Module 15 of 20

Errors and exceptions

throw, try and catch with the standard hierarchy and your own, the three exception-safety guarantees with RAII and noexcept, std::optional for absence, std::variant with std::visit for one-of-several, and error codes, assertions and a per-layer strategy.

A function that cannot do what it was asked has to say so, and C++ gives it four ways: throw, return an `std::optional`, return a `std::variant` of value-or-reason, or return a code. None of them is "the" answer. An exception cannot be ignored and carries itself through frames that have nothing to add, but it costs microseconds and hides the failure path from the reader; a return value is visible and cheap, but a caller can forget to check it. This module is where you learn each mechanism precisely — what `throw` does to the stack, what `what()` promises, which library calls throw and which never do, what `*opt` on an empty optional does (not what you hope) — and then learn to choose between them by the shape of the failure rather than by habit.

The five lessons build the toolbox and then the strategy. Exceptions covers `throw`, handler matching, catching by `const&` most-derived-first, the standard hierarchy and the split between `logic_error` and `runtime_error`, your own types with data members, `throw;` against `throw e;`, and stack unwinding. Exception safety names the no-throw, strong and basic guarantees, shows RAII delivering the basic one for free and a swap delivering the strong one, and explains `noexcept` — the promise, the operator, and why `std::vector` copies a type whose move constructor is missing the word. `std::optional` makes absence part of the type, with `value_or`, the `if (auto r = f())` idiom and the reference it cannot hold. `std::variant` with `std::visit` is the union that remembers, with the `overloaded` visitor, `std::monostate` and a small command language. Error codes and design closes with `errno`, `enum class` codes under `[[nodiscard]]`, `std::error_code` and `std::from_chars`, `std::expected` as C++23 reading, assertions that check bugs and never input, and the rule *return values inside, throw at the boundary*.

Every exercise is a whole program that provokes a failure and handles it — an exception that escapes `main` is a runtime error, so each one ends with a clean exit and its own fixed text, never the library's message. You write a parser that translates `std::stoi`'s exceptions, a bank with a `BankError` / `InsufficientFunds` hierarchy and handlers in the only order that works, a resource guard whose acquire/release trace proves unwinding runs destructors, four instrumented types that show `std::move_if_noexcept` moving or copying, an optional-returning price lookup and statistics

over a possibly empty range, a token stream and a turtle visited through `overloaded`, a calculator that returns codes, and a parse–validate–execute pipeline that reports the first layer to fail. The checkpoint's three programs — a score reader with its own hierarchy, an RPN evaluator returning a `std::variant`, and a key–value store whose transactions roll back through an RAII guard — draw on all of it.

CONTENTS

1	Exceptions — throw, try, catch and the standard hierarchy	14 min
2	Exception safety — guarantees, noexcept and RAII	14 min
3	std::optional — a value that may be absent	13 min
4	std::variant and std::visit — a type-safe union	14 min
5	Error codes, assertions and an error-handling strategy	14 min
✓	Checkpoint — Errors and exceptions	25 min

Exercises in this module compile on Clang 18 · C++20. Reference solutions and the answer key follow each lesson.

LESSON 1 · 14 MIN

Exceptions — throw, try, catch and the standard hierarchy

A function that cannot do what it was asked can return a special value, set a flag, or throw. The first two leave every caller to check, and a caller that forgets carries on with garbage. An exception cannot be ignored: `throw` abandons the current function, destroys everything on the way out and lands in the nearest handler that can deal with the failure — or, if there is none, ends the program. This lesson settles what a `throw` does, how a handler is chosen, what the standard library throws, what `what()` promises, and what stack unwinding runs.

Throwing and catching

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

double ratio(int numerator, int denominator) {
    if (denominator == 0) {
        throw std::invalid_argument("ratio: denominator is zero");
    }
    return static_cast<double>(numerator) / denominator;
}

int main() {
    try {
        std::cout << ratio(6, 3) << '\n';
        std::cout << ratio(1, 0) << '\n';   // throws: nothing after this line
        std::cout << "unreachable\n";
    } catch (const std::invalid_argument& e) {
        std::cout << "invalid: " << e.what() << '\n';
    }
    std::cout << "still running\n";
    return 0;
}
```

```
2
invalid: ratio: denominator is zero
still running
```

`throw` takes an expression of any type and creates the *exception object* from it — a copy or move into storage that survives the unwinding, so throwing a local is safe. The runtime then walks up the call stack looking for a `try` block with a handler that matches the object's type. When it finds one, every frame in between is discarded, the handler runs, and execution continues after the whole `try` / `catch` statement — never at the line that threw. Handlers are tested top to bottom and the first match wins; `catch (...)` matches anything and belongs only at the very top of a program. Throw objects derived from `std::exception`: `throw 42;` compiles, but leaves the catcher nothing to call `what()` on and no category to catch by.

Catch by const reference, most derived first

`catch (const std::exception& e)` is the form to use every time. Catching by value copies the exception object and, worse, *slices* it (Module 10, lesson 4): a `std::out_of_range` caught as a `std::exception` by value is a plain `std::exception` inside the handler — `what()` survives, the dynamic type and every derived member do not — and `throw e;` from there throws that slice. (A bare `throw;` still rethrows the original object, because the parameter is only a copy of it; both compilers warn about the by-value catch regardless.) A reference binds to the real object, keeps its dynamic type and costs nothing.

A handler for a base class matches every derived type, so order matters: with `catch (const std::exception&)` written before `catch (const std::out_of_range&)`, the second handler can never run, and both GCC and Clang warn that the exception "will be caught by earlier handler". Most derived first, most general last.

The standard hierarchy

Every exception the standard library throws derives from `std::exception` (`<exception>`); the useful subclasses live in `<stdexcept>`.

Type	Base	Thrown by
`std::logic_error`	`exception`	a precondition the caller should have checked
`std::invalid_argument`	`logic_error`	`std::stoi("abc")`, `std::bitset` built from a bad string
`std::out_of_range`	`logic_error`	`at()` on a vector, string or map; `std::stoi("99999999999")`
`std::runtime_error`	`exception`	a failure only detectable while running
`std::overflow_error`, `range_error`	`runtime_error`	arithmetic results; `std::bitset::to_ulong`
`std::system_error`	`runtime_error`	the OS said no; carries a `std::error_code` (lesson 5)
`std::bad_alloc`	`exception`	`new` cannot get memory
`std::bad_optional_access`, `std::bad_variant_access`	`exception`	lessons 3 and 4

The split at the top is the one to remember. A `logic_error` says *the program is wrong* — an argument was outside its documented range, and a check before the call would have avoided it. A `runtime_error` says *the world is wrong* — a file was missing, a number in the input did not fit. Derive your own types from `std::runtime_error` unless you mean "this is a bug".

Which calls throw, and which do not

`v.at(i)`, `s.at(i)` and `m.at(key)` throw `std::out_of_range`; `v[i]` and `s[i]` never throw — an index out of range is undefined behaviour — and `m[key]` silently inserts.

`std::stoi`, `stol`, `stoll` and `stod` throw `std::invalid_argument` when no digits can be read and `std::out_of_range` when the value does not fit; `std::from_chars` throws nothing (Module 5, lesson 3).

`new T` throws `std::bad_alloc` when memory runs out, so every container operation that allocates can throw it too.

`std::cin >> n` never throws by default: a bad token sets the fail bit and leaves `n` zero (Module 18, lesson 1).

---

what() and the message

`what()` returns a `const char*`. For `std::runtime_error`, `std::logic_error` and their subclasses the standard promises it is the string you passed to the constructor, so an exception you built carries exactly your words. For the exceptions the library throws itself, the text is whatever the implementation chose: libstdc++ says `stoi` where MSVC says `invalid stoi argument`. Code whose output must be exact — every exercise in this track — catches the library's exceptions and prints its own fixed text, and prints `what()` only for exceptions it threw itself.

Your own exception types

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

class BankError : public std::runtime_error {
public:
    using std::runtime_error::runtime_error;   // inherit the constructor that
};

class InsufficientFunds : public BankError {
public:
    InsufficientFunds(long long needed, long long available)
        : BankError("insufficient funds"), needed_(needed), available_(availab
    long long needed() const noexcept { return needed_; }
    long long available() const noexcept { return available_; }
private:
    long long needed_;
    long long available_;
};
```

A small hierarchy lets each caller catch at the level it can handle: the teller loop catches `InsufficientFunds` and asks for a smaller amount, the layer above catches `BankError` and

abandons the transaction, `main` catches `std::exception` and reports. Data members carry what a message cannot — the numbers a handler needs in order to act.

Rethrowing and translating

```
try {
    applyLine(line);
} catch (const std::out_of_range&) {
    throw ConfigError("line " + std::to_string(lineNo) + ": index out of range
} catch (const std::exception& e) {
    std::cerr << "line " << lineNo << ": " << e.what() << '\n';
    throw;
}
```

`throw;` with no operand rethrows the exception currently being handled — the original object, dynamic type intact. `throw e;` throws a *copy* of `e`, sliced if `e` was caught as a base reference. The first handler shows *translation*: a low-level exception is caught at a module boundary and rethrown as a type that means something to the caller, with context added.

Stack unwinding

Between the `throw` and the handler, every automatic object in every abandoned frame is destroyed, in reverse order of construction, exactly as if each function had returned. This is *stack unwinding*, and it is why destructors — not `finally` blocks — are where C++ cleans up:

```
struct Trace {
    const char* name;
    ~Trace() { std::cout << "leave " << name << '\n'; }
};

void inner() { Trace t{"inner"}; throw std::runtime_error("boom"); }
void outer() { Trace t{"outer"}; inner(); std::cout << "after inner\n"; }

int main() {
    try { outer(); } catch (const std::runtime_error& e) { std::cout << "caugh
}
```

```
leave inner
leave outer
caught boom
```

`after inner` never prints: the throw leaves `inner`, then `outer`, running both destructors before the handler's first statement. An exception nobody catches calls `std::terminate()`, which aborts the process — the judge reports it as a runtime error — and whether the destructors run before that is implementation-defined, so "an uncaught exception will still close my file" is not a promise you have. `main` gets no handler for free.

Compilers implement exceptions with tables, not checks: a `try` block that never throws costs nothing, while a `throw` that happens costs microseconds — a thousand times a `return`. Exceptions are for the *exceptional* path, not for "key not found" inside a loop; lesson 5 turns that into a strategy per layer.

Pitfalls

Mistake	What happens
`catch (std::exception e)` by value	A copy, sliced to the base; the compiler warns, and `throw e;` from inside throws the slice
General handler before a specific one	The specific handler is unreachable; the compiler warns
`throw e;` to rethrow	Copies, and slices if `e` is a base reference; use `throw;`
Printing `what()` of `std::stoi`'s exception	Implementation-specific text; print your own
Expecting `v[i]` to throw	It is undefined behaviour; only `at()` throws
No `try` in `main`	An uncaught exception terminates the program

Key takeaways

`throw` creates an exception object and unwinds to the first matching handler; execution resumes after the `try` / `catch`, never at the throw.

Catch by `const&`, most derived first; a base handler matches every derived type.

`logic_error` means the program is wrong, `runtime_error` means the world is; `at()`, `stoi` and `new` throw, `[]`, `from_chars` and `>>` do not.

`what()` is your text for your exceptions and unspecified text for the library's — print your own.

`throw;` rethrows the original; `throw e;` copies and slices.

Unwinding runs destructors in reverse; an uncaught exception calls `std::terminate`.

EXERCISE 1

Safe integer parser

Write a parser that never lets `std::stoi`'s exceptions escape. Read whitespace-separated tokens until the end of input and pass each to `int parseInt(const std::string& text)`, which calls `std::stoi(text, &consumed)` and treats a token that `std::stoi` only partly consumes (`consumed ≠ text.size()`, as in `12x` or `3.7`) as invalid by throwing `std::invalid_argument` yourself.

In `main`, catch `std::invalid_argument` and `std::out_of_range` in separate handlers and print your own fixed words — never `what()`: the text of the library's exception is implementation-specific. Count how many tokens parsed.

Input: tokens separated by whitespace, until EOF. **Output:** one line per token — `<token> → <value>`, `<token> → invalid` or `<token> → out of range` — then `parsed <k> of <n>`.

```
42 -7 abc 12x 2147483648 +8
```

prints

```
42 → 42
-7 → -7
abc → invalid
12x → invalid
2147483648 → out of range
+8 → 8
parsed 3 of 6
```

INPUT 1

```
42 -7 abc 12x 2147483648 +8
```

OUTPUT 1

```
42 → 42
-7 → -7
abc → invalid
12x → invalid
2147483648 → out of range
+8 → 8
parsed 3 of 6
```

INPUT 2

```
007 -2147483648 1e3
```

OUTPUT 2

```
007 → 7
-2147483648 → -2147483648
1e3 → invalid
parsed 2 of 3
```

REFERENCE SOLUTION

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

// Returns the value; lets std::stoi's std::invalid_argument and std::out_of_range
// pass through, and refuses a token that was only partly consumed.
int parseInt(const std::string& text) {
    std::size_t consumed = 0;
    int value = std::stoi(text, &consumed);
    if (consumed != text.size()) throw std::invalid_argument("trailing characters");
    return value;
}

int main() {
    std::string token;
    int total = 0;
    int parsed = 0;
    while (std::cin >> token) {
        ++total;
        try {
            int value = parseInt(token);
            ++parsed;
            std::cout << token << " -> " << value << '\n';
        } catch (const std::invalid_argument&) {
            std::cout << token << " -> invalid\n";
        } catch (const std::out_of_range&) {
            std::cout << token << " -> out of range\n";
        }
    }
    std::cout << "parsed " << parsed << " of " << total << '\n';
    return 0;
}
```

EXERCISE 2

A bank with its own hierarchy

Write a small exception hierarchy and a bank that throws it. `BankError` derives from `std::runtime_error` and inherits its constructor; `InsufficientFunds` derives from `BankError` and carries `needed()` and `available()` as `long long` data members (its message can be anything). `Bank` keeps a `std::map<std::string, long long>` and has `open`, `deposit` and `withdraw`, each returning the account's new balance. Rules, checked in this order: a negative amount throws `std::invalid_argument("negative amount")`; a `deposit` or `withdraw` on an unknown account throws `BankError("unknown account <name>")`; a `withdraw` larger than the balance throws

`InsufficientFunds(amount, balance)`. `open` sets the balance whether or not the account exists. Any other command word throws `std::invalid_argument("unknown command <word>")`.

One `try` per command in `main`, with three handlers **most derived first** — `InsufficientFunds`, then `BankError`, then `std::invalid_argument` — each printing the line shown below. Every message you print comes from an exception you threw yourself, so `what()` is safe here.

Input: lines of `<command> <name> <amount>` until EOF. **Output:** per command, one of `ok <name> <balance>`, `insufficient: need <amount>, have <balance>`, `bank error: <what()>`, `invalid: <what()>`.

