

UNIVERSITÉ
PARIS SCIENCES
& LETTRES

Dealing with Exceptional Situations

The Art of Computer Programming

Pierre Senellart



6 November 2026

Outline

- Motivation
- Without exceptions
- Exceptions
- In C++
- In Python
- Practice
- Coda

UNIVERSITÉ
PARIS
SCIENCES
& LETTRES

Motivation



Programs run in an imperfect world

- A file is missing, or unreadable; the input is not what the program expects; the network drops; the disk is full; the allocation fails (lecture 4); an index is out of range
- A useful program must **detect** such a situation, **report** it in a way its user can act on, and **recover** where recovery makes sense
- Poor error handling gives crashes, silently wrong results, corrupted data, and **security holes** – many vulnerabilities hide in error paths nobody tested
- **The difficulty**: the code that **detects** a problem is rarely the code that can **decide what to do** about it. A function reading a file cannot know whether a missing file should end the program, prompt the user, or be ignored

What counts as exceptional

- **Exceptional does not mean rare**: a missing file or an invalid input happens every day. It means **outside the normal course** of the function: the function cannot do what its name promises
- **Not exceptional**: a search that finds nothing, a queue that is empty, the end of a file. These are **expected outcomes**, and belong in the return value
- **Exceptional**: the arguments violate what the function requires, a resource is unavailable, an invariant of the program is broken
- **A bug of our own** is a third case: an impossible state, best detected by an **assertion** that stops the program where the mistake is, rather than reported to a user who can do nothing about it

Resources must be released

- A program acquires resources that must be given back, and that the language does not manage on its own:
  - **memory** on the heap (lecture 4), outside garbage-collected languages
  - **open files** (lecture 10): the operating system limits how many a process may hold
  - **network connections**, handles of every kind
- On the **normal** path, releasing them is easy to remember
- On an **error** path, it is easy to forget – and error paths are the least tested code in any program
- Two questions run through this lecture: **how is a problem reported** to the code that can handle it, and **how is everything released** on the way there

UNIVERSITÉ
PARIS
SCIENCES
& LETTRES

Without exceptions



Returning an error code

C

```c
long file_size(const char *filename) {
  FILE *f = fopen(filename, "rb");
  if (f == nullptr) return -1;          // a value that cannot be a size
  if (fseek(f, 0, SEEK_END) != 0) {
    fclose(f);                          // do not leak the open file
    return -1;
  }
  long size = ftell(f);
  fclose(f);
  return size;                          // >= 0: the answer
}
```

- The C way: a **special value** of the return type means failure – `-1`, `0`, `nullptr`, depending on the function
- **Drawbacks**: nothing forces the caller to look; the checks clutter every call; and the error value must not be a possible answer

Saying why: `errno`

C

```c
#include <errno.h>
#include <string.h>

FILE *f = fopen(filename, "rb");
if (f == nullptr)
  fprintf(stderr, "cannot open %s: %s\n", filename, strerror(errno));
// cannot open nosuchfile: No such file or directory
```

- A return code says **that** something failed, not **what**: the C library also sets the global variable `errno` to a standard error number, which `strerror` turns into a message
- **Drawbacks**: it is **global state** (lecture 4), overwritten by the next failing call, so it must be read at once; and it needs a catalog of numbers

Two ways of getting it wrong

ignoring the error

```
FILE *f = fopen("data.txt", "r");
fgets(line, sizeof line, f);
// if the file is missing, f is
// nullptr, and fgets crashes, far
// from the actual cause
```

crashing on purpose

```
double mean(double sum, int n) {
  if (n == 0) abort();   // dies
  return sum / n;
}
```

- **Ignoring** an error is a common mistake: it turns a clear failure into undefined behavior later, far from the cause
- **Failing fast** (`abort`, `assert`, `exit`) is the right answer for **a bug of our own**: it stops at the fault
- But it is the wrong answer for a situation the **user** caused: a program that dies on a mistyped filename is unusable, and one that dies mid-write corrupts its data

A structured result

C

```c
struct long_result {                // no integer can mean "failed" here
  bool ok;
  long value;
};

struct long_result r = to_long("12x");
if (r.ok) printf("%ld\n", r.value);
else      fprintf(stderr, "not a number\n");
```

- When **every** value of the type is a possible answer, the function returns a `struct` (lecture 5) holding a flag and the value – the caller cannot read the value without seeing the flag
- C++ has `std::optional<T>` for “a value or nothing”
- Still up to the caller to check, but now **visibly** so

UNIVERSITÉ
PARIS
SCIENCES
& LETTRES

Exceptions

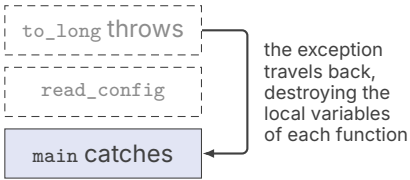


Exceptions

- An **exception** is an object, thrown at the point where a problem is **detected**, and caught at the point where it can be **handled** – with nothing to write in between
- Three constructs, in almost every language since the 1980s (C++, Java, Python, C#, JavaScript… but **not C**):

