Gorgo Starting C++

April 11, 2026

Contents

11. Exceptions

So far, when something goes wrong in your programs you have printed an error message and returned early. That works when the error happens in `main()`, but what about a function buried three or four calls deep? You would have to thread error codes back through every function in the chain, and every caller would have to check the return value — tedious and easy to get wrong. C++ provides **exceptions**, a mechanism that lets a function signal an error and lets code much higher up in the call stack handle it. C++23 also introduces `std::expected`, which gives you a way to return either a value or an error without the overhead of exceptions. In this chapter you will learn how to throw and catch exceptions, how the stack unwinds when an exception is thrown, when to use `noexcept`, and how `std::expected` offers a lightweight alternative.

Throwing Exceptions

When a function encounters a situation it cannot handle, it **throws** an exception using the `throw` keyword:

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

int parse_track(const std::string &s) {
    int n = std::stoi(s);
    if (n < 1) {
        throw std::out_of_range("track number must be positive");
    }
    return n;
}
```

When `throw` executes, the function stops immediately. It does not return a value — control leaves the function and travels up the call stack looking for something that can handle the error.

The `<stdexcept>` header provides several standard exception types, all derived from `std::exception`:

Type	When to use
`std::runtime_error`	general errors detected at runtime
`std::out_of_range`	a value is outside an acceptable range
`std::invalid_argument`	an argument does not make sense
`std::logic_error`	a bug in the program's logic
`std::overflow_error`	arithmetic overflow

All of them take a `std::string` message in their constructor and provide a `what()` member function that returns it.

Catching Exceptions

To handle an exception, wrap the code that might throw in a **try block** and follow it with one or more **catch blocks**:

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

int parse_track(const std::string &s) {
    int n = std::stoi(s);
    if (n < 1) {
        throw std::out_of_range("track number must be positive");
```

```
    }
    return n;
}

int main()
{
    try {
        int track = parse_track("0");
        std::cout << "Track: " << track << std::endl;
    } catch (const std::out_of_range &e) {
        std::cout << "Error: " << e.what() << std::endl;
    }

    return 0;
}
```

Output:

```
Error: track number must be positive
```

When `parse_track` throws `std::out_of_range`, the rest of the `try` block is skipped and the matching `catch` block runs. After the `catch` block finishes, execution continues normally after it — the program does not crash.

Multiple Catch Blocks

You can have multiple `catch` blocks to handle different exception types. The compiler tries them in order and uses the first one that matches:

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

int parse_volume(const std::string &s) {
    int v = std::stoi(s);
    if (v < 0 || v > 11) {
        throw std::out_of_range("volume must be 0-11");
    }
    return v;
}

int main()
{
    try {
        int v = parse_volume("abc");
        std::cout << "Volume: " << v << std::endl;
    } catch (const std::out_of_range &e) {
        std::cout << "Out of range: " << e.what() << std::endl;
    } catch (const std::invalid_argument &e) {
        std::cout << "Bad input: " << e.what() << std::endl;
    }

    return 0;
}
```

Output:

```
Bad input: stoi
```

Here `std::stoi("abc")` throws `std::invalid_argument`, so the second `catch` block handles it. If you passed `"99"` instead, `parse_volume` would throw `std::out_of_range` and the first `catch` would handle it.

Catching Everything

You can catch any exception with `catch (...)`:

```
try {
    risky_function();
} catch (const std::exception &e) {
    std::cout << "Known error: " << e.what() << std::endl;
} catch (...) {
    std::cout << "Unknown error" << std::endl;
}
```

The `catch (...)` block is a last resort — it catches any exception, including types that do not derive from `std::exception`. Always catch specific types first and use `catch (...)` only as a safety net.

Tip: Always catch exceptions by `const` reference (`const std::exception &e`). Catching by value makes a copy and can slice off information from derived exception types.

Stack Unwinding

When an exception is thrown, C++ **unwinds the stack** — it walks back up the call chain, destroying local variables in each function along the way, until it finds a matching `catch` block. This means destructors run automatically, which is critical for cleaning up resources.

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

struct Song {
    std::string title;
    Song(const std::string &t) : title(t) {
        std::cout << "  created: " << title << std::endl;
    }
    ~Song() {
        std::cout << "  destroyed: " << title << std::endl;
    }
};

void deep_function() {
    Song s("The Freshmen");
    throw std::runtime_error("something went wrong");
}

void middle_function() {
    Song s("Save Tonight");
    deep_function();
}

int main()
```

```
{
    try {
        middle_function();
    } catch (const std::runtime_error &e) {
        std::cout << "Caught: " << e.what() << std::endl;
    }

    return 0;
}
```