```
open alice 100
withdraw alice 30
withdraw alice 100
withdraw bob 5
deposit alice -5
deposit alice 10
```

prints

```
ok alice 100
ok alice 70
insufficient: need 100, have 70
bank error: unknown account bob
invalid: negative amount
ok alice 80
```

INPUT 1

```
open alice 100
withdraw alice 30
withdraw alice 100
withdraw bob 5
deposit alice -5
deposit alice 10
```

OUTPUT 1

```
ok alice 100
ok alice 70
insufficient: need 100, have 70
bank error: unknown account bob
invalid: negative amount
ok alice 80
```

INPUT 2

```
open bob 0
withdraw bob 0
withdraw bob 1
```

OUTPUT 2

```
ok bob 0
ok bob 0
insufficient: need 1, have 0
```

REFERENCE SOLUTION

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

class BankError : public std::runtime_error {
public:
    using std::runtime_error::runtime_error;   // BankError("message")
};

class InsufficientFunds : public BankError {
public:
    InsufficientFunds(long long needed, long long available)
        : BankError("insufficient funds"), needed_(needed), available_(available) {}
    long long needed() const noexcept { return needed_; }
    long long available() const noexcept { return available_; }
private:
    long long needed_;
    long long available_;
};

class Bank {
public:
    long long open(const std::string& name, long long amount) {
        checkAmount(amount);
        accounts_[name] = amount;
        return amount;
    }
    long long deposit(const std::string& name, long long amount) {
        checkAmount(amount);
        return balanceOf(name) += amount;
    }
    long long withdraw(const std::string& name, long long amount) {
        checkAmount(amount);
        long long& balance = balanceOf(name);
        if (amount > balance) throw InsufficientFunds(amount, balance);
        return balance -= amount;
    }
private:
    std::map<std::string, long long> accounts_;

    static void checkAmount(long long amount) {
        if (amount < 0) throw std::invalid_argument("negative amount");
    }
    long long& balanceOf(const std::string& name) {
        auto it = accounts_.find(name);
        if (it == accounts_.end()) throw BankError("unknown account " + name);
        return it->second;
    }
```

```
};

int main() {
    Bank bank;
    std::string command, name;
    long long amount;
    while (std::cin >> command >> name >> amount) {
        try {
            long long balance = 0;
            if (command == "open") balance = bank.open(name, amount);
            else if (command == "deposit") balance = bank.deposit(name, amount);
            else if (command == "withdraw") balance = bank.withdraw(name, amount);
            else throw std::invalid_argument("unknown command " + command);
            std::cout << "ok " << name << ' ' << balance << '\n';
        } catch (const InsufficientFunds& e) {
            std::cout << "insufficient: need " << e.needed() << ", have " << e.avail
        } catch (const BankError& e) {
            std::cout << "bank error: " << e.what() << '\n';
        } catch (const std::invalid_argument& e) {
            std::cout << "invalid: " << e.what() << '\n';
        }
    }
    return 0;
}
```

QUIZ

1. What does this print?

```
struct T {
    const char* n;
    ~T() { std::cout << n; }
};
void f() { T a{"a"}; T b{"b"}; throw std::runtime_error("x"); }
int main() {
    try { f(); std::cout << "r"; }
    catch (const std::exception&) { std::cout << "c"; }
}
```

A. `bac`

B. `abc`

C. `c`

D. `rbac`

2. Why is `catch (const std::exception& e)` preferred over `catch (std::exception e)`?

A. The reference binds to the real exception object, keeping its dynamic type; catching by value copies and slices a derived exception down to the base

B. Catching by value is a compile error for polymorphic types

C. The reference form is the only one that allows `throw;` inside the handler

D. A `const&` handler also matches exceptions of unrelated types

3. A `try` block has `catch (const std::exception&)` first and `catch (const std::out_of_range&)` second. What happens when `v.at(9)` throws?

A. The `out_of_range` handler runs, because it is the more specific match

B. The `std::exception` handler runs; the second handler can never be reached and the compiler warns

C. Compile error: handlers must be ordered most general first

D. Both handlers run in order

4. Which of these throws an exception?

A. `v[9]` on a `std::vector` of three elements

B. `v.at(9)` on a `std::vector` of three elements

C. `m[key]` on a `std::map` that has no such key

D. `std::from_chars` given `"abc"`

5. Inside `catch (const std::exception& e)`, what is the difference between `throw;` and `throw e;`?

A. None — both rethrow the same object

B. `throw;` rethrows the original exception object with its dynamic type intact; `throw e;` throws a copy of `e`, sliced to `std::exception`

C. `throw e;` is faster because it avoids a lookup of the current exception

D. `throw;` is only permitted in `main`

6. You are writing `Account::withdraw`, which fails when the balance is short. Which standard base should your exception type derive from?

A. `std::logic_error` — the caller should have checked the balance first

B. `std::runtime_error` — a short balance is the state of the world at run time, not a bug in the program

C. `std::out_of_range` — the amount is out of the balance's range

D. `std::bad_alloc` — no funds were allocated

7. What happens when an exception is thrown and no handler anywhere on the call stack matches it?

A. `main` returns 1 automatically

B. `std::terminate` is called and the process aborts — whether destructors run first is implementation-defined

C. The exception is discarded and execution continues after the `throw`

D. The runtime converts it to a `std::exception` and retries the handlers

ANSWER KEY · EXCEPTIONS — THROW, TRY, CATCH AND THE STANDARD HIERARCHY

1. A Unwinding destroys `f`'s automatics in reverse order — `b` then `a` — before the handler runs, so `ba` precedes `c`. `r` never prints: execution resumes after the whole `try` / `catch`, not at the statement after the call.

2. A By value, a `std::out_of_range` becomes a plain `std::exception` inside the handler — compilers warn about it. `throw;` works in either form because it rethrows the original object, not the parameter; a reference handler matches exactly the same types as a value handler.

3. B Handlers are tried top to bottom and the first match wins; a base-class handler matches every derived type. GCC and Clang both warn that the later handler "will be caught by earlier handler". Exactly one handler ever runs.

4. B `at()` checks and throws `std::out_of_range`. `operator[]` on a vector does no check — out of range is undefined behaviour — `map::operator[]` silently inserts a default, and `std::from_chars` reports through an error code and never throws.

5. B A bare `throw;` reuses the exception in flight, so an outer `catch (const std::out_of_range&)` still matches. `throw e;` copy-constructs a new object from the static type of `e`, and the derived part is gone.

6. B `logic_error` means the program is wrong and a check before the call would have avoided the throw; `runtime_error` means the world refused. The lesson's rule: derive from `runtime_error` unless you mean "this is a bug". `out_of_range` and `bad_alloc` have fixed meanings that do not fit.

7. B An uncaught exception ends the program through `std::terminate`; the judge reports it as a runtime error. The standard does not promise unwinding before termination, so RAII cleanup is not guaranteed on that path — `main` gets no handler for free.

LESSON 2 · 14 MIN

Exception safety — guarantees, noexcept and RAII

An exception that passes through a function is a return the function did not plan. What does the caller find afterwards — the old state, a new one, or half of each with a file handle leaked on the way? *Exception safety* is the discipline of answering that question for every function you write, and C++ makes the good answers cheap if you let objects own resources. This lesson settles the three guarantees, how RAII delivers the basic one for free, what `noexcept` promises and why `std::vector` reads it, why a destructor must never throw, and when a return value is the better tool.

The three guarantees

Dave Abrahams named them while building the STL, and every standard library function documents which it gives:

Guarantee	If the operation throws …	Examples
No-throw	it cannot	destructors, `swap`, `std::move` of a well-designed type, `size()`
Strong	the state is exactly as before — commit or roll back	`std::vector::push_back`, `std::map::insert` of one element
Basic	no leak and every invariant holds, but the state may have changed	`std::vector::assign`, most multi-step updates
none	a leak, a half-built object, a broken invariant	a bug

The order is deliberate: no-throw is strongest and rarest, basic is the floor. Consider a transfer between two ledgers:

```
void transfer(Ledger& from, Ledger& to, long long amount) {
    from.withdraw(amount);   // may throw InsufficientFunds: nothing has chang
    to.deposit(amount);      // may throw (say, an audit log allocation fails)
}
```

If `deposit` throws, both ledgers are valid objects and nothing leaked — the *basic* guarantee — but the money is gone. Making it strong means doing every step that can fail before the first step that changes anything, or being able to undo.

RAII makes the basic guarantee automatic

Stack unwinding runs destructors (lesson 1). So a resource held by an object whose destructor releases it is released on every exit path, including the ones you did not write:

```
void writeReport(const std::vector<Row>& rows, std::mutex& m) {
    std::ofstream out("report.txt");            // closed by ~ofstream
    std::lock_guard<std::mutex> lock(m);        // unlocked by ~lock_guard
    auto buffer = std::make_unique<char[]>(4096);   // freed by ~unique_ptr
    for (const auto& r : rows) format(out, r, buffer.get());   // may throw: c
}
```

The raw form — `FILE* f = fopen(...)`, `m.lock()`, `new char[4096]` — leaks all three the moment `format` throws, because the `fclose`, `unlock` and `delete[]` lines are never reached. Module 7, lesson 3 introduced RAII as a lifetime tool; this is its other half. The rule: never hold a resource in a raw variable that a throw could skip. Resources belong to objects.

No finally: the scope guard

Java and JavaScript run cleanup in `finally`. C++ deliberately has none, because a destructor generalises it — the cleanup is attached to the *thing*, not repeated at every call site. When the cleanup is a one-off action rather than a resource, wrap it in a guard:

```
template <typename F>
class ScopeExit {
public:
    explicit ScopeExit(F f) : f_(std::move(f)) {}
    ~ScopeExit() { if (active_) f_(); }
    void release() noexcept { active_ = false; }   // commit: the cleanup is r
    ScopeExit(const ScopeExit&) = delete;
    ScopeExit& operator=(const ScopeExit&) = delete;
private:
    F f_;
    bool active_ = true;
};

void update(Ledger& ledger, const Batch& batch) {
    Ledger backup = ledger;
    ScopeExit rollback([&] { ledger = std::move(backup); });   // runs on any
    for (const auto& entry : batch) ledger.apply(entry);       // ... includi
    rollback.release();                                        // ... unless e
}
```

Class template argument deduction names the lambda's type for you. The guard turns a basic-guarantee loop into a strong one: on a throw, the destructor restores the backup during unwinding; on success, `release()` disarms it.

The strong guarantee: risky work first, commit last

The general recipe needs no guard at all — do everything that can throw on a temporary, then commit with operations that cannot throw:

```
void Inventory::load(const std::string& path) {
    std::map<std::string, int> fresh = parse(path);   // every allocation and
    items_.swap(fresh);                               // swap is no-throw: the
}
```

If `parse` throws, `items_` was never touched. Copy-and-swap assignment (Module 9, lesson 2) is this recipe applied to `operator=`. The price is a temporary — a full copy for a large object — which is why the standard library gives the strong guarantee where it is cheap and documents basic where it is not. Say which you give in the function's comment.

noexcept

`void f() noexcept;` declares that `f` will not let an exception escape. It is a promise, not a check: the compiler will happily compile a `throw` inside `f`, and when one gets out, `std::terminate` is called with no unwinding to the caller. The specifier can be conditional — `noexcept(std::is_nothrow_move_constructible_v<T>)` — and the operator form `noexcept(expr)` asks whether an expression could throw without evaluating it: `noexcept(v.at(0))` is `false`, `noexcept(v.size())` is `true`.

Some functions are `noexcept` without saying so. Destructors are implicitly `noexcept` unless a member's destructor is not; defaulted special members (Module 9, lesson 4) are `noexcept` when every member's corresponding operation is. So a class whose members are `std::string` and `std::vector` gets a `noexcept` defaulted move for free, and a class with a user-written move constructor gets nothing unless you write the word.

Why std::vector reads noexcept

When a vector grows, it must relocate every existing element into the new buffer. Moving them is fast, but if a move throws halfway, the old buffer is already partly gutted and the strong guarantee `push_back` promises is impossible. Copying is safe — the old buffer stays intact until the last copy succeeds — but slow. The library resolves this with `std::move_if_noexcept(elem)`, which yields `T&&` when `T`'s move constructor is `noexcept` (or `T` has no copy constructor at all) and `const T&` otherwise. So a type whose move constructor is `noexcept` is moved; a type whose move constructor may throw is *copied* on every reallocation, silently, forever. Declare move constructors and move assignment `noexcept`. The instrumented exercise makes the decision visible.

Never throw from a destructor

Destructors run during unwinding. If one throws while an exception is already in flight, two exceptions exist at once and `std::terminate` is called — there is no way to handle both. Since C++11 destructors are implicitly `noexcept`, so a throwing destructor terminates even outside unwinding. The design consequence: a cleanup that can genuinely fail — flushing a file, committing a database transaction — gets an explicit member function that may throw, and the destructor calls a swallowing version:

```
class Connection {
public:
    void close() { if (open_) { open_ = false; sendClose(); } }   // may throw
    ~Connection() {
        try { close(); } catch (...) { /* log at most; never rethrow */ }
    }
private:
    bool open_ = true;
    void sendClose();
};
```

`std::fstream` does the same: its destructor closes and swallows any failure, and a program that needs to know calls `close()` and checks the stream.

Exceptions or return values?

