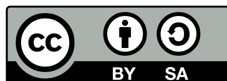
Dealing with Exceptional Situations

The Art of Computer Programming

Pierre Senellart



17 November 2025

Outline

Motivation

Traditional Approaches

Exceptions

Exceptions in C, C++, and Python

Advanced C++ Features

Summary

Why Error Handling?

- Programs operate in an imperfect world
- Inputs can be wrong, files may be missing, networks can fail
- Good programs must:
  - Detect problems
  - Report them clearly
  - Recover when possible
- Poor error handling leads to:
  - Crashes
  - Silent incorrect results
  - Corrupted data
  - Security vulnerabilities

What Is an Exceptional Situation?

- An unexpected or abnormal event during execution
- Examples:
  - File not found
  - Invalid user input
  - Out-of-range array access
  - Division by zero
- Exceptional does not mean rare!

Additional Motivation: Avoiding Resource Leaks

- A program handles some resources, which must be manually managed (allocated and freed):
  - memory on the heap (except in garbage-collected languages)
  - open files (the operating system limits the number of files that can be opened)
  - network connections (only a given number of network connections can be established at the same time)
  - locks (for parallel programming)
  - etc.
- When an exceptional event occurs, important not to forget to free allocated resources!

Outline

Motivation

Traditional Approaches

Exceptions

Exceptions in C, C++, and Python

Advanced C++ Features

Summary

Returning Error Codes

- A simple, common technique in C: a function returns a special value (NULL, -1, 0, etc.) indicating an error occurred
- Drawbacks: easy to ignore; error checks clutter the code

C

```c
#include <stdio.h>

int get_file_size(const char *filename)
{
  /* Open a file from disk */
  FILE *f = fopen(filename, "r");
  if (f == NULL) {
    return -1; // the file could not be opened
  }
  fseek(f, 0, SEEK_END);
  int size = ftell(f);
  fclose(f);
  return size;
}
```

Global Error Variables

- In C, some library calls (in addition to returning an error code) set the global variable errno that contains a standardized error number
- Drawbacks: global state, requires explicit checking, requires a catalog of error numbers, global variables poorly interact with parallel programming

C

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

int main(void) {
  // ...
  if (rename("a", "b") != 0) {
    printf("Could not rename file a to b. Error number: %d\n", errno);
  }
  // ...
}
```

Ignoring Errors

- A common mistake among beginners: ignoring errors
- Leads to undefined behavior (such as dereferencing a NULL pointer), crashes, and security flaws

C

```c
fopen("data.txt", "r"); /* ignoring return value;
                           what if data.txt does not exist? */
```

Crashing Intentionally

- Sometimes used to avoid data corruption: "fail fast"
- Useful during testing or for programs only used by their programmers, not for user-facing programs

C

```c
#include <stdlib.h>

double mean(double sum, int count)
{
  if(count == 0) {
    abort(); // immediate termination
  } else {
    return sum / count;
  }
}
```

Limitations of Traditional Approaches

- Traditional error-handling techniques work, but have important drawbacks

  Scattered checks: Every function call must be verified manually

  Ignored errors: Return values and `errno` are easy to forget

  No automatic cleanup: Resources (files, memory) may leak (remain allocated)

  Complex control flow: Many nested `if` statements reduce readability.

  Global state: `errno` is shared, fragile, and may not be parallel-friendly

  Inconsistent conventions: Different libraries use different error codes
- These issues motivate a more structured, automatic, and language-level mechanism for handling exceptional situations: exceptions

Outline

Motivation

Traditional Approaches

Exceptions

Exceptions in C, C++, and Python

Advanced C++ Features

Summary

Exceptions: A Cleaner Approach

- An exception is a runtime signal that something abnormal happened
- Separates normal logic from error-handling logic
- Used in Python and C++, and in most modern languages; absent in C

Exception Propagation

- When an exception is thrown, the runtime unwinds the call stack: we iteratively go back to the calling function, until (and if) the exception is caught
- If an exception is not caught by any of the functions in the call stack, the program terminates
- Local variables of the current function and every function we return from are destroyed once the exception causes to go back to the caller, as usual when leaving functions

How Exceptions Address Traditional Error Handling Limitations

- Exceptions provide structured, language-supported error handling

No need for scattered checks No need to check every function call manually; the error is automatically propagated

Errors are not ignored Exceptions cannot be silently forgotten: uncaught exceptions terminate the program

Automatic cleanup Resources are cleaned up automatically during stack unwinding (in Python, thanks to `finally` blocks, and in C++ thanks to RAII, see later)

Cleaner control flow Normal logic stays readable; error-handling code is separated in `except`/`catch` blocks

No fragile global state No need for shared variables like errno

Consistent mechanism A unified, language-level method for reporting and handling errors

Low-Level Origins of Exceptions

- Computers have long needed ways to react when something unexpected happens (e.g., a division by zero, an invalid memory access, or the user pushes a button)
- Hardware interrupts allow the processor to stop normal execution and jump to special code that handles specific hardware events
- In C, `setjmp`/`longjmp` (from `<setjmp.h>`) provide a way for programs to manually jump out of deeply nested function calls in case of an error (but hard to use by the programmer)
- Modern exceptions build on these ideas: the language itself provides a structured and safer way to interrupt execution and transfer control to an appropriate error handler

Outline

Motivation

Traditional Approaches

Exceptions

Exceptions in C, C++, and Python

Advanced C++ Features

Summary

C: No Exceptions

- C has no exception system
- Use return codes
- Less error-prone: Structured return types

C

