C++ Exception Handling - Detailed Tutorial

1. Introduction to Exceptions in C++

Exceptions in C++ provide a mechanism to handle runtime errors or unexpected situations in a clean and organized way. Instead of cluttering code with error checks, exceptions allow separation of normal logic from error-handling logic.

2. How Exceptions Work

Exception Handling Flow:

1. **throw**: Used to signal an error.
2. **try**: Wrap code that may throw exceptions.
3. **catch**: Capture and handle exceptions.
4. **Stack unwinding**: Call stack is unwound to find a suitable catch block.
5. **If not caught**: `std::terminate()` is called, crashing the program.

3. Built-in Exception Classes and Hierarchy

Root Exception Class: `std::exception`

```
namespace std {
    class exception {
    public:
        exception() noexcept;
        exception(const exception&) noexcept;
        exception& operator=(const exception&) noexcept;
        virtual ~exception();

        virtual const char* what() const noexcept;
    };
}
```

Example: `std::runtime_error`

```
namespace std {
    class runtime_error : public exception {
    private:
        std::string _msg;

    public:
        explicit runtime_error(const std::string& what_arg)
            : _msg(what_arg) {}

        const char* what() const noexcept override {
            return _msg.c_str();
        }
    };
}
```

`override` is optional but recommended. It ensures the function correctly overrides a virtual base function.

4. Full Exception Hierarchy in Standard Library

```
std::exception
├── std::logic_error
│    ├── std::invalid_argument
│    ├── std::domain_error
│    ├── std::length_error
│    └── std::out_of_range
├── std::runtime_error
│    ├── std::range_error
│    ├── std::overflow_error
│    └── std::underflow_error
├── std::bad_alloc
├── std::bad_cast
├── std::bad_typeid
├── std::bad_exception
├── std::ios_base::failure
├── std::system_error
│    ├── std::future_error
│    └── std::filesystem_error
└── std::regex_error
```

Common Examples:

- `std::out_of_range`: Accessing a vector out of bounds
- `std::bad_alloc`: Allocation failure (e.g., `new` with too much memory)
- `std::invalid_argument`: Passing invalid values to a function

5. Important Headers

```
#include <exception>   // For std::exception and base types
#include <stdexcept>   // For logic_error, runtime_error, etc.
```

6. noexcept and std::nothrow

`noexcept`

- Specifies a function **cannot throw**.
- Violating this causes `std::terminate()`.

```
void safeFunc() noexcept {
    // cannot throw
}
```

`std::nothrow`

- Ensures `new` returns `nullptr` on failure instead of throwing `std::bad_alloc`

```
int* arr = new(std::nothrow) int[100000000];
```

7. Writing a Custom Exception

Steps:

1. Derive from `std::exception` or derived class.
2. Override `what()`.
3. Add custom members if needed.

```
class MyError : public std::exception {
    std::string message;
public:
    explicit MyError(const std::string& msg) : message(msg) {}
    const char* what() const noexcept override {
        return message.c_str();
    }
};
```

8. Catching Exceptions

Basic:

```
try {
    // risk code
} catch (const std::exception& e) {
    std::cerr << e.what();
}
```

Catch Order Matters:

- Catch **derived** types before **base**:

```
try {
    // ...
} catch (const std::out_of_range& e) {
    // specific
} catch (const std::exception& e) {
    // general
}
```

Catch All:

```
catch (...) {
    // handle unknown exception
}
```

9. Rethrowing Exceptions

Use `throw;` to rethrow from a `catch` block

```
try {
    // ...
} catch (...) {
    log();
    throw; // rethrows the same exception
```

```
}
```

10. Best Practices

- Catch by `const&` to avoid slicing.
- Use `noexcept` where possible.
- Never throw from a destructor.
- Always clean up resources using RAII or smart pointers.
- Handle at least at the `main()` level.
- Only use `throw ...` for unexpected errors — not flow control.

Summary

C++ provides a rich set of tools for exception handling. Understanding the hierarchy, usage of `try/catch`, creating custom exceptions, and applying best practices ensures safe, robust, and maintainable C++ code.