Situation	Prefer
The caller expects the failure often — a lookup miss, user input that fails to parse	a return value: `std::optional` or an error code (lessons 3 and 5)
The failure is rare and the immediate caller cannot fix it	an exception
A constructor cannot establish its invariant	an exception — constructors have no return value
Operators, callbacks, deep call chains with no channel for a code	an exception
A hot loop, or a codebase built with `-fno-exceptions`	a return value
Destructors, `swap`, moves	neither — they must not fail

Pitfalls

Mistake	What happens
`new` / `lock()` / `fopen` in a raw variable	Leaked on any throw before the release line
A user-written move constructor without `noexcept`	`std::vector` copies on every reallocation
A destructor that throws	`std::terminate`, immediately
Changing state before the last step that can throw	Basic guarantee where the caller assumed strong
Marking a function `noexcept` that calls `at()` or allocates	A throw terminates instead of propagating

Key takeaways

- No-throw, strong, basic: know which one each function gives and say so.
- RAII plus unwinding gives the basic guarantee for free; a raw resource in a local is a leak waiting for a throw.
- Strong guarantee recipe: do the throwing work on a temporary, commit with `swap` or a move; a scope guard undoes on failure.
- `noexcept` is a promise enforced by `std::terminate`; destructors have it implicitly.
- `std::vector` moves on reallocation only when the move is `noexcept` — write the word on every move constructor.
- Never let a destructor throw; give failing cleanup an explicit `close()`.

EXERCISE 1

Released on every path

Prove that unwinding runs destructors, in order, before the handler. Write `Guard`, an RAII class whose constructor prints `acquire <name>` and whose destructor prints `release <name>` (copying deleted). Then run each job through a recursive `run(const Job&, std::size_t i)` that declares a `Guard` for resource `i` and recurses for `i + 1`, so every resource is held in its own frame. When all resources are held, the job either prints `work <name>` and returns, or throws `std::runtime_error("step failed in <name>")`. `main` catches per job. Do not write a `release` line by hand anywhere: the destructors produce every one of them, on the success path and on the exception path alike.

Input: `n`, then `n` lines `<name> <k> <resource-1> … <resource-k> <ok|fail>`. **Output:** the acquire/work/release trace as it happens, then `<name>: ok` or `<name>: failed: <what()>` per job.

```
3
backup 2 db file ok
sync 1 net fail
noop 0 ok
```

prints

```
acquire db
acquire file
work backup
release file
release db
backup: ok
acquire net
release net
sync: failed: step failed in sync
work noop
noop: ok
```

INPUT 1

```
3
backup 2 db file ok
sync 1 net fail
noop 0 ok
```

OUTPUT 1

```
acquire db
acquire file
work backup
release file
release db
backup: ok
acquire net
release net
sync: failed: step failed in sync
work noop
noop: ok
```

INPUT 2

```
1
deploy 3 lock db cache fail
```

OUTPUT 2

```
acquire lock
acquire db
acquire cache
release cache
release db
release lock
deploy: failed: step failed in deploy
```

REFERENCE SOLUTION

```cpp
#include <iostream>
#include <stdexcept>
#include <string>
#include <utility>
#include <vector>

struct Job {
    std::string name;
    std::vector<std::string> resources;
    bool fails = false;
};

class Guard {
public:
    explicit Guard(std::string name) : name_(std::move(name)) {
        std::cout << "acquire " << name_ << '\n';
    }
    ~Guard() {
        std::cout << "release " << name_ << '\n';
    }
    Guard(const Guard&) = delete;
    Guard& operator=(const Guard&) = delete;
private:
    std::string name_;
};

// Each resource is held in its own frame, so the unwinding has to pass
// through every one of them — and run every destructor — before main's
// handler gets control.
void run(const Job& job, std::size_t i) {
    if (i == job.resources.size()) {
        if (job.fails) throw std::runtime_error("step failed in " + job.name);
        std::cout << "work " << job.name << '\n';
        return;
    }
    Guard guard(job.resources[i]);
    run(job, i + 1);
}

int main() {
    int n;
    std::cin >> n;
    for (int k = 0; k < n; ++k) {
        Job job;
        int count;
        std::cin >> job.name >> count;
        for (int r = 0; r < count; ++r) {
            std::string resource;
            std::cin >> resource;
```

```
            job.resources.push_back(resource);
        }
        std::string outcome;
        std::cin >> outcome;
        job.fails = (outcome == "fail");
        try {
            run(job, 0);
            std::cout << job.name << ": ok\n";
        } catch (const std::runtime_error& e) {
            std::cout << job.name << ": failed: " << e.what() << '\n';
        }
    }
    return 0;
}
```

EXERCISE 2

Moved or copied?

Make the `noexcept` decision visible. `Probe` (given) records which constructor built it: `copied` or `moved`. Define four types, each holding a `std::string label` and a `Probe probe` and constructible from a string:

- `Defaulted` — declares no copy or move at all (rule of zero);
- `Safe` — a user-written copy constructor and a user-written move constructor marked `noexcept`, each forwarding to the members' corresponding constructor;
- `Risky` — the same two constructors, but the move constructor **without** `noexcept`;
- `CopyOnly` — a user-written copy constructor and no move constructor.

`relocate<T>` (given) builds a `T`, relocates it with `T target(std::move_if_noexcept(source))` — exactly what `std::vector` does to each element on reallocation — and prints `<label>: nothrow move <yes|no>, <moved|copied>`, the first word from `std::is_nothrow_move_constructible_v<T>` and the second from `target.probe.how`.

Input: `n`, then `n` lines `<kind> <label>` where kind is `defaulted`, `safe`, `risky` or `copyonly`.
Output: one line per object, then `moved <a>, copied <b>`.

```
4
defaulted d
safe s
risky r
copyonly c
```

prints

```
d: nothrow move yes, moved
s: nothrow move yes, moved
r: nothrow move no, copied
c: nothrow move no, copied
moved 2, copied 2
```

INPUT 1

```
4
defaulted d
safe s
risky r
copyonly c
```

OUTPUT 1

```
d: nothrow move yes, moved
s: nothrow move yes, moved
r: nothrow move no, copied
c: nothrow move no, copied
moved 2, copied 2
```

INPUT 2

```
2
risky big
safe big
```

OUTPUT 2

```
big: nothrow move no, copied
big: nothrow move yes, moved
moved 1, copied 1
```

REFERENCE SOLUTION

```cpp
#include <iostream>
#include <string>
#include <type_traits>
#include <utility>

// Records which constructor built it: the instrument every kind carries.
struct Probe {
    std::string how = "fresh";
    Probe() = default;
    Probe(const Probe&) : how("copied") {}
    Probe(Probe&&) noexcept : how("moved") {}
};

// Rule of zero: the compiler writes copy and move, and the move is noexcept
// because std::string's and Probe's are.
struct Defaulted {
    std::string label;
    Probe probe;
    explicit Defaulted(std::string l) : label(std::move(l)) {}
};

// A user-written move that promises not to throw.
struct Safe {
    std::string label;
    Probe probe;
    explicit Safe(std::string l) : label(std::move(l)) {}
    Safe(const Safe& o) : label(o.label), probe(o.probe) {}
    Safe(Safe&& o) noexcept : label(std::move(o.label)), probe(std::move(o.probe)) {
};

// The same move without the word: the library must assume it can throw.
struct Risky {
    std::string label;
    Probe probe;
    explicit Risky(std::string l) : label(std::move(l)) {}
    Risky(const Risky& o) : label(o.label), probe(o.probe) {}
    Risky(Risky&& o) : label(std::move(o.label)), probe(std::move(o.probe)) {}
};

// A copy constructor and nothing else: declaring it suppresses the implicit move.
struct CopyOnly {
    std::string label;
    Probe probe;
    explicit CopyOnly(std::string l) : label(std::move(l)) {}
    CopyOnly(const CopyOnly& o) : label(o.label), probe(o.probe) {}
};

template <typename T>
void relocate(const std::string& label, int& moved, int& copied) {
```

```
    T source(label);
    T target(std::move_if_noexcept(source));   // what std::vector does on reallocat
    const bool nothrow = std::is_nothrow_move_constructible_v<T>;
    std::cout << label << ": nothrow move " << (nothrow ? "yes" : "no") << ", " << t
    if (target.probe.how == "moved") ++moved; else ++copied;
}

int main() {
    int n;
    std::cin >> n;
    int moved = 0, copied = 0;
    for (int i = 0; i < n; ++i) {
        std::string kind, label;
        std::cin >> kind >> label;
        if (kind == "defaulted") relocate<Defaulted>(label, moved, copied);
        else if (kind == "safe") relocate<Safe>(label, moved, copied);
        else if (kind == "risky") relocate<Risky>(label, moved, copied);
        else if (kind == "copyonly") relocate<CopyOnly>(label, moved, copied);
    }
    std::cout << "moved " << moved << ", copied " << copied << '\n';
    return 0;
}
```

QUIZ

1. Which exception-safety guarantee does `std::vector::push_back` give?

A. No-throw: it never throws

B. Strong: if it throws, the vector is exactly as it was before the call

C. Basic: no leak, but the vector may hold a partial result

D. None: a throw leaves the vector unusable

2. What does this print?

```
struct M {
    M() = default;
    M(const M&) { std::cout << "copy"; }
    M(M&&) { std::cout << "move"; }
};
int main() {
    M a;
    M b(std::move_if_noexcept(a));
}
```

A. `move`

B. `copy`

C. Nothing

D. Compile error: `M` is not movable

3. What does `noexcept` on a function actually guarantee?

A. The compiler rejects any `throw` inside the function

B. If an exception escapes the function, `std::terminate` is called — it is a promise enforced at run time, not a check at compile time

C. The function is inlined and cannot allocate

D. Exceptions thrown inside are converted to error codes

4. `Inventory::load` currently clears `items_` and inserts each parsed line. Which rewrite gives the strong guarantee?

A. Parse everything into a temporary map, then `items_.swap(fresh)` as the last statement

B. Wrap the loop in `try` / `catch`, log the error and return

C. Mark `load` as `noexcept`

D. Insert each line inside its own `try` block and skip the bad ones

5. Which of these is `noexcept` without the word being written?

A. A destructor with no explicit specifier

B. A user-written move constructor

C. `std::vector::at`

D. Any function compiled with `-O2`

6. A `Connection`'s cleanup can genuinely fail — the close message may not be sent. Where should that failure go?

A. Throw from the destructor so the owner learns of it

B. Give the class an explicit `close()` that may throw; the destructor calls it inside `try` / `catch (...)` and never rethrows

C. Print it from the destructor and call `std::exit`

D. Mark the destructor `noexcept(false)` and throw only when no other exception is in flight

7. In which situation is a return value the better choice over an exception?

A. A failure the caller expects often — a lookup that misses, user input that fails to parse

B. A constructor that cannot establish its invariant

C. A failure discovered five calls below the only code that can act on it

D. Always — exceptions are slower than returns

ANSWER KEY · EXCEPTION SAFETY — GUARANTEES, NOEXCEPT AND RAII

1. B `push_back` commits or rolls back — a failed allocation or a throwing element constructor leaves the old contents intact. It can throw (`std::bad_alloc` at least), so it is not no-throw; `assign` is the example of a basic-only operation.

2. B The move constructor is user-written without `noexcept`, so `std::move_if_noexcept` yields `const M&` and the copy constructor is selected. Add `noexcept` to the move constructor and it prints `move` — this is the decision `std::vector` makes on every reallocation.

3. B A `throw` inside a `noexcept` function compiles; the consequence of one escaping is immediate termination with no unwinding to the caller. The specifier tells callers and the library what to rely on — nothing about inlining or allocation.

4. A Do every step that can throw on a temporary; commit with an operation that cannot throw. If `parse` fails, `items_` was never touched. Logging or skipping leaves a partial update, and `noexcept` would turn the parse error into a call to `std::terminate`.

5. A Since C++11 destructors are implicitly `noexcept` (unless a member's destructor is not), and defaulted special members inherit `noexcept` from their members. A user-written move constructor gets nothing unless you write the word; `at()` throws by design; optimisation level changes no specifier.

6. B A destructor that throws during unwinding terminates the program, and a destructor is implicitly `noexcept` anyway. `std::fstream` follows the same design: `close()` for callers who care, a swallowing destructor for everyone else. Checking for an exception in flight is fragile and still leaves the caller no way to handle the failure.

7. A A throw costs a thousand times a return and belongs on the exceptional path. Constructors have no return value, and a deep chain with no channel for a code is exactly what exceptions are for — so "always" is wrong in both directions.

LESSON 3 · 13 MIN

std::optional — a value that may be absent

Some functions sometimes have no answer: a search that finds nothing, a configuration key that was never set, an average of zero numbers. C used a *sentinel* — `-1`, `NULL`, `EOF`, `std::string::npos` — a value smuggled through the normal channel that the caller must know to check and can forget, and that some types cannot spare at all (which `int` is "no int"?). `std::optional<T>`, in `<optional>` since C++17, makes absence part of the type: it either holds a `T` or holds nothing, it says which, and the compiler makes the caller ask. This lesson settles how to build, test, read and compare one, when to return it, and the one thing it cannot hold.

Constructing and inspecting

```
#include <optional>
#include <string>

std::optional<int> a;                                  // empty
std::optional<int> b = 42;                             // holds 42
std::optional<int> c = std::nullopt;                   // empty, said explici
auto d = std::make_optional<std::string>("ada");      // optional<std::string>
std::optional<std::string> e{std::in_place, 3, 'x'};   // constructs "xxx" ins

if (b) { /* engaged */ }              // explicit operator bool
if (b.has_value()) { /* the same */ }
```

An `optional` is *engaged* when it holds a value and *empty* (or *disengaged*) otherwise. `std::nullopt` is the empty value's name, and the type is default-constructed empty, which is why a `std::optional` member needs no initialiser to be safe.

Reading the value has four spellings with three different behaviours:

Expression	When engaged	When empty
`*o`, `o->member`	the value	**undefined behaviour** — no check
`o.value()`	the value	throws `std::bad_optional_access`
`o.value_or(x)`	a copy of the value	`x` converted to `T`
`o.has_value()`, `if (o)`	`true`	`false`

`*o` reads like a pointer and is checked like one — not at all. Use it only right after a test. `value()` is the checked form for code that would rather throw than continue. `value_or` is the everyday form when a default exists.

