Effizientes Programmieren in C, C++ und Rust

Exceptional C++

Colin Bretl, Nikolai Maas, Peter Maucher | 24.10.2024

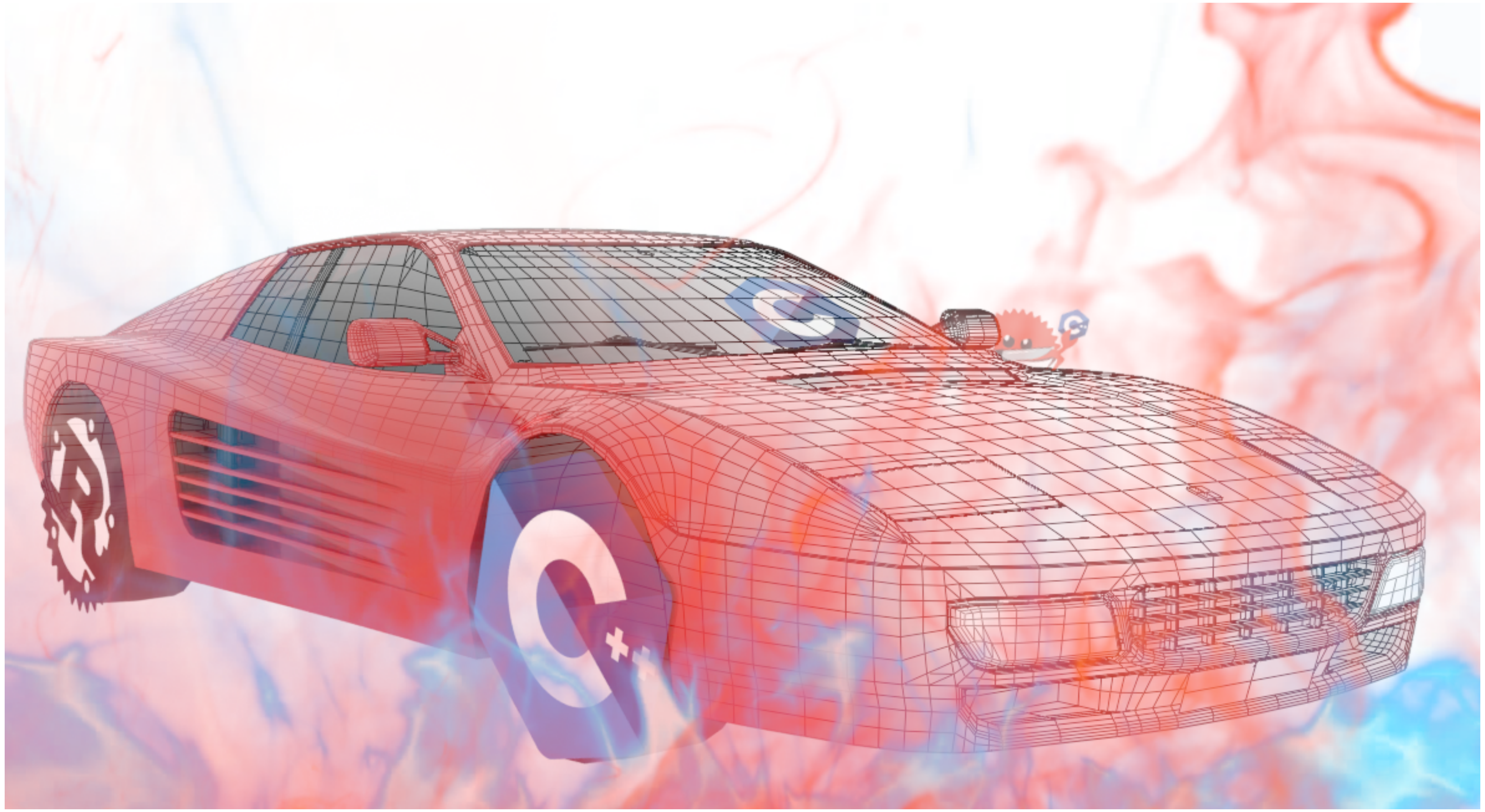

Table of Contents

Failing In Place

Add a Way Out

- Constructors can fail
- Constructors are *not* functions
  - Constructors do *not* return
  - Constructors have initializer lists
  - Constructors run in-place
  - Constructors ignore `const`
  - Constructors cannot *fail partially*
- Where failure in construction?
  - In subconstructors
  - In own code
- On failure in construction: roll-back, abort, and report
  - Undo own code (if we got so far)
  - Call destructors of already called constructors (but *only* those)
  - Abort and report failure

Designing Our Way Out

The Road to Error Handling is Paved With Good Options

- Hard qualifications of mechanism:
  - Separate in normal and exceptional state
  - Find caller that handles report ⇒ stack unwinding
  - In exceptional state: destroy all destructors on path to handler
  - Report arbitrary error information
- Soft qualifications of mechanism:
  - Fast for normal (non-failure) path
  - Clear separation between production code and error handling
  - Nice syntax