Output:

```
  created: Save Tonight
  created: The Freshmen
  destroyed: The Freshmen
  destroyed: Save Tonight
Caught: something went wrong
```

Even though no function returned normally, both `Song` destructors ran. `deep_function`'s `Song` ("The Freshmen") is destroyed first (most recent), then `middle_function`'s ("Save Tonight"). This automatic cleanup during stack unwinding is why destructors are so important — and why you should manage resources through objects rather than raw `new/delete`.

Trap: If a destructor throws an exception during stack unwinding (while another exception is already in flight), the program calls `std::terminate()` and crashes. Never throw from a destructor.

noexcept

The `noexcept` keyword promises the compiler that a function will not throw any exceptions:

```
int add(int a, int b) noexcept {
    return a + b;
}
```

If a `noexcept` function does throw (for example, by calling a function that throws), the program calls `std::terminate()` immediately — there is no stack unwinding, no catch blocks, just a crash.

`noexcept` is not just documentation — the compiler uses it to generate more efficient code. Standard library containers like `std::vector` check whether your move operations are `noexcept` before deciding whether to move or copy during reallocation. If your move constructor is `noexcept`, the vector moves elements (fast). If it is not, the vector falls back to copying (slow) because a failed move would leave the container in a broken state.

Tip: Mark functions `noexcept` when you are certain they will not throw. This is especially important for move constructors, move assignment operators, and destructors.

Trap: The compiler does not verify that a `noexcept` function actually avoids throwing. If you mark a function `noexcept` but it calls something that throws, you get `std::terminate()` at runtime with no warning at compile time.

std::expected

Exceptions are powerful, but they are not always the right tool. They are best for truly exceptional situations — file not found, out of memory, network failure. For errors that are a normal part of a function's contract (like parsing invalid user input), the overhead of exception handling can be unnecessary.

C++23 introduces `std::expected<T, E>` in the `<expected>` header. It holds either a value of type `T` (the success case) or an error of type `E` (the failure case) — but never both.

```
std::expected<T, E>
```

You return the value normally for success, and wrap the error in `std::unexpected` for failure:

```
#include <expected>
#include <iostream>
#include <string>

std::expected<int, std::string> parse_track(const std::string &s) {
    try {
        int n = std::stoi(s);
        if (n < 1) {
            return std::unexpected("track must be positive");
        }
        return n;
    } catch (...) {
        return std::unexpected("not a number");
    }
}

int main()
{
    auto result = parse_track("5");
    if (result) {
        std::cout << "Track: " << *result << std::endl;
    }

    auto error = parse_track("abc");
    if (!error) {
        std::cout << "Error: " << error.error() << std::endl;
    }

    return 0;
}
```

Output:

```
Track: 5
Error: not a number
```

Use `*result` or `result.value()` to get the value, and `result.error()` to get the error. The boolean check (`if (result)`) tells you whether it holds a value or an error.

Exceptions vs std::expected

When should you use which?

	Exceptions	`std::expected`
Best for	rare, truly exceptional failures	expected, routine failures
Error path	unwinds the stack	returns normally
Caller must check?	no — propagates automatically	yes — must inspect the return value
Performance	zero cost when no exception is thrown; expensive when thrown	small constant cost (size of the return type)

A good rule of thumb: if the caller is *likely* to handle the error immediately, use `std::expected`. If the error should propagate up several layers, use exceptions.

Tip: `std::expected` makes error handling explicit and visible in the return type. This is especially useful for functions where failure is a normal outcome, like parsing user input or looking up a key in a map.

