Exceptions

1 Overview

- Exceptions are used to handle runtime errors in C++.
- Control is transferred from the point of error to a handler.
- Main keywords: `try`, `catch`, `throw`.

2 Basic Syntax

```
try {
    // Code that may throw an exception
    if (x == 0) {
        throw std::runtime_error("Division by zero");
    }
    result = y / x;
} catch (const std::exception& e) {
    std::cerr << "Error: " << e.what() << std::endl;
}
```

3 Key Points

- `throw` raises an exception.
- `catch` defines how to handle it.
- Handlers can catch by type (e.g., `int`, `std::exception`).
- Uncaught exceptions cause program termination.
- Multiple `catch` blocks can be used for different types.

4 Deallocation and Resource Management

- When an exception is thrown, C++ automatically destroys all local objects created since entering the `try` block.
- This process is called **stack unwinding**.
- Destructors of objects are called, so resources (memory, files, etc.) are released correctly.
- If raw pointers are used, memory leaks may occur unless managed carefully.
- Best practice: use RAII (Resource Acquisition Is Initialization) with smart pointers or resource-managing classes.

5 Example: Automatic Deallocation

```
struct File {
    File(const char* name) { f = fopen(name, "r"); }
    ~File() { if (f) fclose(f); }  // destructor cleans up
    FILE* f;
};

try {
    File file("data.txt");  // allocated here
    throw std::runtime_error("Some error");
} catch (const std::exception& e) {
    std::cerr << "Caught: " << e.what() << std::endl;
}
// file is automatically closed by destructor
```

6 Notes

- Prefer using standard exceptions from `<stdexcept>`.
- Exception handling separates error-handling code from normal logic.
- Use exceptions for exceptional cases, not regular control flow.
- RAII ensures resources are always freed, even when exceptions occur.