << std::endl;
    func1();
}

int main() {
    try {
        func2();
    }
    catch (const std::runtime_error& e) {
        std::cerr << "Caught exception: " << e.what() << std::endl;
    }
    return 0;
}
```

This example demonstrates stack unwinding. When the exception is thrown in `func1()`, the `MyClass` destructor is called for `obj1` before the exception is caught in `main()`.

12. **Resource Acquisition Is Initialization (RAII)**

- RAII is a C++ programming technique that ties the management of resources (e.g., memory, file handles, locks) to the lifetime of objects.
- In RAII, a constructor acquires the resource, and the destructor releases the resource.
- RAII works very well with exception handling because during stack unwinding, destructors are automatically called for objects that have gone out of scope.
- **Smart Pointers:** `std::unique_ptr`, `std::shared_ptr`, and `std::weak_ptr` are examples of RAII classes that manage dynamically allocated memory.

```
#include <iostream>
#include <memory>
#include <fstream>

void processFile(const std::string& filename) {
    // Use RAII to manage the file
    std::unique_ptr<std::fstream> filePtr(new std::fstream(filename,
std::ios::in));

    if (!filePtr->is_open()) {
        throw std::runtime_error("Could not open file");
    }

    std::cout << "File opened successfully." << std::endl;
    // ... use the file ...
    // The file will be automatically closed when filePtr goes out of
scope, even if an exception occurs.
}

int main() {
    try {
        processFile("myFile.txt"); // Create a file named "myFile.txt"
in the same directory.
    }
    catch (const std::exception& e) {
        std::cerr << "Exception: " << e.what() << std::endl;
        return 1;
    }
    return 0;
}
```

In this example, the `std::unique_ptr` manages the file stream. The file is automatically closed when the `unique_ptr` goes out of scope, whether an exception is thrown or not.

13. Exception Safety

- Exception safety refers to how a function or code handles exceptions.
- A function is exception-safe if it meets certain guarantees in the presence of exceptions.
- Levels of Exception Safety:
  - **No-throw guarantee:** The function will not throw an exception. Functions declared `noexcept` provide this guarantee.
  - **Strong exception safety:** If an exception is thrown, the function has no side effects. The state of the program remains as it was before the function was called.
  - **Basic exception safety:** If an exception is thrown, the function ensures that no resources are leaked and the program remains in a valid (but possibly unspecified) state.
  - **No guarantee:** The function might leave the program in an invalid or unpredictable state if an exception is thrown.
- Aim for strong or basic exception safety whenever possible.

```
#include <iostream>
#include <vector>

// Example demonstrating basic exception safety
void resizeVector(std::vector<int>& vec, size_t newSize) {
    try {
        std::vector<int> temp(newSize); // Allocate new memory
        // Copy elements (potential exception during copy)
        for (size_t i = 0; i < std::min(vec.size(), newSize); ++i) {
            temp[i] = vec[i];
        }
        vec.swap(temp); // Swap data (no-throw operation)
    }
    catch (const std::exception& e) {
        // If an exception occurs during the copy, vec is unchanged
(basic guarantee).
        std::cerr << "Exception: " << e.what() << std::endl;
        // You might rethrow the exception here if you can't handle it.
        throw;
    }
    // If no exception, vec now has the new size (strong guarantee).
}

int main() {
```

```
    std::vector<int> myVector = { 1, 2, 3, 4, 5 };
    try {
        resizeVector(myVector, 10);
        std::cout << "Vector resized successfully." << std::endl;
    }
    catch (const std::exception& e) {
        std::cerr << "Failed to resize vector." << std::endl;
    }

    for (int val : myVector) {
        std::cout << val << " ";
    }
    std::cout << std::endl;
    return 0;
}
```

The `resizeVector` function demonstrates basic exception safety. If the memory allocation or element copying throws an exception, the original vector is still in a valid state.

14. `std::terminate()`

- The `std::terminate()` function is called in the following situations:
  - An exception is thrown and no matching `catch` handler is found.
  - A function declared `noexcept` throws an exception.
  - The destructor of a static or global object throws an exception during stack unwinding.
- The default behavior of `std::terminate()` is to call `std::abort()`, which terminates the program immediately.
- You can use the `std::set_terminate()` function to install your own termination handler.

```
#include <iostream>
#include <stdexcept>
#include <cstdlib> // For std::abort

void myTerminateHandler() {
    std::cerr << "Custom terminate handler called.  Program aborting."
<< std::endl;
    std::abort(); // Terminate the program.  You could also log the
error, etc.
}

int main() {
```

```
    std::set_terminate(myTerminateHandler); // Set the custom terminate
handler

    try {
        throw std::runtime_error("Unhandled exception");
    }
    catch (int i) {  // Added a catch block, but it doesn't catch the
exception thrown.
        std::cout << "Caught int exception" << std::endl;
    }
    // If the above catch block is removed, std::terminate will be
called.
    std::cout << "This will not be printed." << std::endl;
    return 0;
}
```

This example shows how to use `std::set_terminate` to replace the default terminate handler with a custom function.

15. **Best Practices**

- **Use Exceptions for Exceptional Circumstances:** Don't use exceptions for normal control flow. Use them for error conditions that are unexpected and cannot be easily handled locally.
- **Throw by Value, Catch by Reference (const):** This avoids unnecessary copying and object slicing.
- **Use Standard Exceptions When Possible:** Use the standard exception classes from `<stdexcept>` whenever they are appropriate.
- **Create Custom Exception Classes When Necessary:** Derive custom exception classes from `std::exception` to represent application-specific errors.
- **Be Specific in `catch` Blocks:** Catch specific exception types rather than using a generic `catch (...)` whenever possible. This allows you to handle different errors in different ways.
- **Keep `try` Blocks Small:** Enclose only the code that might directly throw an exception in a `try` block. This improves code readability.
- **Handle Exceptions at the Appropriate Level:** Catch exceptions where you can take meaningful action. If you can't handle an exception in a function, let it propagate to the caller.
- **Use RAII to Manage Resources:** Use RAII (e.g., smart pointers) to ensure that resources are properly released, even if exceptions are thrown.
- **Write Exception-Safe Code:** Strive to write functions that provide at least basic exception safety.

- **Provide Informative Error Messages:** Make sure your exception objects provide useful information about the error. The `what()` method is used for this purpose.
- **Log Exceptions:** Consider logging exceptions, especially those that are not caught and lead to program termination. This can help in debugging.
- **Avoid throwing exceptions in destructors:** Destructors should not throw exceptions. If a destructor throws an exception during stack unwinding, `std::terminate` will be called.