Key Points

- Use `throw` to signal an error and `try/catch` to handle it.
- The standard exception types in `<stdexcept>` cover most common error categories.
- Always catch exceptions by `const` reference.
- Stack unwinding destroys local variables automatically when an exception propagates — this is why resource management through objects (RAII) matters.
- Never throw from a destructor.
- `noexcept` promises a function will not throw; violating the promise calls `std::terminate()`.
- Mark move constructors, move assignment operators, and destructors `noexcept`.
- `std::expected<T, E>` (C++23) returns either a value or an error without using exceptions.
- Use exceptions for rare failures that should propagate; use `std::expected` for routine errors the caller handles immediately.

Exercises

1. What does the following program print?

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

void step3() { throw std::runtime_error("oops"); }
void step2() { step3(); }
void step1() { step2(); }

int main()
{
    try {
        step1();
        std::cout << "A" << std::endl;
    } catch (const std::runtime_error &e) {
        std::cout << "B: " << e.what() << std::endl;
    }
    std::cout << "C" << std::endl;
```

```
    return 0;
}
```

2. What is wrong with this code?

```
try {
    int n = std::stoi(input);
} catch (const std::out_of_range &e) {
    std::cout << "out of range" << std::endl;
} catch (const std::exception &e) {
    std::cout << "error" << std::endl;
} catch (const std::invalid_argument &e) {
    std::cout << "bad input" << std::endl;
}
```

3. Why should you always catch exceptions by `const` reference rather than by value?

4. What does the following program print?

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

struct Amp {
    std::string name;
    Amp(const std::string &n) : name(n) {
        std::cout << name << " on" << std::endl;
    }
    ~Amp() {
        std::cout << name << " off" << std::endl;
    }
};

void soundcheck() {
    Amp a("Marshall");
    Amp b("Fender");
    throw std::runtime_error("feedback!");
}

int main()
{
    try {
        soundcheck();
    } catch (...) {
        std::cout << "handled" << std::endl;
    }
    return 0;
}
```

5. Will this code compile? If so, what happens when `play()` is called?

```
void load(const std::string &file) {
    throw std::runtime_error("file not found");
}

void play() noexcept {
    load("track01.wav");
```

```
}
```

6. What is the output of this program?

```
#include <expected>
#include <iostream>
#include <string>

std::expected<int, std::string> divide(int a, int b) {
    if (b == 0) {
        return std::unexpected("division by zero");
    }
    return a / b;
}

int main()
{
    auto r1 = divide(10, 3);
    auto r2 = divide(10, 0);

    if (r1) std::cout << *r1 << std::endl;
    if (!r2) std::cout << r2.error() << std::endl;

    return 0;
}
```

7. When would you use `std::expected` instead of throwing an exception? Give an example scenario for each.

8. How many destructors run before the `catch` block executes?

```
struct Song {
    std::string title;
    Song(const std::string &t) : title(t) {}
    ~Song() { std::cout << "destroyed: " << title << std::endl; }
};

void inner() {
    Song a("Torn");
    Song b("Vogue");
    throw std::runtime_error("oops");
}

void outer() {
    Song c("Iris");
    inner();
}

int main() {
    try {
        outer();
    } catch (...) {
        std::cout << "caught" << std::endl;
    }
    return 0;
}
```

9. Write a function `safe_sqrt` that takes a `double` and returns `std::expected<double, std::string>`. If the input is negative, return an error message. Otherwise, return the square root. Test it in `main()` with both a positive and a negative value.

10. **Where is the bug?**

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

int main()
{
    try {
        throw std::out_of_range("nope");
    }
    catch (...) {
        std::cout << "anything\n";
    }
    catch (const std::out_of_range &e) {
        std::cout << "out_of_range: " << e.what() << "\n";
    }
    return 0;
}
```

Catch handlers are tried in source order, top to bottom. Why is the second `catch` block effectively dead code? How would you reorder the handlers so that `out_of_range` is caught specifically and `catch(...)` only acts as a final safety net?

11. **Write a program** that defines a function

```cpp
int parse_age(const std::string &s);
```

that converts `s` to an integer using `std::stoi` and then returns it. Throw `std::invalid_argument("not a number")` if `std::stoi` itself throws `std::invalid_argument`, and throw `std::out_of_range("age must be 0..150")` if the parsed number is outside the range `[0, 150]`. In `main`, call `parse_age` on three inputs — `"42"`, `"abc"`, and `"-1"` — inside `try`/`catch` blocks that catch each of the two exception types separately and print a different message for each one.