Returning optional from a lookup

```
#include <vector>

std::optional<std::size_t> indexOf(const std::vector<int>& v, int target) {
    for (std::size_t i = 0; i < v.size(); ++i) {
        if (v[i] == target) return i;       // a std::size_t converts to optio
    }
    return std::nullopt;
}

int main() {
    std::vector<int> v{4, 8, 15};
    if (auto at = indexOf(v, 4)) {
        std::cout << "found at " << *at << '\n';   // found at 0
    } else {
        std::cout << "absent\n";
    }
}
```

`return i;` builds the optional from the value and `return std::nullopt;` builds the empty one; the function never spells the type twice. The `if` form is the idiom: the C++17 initialiser scopes `at` to the branches, the condition tests *engagement*, and `*at` is safe inside. Note what the condition does not test: index 0 is a perfectly good answer, and `if (auto at = ...)` is true for it, because the optional is engaged. That is the whole point — `0` and `-1` are no longer overloaded with meaning.

The same shape wraps `std::map::find`, whose iterator-or-`end()` protocol is exactly a sentinel:

```
#include <map>

std::optional<std::string> get(const std::map<std::string, std::string>& cfg,
    auto it = cfg.find(key);
    if (it == cfg.end()) return std::nullopt;
    return it->second;                    // copies the mapped string into t
}

std::cout << get(cfg, "port").value_or("8080") << '\n';
```

---

value_or, precisely

`o.value_or(x)` returns a `T` *by value* — a copy of the held value, or `x` converted to `T`. Two consequences follow. The argument is evaluated whether or not it is needed, because it is an ordinary function argument: `o.value_or(expensiveDefault())` always calls `expensiveDefault`. And for a large `T` in a hot path the copy costs; test and dereference instead. For a `std::optional<std::string>` and a literal default, `value_or("none")` is fine — the literal becomes a `std::string` only on the empty path's return.

Modifying

```
std::optional<std::string> name;
name = "ada";                 // engage by assignment
name.emplace(3, 'x');         // construct a new value in place: "xxx"
name.reset();                 // back to empty
name = std::nullopt;          // the same
if (name) *name += "!";       // modify through the dereference — only when en
```

Assignment through `*name` on an empty optional is undefined behaviour like any other dereference; `emplace` and `=` are how an empty one becomes engaged.

Comparing

Optionals compare by value, with *empty* ordered before every engaged value and two empties equal:

```
std::optional<int> none, zero = 0, five = 5;
none == zero;          // false
none < zero;           // true: empty sorts first
zero < five;           // true: compares the values
five == 5;             // true: compares with a plain T
none == 5;             // false, not an error
none == std::nullopt;  // true
```

That ordering makes `std::optional<int>` usable as a `std::map` key or in `std::sort`, and C++20 adds `⟺` so `std::optional<T>` orders whenever `T` does.

---

optional<T&> does not exist

`std::optional<int&>` is ill-formed. The committee could not agree what `o = x` should mean for a reference — rebind to `x`, or assign through to the referent — so the type was left out (C++26 finally adds it, with rebinding). Until then, when a function may or may not return a reference to something the caller owns:

- return `T*` — a pointer that may be `nullptr` *is* an optional reference, and Module 6 said so;
- return `std::optional<std::reference_wrapper<T>>` and read it with `→get()`;
- return an index or an iterator, and let the caller subscript.

The first is idiomatic; the second appears in code that wants "optional" in the signature.

Layout and cost

The `T` lives inside the optional object — no heap — followed by a `bool`, padded to `T`'s alignment. So `std::optional<int>` is bigger than an `int`, copying one copies the `T`, and returning one by value is as cheap as returning the `T`. `emplace` and `std::in_place` avoid a temporary when construction is expensive. A `std::optional<std::unique_ptr<T>>` is redundant — the pointer already has a null state — and `std::optional<bool>` is legal but three-valued (`true`, `false`, empty), which readers reliably misread; a small `enum class` says more.

Monadic operations (C++23, reading only)

C++23 adds three members that chain without an `if` ladder: `and_then(f)` calls `f` with the value and expects `f` to return another optional; `transform(f)` wraps `f`'s plain result in an optional; `or_else(f)` runs `f` when empty.

```
int next = get(cfg, "port")              // optional<std::string>
    .and_then(parseInt)                  // optional<int>: parseInt returns sta
    .transform([](int p) { return p + 1; })
    .value_or(8081);
```

This track's runtime is C++20 and does not have them; write the two `if`s. Knowing the names matters because the same three operations exist on `std::expected` (lesson 5) and in every language with a maybe type.

optional, pointer or exception?

Return	When
`std::optional<T>`	absence is normal and the caller decides; `T` is a value
`T*`	the thing lives elsewhere and may be absent; the caller must not own it
throw	absence is a violated precondition or a failure the caller cannot fix
`std::variant<T, Error>` / `std::expected`	the caller needs to know *why* (lessons 4 and 5)

Pitfalls

Mistake	What happens
`*o` or `o→x` on an empty optional	Undefined behaviour — not a throw
Expecting `value()` to be free	It checks and can throw `std::bad_optional_access`
`o.value_or(compute())`	`compute()` runs even when `o` is engaged
`if (opt)` meaning "value is non-zero"	It means "engaged"; `optional<int>` holding 0 is `true`
`std::optional<T&>`	Compile error; return `T*` or `reference_wrapper`
`std::optional<bool>` for a flag	Three states; use an enum

Key takeaways

- `std::optional<T>` holds a `T` or nothing, in place, and says which through `has_value()` / `if (o)`.
- `*o` is unchecked (UB when empty), `value()` throws, `value_or(x)` returns a `T` by value and evaluates `x` eagerly.
- Return an optional from a lookup: `return value;` engages, `return std::nullopt;` empties; test with `if (auto r = f())`.
- Empty sorts before every value and equals only empty; `o == 5` compares with the held value.
- `optional<T&>` does not exist: return a pointer, a `reference_wrapper` or an index.
- `and_then` / `transform` / `or_else` are C++23 — on this runtime, write the `if`.

EXERCISE 1

Price lookup with value_or

Write `std::optional<int> priceOf(const std::map<std::string, int>& prices, const std::string& item)` — the value when the item is listed, `std::nullopt` when it is not — and answer two kinds of query with it. `price <item>` prints the price or `not listed`, using the `if (auto price = priceOf(...))` idiom. `basket <k> <item-1> … <item-k>` adds up the prices of the listed items with `value_or(0)` and counts the items that are not listed.

Input: `n`, then `n` lines `<item> <price>`, then queries until EOF. **Output:** `<item>: <price>` or `<item>: not listed` per price query; `basket: total <sum>, missing <count>` per basket query.

```
3
apple 3
pear 5
fig 12
price apple
price kiwi
basket 3 apple kiwi fig
basket 2 kiwi plum
```

prints

```
apple: 3
kiwi: not listed
basket: total 15, missing 1
basket: total 0, missing 2
```

INPUT 1

```
3
apple 3
pear 5
fig 12
price apple
price kiwi
basket 3 apple kiwi fig
basket 2 kiwi plum
```

OUTPUT 1

```
apple: 3
kiwi: not listed
basket: total 15, missing 1
basket: total 0, missing 2
```

INPUT 2

```
1
zero 0
price zero
basket 1 zero
```

OUTPUT 2

```
zero: 0
basket: total 0, missing 0
```

REFERENCE SOLUTION

```cpp
#include <iostream>
#include <map>
#include <optional>
#include <string>

std::optional<int> priceOf(const std::map<std::string, int>& prices, const std::stri
    auto it = prices.find(item);
    if (it == prices.end()) return std::nullopt;
    return it->second;
}

int main() {
    int n;
    std::cin >> n;
    std::map<std::string, int> prices;
    for (int i = 0; i < n; ++i) {
        std::string item;
        int price;
        std::cin >> item >> price;
        prices[item] = price;
    }
    std::string query;
    while (std::cin >> query) {
        if (query == "price") {
            std::string item;
            std::cin >> item;
            if (auto price = priceOf(prices, item)) {
                std::cout << item << ": " << *price << '\n';
            } else {
                std::cout << item << ": not listed\n";
            }
        } else if (query == "basket") {
            int count;
            std::cin >> count;
            long long total = 0;
            int missing = 0;
            for (int i = 0; i < count; ++i) {
                std::string item;
                std::cin >> item;
                std::optional<int> price = priceOf(prices, item);
                total += price.value_or(0);
                if (!price.has_value()) ++missing;
            }
            std::cout << "basket: total " << total << ", missing " << missing << '\r
        }
    }
    return 0;
}
```

EXERCISE 2

Statistics that may have no answer

Three functions over a `std::vector<int>`, each returning an `std::optional` because an empty range has no answer: `std::optional<double> mean(const std::vector<int>&)`, `std::optional<int> maxOf(const std::vector<int>&)` and `std::optional<std::size_t> indexOf(const std::vector<int>&, int target)` (the first index holding `target`). Return the value directly and `std::nullopt` for "no answer"; never use `-1` or `npos`. Accumulate the mean in `long long`. Print each result with the `if (auto r = f())` idiom.

Input: `n`, then `n` integers, then `target`. **Output:** `count: <n>`, then `mean: <x.xx>` or `mean: none`, `max: <v>` or `max: none`, `index of <target>: <i>` or `index of <target>: absent`.

```
5
4 8 15 16 23
15
```

prints

```
count: 5
mean: 13.20
max: 23
index of 15: 2
```

INPUT 1

```
5
4 8 15 16 23
15
```

OUTPUT 1

```
count: 5
mean: 13.20
max: 23
index of 15: 2
```

INPUT 2

```
0
3
```

OUTPUT 2

```
count: 0
mean: none
max: none
index of 3: absent
```

REFERENCE SOLUTION

```cpp
#include <iomanip>
#include <iostream>
#include <optional>
#include <vector>

std::optional<double> mean(const std::vector<int>& v) {
    if (v.empty()) return std::nullopt;
    long long sum = 0;
    for (int x : v) sum += x;
    return static_cast<double>(sum) / static_cast<double>(v.size());
}

std::optional<int> maxOf(const std::vector<int>& v) {
    if (v.empty()) return std::nullopt;
    int best = v[0];
    for (int x : v) {
        if (x > best) best = x;
    }
    return best;
}

std::optional<std::size_t> indexOf(const std::vector<int>& v, int target) {
    for (std::size_t i = 0; i < v.size(); ++i) {
        if (v[i] == target) return i;      // index 0 is a perfectly good answer
    }
    return std::nullopt;
}

int main() {
    int n;
    std::cin >> n;
    std::vector<int> v(n);
    for (int& x : v) std::cin >> x;
    int target;
    std::cin >> target;

    std::cout << "count: " << v.size() << '\n';
    std::cout << std::fixed << std::setprecision(2);
    if (auto m = mean(v)) std::cout << "mean: " << *m << '\n';
    else std::cout << "mean: none\n";
    if (auto m = maxOf(v)) std::cout << "max: " << *m << '\n';
    else std::cout << "max: none\n";
    if (auto i = indexOf(v, target)) std::cout << "index of " << target << ": " << *
    else std::cout << "index of " << target << ": absent\n";
    return 0;
}
```

QUIZ

1. What does this print?

```
std::optional<int> o;
std::cout << o.value_or(7);
o = 0;
std::cout << (o ? "y" : "n");
```

A. `7y`

B. `7n`

C. `0y`

D. It throws `std::bad_optional_access`

2. What does `*o` do when `o` is an empty `std::optional<int>`?

A. Throws `std::bad_optional_access`

B. Undefined behaviour — no check is made

C. Returns a default-constructed `int`

D. Returns `std::nullopt`

3. Which statement about `o.value_or(compute())` is correct?

A. `compute()` is called only when `o` is empty

B. `compute()` is always called, because it is an ordinary function argument evaluated before the call

C. It does not compile unless `compute` returns an `std::optional`

D. It returns a reference to the held value when `o` is engaged

4. What is `std::optional<int&>`?

A. An optional reference that rebinds on assignment

B. Ill-formed: `std::optional` cannot hold a reference; return a pointer, a `std::reference_wrapper` or an index instead

C. Legal, but always empty

D. Legal only when the referent outlives the optional

5. `std::optional<int> a; std::optional<int> b = 0;` — what is `a < b`?

A. `true`: an empty optional orders before every engaged value

B. `false`: empty compares as zero

C. Undefined behaviour

D. Compile error: an empty optional cannot be compared

6. What does this print?

```
std::optional<std::string> name;
name.emplace(3, 'x');
std::cout << name→size();
name.reset();
std::cout << name.has_value();
```

A. `30`

B. `31`

C. `10`

D. It throws

ANSWER KEY · STD::OPTIONAL — A VALUE THAT MAY BE ABSENT

1. A `value_or(7)` returns the default while `o` is empty. After `o = 0` the optional is *engaged* — the condition tests engagement, not the value — so it is `true` even though the held `int` is zero. Nothing here throws.

2. B `operator*` and `operator→` are unchecked, like a pointer. `value()` is the form that checks and throws. The optional never hands you a made-up `int`.

3. B `value_or` is a member function, not lazy syntax: its argument is evaluated whether or not it is needed, and the result is a `T` by value. When the default is expensive, test and dereference instead.

4. B The committee could not agree what assigning to an optional reference should mean, so the specialisation does not exist before C++26. A `T*` that may be null already is an optional reference, and `std::optional<std::reference_wrapper<T>>` spells it out when the signature must say "optional".

5. A Optionals compare by value with *empty* first: two empties are equal, and empty is less than any engaged value, including zero. That ordering is what lets `std::optional<int>` be a `std::map` key.

6. A `emplace(3, 'x')` constructs `"xxx"` in place, so `size()` is 3; `reset()` returns the optional to empty and `has_value()` streams as `0`. `→` is safe here because the optional was engaged at that point.