```c
struct result {
  int ok;
  int value;
};
struct result parse(const char *s);

int main(void) {
  struct result r = parse("123456");
  if(r.ok) {
    // ...
  } else {
    // ...
  }
  return 0;
}
```

C++ Exceptions

- Introduced in C++98
- Use `throw`, `try`, `catch`

C++

```
try {
  std::vector<int> v(5);
  v.at(10); // throws std::out_of_range,
            // which derives from std::exception
} catch (const std::exception &e) {
  std::cout << e.what();
}
```

Common C++ Exception Types

- C++ uses a hierarchy of exception classes rooted in `std::exception`
- Most standard library exceptions inherit from this base class

`std::exception` The root of the standard exception hierarchy. Provides the virtual method `what()` for error messages

`std::runtime_error` General category for errors detected during program execution that cannot be classified more precisely

`std::logic_error` Errors in the program's logic (e.g., invalid arguments, contract violations)

`std::out_of_range` Access outside the valid range of a container

`std::invalid_argument` A function received an argument that does not satisfy its requirements

RAII: Resource Acquisition Is Initialization

Definition: RAII is a C++ programming idiom where resource ownership is tied to object lifetime

Principle: A resource is acquired in an object's constructor and automatically released in its destructor

Key Ideas:
- Resources (memory, opened files, locks, network connections. . .) are owned by objects
- Acquisition happens at initialization (constructor)
- Release happens automatically at destruction (end of scope)
- Guarantees exception-safety and prevents leaks

Benefit: No need to manually free resources: cleanup always happens, even on exceptions.

C++ RAII Example (1/2)

C++

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

class File {
  std::ifstream f;
public:
  File(const std::string& name) : f(name) {
    if(!f) throw std::runtime_error("Could not open file");
  }

  char readOneByte() {
    int c = f.get();
    if(c == EOF)
      throw std::runtime_error("No character to read");
    else
      return c;
  }
};
```

C++ RAII Example (2/2)

C++

```
int main() {
  File file("data.txt");
  std::cout << file.readOneByte() << std::endl;
  return 0;
}
```

- If the file cannot be opened, the constructor of `File` throws an exception – the partially constructed `File` object is cleaned up on leaving the constructor
- If a character cannot be read, the function `readOneByte` throws an exception – if not caught, the `File` is cleanly destroyed
- Once a `File` object is destroyed, its default destructor calls the default destructor of `std::fstream` on `f`, which closes the file

Python Exceptions

- Python's standard mechanism for handling errors.

Python

```python
try:
    f = open("data.txt")
except FileNotFoundError:
    print("File missing!")
except:
    print("Other kind of error!")
finally:
    print("Done")
```

Common Python Exception Types

- Python provides many built-in exception classes
- Most of them inherit from the general base class `Exception`

`Exception` The base class for almost all errors. Most user-defined exceptions should inherit from it

`RuntimeError` Raised when an error is detected that does not fall into a more specific category (e.g., internal inconsistencies, unexpected states)

`ValueError` A function received an argument of the right type but with an invalid value

`TypeError` An operation or function was applied to an object of inappropriate type

`FileNotFoundError` A file or directory operation failed because the target does not exist

Python: Context Managers

- Some functions can return a context manager, an object of a special class for handling resources
- Example: open standard function to open a file
- With such functions, with ensures cleanup even if exceptions occur

Python

```python
with open("data.txt") as f:
    process(f)
```

- Internally, object managers are objects of classes that define __enter__ and __exit__ methods, which should respectively acquire resources and free them
- Plays a similar role to RAII in Python, except that context managers are used in lieu of standard constructors and destructors (because Python does not guarantee when a destructor is called, but guarantees __exit__ methods are called on the exit of a with block)

Python: Multiple and Chained Exceptions

- One can catch exceptions of different types

Python
```python
try:
    ...
except (ValueError, TypeError) as e:
    ...
```

- One can rethrow a new exception that includes the original exception in its context (to preserve the original root cause of the exception)

Python
```python
try:
    ...
except Exception as e:
    raise RuntimeError("failed") from e
```

Defining Custom Exceptions

Python

```python
class MyError(Exception):
    pass
```

C++

```cpp
class MyError : public std::exception {
  const char* what() const noexcept override {
    return "MyError";
  }
};
```

Outline

Motivation

Traditional Approaches

Exceptions

Exceptions in C, C++, and Python

Advanced C++ Features

Summary

noexcept in C++

- Declares that a function will not throw any exceptions
- Helps optimization and reasoning
- Careful: If violated, program may terminate

C++

```
void f() noexcept {
  // must not throw
}
```

C++: Constructors and Destructors

- Constructors may throw: partially built objects cleaned up when this happens
- Destructors must not throw! (`noexcept` implicit on destructors)

Outline

Motivation

Traditional Approaches

Exceptions

Exceptions in C, C++, and Python

Advanced C++ Features

Summary

Exceptions: Things to Watch Out For

- Do not use exceptions for normal control flow
- Uncaught exceptions terminate the program
- Make sure to clean up resources (using `finally` or `with` in Python; using RAII in C++)
- When you document a function, document which exceptions may be thrown (see documentation of standard library)
- Keep exception hierarchies simple

Summary

- Many approaches exist for handling exceptional situations
- Exceptions offer structured, clean error handling
- Python: built around exceptions
- C++: rich system with RAII and `noexcept`
- C: no exceptions, use structured patterns manually

Licensing

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