CS 247

Software Engineering Principles

Chapter 6

Assertions, Exceptions, and Smart Pointers

Victoria Sakhnini

Table of Contents

Introduction 4
Defensive Programming 4
Categories of Failure 4
Assertions 5
What Assertions Are For 6
Why Halt Immediately? 6
Turning Assertions Off in Production 6
Compiler Warnings: The Cheapest Defence 7
Traditional Error Handling 8
Approach 1: Terminate 8
Approach 2: Return Codes and Status Variables 8
Why Return Codes Get Painful 9
Exceptions: The C++ Approach 10
The Core Idea: Detection vs. Handling 11
Why This Is an Improvement 11
Throwing and Catching Exceptions 12
Throwing 12
Building Your Own Exception Class 12
Catching 13
The Matching Rules 14
The Standard Exception Hierarchy 15
The Root: std::exception 15
Standard-Library Subclasses 16
Implementing a Subclass: runtime_error 16
Stack Unwinding 17
The Four Steps of Throwing 18
A Picture of the Stack 18
Destructors During Unwinding 19
Rethrowing and Catch-All Handlers 21
A Worked Example 21
Rethrow vs. Throw-New 22
Catch-All 23
Exceptions in Constructors 23
What Happens When a Constructor Throws 24

Function-Try-Blocks for Member Initializers 24
Using the Constructor Exception 25
Exception Safety Guarantees 26
The Three Guarantees 26
Conditions for the Guarantees 27
Resources Beyond Memory 27
The Resource-Leak Problem 27
The Stack-Object Guarantee 28
Smart Pointers 29
How a Smart Pointer Works 29
std::unique_ptr 30
Transferring ownership 31
Why auto_ptr was removed 32
std::shared_ptr 32
std::weak_ptr 33
The circular-reference problem 33
The fix: make one direction weak 34
The RAII Idiom 35
Why It Works 36
A File-Handling Example 36
The Resources Worth Wrapping 37
A Complete Example: Rational with pImpl and unique_ptr 38
noexcept and terminate() 41
When to Mark Functions noexcept 41
set_terminate and terminate() 42
Without noexcept, Same Code Works Fine 43
Exceptions vs. Assertions 43
The Two Failure Modes Compared 44
Best Practices 45
Summary 47
Practice Problems 48

Introduction

Every non-trivial program will encounter situations that its straight-line logic cannot handle: a file that does not exist, an integer divided by zero, memory that cannot be allocated, a client who passes garbage to one of your functions, a precondition that your own implementation forgot to maintain. The interesting question is not whether such situations will arise; they will, but how the program should detect, report, and recover from them while keeping the rest of the code readable.

This chapter develops three complementary techniques that C++ provides for that purpose. Assertions catch programming errors as close to their cause as possible, halting the program before the bug can corrupt anything else. Exceptions transmit error information from the point where a problem is detected to a distant point where the program can sensibly respond, without polluting the intervening code with status checks. Smart pointers and the RAII idiom then make exception-handling code safe: they guarantee that resources acquired in normal code are released even when control is unwound by an exception.

These tools are not interchangeable. Assertions are for **our own** mistakes and broken assumptions inside the program. Exceptions are for **environmental** problems the program could not prevent, missing files, full disks, and bad user input. Smart pointers are the mechanism that lets those two ideas coexist without leaking memory. Understanding which tool fits which kind of failure is one of the deepest design skills in C++.

“The question isn't whether or not we will make programming errors… the question is whether or not we will arrange for our compiler and tools to help us find them,” , Sutter & Alexandrescu, *C++ Coding Standards*

Defensive Programming

Defensive programming is the deliberate practice of writing code that fails loudly and early rather than silently and late. The goal is not to prevent every conceivable failure, that would be impossible, but to arrange the program so that when something does go wrong, the symptom appears as close as possible to the cause. A bug that crashes the program on the line that violates an invariant is enormously easier to fix than one that crashes ten thousand lines later, after the corrupted data has been written to disk, copied across structures, and finally dereferenced as a null pointer.

Categories of Failure

Before deciding how to defend against errors, it helps to know what kinds of errors exist. They fall into three broad categories, each of which calls for a different response:

Errors in your own code. Bugs you wrote: off-by-one indices, uninitialized variables, broken representation invariants, accidentally exposing the data representation of an ADT. These are mistakes you want to catch before the code ships, ideally on the very first run during development.

Misuse of your code by a client. A caller violates the precondition of a function you wrote: passes a negative size to a `resize()` call, calls `pop()` on an empty stack, hands a `nullptr` to a function whose contract requires a real object. From your point of view, this is still a programming error, but the bug lives in the client's code, not yours.

External problems. Failures that no amount of careful coding can prevent, the program runs out of memory, an input file does not exist or has wrong permissions, hardware arithmetic overflows, a network connection drops, or the user types something nonsensical. Your program did nothing wrong; the world simply did not cooperate.

The crucial distinction is between the first two categories (which we will call logic errors or programming errors) and the third (environmental errors). C++ uses assertions to handle the first two and exceptions to handle the third. Mixing them up, using exceptions to report a broken invariant, say, or assertions to detect a missing file, leads to code that is both harder to debug and harder to recover from.

> 💡 **Tip: Mental model.** Ask of every failure: *could the program have been written so that this never happens?* If yes (a bad index, a violated precondition, a corrupted invariant), it is a **programming error**; use an assertion. If no (a missing file, an out-of-memory condition, malformed user input), it is an **environmental error**; use an exception.

Assertions

An **assertion** is a Boolean expression that the programmer claims must be true at a particular point in the program. If the claim turns out to be false, the assertion macro prints a diagnostic message naming the file, line, and the expression that failed, and then aborts the program. The header is `<cassert>` (or, in older code, `<assert.h>`), and the macro is simply called assert.

```
#include <cassert>
double squareRoot(double x) {
    assert(x >= 0);          // Precondition: x must be non-negative
    // ... computation ...
}
void Stack::pop() {
    assert(!isEmpty());  // Precondition: there must be something to pop
    --top_;}
```

What Assertions Are For

Assertions document and check the assumptions a piece of code makes about the world it lives in. The four most common places to put them are:

Preconditions of a function are facts that must be true when the function is called. `assert(x >= 0)` at the top of `squareRoot` enforces the precondition that the argument is non-negative.

Postconditions of a function, facts that must be true when the function returns. After a sorting routine, `assert(isSorted(arr))` would verify that the function delivered what it promised.

Representation invariants of an ADT are facts that must remain true between operations. A Rational class might assert that its denominator is never zero and its numerator and denominator are coprime; a List might assert that its size counter agrees with the actual number of nodes.

Postconditions of called functions, once a function returns, an assertion can verify that the value it produced makes sense, defending you against a bug elsewhere that you have not yet found.

In each case, the assertion serves two purposes simultaneously. To the human reader, it is a piece of documentation, written in code, that states what the surrounding logic assumes. To the running program, it is a checkpoint: if reality ever disagrees with the documentation, the program halts immediately so you can investigate while the failure is still fresh.

Why Halt Immediately?

It might seem heavy-handed to abort the whole program because of one violated assumption. The reason is diagnostic. When an assertion fires, the stack is still intact, all local variables of all active functions are still in memory, and the debugger can show you exactly which call sequence led to the failure. If you let the program limp along after a broken invariant, the eventual crash will happen somewhere unrelated, perhaps inside a standard library function several stack frames away, and the chain of cause and effect will be much harder to reconstruct.

> 💡 **Tip:** The goal is to abort as close to the error as possible so that the program state at the moment of failure still reflects the cause. Assertions terminate execution **before** the failure has a chance to contaminate (and be obscured by) further computation.

Turning Assertions Off in Production

Whether to keep assertions enabled in a release build is a judgment call. The `assert` macro is controlled by the preprocessor symbol `NDEBUG`. When `NDEBUG` is *not* defined, every `assert(expr)`

expands to code that evaluates expr and aborts on failure. When `NDEBUG` *is* defined, every `assert(expr)` expands to nothing; the expression is not even evaluated.

```
% g++ -DNDEBUG main.cpp     # disables all assertions in this build
% g++ main.cpp              # leaves assertions enabled (the default)
```

There are two schools of thought on this. One holds that assertions should always be left on, that the cost of a Boolean check is negligible, and that you would rather have the program abort cleanly than continue in a corrupted state and risk writing bad data to persistent storage. The other holds that there are environments (medical devices, flight controllers, long-running servers) where stopping the program is itself an unacceptable outcome, and a corrupted-but-running program is at least giving you a chance to recover. In practice, *most* assertions stay on by default; you turn them off only when profiling shows that a particular hot inner loop is spending real time on checks.

> ⚠️ **Warning:** **Assertions must not have side effects.** Because assertions can be compiled out, any state change you embed in one will silently vanish in production. The classic mistake:
>
> assert(i++ == 10); // i is incremented in debug builds, NOT in release builds
>
> Now the program behaves differently depending on whether assertions are enabled, exactly the opposite of what assertions are supposed to give you. Always write assert(i == 10); ++i; instead.

Compiler Warnings: The Cheapest Defence

Before any runtime check, the cheapest line of defence is the compiler itself. Modern C++ compilers can detect a remarkable range of bugs at compile time, but only if you ask them to. The default warning level is far too quiet for serious work.

```
# Enable maximum useful warnings, and turn them into errors
g++ -Wall -Wextra -Wpedantic -Werror -std=c++17 main.cpp

# C++17-specific warnings
g++ -Wc++17-extensions ...
```

These flags expose, among other things, uninitialized variables, type mismatches in printf-style format strings, unused variables and parameters, implicit narrowing conversions, potential null-pointer

dereferences, signed/unsigned comparison bugs, missing return statements, and uninitialized class members. Each warning is a bug that the compiler caught before the code ever ran.