LESSON 4 · 14 MIN

std::variant and std::visit — a type-safe union

A value that is *one of several types* — a token that is a number or a word, a command that is a push or a pop, a result that is a value or an error — is a sum type, and C's tool for it was the `union`: overlapping storage with no record of which member is live, so reading the wrong one is undefined behaviour. `std::variant`, in `<variant>` since C++17, is a union that remembers. It stores exactly one of its alternatives at a time, tracks which by index, checks every access, and — through `std::visit` — makes the compiler prove that your code handles every alternative. This lesson settles construction and access, the visitor patterns, `std::monostate`, modelling a small command language, and why `std::any` is almost never the tool.

Declaring, assigning, asking

```
#include <iostream>
#include <string>
#include <variant>

int main() {
    std::variant<int, double, std::string> v;   // holds int 0: the first alte
    v = 2.5;                                    // now a double
    v = "text";                                 // now a std::string (the cons
    std::cout << v.index() << '\n';             // 2

    if (std::holds_alternative<std::string>(v)) {
        std::cout << std::get<std::string>(v).size() << '\n';   // 4
    }
    if (auto* d = std::get_if<double>(&v)) {    // pointer to the double, or n
        std::cout << *d << '\n';                // not printed: v holds a stri
    }
    std::cout << std::get<2>(v) << '\n';        // by index: text
    return 0;
}
```

A variant is never empty in normal use: default construction builds the first alternative, and assigning a value of another type destroys the current alternative and constructs the new one in the same storage. Access comes in two checked forms. `std::get<T>` (or `std::get<I>`) returns a reference and throws `std::bad_variant_access` when the variant holds something else; `std::get_if<T>` takes a *pointer* to the variant and returns a pointer to the value or

`nullptr`, which is the form for an `if`. Two alternatives of the same type — `std::variant<int, int>` — are legal but make `get<int>` ambiguous; address them by index or, better, wrap each in a named struct.

Visiting

Testing `index()` in an `if` / `else` chain works and is the anti-pattern: add a fourth alternative and every chain silently falls through its last `else`. `std::visit(visitor, v)` calls `visitor(held)` with the held value at its real type, and refuses to compile unless the visitor accepts *every* alternative.

The smallest visitor is a generic lambda, for operations that read the same for every type:

```
std::visit([](const auto& x) { std::cout << x << '\n'; }, v);   // int, double
```

When each alternative needs its own code, overload the call operator. The standard idiom is a struct that inherits every lambda you hand it:

```
template <typename... Ts>
struct overloaded : Ts... {
    using Ts::operator()...;
};
template <typename... Ts> overloaded(Ts...) -> overloaded<Ts...>;   // C++20 d

std::visit(overloaded{
    [](int i) { std::cout << "int " << i << '\n'; },
    [](double d) { std::cout << "double " << d << '\n'; },
    [](const std::string& s) { std::cout << "string " << s << '\n'; },
}, v);
```

`overloaded` is a class derived from each lambda's closure type (Module 12, lesson 4 covered the pack), and `using Ts::operator()...` pulls every call operator into one overload set, so the call resolves like any overloaded function — exact match first. Leave one alternative out and the error names the type that has no match. A trailing `[](const auto&) { }` is the catch-all when only some alternatives matter, at the cost of the compile-time check.

A visitor may return a value, and then every overload must return the *same* type:

```
int width = std::visit(overloaded{
    [](int) { return 1; },
    [](double) { return 2; },
    [](const std::string& s) { return static_cast<int>(s.size()); },   // size
}, v);
```

A visitor with state

Lambdas capture, but a visitor that carries state across many visits is clearer as a struct with overloaded `operator()`s:

```
struct Push { long long value; };
struct Pop {};
struct Add {};
using Command = std::variant<Push, Pop, Add>;

struct Executor {
    std::vector<long long>& stack;
    void operator()(const Push& p) const { stack.push_back(p.value); }
    void operator()(const Pop&) const { stack.pop_back(); }
    void operator()(const Add&) const {
        long long b = stack.back(); stack.pop_back();
        stack.back() += b;
    }
};

for (const Command& c : program) std::visit(Executor{stack}, c);
```

Empty structs as alternatives are the C++ spelling of an enum with payloads: `Pop` carries nothing but is still a distinct type the visitor can overload on.

std::monostate

Because a variant default-constructs its first alternative, a variant whose alternatives are all non-default-constructible cannot be default-constructed at all. `std::monostate` — an empty type that compares equal to itself — goes first to say "nothing yet":

```
std::variant<std::monostate, Connection, Error> state;   // default: monostate
```

It also models a legitimately empty state without reaching for `std::optional<std::variant<...>>`.

Modelling a small language

A variant of command structs plus a parser and a visitor is a complete interpreter, and the pattern scales to tokens, JSON values and abstract syntax trees. The trade-off it embodies has a name — the *expression problem*. A `std::variant` fixes the set of *types* and leaves the set of *operations* open: anyone can write a new visitor without touching the types. A class hierarchy with virtual functions (Module 10) fixes the set of *operations* and leaves the *types* open: anyone can derive a new class, but a new operation means editing every class. Choose by which axis your program grows. Recursive shapes — an expression that contains expressions — need indirection, because a type cannot contain itself by value:

```
struct Expr;
using Node = std::variant<long long, std::unique_ptr<Expr>>;   // the pointer
```

---

std::any, and why it is rare

`std::any` holds a value of *any* copyable type and remembers which with `typeid`:

```
#include <any>
std::any a = 5;
a = std::string("five");
if (auto* s = std::any_cast<std::string>(&a)) std::cout << *s << '\n';   // po
int n = std::any_cast<int>(a);                                            // t
```

There is no `visit` — the holder cannot enumerate what it does not know — and large values are heap-allocated. That is the tell: `any` is for a set of types that is genuinely open at compile time, such as a scripting bridge or a plugin's opaque configuration. When the set is known, `std::variant` is checked and faster; when the code is generic over the type, a template is. Most codebases contain no `std::any` at all.

Cost

A variant is as large as its largest alternative plus an index, padded, and lives wherever you put it — no heap. `std::visit` compiles to a table dispatch on the index, comparable to a virtual call. One rarely met state exists: if constructing the new alternative during an assignment throws after the old one was destroyed, the variant becomes `valueless_by_exception()` and `visit` throws; a `noexcept` move constructor (lesson 2) makes this impossible for your types.

Pitfalls

Mistake	What happens
`std::get<T>` without checking	Throws `std::bad_variant_access` when another alternative is held
`if` / `else` on `index()`	A new alternative is silently unhandled; `std::visit` would refuse to compile
Visitor overloads returning different types	Compile error: every return must convert to one type
`std::variant<int, int>` and `get<int>`	Ambiguous; wrap each in a named struct
First alternative not default-constructible	The variant is not default-constructible; put `std::monostate` first
A variant that contains itself by value	Incomplete type; go through `std::unique_ptr` or a `std::vector`
Reaching for `std::any` with a known set of types	No `visit`, possible heap allocation, a cast per read

---

Key takeaways

- `std::variant<A, B, C>` holds exactly one alternative, tracked by `index()`; default construction builds the first.
- `std::get<T>` throws on a mismatch; `std::get_if<T>(&v)` returns `nullptr` instead — use it in an `if`.

`std::visit` calls the visitor with the real type and refuses to compile unless every alternative is handled; `overloaded{…}` builds the visitor from lambdas.

Empty structs are alternatives too; a visitor struct with a reference member carries state.

`std::monostate` is the "nothing yet" alternative; recursion needs a pointer.

`std::any` is for a truly open set of types and is almost never the right tool.

EXERCISE 1

A token stream

Classify a stream of tokens with `using Token = std::variant<int, double, std::string>;` and visit each one. `parseToken` (given) decides the alternative by shape: an optional sign followed by digits is an `int`, digits with one `.` is a `double`, anything else is a `std::string` (integers fit in `int`). Write the `overloaded` helper from the lesson and a single `std::visit(overloaded{...}, token)` with three lambdas that print the alternative's name and value, count it, and add numbers to a running `double` sum. Doubles print with two decimals.

Input: whitespace-separated tokens until EOF. **Output:** `int <v>`, `double <v>` or `string <v>` per token, then `ints <a>, doubles <b>, strings <c>, sum <s>`.

```
42 -7 2.5 hello .5 3. 1e5 +8
```

prints

```
int 42
int -7
double 2.50
string hello
double 0.50
double 3.00
string 1e5
int 8
ints 3, doubles 3, strings 2, sum 49.00
```

INPUT 1

```
42 -7 2.5 hello .5 3. 1e5 +8
```

OUTPUT 1

```
int 42
int -7
double 2.50
string hello
double 0.50
double 3.00
string 1e5
int 8
ints 3, doubles 3, strings 2, sum 49.00
```

INPUT 2

```
a b c
```

OUTPUT 2

```
string a
string b
string c
ints 0, doubles 0, strings 3, sum 0.00
```

REFERENCE SOLUTION

```cpp
#include <cctype>
#include <iomanip>
#include <iostream>
#include <string>
#include <variant>

using Token = std::variant<int, double, std::string>;

template <typename... Ts>
struct overloaded : Ts... {
    using Ts::operator()...;
};

// Given: classifies by shape. An optional sign, then digits with at most one
// '.', is a number; a '.' makes it a double; anything else is a word.
Token parseToken(const std::string& s) {
    std::size_t start = (s[0] == '+' || s[0] == '-') ? 1 : 0;
    bool digits = false, dot = false, ok = true;
    for (std::size_t i = start; i < s.size(); ++i) {
        if (std::isdigit(static_cast<unsigned char>(s[i]))) digits = true;
        else if (s[i] == '.' && !dot) dot = true;
        else ok = false;
    }
    if (!ok || !digits) return s;
    if (dot) return std::stod(s);
    return std::stoi(s);
}

int main() {
    std::cout << std::fixed << std::setprecision(2);
    int ints = 0, doubles = 0, strings = 0;
    double sum = 0.0;
    std::string text;
    while (std::cin >> text) {
        Token token = parseToken(text);
        std::visit(overloaded{
            [&](int i) { std::cout << "int " << i << '\n'; ++ints; sum += i; },
            [&](double d) { std::cout << "double " << d << '\n'; ++doubles; sum += d
            [&](const std::string& w) { std::cout << "string " << w << '\n'; ++strin
        }, token);
    }
    std::cout << "ints " << ints << ", doubles " << doubles << ", strings " << strin
    return 0;
}
```

EXERCISE 2

Turtle commands

Model a small command language as a variant and interpret it with a stateful visitor. Define `struct Move { int steps; }`, `struct Turn { bool right; }`, `struct Report {}`, `struct Unknown { std::string line; }` and `using Command = std::variant<Move, Turn, Report, Unknown>;`. `parse(line)` turns `move <n>` into `Move`, `turn left` / `turn right` into `Turn`, `report` into `Report`, and any other line into `Unknown{line}`. `Turtle` is a struct with `x`, `y`, a heading (north, east, south, west; starts at `(0, 0)` facing north) and a step total, and one `operator()` overload per alternative: `Move` walks `steps` in the current heading and adds `steps` to the total, `Turn` rotates a quarter turn, `Report` prints `at (<x>, <y>) facing <N|E|S|W>`, `Unknown` prints `unknown: <line>`. Drive it with `std::visit(turtle, parse(line))` — pass the visitor by name so its state persists.

Input: one command per line until EOF. **Output:** the report and unknown lines as they occur, then `total <steps>`.

```
move 3
turn right
move 2
report
jump 5
turn left
turn left
move 4
report
```

prints

```
at (2, 3) facing E
unknown: jump 5
at (-2, 3) facing W
total 9
```

INPUT 1

```
move 3
turn right
move 2
report
jump 5
turn left
turn left
move 4
report
```

OUTPUT 1

```
at (2, 3) facing E
unknown: jump 5
at (-2, 3) facing W
total 9
```

INPUT 2

```
report
turn left
move 1
report
```

OUTPUT 2

```
at (0, 0) facing N
at (-1, 0) facing W
total 1
```

REFERENCE SOLUTION

```cpp
#include <iostream>
#include <sstream>
#include <string>
#include <variant>

struct Move { int steps; };
struct Turn { bool right; };
struct Report {};
struct Unknown { std::string line; };
using Command = std::variant<Move, Turn, Report, Unknown>;

Command parse(const std::string& line) {
    std::istringstream in(line);
    std::string word;
    in >> word;
    if (word == "move") {
        int steps;
        if (in >> steps) return Move{steps};
    } else if (word == "turn") {
        std::string direction;
        in >> direction;
        if (direction == "left") return Turn{false};
        if (direction == "right") return Turn{true};
    } else if (word == "report") {
        return Report{};
    }
    return Unknown{line};
}

struct Turtle {
    int x = 0, y = 0;
    int heading = 0;            // 0 N, 1 E, 2 S, 3 W
    long long travelled = 0;

    void operator()(const Move& m) {
        static const int dx[4] = {0, 1, 0, -1};
        static const int dy[4] = {1, 0, -1, 0};
        x += dx[heading] * m.steps;
        y += dy[heading] * m.steps;
        travelled += m.steps;
    }
    void operator()(const Turn& t) {
        heading = (heading + (t.right ? 1 : 3)) % 4;
    }
    void operator()(const Report&) const {
        static const char* const names[4] = {"N", "E", "S", "W"};
        std::cout << "at (" << x << ", " << y << ") facing " << names[heading] << '\
    }
    void operator()(const Unknown& u) const {
```

```
        std::cout << "unknown: " << u.line << '\n';
    }
};

int main() {
    Turtle turtle;
    std::string line;
    while (std::getline(std::cin, line)) {
        if (line.empty()) continue;
        std::visit(turtle, parse(line));
    }
    std::cout << "total " << turtle.travelled << '\n';
    return 0;
}
```

QUIZ

1. What does this print?

```
std::variant<int, double, std::string> v;
std::cout << v.index();
v = "hi";
std::cout << v.index();
```

A. `02`

B. `01`

C. `20`

D. Compile error: a `const char*` is not an alternative

2. `v` is a `std::variant<int, double, std::string>` currently holding a string. What does `std::get<double>(v)` do?