  `throw` / `raise` report a problem, abandoning the current computation
  `try` mark a region whose failures we are prepared to handle
  `catch` / `except` handle a problem of a given kind
- The normal logic stays readable; the error handling sits **apart**, where it belongs
- An exception carries **information** (a class, a message, any data we put in it) and cannot be ignored: uncaught, it **terminates the program** with a report

Propagation: unwinding the stack



- The exception leaves the function that threw it, then its caller, then its caller… until a `try` block prepared for **this kind** of exception is found: **stack unwinding** (call stack: lecture 2)
- Each function it leaves ends as if it had returned: its local variables are **destroyed**, and in C++ their **destructors run** (lecture 5)
- Functions in between need **no code at all**, which the return-code style cannot achieve

What exceptions achieve

- **Automatic propagation**: a problem travels to the code that can handle it, and the functions in between contain no error-handling code
- **Errors are noticed**: an uncaught exception stops the program with a report
- **Automatic cleanup**: destructors in C++ (RAII), `finally` and `with` in Python release resources on every path
- **Readable code**: the normal logic comes first, and the handlers are separate
- **Informative errors**: the exception object carries its own information
- **A single mechanism**: one way to report errors and one hierarchy of classes for the whole language

Exceptions do not decide **what to do** when a problem occurs: that remains up to the programmer

UNIVERSITÉ
PARIS
SCIENCES
& LETTRES

In C++



Throwing and catching

C++

```cpp
#include <stdexcept>
double mean(double sum, int n) {
  if (n == 0) throw std::invalid_argument("mean of no values");
  return sum / n;
}

try {
  std::cout << mean(10.0, 0);
} catch (const std::invalid_argument &e) {
  std::cout << "invalid argument: " << e.what() << "\n";
}
```

- `throw expr;` throws any value; by convention an object of a class derived from `std::exception`
- Catch **by const reference**: by value would copy the object and lose its class (lecture 5: slicing)

Several handlers

C++

```
try {
  std::vector<int> v(5);
  std::cout << v.at(10);                        // at() checks and throws
} catch (const std::out_of_range &e) {          // the specific class first
  std::cout << "out of range: " << e.what() << "\n";
} catch (const std::exception &e) {             // then the general one
  std::cout << "something else: " << e.what() << "\n";
} catch (...) {                                 // anything at all: rare
  throw;                                        // re-throw what we caught
}
```

- Handlers are tried **in order**, and the first one whose type matches wins: put the **specific** classes before the general ones

The standard hierarchy

Class (mostly in `<stdexcept>`)	**Meaning**
`std::exception`	the root; gives `what()`, a message
`  std::logic_error`	the program broke its own contract
`    std::invalid_argument`	an argument the function rejects
`    std::out_of_range`	an index or key outside the valid range
`  std::runtime_error`	a failure detected while running
`  std::bad_alloc`	`new` could not allocate (lecture 4)

- A class of our own derives from the closest one, so that a caller catching the base class catches it too:

C++

```
class ConversionError : public std::runtime_error {
public:
  explicit ConversionError(const std::string &s)
      : std::runtime_error("cannot convert: " + s) {}
};
```

RAII: the destructor does the cleanup

C++

```
class Buffer {
  int *data;
public:
  explicit Buffer(std::size_t n) : data(new int[n]()) {}    // acquires
  ~Buffer() { delete[] data; }                               // releases
  int &operator[](std::size_t i) { return data[i]; }
};

void store(const std::string &s) {
  Buffer b(1000);                    // acquires
  b[0] = std::stoi(s);               // may throw: we write no delete[]
}                                    // ~Buffer() runs, on both paths
```

Resource acquisition is initialization: the constructor acquires, the destructor releases, and unwinding runs destructors – so an object that owns a resource cannot leak it (lecture 5)

RAII in practice

- **The rule**: every resource is owned by an **object** whose destructor releases it. Then no `try` block is needed for cleanup – only for handling
- The standard library is built this way: `std::vector` and `std::string` own memory
- **Two rules that follow**:
  - a **constructor may throw**: the object never exists, and what it had already built is destroyed
  - a **destructor must not throw**: it runs **during** unwinding, and a second exception then terminates the program. Destructors are `noexcept` by default
- `noexcept` on a function promises it throws nothing; if it does, the program terminates

UNIVERSITÉ
PARIS
SCIENCES
& LETTRES

In Python



The four clauses of `try`

Python

```python
try:
    f = open(name)                    # what we are prepared to handle
except FileNotFoundError:             # one kind of failure
    print("no such file")
except (PermissionError, IsADirectoryError) as e:    # several at once
    print("cannot read it:", e)
else:
    print(f.readline()); f.close()    # only if nothing was raised
finally:
    print("done")                     # always, error or not, even on return
```

- `raise SomeError("message")` throws; `raise` alone, inside a handler, re-raises what was caught
- `else` is the code that belongs **after** the risky call, but should not be protected by the handlers
- `finally` is where cleanup goes: it runs on every path out of the block