Why -Werror? Once a warning exists in a codebase, it tends to accumulate company: people get used to seeing it and stop noticing when new ones appear. Promoting warnings to errors forces the team to either fix the issue or explicitly suppress it for a documented reason. A clean compile is much easier to keep clean than to recover.

> ❓ **Question:** If assert can be compiled out, why bother writing it at all? Why not just put the check in a regular if statement that does the same thing?
>
> Because the assertion documents *intent*, a reader of the code (or a static analysis tool) immediately knows that this is an invariant, not part of the normal control flow. Because it gives a uniform crash format (file, line, expression) that lets every assertion failure be diagnosed the same way. And because there are environments where you genuinely want assertions disabled in release builds for performance, while keeping the documentation in the source.

Traditional Error Handling

Before C++ added exceptions to the language, programmers reported environmental failures using a few well-worn techniques: terminate the program, return a special error code, set a global status variable, or both. These techniques still work and still have their place, but they accumulate well-known problems as programs grow. Understanding those problems is the best motivation for the exception machinery we will study next.

Approach 1: Terminate

The simplest response to an error is to print a message and call `std::exit` or `std::abort`. This is fine for short throwaway scripts, but unusable for any program that cannot afford to vanish without warning, anything interactive, anything that holds open files or network connections, anything that is part of a larger system. A library function, in particular, has no right to terminate the entire process because of a single failed operation; that decision belongs to the application.

Approach 2: Return Codes and Status Variables

The more general technique, ubiquitous in C, is to have every function that might fail return a special value indicating success or failure, and for every caller to check that value before proceeding. The classic file-reading idiom looks like this:

```
#include <iostream>
#include <fstream>
#include <string>

int main() {
    std::ifstream infile;
    infile.open("data");
    if (!infile) {
        std::cout << "Error opening input file data." << std::endl;
        return 1;
    }
    while (!infile.eof()) {
        std::string name;
        infile >> name;
        if (infile.fail()) {
            // react to read failure (likely format error)
        }
        else if (infile.bad()) {
            // react to read failure (likely some data lost)
        }
        else {
            // process data
        }
    }
    return 0;
}
```

Notice how much of the source is dedicated to error checking compared to actually processing the data. In industrial code, this ratio is often inverted from what you might expect; measurements regularly show that more than half of a codebase is devoted to input validation, error checking, and recovery. The functional core of the program drowns in defensive scaffolding.

Why Return Codes Get Painful

Return-code error handling has six recurring problems, and a serious program will hit all of them sooner or later.

It is incomplete. The caller may simply forget to check the return value. The compiler will not complain, because ignoring a return value is legal C++. The bug is silent until something downstream breaks.

It is error-prone. Error information stored in a global status variable (think C's errno) may be overwritten by the next call before anyone checks it.

It is inefficient. Every single call must be wrapped in a check, even though failure is rare. The branch predictor handles this well, but the source code becomes a forest of if statements.

It is inelegant. Normal return values and error indicators must share the same return type. A function that should return an integer count of items now has to reserve -1 or UINT_MAX as an “error” value,

contaminating the type. If every integer is a valid result, you are forced into clumsy workarounds like std::pair<Value, ErrorCode> everywhere, or std::optional<T>, or out-parameters.

It is sometimes impossible. Constructors have no return value. There is literally no way to report a constructor failure using return codes; you would have to leave the object in a half-built, "invalid" state and require every caller to check after construction.

It forces local recovery. The function that detects the error is rarely the function that knows what to do about it. A low-level read function can detect that a file is unreadable, but the right place to *react* is the user-facing layer, where the program can pop up a dialogue or fall back to a default. Return codes mix detection and recovery at the same level. To pass an error up several layers, you have to invent a chain of return-and-check, each one re-encoding the failure and possibly losing detail along the way.

Beyond these six, there is a more fundamental issue: error-handling code is interleaved with normal code at every line. Every function call is followed by an if-block. This destroys the readability of the program's *happy path*, the sequence of operations that executes when nothing goes wrong. The reader has to mentally strip out the error checks to see what the code is actually doing.

> **Tricky:** In industrial software, **more than 50% of code** is typically devoted to input validation, error checking, and exception handling. Embedding all of that within the normal application logic distracts from the coherence of the overall functional design. The fact that exception handling looks like "extra syntax" is misleading; it is *less* syntax than what return-code style requires, just concentrated in different places.

Exceptions: The C++ Approach

C++ provides a dedicated language mechanism, exceptions, for transporting error information from the point where the error is detected to the point where it can be handled. The mechanism is built around three keywords:

`throw`, emit an exception. The function that detects the problem creates an object describing the error and throws it. Control leaves the function immediately, before the next statement executes.

`try`, mark a region of code as guarded. Any exception thrown inside a `try` block, including from functions called inside it, will be considered for delivery to the handlers that follow.

`catch`, declare a handler. A `catch` clause associated with a try block specifies which kinds of exceptions it can handle and what to do with them.

The Core Idea: Detection vs. Handling

The point of exceptions is the separation of detection from handling. The function that notices the problem rarely knows the right way to recover; the function that can recover often does not know how to detect the problem. Exceptions decouple the two: the detector throws, and the handler, possibly many stack frames away, catches.

An exception, in C++, is **an object**, a value of some type that the throwing code creates and the catching code receives. It can be a primitive (`int`, `const char*`), a user-defined class, or, by overwhelming convention, an instance of a class derived from `std::exception`. The object carries information about what went wrong; its type identifies the kind of failure.

Why This Is an Improvement

Compared to return codes, exceptions deliver four important benefits:

They separate normal code from error-handling code. The happy path stays in the try block; the recovery logic sits in the catch clauses that follow. A reader can scan the normal flow without being constantly diverted into `if (err) {...}` branches.

They separate risk-free code from risky code. A function declared `noexcept` advertises that it cannot `throw`. The compiler enforces this and can optimize accordingly. The caller of such a function does not need a `try` block at all.

They allow different layers to detect versus recover. Detection happens deep inside the library, where the failure is visible. Recovery happens up in the application, where context is available to decide what to do. The intervening stack frames are unwound automatically and need no error-handling code of their own.

They cannot be ignored. A return value can be silently dropped. An exception, if not caught anywhere, terminates the program. Silence is no longer an option.

> 💡 **Tip: Key C++17 improvements.** Modern C++ exceptions are stricter and cleaner than their early versions: catch exception objects by const reference (catch (const std::exception& e)), mark non-throwing functions with noexcept rather than the deprecated throw() specifier, and use the standard exception hierarchy plus a catch-all catch (...) for unknown types.

Throwing and Catching Exceptions

Throwing

To throw an exception, use the throw keyword followed by an expression. The result of the expression is copied into a special compiler-managed area (because the stack frame containing the original value is about to be destroyed), and control transfers to the nearest matching handler.

```
class Rational {
public:
    class DivideByZeroException {};   // nested exception type
    Rational(int, int);
    // ...
};

Rational::Rational(int numer, int denom) {
    if (denom == 0) {
      //throw 0;                                  // (1) throw an int
      //throw "Denominator cannot be 0.";  // (2) throw a string literal
      //throw DivideByZeroException();  // (3) throw a user-defined object
    }
    reduce(numer, denom);
    numerator_   = numer;
    denominator_ = denom;
}
```

The three commented-out alternatives are all legal; you can throw a value of any copyable type. The third form, throwing a user-defined object, is what you should almost always do, for three reasons:

Class hierarchies enable polymorphic handlers. If you organize your exception types under a common base class, one catch clause can handle a whole family. You will see this pattern throughout the standard library.

Objects carry information. A bare int or string literal cannot tell the handler *which* denominator was zero, *where* the call came from, or what to do next. A class can carry as many fields as you need.

Type names are documentation. Catching `DivideByZeroException` says exactly what is being handled. Catching int says nothing. Was it a division by zero, a bad index, or a corrupt file handle?

Building Your Own Exception Class

It is conventional to nest exception classes inside the ADT they belong to. This keeps the global namespace uncluttered and makes the relationship between class and exception immediately visible.

```
class Rational {
public:
    class DivideByZeroException {       // An ADT-specific exception type
    public:
        DivideByZeroException(int n) : numer_(n) {}
        int numer() const { return numer_; }
    private:
        int numer_;
    };
    // ...
};

Rational::Rational(int numer, int denom)
    : numerator_(numer), denominator_(denom) {
    if (denom == 0) {
        throw DivideByZeroException(numer);
    }
    reduce();
}
```

The exception object now carries the offending numerator, useful diagnostic information that a handler can include in an error message. Note also that, because the exception is a nested type, the qualified name `Rational::DivideByZeroException` makes the source crystal clear about which class the failure came from.

Catching

A try block groups statements that might throw. After the closing brace of the try come one or more catch clauses, each specifying a type. If an exception is thrown anywhere inside the try, control transfers to the first catch whose declared type matches the exception object.

```
int main() {
    Rational r(2000000000);
    Rational s(2000000000);
    try {
        Rational t(r * s);
        std::cout << t << std::endl;
    }
    catch (const std::runtime_error& e) {
        std::cout << e.what() << ": " << r << " * " << s << std::endl;
    }
    return 0;
}
```

The handler's identifier (e) works just like a function parameter: it names the exception object so that the handler's body can inspect it. You may omit the identifier if the type alone gives you everything you need.