A. Returns `0.0`

B. Throws `std::bad_variant_access`

C. Undefined behaviour

D. Returns `nullptr`

3. What does `std::visit` guarantee that an `if` / `else` chain on `v.index()` does not?

A. It is faster because it skips the index check

B. A compile error when the visitor has no overload for some alternative, so adding a type to the variant cannot be silently unhandled

C. A run-time exception when an alternative is unhandled

D. That the visitor is called at most once per program

4. What is wrong with this visitor?

```
auto w = std::visit(overloaded{
    [](int i) { return i; },
    [](double d) { return d; },
    [](const std::string& s) { return s.size(); },
}, v);
```

A. Nothing — `w` becomes whichever type the held alternative produced

B. Compile error: every overload of a visitor must return the same type, and these return `int`, `double` and `std::size_t`

C. It compiles but `w` is always an `int`

D. Undefined behaviour when the variant holds a string

5. What is `std::monostate` for?

A. As the first alternative it lets a variant of non-default-constructible types be default-constructed, and models "nothing yet"

B. It is what `std::get` returns when the variant holds a different alternative

C. A faster replacement for `std::any`

D. It marks a variant as immutable

6. When is `std::any` the right tool rather than `std::variant`?

A. Whenever the set of possible types is known in advance

B. When the set of types is genuinely open at compile time — a plugin's opaque configuration or a scripting bridge

C. When you need `std::visit` over many types

D. Never — it exists only for C compatibility

7. What does `std::get_if<int>(&v)` return?

A. A reference to the `int`, or it throws

B. A pointer to the `int` if the variant holds one, otherwise `nullptr`

C. An `std::optional<int>`

D. `true` or `false`

ANSWER KEY · STD::VARIANT AND STD::VISIT — A TYPE-SAFE UNION

1. A A default-constructed variant holds its first alternative, `int`, at index 0. Assigning a string literal converts it to the one alternative it can become, `std::string`, at index 2 — the same overload-resolution rules as a function call.

2. B `std::get` is checked: a mismatch throws. `std::get_if<double>(&v)` is the form that returns a pointer — `nullptr` on a mismatch — for use in an `if`. Nothing about variant access is undefined.

3. B The visitor must accept every alternative or the call does not compile; an `if` chain simply falls off its last `else`. Both are index dispatches at run time — the difference is what the compiler checks.

4. B `std::visit` has one return type for all alternatives; the lambdas must agree (or convert to one type). Casting each result to a common type — `static_cast<double>` say — fixes it.

5. A A variant default-constructs its first alternative, so a variant whose first alternative cannot be default-constructed cannot be either. `std::monostate` is an empty, comparable type that goes first to say the variant holds nothing meaningful yet.

6. B `std::any` cannot be visited — it does not know its own set of types — and may heap-allocate. With a known set, `std::variant` is checked and faster; with generic code, a template is. Its legitimate use is a set that is open by design.

7. B `get_if` takes a pointer to the variant and answers with a pointer, which makes `if (auto* p = std::get_if<int>(&v))` the idiomatic checked access. The throwing form is `std::get`; the boolean test is `std::holds_alternative`.

LESSON 5 · 14 MIN

Error codes, assertions and an error-handling strategy

Exceptions are one channel for failure, and the previous lessons gave you two more — `std::optional` for "no value" and `std::variant` for "a value or a reason". This lesson settles the rest of the toolbox and when to use which: the return-code tradition C left behind and its modern form in `std::error_code`, the C++23 `std::expected` that unifies value and error, assertions for bugs as opposed to failures, and a strategy that assigns each layer of a program the mechanism that fits it. The strategy is what interviewers are asking about when they say "how do you handle errors in C++?"

Return codes and errno: the C inheritance

C reports failure through the return value — `-1`, `NULL`, `EOF` — and through `errno`, a thread-local integer that library calls set and never clear:

```
#include <cerrno>
#include <cstdlib>

errno = 0;                                          // must be cleared first: r
char* end = nullptr;
long value = std::strtol(text, &end, 10);
if (end == text)          { /* no digits at all */ }
else if (errno == ERANGE) { /* the value did not fit */ }
```

The weaknesses are structural. The check is optional, so it is forgotten. The error shares the channel with the value, so a function returning any `long` has no free value to spend. `errno` must be reset before the call and read before anything else overwrites it. And the code carries no context — `ERANGE` says "too big", not what was too big. Every C API — `fopen`, `read`, sockets — still works this way, so you need to read the pattern even if you never write it.

enum class codes, done properly

The modern return code keeps what worked and fixes what did not: an `enum class` per subsystem, `[[nodiscard]]` so ignoring it is a warning, and the value on a separate channel.

```cpp
enum class CalcError { Ok, DivisionByZero, UnknownOperator, Overflow };

[[nodiscard]] CalcError compute(int a, char op, int b, long long& out) {
    switch (op) {
        case '+': out = static_cast<long long>(a) + b; break;
        case '/': if (b == 0) return CalcError::DivisionByZero;
                  out = static_cast<long long>(a) / b; break;
        default:  return CalcError::UnknownOperator;
    }
    if (out < std::numeric_limits<int>::min() || out > std::numeric_limits<int
    return CalcError::Ok;
}

const char* describe(CalcError e) {
    switch (e) {
        case CalcError::Ok: return "ok";
        case CalcError::DivisionByZero: return "division by zero";
        case CalcError::UnknownOperator: return "unknown operator";
        case CalcError::Overflow: return "overflow";
    }
    return "unknown error";
}
```

The `switch` in `describe` has no `default` on purpose: add a code to the enum and `-Wall`'s `-Wswitch` reports every `switch` that does not handle it. The output parameter is the honest C++20 shape when the value and the code must travel together; a small `struct Result { long long value; CalcError error; }` is the alternative when the pair is returned often.

---

std::error_code

`<system_error>` generalises the enum into something libraries can share. A `std::error_code` is an integer plus a pointer to a *category* that knows what the integer means, so codes from the OS, from the C library and from your own subsystem travel through one type without colliding. `std::errc` is the portable enumeration of POSIX conditions, and `std::from_chars` is the standard function you will meet that reports through it:

```
int value = 0;
auto [ptr, ec] = std::from_chars(text.data(), text.data() + text.size(), value
if (ec == std::errc::invalid_argument)         { /* no digits */ }
else if (ec == std::errc::result_out_of_range) { /* too big for int */ }
else if (ptr ≠ text.data() + text.size())      { /* digits, then junk */ }
```

`ec == std::errc{}` is success; `ptr` marks where parsing stopped, so "the whole token was a number" is a comparison with the end, not a guess. `ec.message()` renders the code as text — implementation-specific text, so on this track print your own. `std::system_error` is the exception that carries an `error_code` across a boundary, and `std::filesystem` (Module 18, lesson 5) offers every operation twice, throwing or taking an `std::error_code&` — the standard library's own acknowledgement that both styles have a place.

---

std::expected (C++23, reading only)

`std::expected<T, E>` holds either a `T` or an `E`, never both, with the optional's interface on the value side and `error()` on the other:

```
std::expected<int, ParseError> parseInt(std::string_view s);   // C++23

auto r = parseInt("42");
if (r) use(*r); else report(r.error());
return std::unexpected(ParseError::Empty);                     // how the funct

int port = parseInt(text).and_then(checkRange).value_or(8080);
```

It is the type the return-code tradition was reaching for: value and error are one object, ignoring the error is impossible without ignoring the value, and the monadic members chain the happy path. This runtime is C++20 and does not have it. The C++20 spellings are `std::variant<T, E>` visited (lesson 4), or `std::optional<T>` beside a code — the checkpoint uses the first.

Assertions

An *assertion* states a fact about your own code that cannot be false unless the program has a bug:

```cpp
#include <cassert>

int median(std::vector<int>& v) {
    assert(!v.empty() && "median of an empty range");   // a caller broke the
    std::nth_element(v.begin(), v.begin() + v.size() / 2, v.end());
    return v[v.size() / 2];
}

static_assert(sizeof(long long) == 8, "this code assumes 64-bit long long");
```

`assert(expr)` aborts the process with the file and line when `expr` is false — a runtime error on the judge — and is compiled out *entirely* when `NDEBUG` is defined, expression included, so an assertion must never have a side effect. `-O2` does not define `NDEBUG`; release builds usually do. Hence the rule: assertions check *bugs* — invariants and preconditions inside code you control — and never validate input, because in release the check vanishes and the bad input sails through. `static_assert` is the compile-time form — free, always on, for facts about types and constants. C++26 adds contracts (`pre` and `post` on a declaration) as a configurable successor to `assert`; reading only.

A strategy per layer

The question is never "exceptions or codes?" for a whole program but "which mechanism at this layer?". A processing pipeline has a natural shape:

```
input line  →  parse  →  validate  →  execute  →  report
```

Parse turns text into a structure. Malformed input is ordinary — it is what input does — so parse returns `std::optional<Request>`, and the caller handles the empty case in one `if`.

Validate applies the rules of the domain. Violations are ordinary and enumerable, so validate returns an error code the caller can switch on.

Execute performs the operation. Here a failure means the world refused — the account is short, the file is gone — and the code that discovers it is usually several calls below anyone who can decide what to do. It throws, with context, and the exception carries the failure up through frames that have nothing to add.

Report is the boundary. It catches, translates to output and an exit code, and is the one place `catch (...)` is acceptable.

Return values inside, throw at the boundary. Two corollaries: do not catch what you cannot handle — a handler that only logs and swallows hides the failure from the layer that could act — and translate at module boundaries, so a caller sees `ConfigError`, not `std::out_of_range` from three libraries down.

Logging versus failing

Diagnostics go to `std::cerr`; results go to `std::cout`. The judge reads only standard output, which makes `cerr` a free channel for the trace you use to debug a failing case. A message worth logging says what was attempted, what was found and what was expected — `line 7: expected an integer, found "ten"` — and the same three parts make a good exception message. Fail fast on bugs; recover from expected failures; never `catch (...) {}` silently — an empty handler is a bug with permission to hide.

Choosing the mechanism

Mechanism	Use it for
`std::optional<T>`	absence with no reason needed
`enum class` code / `std::error_code`	enumerable failures a caller switches on; C and OS boundaries
`std::variant<T, E>` / `std::expected`	a value or a reason, in one object
exception	rare failures the immediate caller cannot fix; constructors; deep chains
`assert` / `static_assert`	bugs, never input

Pitfalls

Mistake	What happens
Reading `errno` without clearing it first	A stale code from an earlier call
Ignoring a returned code	Silent garbage; `[[nodiscard]]` turns it into a warning
`assert(read(fd, buf, n) > 0)`	The read disappears under `NDEBUG`
`assert` on user input	Aborts in debug, passes bad data in release
`catch (...) {}`	The failure vanishes; the next symptom is far away
Throwing from `parse` for every bad line	Microseconds per line; a return value is the shape of the job

Key takeaways

C reports through return values and `errno`; the pattern survives in every C API, so know its rules: clear, call, check.

A modern code is an `enum class` with `[[nodiscard]]` and a `switch` the compiler checks; `std::error_code` lets categories coexist, and `std::from_chars` reports through `std::errc`.

`std::expected<T, E>` is C++23; on this runtime use `std::variant<T, E>` or `optional` plus a code.

`assert` is for bugs and vanishes under `NDEBUG`; `static_assert` is compile-time; neither validates input.

Return values inside a layer, throw at the boundary, catch where you can act, translate between modules.

`cerr` for diagnostics, `cout` for answers; never swallow silently.

EXERCISE 1

A calculator that returns codes

Write a calculator whose failures are an `enum class CalcError { Ok, DivisionByZero, UnknownOperator, Overflow };` returned from `[[nodiscard]] CalcError compute(int a, char op, int b, long long& out)`. Operators are `+ - * / %`; compute in `long long`; `/` and `%` by zero are `DivisionByZero`; any other operator is `UnknownOperator`; a result outside the range of `int` is

`Overflow`. `const char* describe(CalcError)` names each code through a `switch` with no `default`, so the compiler reports a code you forget.

Input: lines `<a> <op> <b>` until EOF; `a` and `b` fit in `int`. **Output:** `<a> <op> <b> = <result>` on success, `<a> <op> <b>: <description>` otherwise, with descriptions `division by zero`, `unknown operator`, `overflow`.

```
7 + 5
9 / 0
2147483647 + 1
6 ^ 2
-7 / 2
```

prints

```
7 + 5 = 12
9 / 0: division by zero
2147483647 + 1: overflow
6 ^ 2: unknown operator
-7 / 2 = -3
```

INPUT 1

```
7 + 5
9 / 0
2147483647 + 1
6 ^ 2
-7 / 2
```

OUTPUT 1

```
7 + 5 = 12
9 / 0: division by zero
2147483647 + 1: overflow
6 ^ 2: unknown operator
-7 / 2 = -3
```

INPUT 2

```
10 % 3
-7 % 3
5 % 0
```

OUTPUT 2

```
10 % 3 = 1
-7 % 3 = -1
5 % 0: division by zero
```

REFERENCE SOLUTION

```cpp
#include <iostream>
#include <limits>

enum class CalcError { Ok, DivisionByZero, UnknownOperator, Overflow };

[[nodiscard]] CalcError compute(int a, char op, int b, long long& out) {
    const long long x = a;
    const long long y = b;
    switch (op) {
        case '+': out = x + y; break;
        case '-': out = x - y; break;
        case '*': out = x * y; break;
        case '/':
            if (y == 0) return CalcError::DivisionByZero;
            out = x / y;
            break;
        case '%':
            if (y == 0) return CalcError::DivisionByZero;
            out = x % y;
            break;
        default:
            return CalcError::UnknownOperator;
    }
    if (out < std::numeric_limits<int>::min() || out > std::numeric_limits<int>::max
    return CalcError::Ok;
}

const char* describe(CalcError e) {
    switch (e) {
        case CalcError::Ok: return "ok";
        case CalcError::DivisionByZero: return "division by zero";
        case CalcError::UnknownOperator: return "unknown operator";
        case CalcError::Overflow: return "overflow";
    }
    return "unknown error";   // unreachable while every code has a case; -Wswitch k
}

int main() {
    int a, b;
    char op;
    while (std::cin >> a >> op >> b) {
        long long result = 0;
        const CalcError error = compute(a, op, b, result);
        if (error == CalcError::Ok) {
            std::cout << a << ' ' << op << ' ' << b << " = " << result << '\n';
        } else {
            std::cout << a << ' ' << op << ' ' << b << ": " << describe(error) << '\
        }
    }
```

```
    return 0;
}
```

EXERCISE 2

Parse, validate, execute

Build the pipeline from the lesson and report the first layer that fails. A stock of items is loaded first; then each command line passes through three layers:

- **parse** — `std::optional<Request> parse(const std::string& line)`: exactly three tokens, `<command> <item> <quantity>`, the quantity converted with `std::from_chars` and consumed entirely (no `+`, no trailing characters). Anything else is `std::nullopt` → `parse error: <line>`.
- **validate** — `[[nodiscard]] ValidationError validate(const Request&)` with `enum class ValidationError { Ok, UnknownCommand, BadQuantity }`: the command must be `take` or `add` (checked first), the quantity must be 1–1000 → `validate error: unknown command` / `validate error: bad quantity`.
- **execute** — `int execute(std::map<std::string, int>& stock, const Request&)` returns the item's new quantity; `add` creates the item if needed; `take` throws `StockError("unknown item <item>")` (derived from `std::runtime_error`) when the item is not stocked and `StockError("not enough <item>")` when the stock is short.
- **report** — `main` prints `ok <item> <quantity>` or catches the `StockError` and prints `execute error: <what()>`.