The standard hierarchy, and catching well

```
BaseException
  KeyboardInterrupt, SystemExit
  Exception
    ValueError
    TypeError
    KeyError, IndexError
    ZeroDivisionError
    OSError
      FileNotFoundError
```

- Catch the **most specific** class that you can actually handle
- Never write a bare `except:` – it catches `KeyboardInterrupt` and `SystemExit` too, and hides your own bugs. `except Exception:` if you must
- Never write `except ...: pass` unless you can say, in a comment, why the failure is harmless
- A class of your own derives from `Exception`:

Python

```
class ConversionError(Exception):
    pass
```

Chaining: keeping the cause

Python

```python
def to_int(s):
    try:
        return int(s)
    except ValueError as e:
        raise ConversionError(f"cannot convert: {s}") from e  # keep the cause
```

- A library rarely lets a low-level exception escape as it is: it **translates** it into a class of its own, more meaningful to its users
- `raise ... from e` records the original in `__cause__`, and the traceback shows both: “the above exception was the direct cause of the following exception”
- Without `from`, a failure inside a handler is still reported with the first one (“during handling of the above exception, another exception occurred”)
- **Why bother**: the class tells the caller what to do, the cause tells the programmer what happened

`with`: Python's answer to RAII

Python

```python
with open("data.txt") as f:          # __enter__ acquires
    process(f)                       # may raise: nothing to clean up here
# the file is closed here, on both paths: __exit__ has run

class Timer:                         # a context manager of our own
    def __enter__(self):
        self.start = perf_counter(); return self
    def __exit__(self, kind, value, traceback):
        print(f"{perf_counter() - self.start:.4f} s")
        return False                 # False: let the exception propagate
```

- A **context manager** is an object with `__enter__` and `__exit__`; `with` calls the second one **whatever happens** in the block
- Why Python needs it: `__del__` runs at an unpredictable moment, if at all, so it cannot be trusted to release a resource – `with` can

UNIVERSITÉ
PARIS
SCIENCES
& LETTRES

Practice



Where to catch, and where not to

- **Catch where you can act**: retry, fall back on a default, ask the user again, skip this item and continue, or report and exit with a message. If none of these applies here, let the exception **propagate**
- **Do not use exceptions as ordinary control flow**: `d.get(k, 0)` rather than `try: d[k] except KeyError`, a test rather than a throw, for outcomes you **expect**
- **Throwing an exception is expensive**, in every language, compared to a normal return or a test: keep exceptions for exceptional situations

Habits worth having

- **Document** what a function throws, and when, exactly as you document what it returns
- Throw an exception **carrying** what is needed to understand it: the filename, the value, the index – not just “invalid input”
- Keep the hierarchy **small**: one class per kind of failure a caller might treat differently, and no more
- Do not let a resource live outside an owner: an object in C++, a `with` block in Python
- **Test the error paths**: give the program a missing file, a full disk, a wrong input. They are the least tested part of every program, and where the worst bugs live
- Distinguish **your bug** (assert, and fix it) from **their mistake** (report it, and carry on)

The three languages side by side

	C	C++	Python
Reporting	return code, `errno`	`throw`	`raise`
Handling	`if` at every call	`try`/`catch`	`try`/`except`
Propagation	by hand	automatic	automatic
Cleanup	by hand	destructors (RAII)	`finally`, `with`
Classes	none	`std::exception` tree	`Exception` tree
Ignored error	silent	program stops	program stops

- **Python** is built around exceptions: they are the normal way to report anything unusual
- **C++** has the richer system, and RAII, which is what makes exceptions safe there
- **C** has none, so the discipline is entirely the programmer's – which is why its error paths are worth reviewing twice

UNIVERSITÉ
PARIS
SCIENCES
& LETTRES

Coda

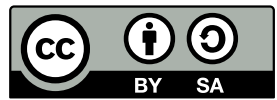


References and what follows

- **In the tutorials that follow**: turning a program that ignores its errors into one that reports them – return codes and `errno` in C, exceptions and RAII in C++, `try`/`except` and `with` in Python, and an exception class of your own
- **Next lecture**: sorting algorithms, then input and output (lecture 10), where files and their failures come back in detail
- References: Python Software Foundation [2026]; Stroustrup [2022], chapter 4 (error handling, invariants) and cppreference.com [2026]; King [2008], chapter 24 (error handling in the C library)

Licensing

This work is licensed under a Creative Commons “Attribution-ShareAlike 4.0 International” license.



Bibliography I

cppreference.com. Exceptions. https://en.cppreference.com/w/cpp/language/exceptions, 2026.

K. N. King. *C Programming: A Modern Approach*. W.W. Norton & Company, 2 edition, 2008. ISBN 0393979504. URL http://knking.com/books/c2/index.html.

Python Software Foundation. The python tutorial: Errors and exceptions. https://docs.python.org/3/tutorial/errors.html, 2026.

Bjarne Stroustrup. *A Tour of C++*. Addison-Wesley, 3 edition, 2022. ISBN 978-0-13-681648-5. URL https://www.stroustrup.com/tour3.html.