The Matching Rules

When an exception is thrown, the runtime searches for a handler in a very specific way. Understanding these rules is essential to writing exception code that behaves the way you expect.

Stricter than function-parameter matching. The type of the thrown object and the type in the catch clause must match exactly, with only four allowed conversions:

- Conversions from non-const exception objects to const catch arguments.
- Conversions from a derived-class exception to a base-class catch argument (the polymorphic case).
- Arrays converted to pointers to the first element.
- Functions converted to pointers to functions.

Notably absent. There is no implicit numeric conversion. A thrown int will not be caught by a catch (double). A thrown `char*` will not be caught by a catch (`std::string`).

First match wins. Handlers are tried in source order; the first catch whose type matches gets the exception, even if a later catch would have been a better fit. This means you should always order catch clauses from **most derived to most general**: specific child classes first, then base classes, with catch (...) last.

⚠️ **Warning:** **Catch by reference, not by value.** Always write catch (const ExceptionType& e) rather than catch (ExceptionType e). Catching by value triggers a copy, which can slice off the derived parts of a polymorphic exception (only the base subobject is copied) and silently destroy the information you needed. Catching by const reference preserves the dynamic type and lets virtual functions like what() do the right thing.

❓ **Question:** What happens if no catch handler matches the thrown exception?

The runtime walks *up* the call stack, leaving the current function and looking for a matching handler in the caller, and the caller's caller, and so on. If it walks all the way out of main without finding one, the program aborts by calling std::terminate(). This is one of the key behaviors that makes exceptions “impossible to ignore”, the only way to silence one is to actively catch it somewhere.

The Standard Exception Hierarchy

Rather than invent a fresh exception class for every situation, you should usually derive your exceptions from the C++ standard library hierarchy. This gives client code a uniform interface, every standard exception has a what() method that returns a human-readable description, and lets a single catch (const std::exception&) handle anything thrown from your code or from the standard library.

The Root: std::exception

All standard exceptions derive from `std::exception` (declared in `<exception>`). The base class is small and deliberate:

```
#include <exception>
#include <string>

class exception {
public:
    exception() noexcept = default;
    exception(const exception&) noexcept = default;
    exception& operator=(const exception&) noexcept = default;
    virtual ~exception() noexcept = default;

    virtual const char* what() const noexcept {
        return "exception";
    }
};
```

Three details deserve attention:

noexcept everywhere. Every member function is marked `noexcept`, promising not to throw. This is critical: if an exception object's own methods could throw while you were trying to handle a previous exception, you would have two live exceptions at once, and the program would terminate. The handler must always be able to talk to the exception safely.

= default for special members. Instead of writing empty constructors, copy constructors, and assignment operators, we ask the compiler to generate them. This is C++11 syntax and is cleaner and more honest about intent.

Virtual destructor. Because the class is intended as a polymorphic base, handlers will catch derived exceptions through a `std::exception& parameter;` the destructor must be virtual so that deleting through a base pointer cleans up correctly.

Standard-Library Subclasses

The standard library splits its exceptions into two broad families, plus several special-purpose siblings:

`std::logic_error` (in `<stdexcept>`) and its subclasses indicate errors in program logic that *could* have been prevented by proper checks. The subclasses include:

- `std::domain_error`, a mathematical function received an argument outside its domain (e.g., `sqrt(-1)` if the function only accepts non-negatives).
- `std::invalid_argument`, a function received a malformed argument (e.g., wrong format string).
- `std::length_error`, an operation would exceed the maximum allowed size (e.g., resizing a container beyond its limit).
- `std::out_of_range`, accessed an element outside the valid range (e.g., `vector.at(i)` with `i` too large).

`std::runtime_error` and its subclasses indicate errors that are only detectable at runtime; they could not have been prevented by static checks. The subclasses include:

- `std::range_error`, a computation produced a result outside the expected range.
- `std::overflow_error`, arithmetic overflow.
- `std::underflow_error`, arithmetic underflow.
- `std::system_error`, an OS-level failure, paired with a `std::error_code`.
- `std::filesystem_error`, a file-system operation failed (missing file, permissions).
- `std::regex_error`, a regular expression is invalid or cannot be processed.
- `std::future_error`, a concurrency operation on `std::promise` or `std::future` was invalid.

Special-purpose siblings that derive directly from `std::exception`:

- `std::bad_alloc`, thrown by new when memory allocation fails (in `<new>`).
- `std::bad_array_new_length`, `new[]` called with an invalid length (negative or too large).
- `std::bad_cast`, `dynamic_cast` failed on a reference type.
- `std::bad_typeid`, typeid applied to a null pointer of polymorphic type.

Implementing a Subclass: runtime_error

Here is the typical implementation of `std::runtime_error`, which gives you a template for writing your own exception types:

```
class runtime_error : public exception {
public:
    explicit runtime_error(const std::string& what_arg)
        : message_(what_arg) {}

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

private:
    std::string message_;
};
```

Four points worth noting:

Inheritance: `runtime_error : public exception`. A runtime_error *is-a* exception, and a catch (`const std::exception&`) handler will catch it.

explicit constructor. Without explicit, `runtime_error err = "hello";` would silently construct an exception object from a string literal. That is too easy to do by accident; explicit requires the conversion to be deliberate.

The override keyword. Marking `what()` as override tells the compiler we believe we are overriding a virtual function from the base. If the signature doesn't match (wrong name, wrong constness, wrong return type), the compiler produces a diagnostic instead of silently giving us a *new* function that hides the base's. Always use override; it is one of the cheapest pieces of safety in modern C++.

noexcept on what(). A handler may want to print the exception's message. If `what()` could throw, that print might trigger a second exception while the first is still being handled, which is the recipe for `std::terminate`. The `noexcept` specifier promises this can never happen.

> 💡 **Tip:** When designing your own exceptions, follow this pattern: derive from the most specific standard subclass that fits, store any extra information you want to convey, and override what() to return a useful message. If nothing in the standard hierarchy fits well, derive from std::exception directly.

Stack Unwinding

Throwing an exception does much more than transfer control. The keyword throw sets off a cascade of “relatively magical” things, in Bjarne Stroustrup's phrase, that together make exception handling safe.

The Four Steps of Throwing

(1) The exception object is created. A *copy* of the thrown expression is made and stored in a compiler-managed location that survives the destruction of any stack frame. Why a copy? Because the original lives in the stack frame of the function that did the throw, and that frame is about to be torn down. The copy persists until the handler finishes.

(2) Control transfers to a matching handler. The runtime walks up the call stack searching for a catch clause whose declared type is compatible with the exception object's type, using the matching rules described in Section 5.

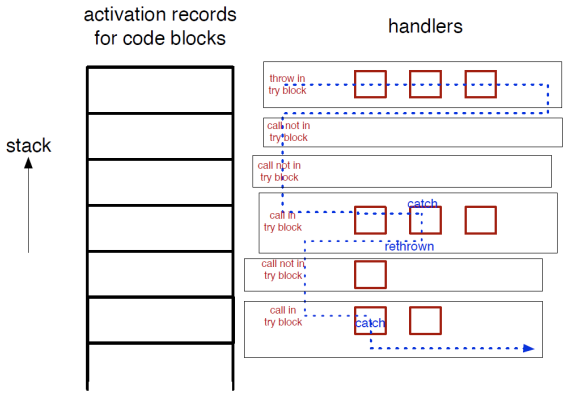
(3) Functions on the path are exited prematurely. Every function between the throw site and the handler is abandoned, its current statement never finishes, and any statements after it are skipped.

(4) Destructors run for stack-allocated objects along the way. This is the critical part. As each function frame is dismantled, every fully constructed local object in that frame has its destructor called in reverse order of construction. Partially constructed objects (those that failed in their own constructor) are also handled correctly: only the fields that were successfully constructed get destroyed.

This automatic invocation of the destructor is what makes exception handling viable. Without it, every function on the call chain would have to know about every exception that might pass through and arrange to clean up its local resources, defeating the whole point of separation of detection from recovery.

A Picture of the Stack

Imagine the call stack as a vertical stack of activation records, with the most recent call on top. Each record may or may not be inside a try block. When an exception is thrown from the top frame, the runtime peels off records one at a time, looking down for the nearest enclosing try block that has a matching handler.



If a matching handler is found, the exception object is delivered to it, and the program resumes normally after the handler. If the runtime walks off the bottom of the stack without finding a match, `std::terminate` is called and the program aborts.

Destructors During Unwinding

Here is a concrete example that demonstrates destructors firing during unwinding. The class Trace reports its construction and destruction; the program constructs several of them, then triggers an exception.