Input: `n`, then `n` lines `<item> <quantity>`, then command lines until EOF (blank lines are skipped).
Output: one line per command.

```
2
apple 5
pear 2
take apple 3
take apple 3
take kiwi 1
add kiwi 4
take kiwi 4
grab apple 1
take apple 0
take apple
take pear x
```

prints

```
ok apple 2
execute error: not enough apple
execute error: unknown item kiwi
ok kiwi 4
ok kiwi 0
validate error: unknown command
validate error: bad quantity
parse error: take apple
parse error: take pear x
```

INPUT 1

```
2
apple 5
pear 2
take apple 3
take apple 3
take kiwi 1
add kiwi 4
take kiwi 4
grab apple 1
take apple 0
take apple
take pear x
```

OUTPUT 1

```
ok apple 2
execute error: not enough apple
execute error: unknown item kiwi
ok kiwi 4
ok kiwi 0
validate error: unknown command
validate error: bad quantity
parse error: take apple
parse error: take pear x
```

INPUT 2

```
0
add pen 3
take pen 3
take pen 1
```

OUTPUT 2

```
ok pen 3
ok pen 0
execute error: not enough pen
```

REFERENCE SOLUTION

```cpp
#include <charconv>
#include <iostream>
#include <map>
#include <optional>
#include <sstream>
#include <stdexcept>
#include <string>

struct Request {
    std::string command;
    std::string item;
    int quantity;
};

enum class ValidationError { Ok, UnknownCommand, BadQuantity };

class StockError : public std::runtime_error {
public:
    using std::runtime_error::runtime_error;
};

// Layer 1: text → structure. Malformed lines are ordinary, so: optional.
std::optional<Request> parse(const std::string& line) {
    std::istringstream in(line);
    std::string command, item, number, extra;
    if (!(in >> command >> item >> number) || (in >> extra)) return std::nullopt;
    int quantity = 0;
    const char* end = number.data() + number.size();
    auto [ptr, ec] = std::from_chars(number.data(), end, quantity);
    if (ec ≠ std::errc{} || ptr ≠ end) return std::nullopt;
    return Request{command, item, quantity};
}

// Layer 2: the domain's rules. Enumerable, so: a code.
[[nodiscard]] ValidationError validate(const Request& r) {
    if (r.command ≠ "take" && r.command ≠ "add") return ValidationError::UnknownCo
    if (r.quantity < 1 || r.quantity > 1000) return ValidationError::BadQuantity;
    return ValidationError::Ok;
}

// Layer 3: the world may refuse. Throws.
int execute(std::map<std::string, int>& stock, const Request& r) {
    if (r.command == "add") return stock[r.item] += r.quantity;
    auto it = stock.find(r.item);
    if (it == stock.end()) throw StockError("unknown item " + r.item);
    if (it→second < r.quantity) throw StockError("not enough " + r.item);
    return it→second -= r.quantity;
}
```

```cpp
int main() {
    int n;
    std::cin >> n;
    std::map<std::string, int> stock;
    for (int i = 0; i < n; ++i) {
        std::string item;
        int quantity;
        std::cin >> item >> quantity;
        stock[item] = quantity;
    }
    std::cin.ignore();
    std::string line;
    while (std::getline(std::cin, line)) {
        if (line.empty()) continue;
        const std::optional<Request> request = parse(line);
        if (!request) {
            std::cout << "parse error: " << line << '\n';
            continue;
        }
        switch (validate(*request)) {
            case ValidationError::UnknownCommand: std::cout << "validate error: unkr
            case ValidationError::BadQuantity: std::cout << "validate error: bad qua
            case ValidationError::Ok: break;
        }
        // Layer 4: the boundary. The only place that catches.
        try {
            const int left = execute(stock, *request);
            std::cout << "ok " << request->item << ' ' << left << '\n';
        } catch (const StockError& e) {
            std::cout << "execute error: " << e.what() << '\n';
        }
    }
    return 0;
}
```

QUIZ

1. Why must `errno` be set to `0` before calling `std::strtol`?

A. `strtol` reads it as an input parameter

B. Library calls set `errno` on failure but never clear it, so a stale value from an earlier call would be misread as this call's failure

C. Reading `errno` before writing it is undefined behaviour

D. `strtol` returns `errno` as its result

2. What does `[[nodiscard]]` on `CalcError compute(...)` do when a caller writes `compute(a, op, b, out);` and ignores the result?

A. Makes it a compile error

B. Makes the compiler emit a warning

C. Forces the caller to wrap the call in `try`

D. Converts the ignored code into an exception

3. `std::from_chars` is given the text `12ab` and an `int`. What does it report?

A. `ec == std::errc::invalid_argument`

B. `ec == std::errc{}` (success), `value == 12`, and `ptr` pointing at the `a`

C. It throws `std::invalid_argument`

D. `value == 0` and `ptr` at the start

4. With `NDEBUG` defined, what happens to `assert(read(fd, buf, n) > 0);`?

A. The check is skipped but `read` still runs

B. The entire statement is removed, so `read` is never called

C. The assertion becomes an exception

D. Nothing — `NDEBUG` only affects `static_assert`

5. In the parse → validate → execute → report pipeline, which layer throws?

A. parse — malformed input is an error

B. validate — rule violations are errors

C. execute — the world refused, and the code that finds out is far below anyone who can act

D. report — it is the boundary

6. Which statement about `std::expected<T, E>` is correct on this track's runtime?

A. It is available since C++17 alongside `std::optional`

B. It is C++23; on a C++20 compiler use `std::variant<T, E>` visited, or `std::optional<T>` beside a code

C. It is an alias for `std::optional<T>`

D. It is the same type as `std::error_code`

7. What is wrong with wrapping a step in `catch (...) {}`?

A. Nothing — it is the safe way to keep a program running

B. It swallows every failure silently, so the next symptom appears far from the cause

C. It does not compile outside `main`

D. It catches only exceptions derived from `std::exception`

ANSWER KEY · ERROR CODES, ASSERTIONS AND AN ERROR-HANDLING STRATEGY

1. B `errno` is a thread-local integer that only ever gets *set*. The protocol is clear, call, check — and the check must happen before any other call can overwrite it.

2. B The attribute turns a silently ignored return into a diagnostic — a warning, which `-Werror` can promote. It is the modern fix for the C weakness that a return code is optional to check.

3. B `from_chars` parses the longest valid prefix and never throws; `ptr` says where it stopped. "The whole token was a number" is the comparison `ptr == end`, which is why the exercise checks it — the parse alone does not.

4. B `assert` expands to nothing under `NDEBUG`, expression included. That is why an assertion must never carry a side effect and must never be the check that protects against bad input — in release it is not there.

5. C Parse returns an `optional` because bad lines are ordinary; validate returns a code because violations are enumerable; report catches. Only execute throws, so the failure can carry context up through frames that have nothing to add.

6. B `std::expected` arrived in C++23 with `value()`, `error()` and the monadic members. The runtime here is C++20, so it is reading only; the lesson names the two C++20 spellings, and the checkpoint uses the variant one.

7. B An empty handler is a bug with permission to hide. `catch (...)` belongs at the boundary, where it reports and exits; a handler that cannot act should not exist — let the exception reach one that can.

CHECKPOINT · 25 MIN

Checkpoint — Errors and exceptions

This checkpoint covers the whole module: throwing and catching with the standard hierarchy and your own, the three exception-safety guarantees with RAII and `noexcept`, `std::optional` for a value that may be absent, `std::variant` with `std::visit` for a value that is one of several types, and error codes, assertions and the per-layer strategy.

How it works. Fourteen questions and three programs. You need 70% on the questions and every program accepted to clear the module. You can retake it as often as you like; your best score counts.

Before you start, make sure you can answer these from memory:

- Why is `catch (const std::exception& e)` right and `catch (std::exception e)` wrong?
- What does `v.at(9)` throw, and what does `v[9]` do instead?
- What does `std::vector` do on reallocation when the element's move constructor is not `noexcept`?
- Is `*opt` on an empty optional a throw or undefined behaviour?
- What does `std::visit` refuse to compile?
- What does `NDEBUG` do to an `assert`?
- In a parse → validate → execute pipeline, which layer throws?

The three programs are a score reader that translates `std::stoi`'s exceptions into its own hierarchy, an RPN evaluator that returns a `std::variant` of result or failure, and a key–value store whose transactions roll back through an RAII guard. Every program must catch what it provokes and return 0 — an exception that escapes `main` is a runtime error, not a wrong answer.

EXERCISE 1

Score reader

Read scores from text and translate the standard library's exceptions into your own hierarchy at the boundary. `ScoreError` derives from `std::runtime_error` and inherits its constructor; `NotANumber` and `RangeError` derive from `ScoreError`, each with a constructor taking the offending text and building the message `not a number: <text>` or `out of range: <text>`. `int parseScore(const std::string& text)` calls `std::stoi(text, &consumed)`: catch its `std::invalid_argument` and

throw `NotANumber`, catch its `std::out_of_range` and throw `RangeError`; a token only partly consumed is also `NotANumber`, and a score outside 0–100 is also `RangeError`. The library's `what()` is never printed — only yours.

`main` reads `n` records and, per record, one `try` with a handler per derived type that prints `<name>: <what()>` and counts it. At the end print the average of the accepted scores to two decimals (or `none`) and the rejection counts.

Input: `n`, then `n` lines `<name> <score-text>`. **Output:** `n` lines, then `average: <x.xx>` or `average: none`, then `rejected: <a> not a number, <b> out of range`.

```
4
ada 93
bob 9x
cy 250
dee 100
```

prints

```
ada: 93
bob: not a number: 9x
cy: out of range: 250
dee: 100
average: 96.50
rejected: 1 not a number, 1 out of range
```

INPUT 1

```
4
ada 93
bob 9x
cy 250
dee 100
```

OUTPUT 1

```
ada: 93
bob: not a number: 9x
cy: out of range: 250
dee: 100
average: 96.50
rejected: 1 not a number, 1 out of range
```

INPUT 2

```
2
x abc
y -1
```

OUTPUT 2

```
x: not a number: abc
y: out of range: -1
average: none
rejected: 1 not a number, 1 out of range
```

REFERENCE SOLUTION

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

class ScoreError : public std::runtime_error {
public:
    using std::runtime_error::runtime_error;
};

class NotANumber : public ScoreError {
public:
    explicit NotANumber(const std::string& text) : ScoreError("not a number: " + tex
};

class RangeError : public ScoreError {
public:
    explicit RangeError(const std::string& text) : ScoreError("out of range: " + tex
};

// The boundary with the standard library: its exceptions are translated here,
// so nothing above this function ever sees (or prints) the library's text.
int parseScore(const std::string& text) {
    int score = 0;
    std::size_t consumed = 0;
    try {
        score = std::stoi(text, &consumed);
    } catch (const std::invalid_argument&) {
        throw NotANumber(text);
    } catch (const std::out_of_range&) {
        throw RangeError(text);
    }
    if (consumed ≠ text.size()) throw NotANumber(text);
    if (score < 0 || score > 100) throw RangeError(text);
    return score;
}

int main() {
    int n;
    std::cin >> n;
    long long sum = 0;
    int valid = 0, notNumbers = 0, outOfRange = 0;
    for (int i = 0; i < n; ++i) {
        std::string name, text;
        std::cin >> name >> text;
        try {
            const int score = parseScore(text);
            sum += score;
            ++valid;
```

```
            std::cout << name << ": " << score << '\n';
        } catch (const NotANumber& e) {
            ++notNumbers;
            std::cout << name << ": " << e.what() << '\n';
        } catch (const RangeError& e) {
            ++outOfRange;
            std::cout << name << ": " << e.what() << '\n';
        }
    }
    std::cout << std::fixed << std::setprecision(2);
    if (valid > 0) std::cout << "average: " << static_cast<double>(sum) / valid << '
    else std::cout << "average: none\n";
    std::cout << "rejected: " << notNumbers << " not a number, " << outOfRange << "
    return 0;
}
```

EXERCISE 2

RPN evaluator with a variant result

Evaluate reverse-Polish expressions and return the outcome as a value: `struct EvalError { std::string message; };` and `using Result = std::variant<long long, EvalError>;`. `Result evaluate(const std::string& line)` splits the line into tokens; an integer (an optional `-` then digits, parsed with `std::from_chars` into an `std::optional<long long>`) is pushed, and `+ - * /` pop two operands and push the result. The first failure ends the evaluation with an `EvalError`: `bad token <tok>` for anything else, `stack underflow` when an operator finds fewer than two operands, `division by zero`, `empty expression` when the line has no tokens, and `leftover operands` when more than one value remains at the end. `evaluate` never throws. `main` prints each result with `std::visit` and an `overloaded` visitor.

Input: one expression per line until EOF. **Output:** `= <value>` or `error: <message>` per line.