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

class Trace {
private:
    inline static int counter = 0;   // C++17: inline static member
initialization
    int objid;
public:
    Trace() {
        objid = counter++;
        std::cout << "constructing Trace #" << objid << '\n';
        if (objid == 3) {
            throw std::runtime_error("Trace construction failed at object
3");
        }
    }
    ~Trace() noexcept {        // C++17: Destructors should be noexcept
        std::cout << "destructing Trace #" << objid << '\n';
    }
    Trace(const Trace&) = delete;             // Delete copy operations
    Trace& operator=(const Trace&) = delete;
    Trace(Trace&&) noexcept = default;        // Default move operations
    Trace& operator=(Trace&&) noexcept = default;
};

int main() {
    try {
        Trace n1;
        std::array<Trace, 5> array;   // Throws exception during
                                      // construction of object #3
        Trace n2;                     // Never reached
    }
    catch (const std::exception& e) {
        std::cout << "caught exception: " << e.what() << '\n';
    }
    return 0;
}
```

And the output:

```
constructing Trace #0
constructing Trace #1
constructing Trace #2
constructing Trace #3
destructing Trace #2
destructing Trace #1
destructing Trace #0
caught exception: Trace construction failed at object 3
```

Trace through what happens. The variable n1 is constructed as object #0. Then we begin constructing the array of 5 Traces, object #1, then #2, then #3. The constructor of #3 throws. At this point:

- Object #3 was **not** fully constructed (its constructor body did not return normally), so its destructor is **not** called. This is essential; a destructor cannot safely clean up a half-built object.
- Object #4 is never constructed at all; the array's construction never reaches it.
- Objects #2, #1, and #0 (in that order, reverse of construction) **are** destroyed because they were fully constructed.
- n2 is never constructed because we never reached its declaration.
- Control transfers to the catch handler.

⚠ **Warning:** **Destructors must not throw during unwinding.** If a destructor that runs during stack unwinding itself throws an exception that it does not handle locally, the program has two live exceptions at once. There is no good way to merge them, so the runtime calls std::terminate() immediately and the program dies. This is why C++17 strongly encourages marking every destructor noexcept, it both promises the compiler that the destructor will not throw and enables better code generation.

❓ **Question:** Why is the exception object stored in a special compiler-managed location rather than on the stack?

Because the stack frame of the throwing function is destroyed as part of unwinding. If the exception lived there, it would be gone by the time the handler ran. The compiler arranges special storage that survives unwinding, its details are implementation-defined, but every implementation guarantees the exception object remains alive from the moment of throw until the handler that catches it finishes.

This has an important consequence: **never throw a pointer to a local object**. The compiler stores the pointer in the exception area, not the referent. By the time the handler dereferences it, the local it pointed to has been destroyed. If you must throw something on the heap, throw a smart pointer or the object itself, but in practice, throw a copy of a value type and let the runtime handle storage.

Tricky: The **static type** of the throw expression determines the exception object's type. Consider:

Base* p = new Derived;

throw *p; // throws a Base, not a Derived (object slicing!)

throw p; // throws a Base* (bad, points to a possibly-deleted object)

The first form slices: only the Base subobject is copied into the exception storage. The second form is dangerous unless the referent outlives the handler. The right way is to construct a temporary of the most-derived type directly: throw Derived(...);.

Rethrowing and Catch-All Handlers

Sometimes a handler can handle part of a problem but needs to pass the rest to a caller. C++ supports two forms of this. A bare throw; statement inside a catch rethrows the exception currently being handled. A throw expr; statement throws a new exception, abandoning the original. The choice affects which handler runs next and what information is preserved.

A Worked Example

Consider an exception hierarchy. Trouble with two derived classes, Small and Big, and a function that throws a Small:

```
class X {
public:
    class Trouble {};
    class Small : public Trouble {};
    class Big   : public Trouble {};
    void f() { throw Small(); }
};

int main() {
    X x;
    try {
        try {
            x.f();
```

```
        }
        catch (X::Small&) {
            std::cout << "Small Trouble" << std::endl;
            throw X::Trouble();          // Rethrow as a DIFFERENT
exception
        }
        catch (X::Big&)     { std::cout << "Big Trouble"  << std::endl; }
        catch (X::Trouble&) { std::cout << "Trouble"      << std::endl; }
        catch (...) {
            std::cout << "Catches any type of exception" << std::endl;
            throw;                       // Rethrow the ORIGINAL exception
        }
    }
    catch (X::Trouble&) {
        std::cout << "Outer catch: Trouble (from rethrow)" << std::endl;
    }
    return 0;
}
```

Output:

```
Small Trouble
Outer catch: Trouble (from rethrow)
```

Trace through it:

- `x.f()` throws a Small.
- The inner try has four handlers. The first matching one is catch (`X::Small&`), so its body runs and prints “Small Trouble”.
- Inside that handler, throw `X::Trouble();` creates a *new* exception (of type Trouble) and abandons the original Small.
- The new exception escapes the inner try entirely (you cannot catch the new exception in a sibling handler of the one that just threw it). It propagates up to the outer try.
- The outer handler catches `X::Trouble&` and prints “Outer catch: Trouble (from rethrow)”.

Rethrow vs. Throw-New

Bare throw; rethrows the current exception. The object is exactly the one the inner catch received, same type, same data. This is the form to use when you want to do some local cleanup or logging and then pass the same exception up.

throw expr; throws a different exception. Useful when you want to translate one kind of failure into another, for instance, a library function catches a low-level `std::system_error` and rethrows as a high-level ConfigurationException that means more to the user.

Catch-All

catch (...), the ellipsis, matches any exception type whatsoever. It is your last line of defence, the handler that runs when no specific type matches.

Catch-all handlers are useful in two situations:

Logging and rethrowing. You don't know what was thrown, but you want to log the fact and let a more competent handler higher up deal with it: catch (...) { log("unknown exception"); throw; }.

Boundaries of components. In a plugin host, the host should catch every possible exception at the plugin boundary, since a plugin might throw something exotic that the host knows nothing about. A catch (...) at the boundary turns any error into an orderly fallback.

> ⚠️ **Warning:** **Empty throw outside a catch is a bug.** Writing throw; outside a catch handler (or while no exception is being handled) is a serious error, there is nothing to rethrow. The runtime calls std::terminate() with the message “terminate called without an active exception”. A common version of this bug is putting throw; inside a helper function called from a catch handler and then calling the helper from somewhere else.

Exceptions in Constructors

Constructors are where exceptions earn their keep; they have no return value, so without exceptions, there is no way for a constructor to report failure. The pattern is straightforward: if a constructor cannot establish the class invariants, it throws.

```
Rational::Rational(int numer, int denom)
    : numerator_(numer), denominator_(denom) {
    if (denom == 0) {
        throw DivideByZeroException(numer);
    }
    reduce();
}
```

What Happens When a Constructor Throws

If a constructor throws, the C++ language guarantees that **the object was never constructed**. Its destructor will not run. Any field *subobjects* that did complete construction will have their destructors called in reverse order. Memory allocated by the new expression is reclaimed.

The principle is: the destructor is the companion of a successful constructor. If a constructor completes, the destructor will run when the object's lifetime ends. If a constructor throws, the destructor will not run; there is no fully-constructed object to destroy. This symmetry is the foundation on which RAII relies.

> ⚠️ **Warning:** There is one exception to the rule about field subobjects being cleaned up: raw pointers. If your constructor does new T() into a raw pointer and then throws before storing the result anywhere safe, the allocated memory leaks. The language has no way to know there was something to clean up. This is one of the major reasons to use smart pointers in members, they *are* objects with destructors, so they participate in the subobject cleanup machinery automatically.

Function-Try-Blocks for Member Initializers

By default, an exception thrown during a member initializer escapes the constructor entirely; the constructor body never even runs. But sometimes you want to handle such an exception (to log it, to translate it, to retry). C++ provides a special syntax called a *function-try-block* for this: the try keyword goes *before* the member initializer list, and a single catch clause follows the constructor body.

```
class Base {
public:
    virtual ~Base() = default;
};

class C {
public:
    C() { throw std::runtime_error("C constructor exception"); }
};

class MyClass : public Base {
public:
    class MyClassExcept : public std::exception {
    public:
        [[nodiscard]] const char* what() const noexcept override {
            return "MyClass construction failed";
        }
    };
```

```
    MyClass(C c, std::shared_ptr<C> p2);

private:
    C c_;
    std::unique_ptr<C> p_;
    std::shared_ptr<C> p2_;
};

MyClass::MyClass(C c, std::shared_ptr<C> p2)
try : Base(),
      c_(std::move(c)),
      p_(std::make_unique<C>()),       // This call will throw, C() throws
      p2_(std::move(p2))
{
    // (Body of constructor, never reached if initializer throws)
}
catch (const std::exception& e) {
    // No need to delete p_, unique_ptr handles it automatically.
    // p_ has already been destroyed during stack unwinding.
    throw MyClassExcept();   // Translate the exception
}
```

The try keyword appears before the initializer list. If any initializer throws, control jumps to the catch block. The constructor body acts as if it were inside an implicit try too, so exceptions from the body are also caught.

> 💡 **Tip:** In a function-try-block on a constructor, the catch block has one mandatory behavior: it **must** itself throw an exception (either by rethrow or by throwing a new one). It is *not* allowed to return normally. The reason: if any member initializer threw, the object was never constructed, and there is no sensible value for it to take on; returning normally would hand the caller a non-existent object.

Using the Constructor Exception

Here is how the client of MyClass catches the translated exception:

```
int main() {
    try {
       auto p2 = std::make_shared<C>(); // This itself throws, C() throws!
        // never reached: MyClass m{C{}, p2};
```

```
    }
    catch (const MyClass::MyClassExcept& e) {
        std::cerr << "MyClass exception caught: " << e.what() << '\n';
    }
    catch (const std::exception& e) {
        std::cerr << "Standard exception: " << e.what() << '\n';
    }
  catch (...) {
        std::cerr << "Unknown exception caught\n";
    }
    return 0;
}
```

Output:

```
Standard exception: C constructor exception
```

In this example, the very first line throws (because C() itself throws), so we never get to construct MyClass. The handler order matters: most-derived first (MyClassExcept), then base (std::exception), then catch-all. Because the failure is a runtime_error (a std::exception, but not a MyClassExcept), the second handler runs.

Exception Safety Guarantees

A function is exception-safe if it leaves the program in a *valid* state when it terminates by throwing. Validity is a spectrum, and the C++ standard library defines three levels, strict guarantees that library writers (including you, when you write reusable code) make to their callers.

The Three Guarantees

Basic guarantee. If the function throws, the basic invariants of every object touched by the call remain intact, and no resources have leaked. The exact state of the object may have changed, but it is still a usable object: you can assign to it or destroy it without crashing. This is the minimum every well-written function should provide. The standard library guarantees this for all of its operations.

Strong guarantee. The basic guarantee plus a stronger commitment: if the function throws, the program's state is exactly what it was before the call. Either the operation succeeds completely or it has no effect at all, a true transactional rollback. The standard library guarantees this for specific operations such as `vector::push_back`, `list::insert` for a single element, and uninitialized_copy.

Nothrow guarantee. The basic guarantee plus a promise never to throw at all. The function will complete its work without exception. The standard library guarantees this for operations that must be reliable, swap of two containers, `pop_back` on a non-empty container, and the destructors of standard library types.

Conditions for the Guarantees

The basic and strong guarantees are conditional. The standard library can only honour them if your user-supplied operations behave themselves:

- User-supplied operations (copy assignment, swap, comparisons) do not leave container elements in invalid states.
- User-supplied operations do not leak resources.
- Destructors do not throw exceptions.

These conditions are exactly the things that smart pointers, RAII, and noexcept destructors give you for free.

Resources Beyond Memory

It is easy to think of “leak” as meaning “memory leak,” but in C++ a resource is anything that must be acquired from the system and given back. The full list is much larger:

- Heap memory, new/delete, new[]/delete[], malloc/free.
- File handles, open/close, or ifstream constructor/destructor.
- Locks (mutexes, semaphores), acquire and release.
- Network connections, connect and disconnect.
- Database transactions, begin and commit/rollback.
- Graphics resources, textures, buffers, OpenGL contexts.
- Operating-system handles, process handles, thread handles.

Any of these can leak if an exception fires between the acquire and the release. We need a systematic way to make every kind of resource exception-safe, not a separate pattern for each type. That mechanism is RAII, and the tool that implements RAII for the most common case (heap memory) is the smart pointer.

The Resource-Leak Problem

To see why we need a systematic solution, look at what goes wrong without one. Here is the simplest possible example using a raw pointer:

```
if (someCondition) {
    MyClass* rc = new MyClass();
    // ... do stuff ...
    delete rc;
}
```

Under ideal circumstances, control flows linearly from top to bottom, the memory allocated by new is released by delete. But "ideal circumstances" is exactly what we cannot count on in real programs. Three things can prevent the delete from running:

Exceptions. If "do stuff" throws, the delete line is skipped. The MyClass object remains on the heap, with no pointer left to it. Memory leak.

Early returns. Maintenance programming is often "add a quick return here." If someone adds an early return between the new and the delete, the leak is silent and easy to miss.

Other control transfers. break, continue, goto, same story. Any code path that skips over the delete leaks the resource.

This is not a theoretical problem. In real programs, the gap between new and delete may span hundreds of lines, dozens of functions, and many possible exit paths. Manual tracking is hopeless. We need the language itself to guarantee cleanup.

The Stack-Object Guarantee

Notice that the leak problem is specific to *heap* allocation. If we had written:

```
if (someCondition) {
    MyClass rc;          // on the stack, not heap
    // ... do stuff ...
}
```

There would be no leak. C++ *guarantees* that when a stack-allocated object goes out of scope, for any reason whatsoever, including an exception passing through, its destructor runs. The block end, an early return, a throw, a break, all destroy the object cleanly. This is the only cleanup guarantee C++ gives you for free.

The key insight that motivates smart pointers and RAII is to **piggyback on the stack-object guarantee.** If we can package every resource acquisition inside a class whose destructor releases the resource, then we just declare a stack-allocated instance of that class wherever we need the resource, and the

language takes care of cleanup automatically, no delete, no close, no unlock, no matter how the scope is exited.

For heap memory, this packaging is what smart pointers do. For other resources, the same idea is called RAII (Resource Acquisition Is Initialization). We will study each in turn.

Smart Pointers

Raw pointers in C++ are essentially addresses with type checking. They are powerful but unforgiving: it is legal, and astonishingly common, to access an object that has already been deleted, to delete the same object twice, or to delete an object that another pointer is still using. Each of these is undefined behaviour, which in practice means the program will do something unpredictable, possibly only on Tuesdays in February.

A *smart pointer* is a small class that wraps a raw pointer and adds enough machinery to handle deletion automatically. The C++ standard library provides four smart pointer types:

- `std::unique_ptr<T>`, exclusive ownership; one owner at a time, no copying.
- `std::shared_ptr<T>`, shared ownership with reference counting; many owners, last one out destroys.
- `std::weak_ptr<T>`, non-owning observer of a shared_ptr; sees the object without keeping it alive.
- `std::auto_ptr<T>`, the original, deprecated; removed in C++17. We mention it only as a warning.

All of them live in `<memory>`. All of them automate not just deletion but also ownership semantics, making it impossible to write code that, for example, allows two owners to free the same object.

How a Smart Pointer Works

Underneath, a smart pointer is a templated class with an internal raw pointer. It overloads the dereference operators (* and ->), so it can be used with the same syntax as a raw pointer. The class is designed so that instances are normally allocated on the **stack**, while the pointee they manage is on the **heap**. When the stack-based smart pointer goes out of scope, its destructor runs (the stack-object guarantee), and the destructor takes care of deleting the heap object.

```
{
    std::unique_ptr<MyClass> rc = std::make_unique<MyClass>();
    // ... do stuff ...
}   // <-- rc goes out of scope here.
    //     Its destructor runs and deletes the heap-allocated MyClass.
    //     This happens whether we exited normally, via return, or via
exception.
```

Compare this with the raw-pointer version from the previous section. We have eliminated the delete call entirely, and in doing so, we have made the code exception-safe for free. Any path out of the block, normal end, return, throw, calls the destructor of rc, which deletes the underlying MyClass.

std::unique_ptr

A `unique_ptr` enforces **exclusive ownership**: at any moment, exactly one `unique_ptr` can point to a given object. Its four defining properties are:

- **Sole owner.** Only one `unique_ptr` can point to the object at any time.
- **Non-copyable.** The copy constructor and copy assignment are `= deleted`, the compiler will refuse to compile any attempt to copy a `unique_ptr`. If two unique_ptrs pointed to the same thing, they would both try to delete it.
- **Movable.** Move semantics transfer ownership. `std::move(up1)` gives up ownership from up1 (which becomes null) and hands it to a new unique_ptr.
- **Automatic destruction.** When the `unique_ptr` is destroyed, it deletes the managed object.

Two forms exist: a simple `unique_ptr<T>` that uses the default delete operator, and a parameterized `unique_ptr<T, D>` that lets you supply a custom deleter (useful for resources released by something other than delete, say, a C library that has its own `foo_release()` function).

```
#include <memory>

template <class X>
class unique_ptr {
public:
    using element_type = X;
    explicit unique_ptr(X* p = nullptr) noexcept;
    ~unique_ptr() noexcept;

    unique_ptr(const unique_ptr&)            = delete;   // no copy
    unique_ptr& operator=(const unique_ptr&) = delete;

    unique_ptr(unique_ptr&& other) noexcept;             // move only
    unique_ptr& operator=(unique_ptr&& other) noexcept;
    unique_ptr& operator=(nullptr_t)         noexcept;

    X& operator*()  const noexcept;
    X* operator->() const noexcept;

    [[nodiscard]] X* get()      const noexcept;
    [[nodiscard]] X* release()        noexcept;
    void             reset(X* p = nullptr) noexcept;
    explicit         operator bool() const noexcept;
    void             swap(unique_ptr& other) noexcept;
};
```

A few of the methods are worth highlighting:

• `get()` returns the raw pointer without giving up ownership. Useful when you must pass the raw address to a legacy API. The `[[nodiscard]]` attribute warns you if you call `get()` and ignore the result.

• `release()` returns the raw pointer *and* relinquishes ownership, the `unique_ptr` becomes null and will no longer delete the object. You now own the raw pointer and are responsible for cleanup.

• `reset(p)` deletes the currently-owned object (if any) and takes ownership of p. With no argument, it just resets to null.

• operator `bool()` lets you write if `(up)` … to test whether the pointer is non-null. The explicit keyword prevents accidental conversions like `int n = up;`.

Transferring ownership

Because `unique_ptr` is non-copyable, you cannot pass it to a function by value the way you would with a raw pointer. You must explicitly say “move”:

```
#include <memory>
#include <utility>     // for std::move

template <typename T>
std::unique_ptr<T> T_Factory() {
    return std::make_unique<T>();    // returning a unique_ptr is fine,
implicit move
}

template <typename T>
void Sink(std::unique_ptr<T> pt) {
    // Ownership has been transferred; pt now owns the object.
    // It will be destroyed when this function returns.
    pt.reset();   // optional: release early
}

int main() {
    auto pt = T_Factory<int>();      // pt is unique_ptr<int>
    Sink(std::move(pt));             // explicit move; pt is now nullptr
    // pt has no object to delete; destruction of pt does nothing.
    return 0;
}
```

💡 **Tip: Always use std::make_unique.** Prefer std::make_unique<T>(args...) over the older std::unique_ptr<T>(new T(args...)). It is exception-safe (no chance of leaking if the surrounding

expression throws), more concise, and arguably more readable. The only reason to ever write new by hand is if you need a custom deleter.

Why auto_ptr was removed

Before C++11, the only smart pointer was `std::auto_ptr`. It tried to solve the same problem as `unique_ptr` but used copy semantics instead of move semantics; a copy would transfer ownership, leaving the source null. This led to startling code:

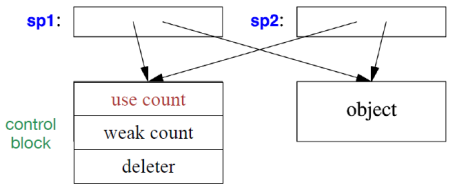
```
std::auto_ptr<int> a(new int(42));
std::auto_ptr<int> b = a;     // copies a, and silently sets a to nullptr!
// a is now empty. Did you know that?
```

Even worse, `auto_ptr`s could not be stored in standard containers, because containers assume that copying produces an equivalent copy. `auto_ptr` was deprecated in C++11, removed in C++17, and you should never use it.

std::shared_ptr

A `shared_ptr` supports **shared ownership**: multiple `shared_ptr`s can point to the same object, and the object lives until the last one is destroyed.

The mechanism is reference counting. Alongside the managed object, `shared_ptr` maintains a small *control block* with two counters and a deleter:



Every `shared_ptr` constructor that takes a raw pointer creates a new control block with use count 1. Every copy constructor and copy assignment increments the use count. Every destructor and assignment-away decrements it. When the count reaches zero, the deleter destroys the object. The weak count tracks `weak_ptr`s; we'll see those next.

In code:

```
auto sp1 = std::make_shared<MyClass>();    // use count == 1
{
    auto sp2 = sp1;                        // use count == 2
    // ... use sp2 ...
}                                          // sp2 destroyed; use count == 1
// ... sp1 still owns the object ...
sp1.reset();                               // use count == 0; object
destroyed
```

Performance note. Shared ownership is more expensive than unique ownership. Every copy and destruction must atomically modify the reference count (because `shared_ptr` is thread-safe with respect to the count), and the control block itself is heap-allocated. Prefer `unique_ptr` when one owner is enough; reach for `shared_ptr` only when sharing is genuinely needed.

> 💡 **Tip:** Use std::make_shared<T>(args...) instead of std::shared_ptr<T>(new T(args...)). It performs a single allocation for both the object and the control block (instead of two), so it is faster and uses less memory in addition to being exception-safe.

std::weak_ptr

A `weak_ptr` is a non-owning, observer-only smart pointer. It holds a reference to an object managed by a `shared_ptr`, but it does not increase the use count or keep the object alive on its own.

You use a `weak_ptr` when you want to observe an object without owning it, a cache that does not want to keep entries alive forever, a back-pointer from a child to a parent, or a notification system where listeners should not extend the publisher's lifetime.

To actually use the object, you ask the `weak_ptr` for a temporary `shared_ptr` via its `lock()` method. If the object is still alive (use `count > 0`), you get a `shared_ptr`; if not, you get an empty one.

The circular-reference problem

The most important application of `weak_ptr` is breaking cycles. Reference counting cannot break a cycle: if A holds a `shared_ptr` to B and B holds a `shared_ptr` to A, then even when no external pointer references either of them, the use counts of both stay at 1, and neither is ever destroyed. A memory leak, despite using smart pointers.

Here is the classic example:

```
class Parent;     // forward declaration

class Child {
public:
    std::shared_ptr<Parent> parent;
    ~Child() { std::cout << "Child destroyed\n"; }
};

class Parent {
public:
  std::shared_ptr<Child> child;
    ~Parent() { std::cout << "Parent destroyed\n"; }
};

int main() {
    {
        auto parent = std::make_shared<Parent>();
        auto child  = std::make_shared<Child>();
        parent->child  = child;
        child->parent  = parent;
    }
    // Neither Parent nor Child will be destroyed due to circular
reference.
    return 0;
}
```

Output: *nothing*, no destructor messages. Both objects leak.

The fix: make one direction weak

To break a cycle, replace one of the shared_ptrs with a weak_ptr. The conventional choice is to make the back-edge weak: parents own children, children merely observe their parents.

```
class Parent;

class Child {
public:
    std::weak_ptr<Parent> parent;     // <-- weak_ptr, not shared_ptr
    ~Child()  { std::cout << "Child destroyed\n"; }
};

class Parent {
public:
    std::shared_ptr<Child> child;
    ~Parent() { std::cout << "Parent destroyed\n"; }
};

int main() {
    {
```

```
        auto parent = std::make_shared<Parent>();
        auto child  = std::make_shared<Child>();
        parent->child  = child;
        child->parent  = parent;        // weak assignment, does not bump
use count
    }
    // Both Parent and Child are destroyed correctly.
    return 0;
}
```

In this code, **only shared_ptr contributes to the reference (use) count**, while weak_ptr does not affect ownership. When parent is created with make_shared, its use count is **1**, and similarly child starts with a use count of **1**. After parent->child = child;, the shared_ptr<Child> stored inside parent increases the child's use count to **2** (one from main, one from parent). When child->parent = parent;, a **weak_ptr** is assigned, which **does not increment** the parent's use count, so the parent remains at **1**. When the scope ends, the shared_ptr<Parent> in main is destroyed, reducing the parent's count to **0**, so the Parent object is destroyed; during its destruction, its shared_ptr<Child> member is released, decreasing the child's count from **2 to 1**, and finally when the shared_ptr<Child> in main is destroyed, the child's count becomes **0**, leading to its destruction. Thus, using weak_ptr prevents a circular ownership cycle, allowing both objects to be properly cleaned up.

❓ **Question:** How do I choose between unique_ptr, shared_ptr, and weak_ptr?

Default to unique_ptr. It is the cheapest, simplest, and most expressive of intent. Use shared_ptr only when ownership is genuinely shared between unrelated parts of the program, and prefer to design the program so that shared ownership is rare. Use weak_ptr when you have a shared_ptr world and need a non-owning observer, especially to break cycles.

The RAII Idiom

Smart pointers are a special case of a much broader pattern. **RAII**, Resource Acquisition Is Initialization, equates the management of *any* resource with the lifetime of an object. The pattern is simple and devastatingly effective:

- Acquire the resource inside an object's **constructor**.
- Release the resource inside the object's **destructor**.

Once a resource is wrapped this way, you manage it by manipulating *object lifetimes* rather than by writing explicit acquire/release calls. Declare a stack-allocated instance of the wrapper at the point

where you need the resource; the object's constructor acquires it, and the destructor, which is guaranteed to run whenever the scope is exited, for any reason, releases it.

Why It Works

In C++, the only code **guaranteed** to run after an exception is thrown is the destructors of stack-allocated objects between the throw point and the matching handler. Normal cleanup code, delete calls, close() calls, unlock() calls, sit in normal statements that are simply skipped when an exception passes through. RAII anchors the cleanup to the only thing the language promises to do during unwinding.

So the strategy is: never hold a raw resource directly in a variable, function parameter, or class member. Always wrap it in an object whose destructor releases it. Then exception safety is automatic; there is nothing to forget because there is no acquire/release pair for a maintenance programmer to break.

A File-Handling Example

To see the pattern outside of memory management, here is a tiny RAII wrapper around a file:

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

class File {
public:
    // Constructor: acquires the resource (opens the file)
    File(const std::string& filename) : file_(filename) {
        if (!file_.is_open()) {
            throw std::runtime_error("Failed to open file");
        }
        std::cout << "File opened: " << filename << '\n';
    }

    // Destructor: releases the resource (closes the file)
    ~File() {
        if (file_.is_open()) {
            file_.close();
            std::cout << "File closed\n";
        }
    }

    void write(const std::string& data) {
        if (file_.is_open()) {
            file_ << data << std::endl;
        } else {
            throw std::runtime_error("File is not open");
        }
    }

    // Prevent copying, two File objects must not manage the same handle
```

```
    File(const File&)            = delete;
    File& operator=(const File&) = delete;

private:
    std::ofstream file_;
};
```

The File constructor opens the file and throws an exception if the open fails. The destructor closes it. The copy operations are deleted because two File instances must not own the same handle. (If you *do* want shared ownership of a handle, wrap a shared resource in a shared_ptr instead.)

Usage:

```
int main() {
    try {
        File myFile("example.txt");
        myFile.write("Hello, RAII!");
        // No need to manually close the file, RAII handles it
    }
    catch (const std::runtime_error& e) {
        std::cerr << "Error: " << e.what() << '\n';
    }
    return 0;
}
```

Whether the write succeeds, fails with an exception, or hits the end of the block, myFile's destructor fires and the file is closed. There is no leak path. The close() call is *not in the source code*, and that is precisely what makes it impossible to forget.

The Resources Worth Wrapping

RAII applies to far more than memory and files. Any acquire-release pair in your program is a candidate:

- `std::lock_guard` and `std::unique_lock`, RAII mutex locks. The constructor locks, the destructor unlocks.
- `std::scoped_lock`, RAII multi-mutex locks with deadlock avoidance.
- `std::ifstream`, `std::ofstream`, `std::fstream`, RAII file handles.
- `std::unique_ptr`, `std::shared_ptr`, RAII heap memory.
- `std::thread` with `std::jthread` (C++20), RAII threads.

And in your own code, every place where you would otherwise write a paired acquire/release should become a small RAII class.

> 💡 **Tip: RAII rule of thumb.** If you find yourself writing acquire(...) at the top of a function and release(...) at the bottom, stop and write a small RAII class instead. It will be more lines once, but every future user of the resource gets exception safety for free.

A Complete Example: Rational with pImpl and unique_ptr

To see assertions, exceptions, smart pointers, and RAII working together, here is a fully revised Rational ADT. It is immutable, employs the pImpl idiom from Chapter 4 to hide its implementation, uses `unique_ptr` to manage the implementation safely, and throws on division by zero.

```
//****************************************************
// Rational ADT, immutable, pImpl, unique_ptr, exceptions
//****************************************************
#include <memory>
#include <iostream>
using namespace std;

class Rational {
public:
    Rational(int numer = 0, int denom = 1);   // user-provided values
    // ~Rational();                            // not needed: unique_ptr
                                               //   destroys Impl for us

    Rational(const Rational&);
    Rational& operator=(const Rational&);

    int numerator()   const;
    int denominator() const;

private:
    struct Impl;
    unique_ptr<Impl> rat_;
    void reduce();
};
```

The header declares `Impl` as a forward declaration; clients of Rational never see its definition. The `unique_ptr<Impl>` member manages the heap-allocated implementation. Because `unique_ptr` has its own destructor, we do not need to write one for Rational.

The implementation file defines `Impl` and the constructors:

```
struct Rational::Impl {
    Impl(int num = 0, int denom = 1) : numerator_(num),
denominator_(denom) {}
    int numerator_;
    int denominator_;
};

namespace {     // anonymous namespace: visible only in this file
    int gcd(int small, int large) {
        if (small == 0) return 1;
        int rem = large % small;
        while (rem != 0) {
            large = small;
            small = rem;
            rem   = large % small;
        }
        return small;
    }
}

Rational::Rational(int numer, int denom)
    : rat_(make_unique<Impl>(numer, denom)) {
    if (denom == 0) {
     throw "Panic! Denominator = 0"; //(for illustration; prefer a class!)
    }
    reduce();
}

Rational::Rational(const Rational& r)
    : rat_(make_unique<Impl>(*r.rat_)) {}

Rational& Rational::operator=(const Rational& r) {
    *rat_ = *r.rat_;
    return *this;
}
```

The constructor uses `make_unique` to allocate the `Impl`, which gives us exception-safe initialization automatically: if anything else in the initializer list were to throw, the `Impl` allocated by `make_unique` would still be cleaned up because unique_ptr's destructor runs as a subobject during unwinding.

> ⚠ **Warning:** This example throws a const char* for illustration. In real code you would throw a proper exception class, likely a nested Rational::DivideByZeroException derived from std::runtime_error. String-literal exceptions cannot carry information and don't fit the standard hierarchy.

A short driver shows the class in use:

```
int main() {
    auto ar(make_unique<Rational>());
    auto as(make_unique<Rational>());

    cout << "Enter rational number (a/b): ";  cin >> *ar;
    cout << "Enter rational number (a/b): ";  cin >> *as;

    cout << *ar << " + " << *as << " = " << *ar + *as << endl;
    cout << *ar << " * " << *as << " = " << *ar * *as << endl;

    // Test if a unique_ptr is bound:
    if (ar) cout << "ar is bound to " << *ar << endl;
    else    cout << "ar is unbound"    << endl;

    // Transfer ownership: ar becomes unbound
    unique_ptr<Rational> at(move(ar));
    if (ar) cout << "ar is bound to " << *ar << endl;
    else    cout << "ar is unbound"    << endl;
    if (at) cout << "at is bound to " << *at << endl;
    else    cout << "at is unbound"    << endl;

// Reassign: ar gets a fresh Rational; whatever it pointed to is destroyed
    ar = make_unique<Rational>(100);
    if (ar) cout << "ar is bound to " << *ar << endl;
    return 0;
}
```

Sample output:

```
Enter rational number (a/b): 1/2
Enter rational number (a/b): 3/4
Numerator of ar is 1, denominator of ar is 2
1/2 + 3/4 = 5/4
1/2 * 3/4 = 3/8
ar is bound to 1/2
as is bound to 3/4
ar is unbound
at is bound to 1/2
ar is bound to 3/4
as is unbound
ar is bound to 100/1
```

Notice three things from the output. After `move(ar)`, `ar` prints “unbound”, the ownership was transferred. After `ar = move(as)`, `ar` now points to the rational that as used to own; as is unbound; and the previous referent of `ar` (originally `at`'s after the first move) was destroyed first. Finally, `ar =`

`make_unique<Rational>(100)` makes `ar` point to a fresh `Rational(100)`; the previous referent of `ar` is destroyed.

noexcept and terminate()

Some functions cannot fail. A swap of two integers, the destructor of an int wrapper, std::move of a pointer, these operations simply have no way to throw, and the rest of the program often depends on that. C++ lets you say so with the noexcept specifier.

```
class Rational {
public:
    class DivideByZeroException {};
    Rational(int, int);
    ~Rational() noexcept;
    int numer() const noexcept;
    int denom() const noexcept;
    // ...
};
```

Two benefits, one obligation. Marking a function noexcept benefits *both* sides of the call:

- **The programmer** does not have to write a try block around the call. There is no exception to handle.
- **The compiler** can generate better code. Normally, when a function call might throw, the compiler must keep the surrounding stack in a state that supports unwinding, preserving destruction order, and keeping local objects discoverable. noexcept relieves it of those obligations, often producing measurably faster code at the call site.

The obligation: if a noexcept function does throw, including via any function it calls, the runtime immediately invokes std::terminate(). There is no second chance. noexcept is a guarantee enforced by program death.

When to Mark Functions noexcept

Some functions *must* be noexcept for the language to work correctly:

- **Destructors.** If a destructor throws during stack unwinding, the program terminates. Modern C++ destructors are implicitly noexcept; you should not write a destructor that throws.
- **Move operations.** Standard library containers will only move-construct elements (rather than copy them) when the move is noexcept. A throwing move would leave the container in a hopeless half-moved state. Always mark your move constructors and move assignments noexcept if they truly cannot throw.

- **swap.** Standard idioms depend on the ability to swap without throwing.
- **Memory deallocation.** Anything in a deallocator chain must not throw.

Other functions *should* be noexcept when they genuinely cannot throw, simply because it documents the fact and helps the compiler. Failing to mark a never-throwing function noexcept is, in Scott Meyers's phrase, “poor interface specification.”

set_terminate and terminate()

`std::terminate()` is the runtime function called when an exception cannot be handled, either because there is no matching catch, because a destructor threw during unwinding, or because a `noexcept` function did throw. By default, it calls `std::abort()` and the program dies.

You can install your own terminate handler with `std::set_terminate`. A custom handler might log a diagnostic, flush important files, send an alert, and *then* call `std::abort`. The handler is not allowed to return normally, terminate is final.

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

void myTerminate() {
    std::cerr << "Terminating due to unhandled exception!\n";
    std::abort();
}

void myFunction() noexcept {       // promises not to throw...
    throw 'a';           // ...but it does, std::terminate() will fire
}

int main() {
    std::set_terminate(myTerminate);
    std::cout << "=== noexcept function ===\n";
    try {
        myFunction();              // calls std::terminate via myTerminate
    } catch (...) {
        std::cerr << "This won't print\n";   // unreachable
    }
    return 0;
}
```

Output:

```
=== noexcept function ===
Terminating due to unhandled exception!
```

Note that the outer catch (...) is irrelevant, because myFunction is noexcept, the exception never has a chance to propagate; std::terminate runs immediately at the throw point.

Without noexcept, Same Code Works Fine

Drop the noexcept, and the same exception propagates normally:

```
void myFunction() { // no noexcept
    throw 'a';
}

int main() {
    try {
        myFunction();
    } catch(int e) {std::cerr << "Caught int exception: "  << e << '\n';}
      catch(char c){ std::cerr << "Caught char exception: " << c << '\n';}
      catch(...)   { std::cerr << "Caught unknown exception\n"; }
    return 0;
}
```

Output:

```
Caught char exception: a
```

Tricky: Older C++ had “exception specifications” like void f() throw(std::runtime_error); that listed which exception types a function might throw. These were deprecated in C++11 and removed in C++17. They could not be checked statically, were expensive at runtime, and led to surprising std::unexpected calls. The modern replacement is the binary noexcept: either you don't throw at all, or you might throw anything, that's the only useful distinction in practice.

Exceptions vs. Assertions

We have now seen both mechanisms in detail; the question becomes when to use which. The answer hinges on what kind of error you are reporting.

Use assertions for programming errors. A precondition violation, a broken invariant, an impossible state, these are bugs in your own code. You want the program to halt *on the exact line* where the bug

manifested, with the stack and all local variables intact, so a debugger can show you the cause. Stack unwinding would destroy precisely the state you need to investigate.

Use exceptions for environmental errors. A missing file, an out-of-memory condition, a user typing nonsense, a network drop. These are not your fault; the right response is to unwind cleanly to a layer that can recover, releasing resources along the way. Aborting the program would be needlessly destructive.

> ⚠ **Warning:** Do **not** use exceptions to report programming errors. Throwing an exception unwinds the program stack and destructs local variables, which is exactly the wrong thing if you're trying to find the cause of a bug. By the time the exception is caught, the state of the program at the moment of failure is gone.

The Two Failure Modes Compared

Aspect	Assertions	Exceptions
For	Programming errors	Environmental errors
When detectable	Could be prevented	Could not be prevented
Response	Abort immediately	Unwind to handler
Stack state at failure	Intact (for debug)	Unwound
Production behavior	Often compiled out	Always active
Recoverable	No	Usually yes

There is a real choice here: you can present the same failure as either kind. vector::at(i) throws out_of_range if i is bad; vector::operator[](i) simply asserts and proceeds with undefined behavior. Both are defensible. The thrown version is right when the index might come from outside the program (a user, a file); the asserting version is right when the index is computed by code that is supposed to know better.

> ❓ **Question:** If a client passes me a bad argument, isn't that an environmental error from *my* point of view? I didn't cause it.

> It is environmental in the sense that you did not cause it, but it is still a *programming* error, somebody's program is buggy, just not yours. The difference matters because the bug needs to be fixed. An assertion in your function makes the bug crash loudly at the call site, where the client can see and fix it. An exception would unwind the stack and the client might catch it and proceed without understanding that their code is broken. Use assertions for precondition violations; reserve exceptions for failures that no amount of correct client code could have avoided.

Best Practices

Putting it all together, here is the set of guidelines that experienced C++ programmers follow. None are absolute, but together they form the default approach you should depart from only with reason.

1. Signal all errors by throwing exceptions

Once you have decided that a failure is an environmental error worth reporting, throw an exception. Do not invent new return codes or status flags. Throwing is uniform, cannot be ignored, and uses the language's cleanup machinery.

2. Check for errors only when they first become detectable

Don't add defensive checks at every layer; check once at the boundary where the information is fresh. A function that wraps a system call should check that call's return value once and throw on failure. The layers above it then need no if (err) cascades.

3. Use RAII for all resource management

Every resource, memory, file, lock, connection, should be owned by an object whose destructor releases it. Once this discipline is in place, you can throw exceptions freely without worrying about cleanup, and you can catch them only when you actually want to handle them.

4. Use assertions for logic errors, exceptions for environmental errors

This is the diagnostic principle restated. Logic errors freeze the program with the bug visible; environmental errors unwind cleanly so the caller can react.

5. Only throw exceptions that callers can reasonably catch

Throwing an exception is making a promise: somewhere up the call stack, someone might catch this and do something useful about it. If no one will ever catch your exception (because it indicates a condition that nobody can possibly recover from), you should not be throwing, that is what assertions and std::terminate are for.

Specifically, throw in constructors and mutators. A constructor throws if it cannot establish the class invariants, there is nowhere else for the failure to go. A mutator throws if asked to put the object into an

invalid state, but should restore the object to its pre-call state first (*strong guarantee*), so the caller can recover.

6. Never throw from places that cannot recover

Some functions are so deeply embedded in the language's cleanup machinery that a thrown exception cannot be handled sensibly. Never throw from:

- **Destructors.** A throwing destructor during stack unwinding terminates the program. There is no good way to handle the situation locally; even a try/catch inside the destructor cannot signal failure to the caller because the destructor's caller is the runtime, not your code.
- **Copy constructors and assignment operators.** These are called automatically by containers, by the standard library, and by all sorts of generic code that has no idea you might throw. A thrown exception from a copy operation often leaves the system in an inconsistent state that nobody knows how to clean up.
- **Comparison operators (**==, <, &&, …). They are conceptually pure functions of their operands. There is no environmental reason they could fail.
- **Accessors.** An accessor returns the current value of an attribute. Nothing can go wrong. If the attribute is missing or invalid, return a sentinel or use std::optional; let the client decide whether that's an error.

7. Use exception class hierarchies and catch by reference

Derive your exception types from std::exception or a suitable subclass. Catch by const reference so polymorphism works and no copy is made. Nest exception classes inside their owning ADT to make the relationship explicit.

8. Use noexcept, not exception specifications

Mark every function that cannot throw with noexcept. Do not use the old throw(…) specifier syntax, it was removed in C++17.

> 💡 **Tip: A complete recipe.** Throw std::runtime_error (or a derived type) by value, catch by const std::exception&, manage resources with smart pointers and RAII, mark destructors and movers noexcept, and reserve assertions for invariants and preconditions. With those defaults, most of your code will be exception-safe automatically and most of your bugs will manifest within sight of their cause.

Summary

Recognition: What to spot in code

- Where and when to use (or *not* use) exceptions.
- Motivation for the RAII idiom, exception-safe resource management.
- Exception-safety levels: basic, strong, nothrow.

Comprehension of how the machinery works

- How C++ exception handling delivers a thrown object to a matching handler.
- How stack unwinding destroys fully-constructed local objects.
- When to use exceptions versus assertions, and why.
- When to use unique_ptr versus shared_ptr versus weak_ptr.

Application: What to do with these tools

- Throw and catch exception objects, with proper handler ordering.
- Nest exception classes within ADT classes for clarity.
- Use smart pointers for heap-allocated ownership.
- Use RAII to make resource cleanup automatic.
- Mark non-throwing functions noexcept to enable optimization and clarify intent.

The big picture: assertions, exceptions, smart pointers, and RAII are not four independent topics. There are four parts of a single discipline for writing code that fails gracefully and is straightforward to debug when it does. Assertions catch your own mistakes early. Exceptions report environmental problems to handlers who can recover. Smart pointers and RAII guarantee that resources are released cleanly along the way. Used together, they turn error handling from a maintenance burden into something the language largely takes care of for you.

Practice Problems

1. For each of the following situations, decide whether to use an **assertion** or an **exception**. Justify each choice in one sentence.

(a) A Stack::pop() is called on an empty stack.

(b) std::ifstream fails to open a user-supplied filename.

(c) An integer Rational constructor receives a zero denominator from cin.

(d) An internal helper computes a negative number where the caller's contract said it should be non-negative.

(e) new fails because the system is out of memory.

2. Consider this code:

```
void process() {
   Trace a, b, c;          // construction order: a, b, c
   throw std::runtime_error("oops");
}
```

List, in order, the destructor calls that occur before the exception escapes process. What if the Trace constructor for b had thrown instead, what would the sequence be then?

3.

Design a Date ADT (with day, month, year) that throws appropriately on invalid input. Specifically:

(a) Define an InvalidDateException class nested inside Date, derived from std::runtime_error, that carries the offending day/month/year fields.

(b) Write a constructor that throws when the date is impossible (e.g., February 30, month 13, day 0). Override what() to return a useful message.

(c) Demonstrate the class with a main that reads a date from cin and prints either the date or the exception's message.

4. The following function leaks memory if processData throws:

```
void buggy(int n) {
   int* arr = new int[n];
   processData(arr, n);    // might throw
   delete[] arr;
}
```

Rewrite it in three different ways: (a) using a raw try/catch to clean up by hand, (b) using std::unique_ptr<int[]>, and (c) using std::vector<int>. For each, state the exception-safety guarantee it provides.

5. Build a **Lock** RAII wrapper around a std::mutex. The constructor locks the mutex; the destructor unlocks it. The class must not be copyable (why?) but should be movable. Demonstrate with a function that increments a shared counter exception-safely:

```
std::mutex m;
int counter = 0;
void increment() {
   Lock guard(m);
   ++counter;
   riskyOperation();   // may throw
}
```

Explain why the lock is released even when riskyOperation throws.

6. Predict the output of this program and explain *why* each line is or is not printed:

```
struct X {
   class A {}; class B : public A {}; class C : public B {};
   void f() { throw C(); }
};
int main() {
   X x;
   try { x.f(); }
   catch (const X::A&) { std::cout << "A\n"; }
   catch (const X::B&) { std::cout << "B\n"; }
   catch (const X::C&) { std::cout << "C\n"; }
   return 0;
}
```

Then explain what the rule is and how you would fix the handler order so that `C` is caught as `C`.

7. Identify the leak in this code and fix it using smart pointers:

```
class Node {
public:
   std::shared_ptr<Node> next;
   std::shared_ptr<Node> prev;
   int value;
};
// Used in a doubly-linked list:
auto a = std::make_shared<Node>();
auto b = std::make_shared<Node>();
a->next = b;  b->prev = a;
```

Which direction of the link should be weak_ptr and why? After your fix, trace through what happens when both a and b go out of scope.

8. ★ **Stretch.** Write a small RAII transaction class for a Database object that exposes begin(), commit(), and rollback() methods. The constructor calls begin; if the user calls commit(), the destructor does nothing; otherwise the destructor calls rollback(). This guarantees that an exception in the middle of a transaction automatically rolls back. Sketch the class and show a usage example that exercises both the success path and the throwing path.

9. ★ **Stretch.** The *strong exception safety guarantee* promises that a failing operation leaves the program state unchanged. Implement a strongly-exception-safe operator= for a class that owns a heap-allocated resource via std::unique_ptr. Use the *copy-and-swap* idiom (construct a copy, then swap members). Explain how this guarantees the strong property: under what conditions on the type T does the copy step throw, and why does the swap step never throw?

10. ★ **Stretch.** Design a SafeStack<T> template that wraps an internal std::vector<T> and provides push, pop, top, size, and empty. Decide for each operation:

(a) Whether it should throw or assert when called illegally.

(b) What exception safety guarantee it provides (basic, strong, nothrow).

(c) Whether it should be marked noexcept.

Defend your design choices in 2-3 sentences each. Consider how a T with a throwing copy constructor affects push's safety properties.