```
3 4 + 2 *
1 0 /
5 +
2 3
7
hello
```

prints

```
= 14
error: division by zero
error: stack underflow
error: leftover operands
= 7
error: bad token hello
```

INPUT 1

```
3 4 + 2 *
1 0 /
5 +
2 3
7
hello
```

OUTPUT 1

```
= 14
error: division by zero
error: stack underflow
error: leftover operands
= 7
error: bad token hello
```

INPUT 2

```
15 7 1 1 + - / 3 * 2 1 1 + + -
```

OUTPUT 2

```
error: empty expression
= 5
```

REFERENCE SOLUTION

```cpp
#include <charconv>
#include <iostream>
#include <optional>
#include <sstream>
#include <string>
#include <variant>
#include <vector>

struct EvalError { std::string message; };
using Result = std::variant<long long, EvalError>;

template <typename... Ts>
struct overloaded : Ts... {
    using Ts::operator()...;
};

std::optional<long long> parseInt(const std::string& s) {
    long long value = 0;
    const char* end = s.data() + s.size();
    auto [ptr, ec] = std::from_chars(s.data(), end, value);
    if (ec != std::errc{} || ptr != end) return std::nullopt;
    return value;
}

// Every failure is a value: the caller gets a Result either way and nothing
// unwinds through main.
Result evaluate(const std::string& line) {
    std::istringstream in(line);
    std::vector<long long> stack;
    std::string token;
    while (in >> token) {
        if (auto number = parseInt(token)) {
            stack.push_back(*number);
            continue;
        }
        if (token != "+" && token != "-" && token != "*" && token != "/") return Eva
        if (stack.size() < 2) return EvalError{"stack underflow"};
        const long long b = stack.back();
        stack.pop_back();
        const long long a = stack.back();
        stack.pop_back();
        if (token == "/" && b == 0) return EvalError{"division by zero"};
        long long r = 0;
        if (token == "+") r = a + b;
        else if (token == "-") r = a - b;
        else if (token == "*") r = a * b;
        else r = a / b;
        stack.push_back(r);
    }
```

```
    if (stack.empty()) return EvalError{"empty expression"};
    if (stack.size() > 1) return EvalError{"leftover operands"};
    return stack.back();
}

int main() {
    std::string line;
    while (std::getline(std::cin, line)) {
        std::visit(overloaded{
            [](long long value) { std::cout << "= " << value << '\n'; },
            [](const EvalError& e) { std::cout << "error: " << e.message << '\n'; },
        }, evaluate(line));
    }
    return 0;
}
```

EXERCISE 3

Transactions that roll back

A key–value store (`std::map<std::string, long long>`) processes transaction blocks, each from a `begin` line to a `commit` or `abort` line. Commands inside a block: `set <key> <value>` , `add <key> <value>` (the key must exist), `del <key>` (the key must exist), `get <key>` (prints `<key>: <value>` or `<key>: (none)` through an `std::optional` lookup). A missing key throws `StoreError("no key <key>")` (derived from `std::runtime_error`).

Write `Rollback` , an RAII guard that copies the store in its constructor and, in its destructor, restores the copy unless `commit()` was called. Run each block's commands with one `Rollback` in scope; call `commit()` only when the block ended with `commit` . You write no rollback code at any call site: an `abort` , a thrown `StoreError` , and a block cut off by the end of input all restore the store because the guard's destructor runs. After each block `main` prints `committed` , `aborted` or `failed: <what()>` , then the store as `store: k1=v1 k2=v2 …` in key order, or `store: (empty)` .

Input: blocks as described, until EOF. **Output:** the `get` lines as they happen; after each block its verdict and the store.

```
begin
set a 1
set b 2
commit
begin
set c 3
del zz
commit
begin
add a 10
get a
abort
begin
add b 5
get b
get q
commit
```

prints

```
committed
store: a=1 b=2
failed: no key zz
store: a=1 b=2
a: 11
aborted
store: a=1 b=2
b: 7
q: (none)
committed
store: a=1 b=7
```

INPUT 1

```
begin
set a 1
set b 2
commit
begin
set c 3
del zz
commit
begin
add a 10
get a
abort
begin
add b 5
get b
get q
commit
```

OUTPUT 1

```
committed
store: a=1 b=2
failed: no key zz
store: a=1 b=2
a: 11
aborted
store: a=1 b=2
b: 7
q: (none)
committed
store: a=1 b=7
```

INPUT 2

```
begin
set k 5
add k 5
get k
del k
get k
commit
```

OUTPUT 2

```
k: 10
k: (none)
committed
store: (empty)
```

REFERENCE SOLUTION

```cpp
#include <iostream>
#include <map>
#include <optional>
#include <sstream>
#include <stdexcept>
#include <string>
#include <utility>
#include <vector>

using Store = std::map<std::string, long long>;

class StoreError : public std::runtime_error {
public:
    using std::runtime_error::runtime_error;
};

// Snapshots the store; restores it on every exit unless commit() was called.
// The strong guarantee for a whole block, paid for with one copy of the map.
class Rollback {
public:
    explicit Rollback(Store& store) : store_(store), snapshot_(store) {}
    ~Rollback() {
        if (!committed_) store_ = std::move(snapshot_);
    }
    void commit() noexcept { committed_ = true; }
    Rollback(const Rollback&) = delete;
    Rollback& operator=(const Rollback&) = delete;
private:
    Store& store_;
    Store snapshot_;
    bool committed_ = false;
};

std::optional<long long> get(const Store& store, const std::string& key) {
    auto it = store.find(key);
    if (it == store.end()) return std::nullopt;
    return it->second;
}

Store::iterator existing(Store& store, const std::string& key) {
    auto it = store.find(key);
    if (it == store.end()) throw StoreError("no key " + key);
    return it;
}

void apply(Store& store, const std::string& line) {
    std::istringstream in(line);
    std::string op, key;
    long long value = 0;
```

```
    in >> op >> key;
    if (op == "set") {
        in >> value;
        store[key] = value;
    } else if (op == "add") {
        in >> value;
        existing(store, key)->second += value;
    } else if (op == "del") {
        store.erase(existing(store, key));
    } else if (op == "get") {
        if (auto v = get(store, key)) std::cout << key << ": " << *v << '\n';
        else std::cout << key << ": (none)\n";
    }
}

// The block's commands run under a guard; the last line decides its fate.
void runBlock(Store& store, const std::vector<std::string>& block, bool commit) {
    Rollback guard(store);
    for (const std::string& line : block) apply(store, line);
    if (commit) guard.commit();
}

void dump(const Store& store) {
    std::cout << "store:";
    if (store.empty()) std::cout << " (empty)";
    for (const auto& [key, value] : store) std::cout << ' ' << key << '=' << value;
    std::cout << '\n';
}

int main() {
    Store store;
    std::string line;
    while (std::getline(std::cin, line)) {
        if (line != "begin") continue;
        std::vector<std::string> block;
        std::string ending = "abort";        // the end of input without a verdict d
        while (std::getline(std::cin, line)) {
            if (line == "commit" || line == "abort") { ending = line; break; }
            block.push_back(line);
        }
        try {
            runBlock(store, block, ending == "commit");
            std::cout << (ending == "commit" ? "committed" : "aborted") << '\n';
        } catch (const StoreError& e) {
            std::cout << "failed: " << e.what() << '\n';
        }
        dump(store);
    }
```

```
    return 0;
}
```

CHECKPOINT QUESTIONS

1. What does this print?

```
struct L {
    char c;
    ~L() { std::cout << c; }
};
void g() { L x{'x'}; throw 1; }
void f() { L y{'y'}; g(); std::cout << 'f'; }
int main() {
    try { f(); } catch (int) { std::cout << 'h'; }
    std::cout << 'e';
}
```

A. `xyhe`

B. `yxhe`

C. `xyfhe`

D. `he`

2. Inside `catch (const std::exception& e)` the code writes `throw e;`. An outer `try` has `catch (const std::out_of_range&)` then `catch (const std::exception&)`. The original exception was a `std::out_of_range`. Which outer handler runs?

A. The `out_of_range` handler — the original type is preserved

B. The `std::exception` handler — `throw e;` throws a copy sliced to the static type of `e`

C. Neither: rethrowing from a handler is a compile error

D. Both, in order

3. `v` is a `std::vector<int>` of three elements. Which statement is correct?

A. `v.at(9)` and `v[9]` both throw `std::out_of_range`

B. `v.at(9)` throws `std::out_of_range`; `v[9]` is undefined behaviour

C. `v.at(9)` and `v[9]` are both undefined behaviour

D. `v[9]` throws `std::out_of_range`; `v.at(9)` is undefined behaviour

4. `Big` has a user-written copy constructor and a user-written move constructor without `noexcept`. What does `std::vector<Big>` do to the existing elements when it reallocates?

A. Moves them

B. Copies them — `std::move_if_noexcept` falls back to the copy constructor because the move might throw

C. Fails to compile

D. Moves them but disables the strong guarantee for that call

5. Which expression on an empty `std::optional<int> o` throws rather than being undefined behaviour?

A. `*o`

B. `o.value()`

C. `o→foo()`

D. `o.value_or(0)`

6. When does `std::visit(visitor, v)` refuse to compile?

A. When the visitor lacks a callable overload for some alternative of `v`, or its overloads return incompatible types

B. When `v` is currently empty

C. When the visitor is a lambda instead of a struct

D. When `v` has more than four alternatives

7. What does `#define NDEBUG` do to `assert(f() > 0);`?

A. Keeps the call to `f()` but drops the check

B. Removes the whole statement, so `f()` is never called

C. Turns a failure into an exception instead of an abort

D. Nothing — `NDEBUG` affects only `static_assert`

8. In a parse → validate → execute → report pipeline, which layer should throw?

A. parse

B. validate

C. execute

D. report

9. What does this print?

```
std::map<std::string, int> m{{"a", 1}};
auto get = [&](const std::string& k) → std::optional<int> {
    if (auto it = m.find(k); it ≠ m.end()) return it→second;
    return std::nullopt;
};
std::cout << get("a").value_or(-1) << get("b").value_or(-1);
```

A. `1-1`

B. `10`

C. `1` and then it throws

D. Compile error: a lambda cannot return `std::optional`

10. What does this print?

```
std::variant<std::monostate, int, std::string> v;
std::cout << v.index();
v = 5;
std::cout << std::holds_alternative<int>(v);
v = std::string("s");
std::cout << v.index();
```

A. `012`

B. `102`

C. `011`

D. `002`

11. A function gives the *strong* exception guarantee when…

A. it cannot throw at all

B. if it throws, the program state is exactly as it was before the call — commit or roll back

C. if it throws, nothing leaks and every invariant holds, but the state may have changed

D. it is declared `noexcept`

12. Which statement rethrows the exception currently being handled, without copying or slicing it?

A. `throw;`

B. `throw e;`

C. `throw std::current_exception();`

D. `return e;`

13. Which of these never throws?

A. `std::stoi("x")`

B. `std::from_chars(first, last, value)`

C. `v.at(3)` on an empty vector

D. `new int[1000000000000]`

14. Why does the score reader translate `std::stoi`'s exception into its own `NotANumber` instead of printing `e.what()` from the library's exception?

A. Calling `what()` on a library exception is undefined behaviour

B. The library's message text is implementation-specific, and a module boundary should hand callers a type in its own vocabulary, with context added

C. `what()` returns a null pointer for `std::invalid_argument`

D. `std::stoi` does not throw, so there is nothing to print

ANSWER KEY · CHECKPOINT — ERRORS AND EXCEPTIONS

1. A Unwinding leaves `g` (destroying `x`) then `f` (destroying `y`) before the handler prints `h`; `f`'s own `'f'` is never reached. Execution then continues after the `try` / `catch` with `e`.

2. B `throw e;` copy-constructs a new `std::exception` from `e`'s static type; the derived part is gone and the `out_of_range` handler no longer matches. A bare `throw;` rethrows the original object and would reach the first handler.

3. B Only `at()` checks. `operator[]` is the unchecked access, and out of range there is undefined behaviour — it may read garbage, crash, or be folded away by the optimiser.

4. B To keep `push_back`'s strong guarantee, the library only moves when the move cannot throw halfway; otherwise it copies, silently and on every reallocation. Adding `noexcept` to the move constructor restores the move.

5. B `value()` checks and throws `std::bad_optional_access`. `*o` and `o→` are unchecked — undefined behaviour when empty — and `value_or` never fails; it returns the default.

6. A The compile-time check is the point of `visit`: every alternative must be accepted and every branch must agree on a return type. A variant is never "empty" in normal use, lambdas are the usual visitors, and there is no alternative limit.

7. B `assert` expands to nothing under `NDEBUG`, expression included. An assertion with a side effect changes the program's behaviour between debug and release, which is why assertions check bugs and never do work.

8. C Malformed input is ordinary, so parse returns `std::optional`; rule violations are enumerable, so validate returns a code; execute discovers that the world refused, far below anyone who can act, so it throws; report is the boundary that catches.

9. A `get("a")` is engaged and yields 1; `get("b")` is empty and `value_or` supplies the default. `value_or` never throws, and a lambda's trailing return type may be any type.

10. A Default construction picks the first alternative, `std::monostate`, at index 0. After `v = 5` the `int` is held, so `holds_alternative<int>` is `1`; the string is the third alternative, index 2.

11. B Strong is the commit-or-rollback promise. "Cannot throw" is the no-throw guarantee, "no leak but changed" is the basic guarantee, and `noexcept` is a specifier, not a guarantee about state.

12. A The operand-less `throw;` reuses the exception object in flight. `throw e;` copies from the handler's parameter and slices; `std::current_exception()` yields an `exception_ptr`, which is itself what would be thrown; `return` is not a throw.

13. B `from_chars` reports through an `std::errc` in its result and is specified never to throw. `stoi` throws `std::invalid_argument`, `at()` throws `std::out_of_range`, and an impossible allocation throws `std::bad_alloc`.

14. B `what()` for the library's own exceptions is whatever the implementation chose (libstdc++ says `stoi`, MSVC something else), so exact output cannot depend on it. Translation at the boundary also lets the caller catch `ScoreError` rather than three types from three libraries down.

CodeKairo · C++ study plan · Module 15: Errors and exceptions. Solve the exercises online to track your progress and earn XP.