Given that our mechanisms deals with the *exceptional* code path, let's call it `exception`

Keyword Matching

`finally` Something that C++ Does Not Support

- Entering the exceptional state: `throw`
  - Enters stack unwinding procedure
  - Can have one object of metadata
    - Any metadata: `throw -1;` is absolutely valid
    - Later today: `std::exception`
- Checking for exceptions:
  `try { throw Ball {}; } catch (Ball&) {}`
  - Checks exception (almost always by reference!)
  - Additionally checks for exception type (RunTime Type Information, RTTI)
  - Multiple `catch` blocks possible
  - Last `catch ( ... )` can rethrow without parameters
- Not in the language: `finally`
  - Use RAII!
- Also not in the language:
  `catch ... else`
  - Use longer `try { /* */ }` clause

`std::exception`

The exception base class to be used

- `std::exception`
- The base class for all your exception needs
- Probably: Select one of the subclasses (e.g. std::runtime_error)
- Use the standard library! ↖
- See the part with "do not exit with an exception"?
- Throwing exceptions while an exception is thrown is bad
- ***Do Not Throw In A Destructor!***
- Chaos!

RTTI

- Sometimes: object with unknown runtime type
  - Reference or Pointer only
  - "Polymorphic Type"
- Query static type: `typeid`
- Result: `std::type_info`
  - Comparison: Are these both for the same type?
  - `before()` allows sorting of types, great
  - `hash_code()` allows hashing of types, great
  - `name()` some name that correlates with the types' name

```
#include <iostream>

struct base {};
struct derived : base {};

struct base2 {virtual ~base2() = default;};
struct derived2 : base2 {};

int main() {
    derived d;
    derived2 d2;
    std::cout << sizeof(d)
        << " " << sizeof(d2) << std::endl;
    base *b = &d;
    base2 *b2 = &d2;
    std::cout << typeid(*b).name()
        << " " << typeid(*b2).name() << std::endl;
}
```

```
g++ typeid.cc && ./a.out
1 8
4base 8derived2
```

dynamic_cast

Verifying runtime types

- Often: child classes have more info
  - Base class: Drawable, children: Rectangle, Circle
- When drawing: Need extra info
- Use `typeid()` for dynamic query? For selection only
- We need to cast

- Use `dynamic_cast`
- Two options:
  - With pointers, returns `nullptr` if wrong type
  - With references, throws exception

```
derived2 *d2p = dynamic_cast<derived2*>(b2);
derived2 &d2r = dynamic_cast<derived2&>(*b2);
base2 bc{};
derived2 &d2rf = dynamic_cast<derived2&>(bc);
```

```
terminate called after throwing an instance
  of 'std::bad_cast'   what():  std::bad_cast
[1]    883256 IOT instruction (core dumped)  ./a.out
```

noexcept

Promises, Promises

- Realistically: Most functions do not throw
  - Pure Math
  - Unrecoverable Errors (out of memory, CPU crashed, ...)
  - Someone used wrong error handling (std::filesystem)
- In this case: Promise not to throw
- Enables optimizations and better code understanding
- `int add1(int a) noexcept;`
- As usual in C++: wrong default ¯_(ツ)_/¯

- *Historical aside: checked exceptions*
- *In C++98: Possibility to explicitly list all throwable exceptions*
- If unexpected exception, call `std::unexpected`, which terminates
- Idea: Simplify arguing about code, increase performance 🚴
- Reality: Not really used
  - Unable to modify code locally
  - Not enforced by compiler, but at runtime
  - Only used with empty list (can throw no exception)
  - Nowadays: checked exceptions considered harmful

Are Exceptions Slow?

Implementing Exceptions Efficiently

- Exception implementation guidelines: 🚴
  - Rule No 1: Optimize the normal, error-free path
  - Rule No 2: The error path can be very slow
  - Rule No 3: See rule 1
- Rough idea: Emit code as if no exception could ever occur

- On throw: Jump to completely different part of code ("shadow stack", Peters name)
- Per function: Store the start of various destructors, catch clauses
- Parse return stack, pc to jump to correct next function handler on shadow stack
- Sometimes: requires use of additional base pointer
- Overhead of exceptions (when not thrown): About one instruction per function

Error handling performance

It's Just Bad

Error handling is never free!

- Optional: Consumes extra stack space, destroys register allocation (calling convention)
- Error Codes: destroy return value, require extra pointer and stack
- Confusing normal and exceptional path, basically universal

The Final Result

Expect the expected

- What we really want:
  - Type that contains *either* the expected result *or* an error
  - Basically an improved `std::optional`
  - Exception-free code as long as we're dealing with the expected or with the error explicitly
  - If we cannot handle the unexpected locally ⇒ throw the unexpected
- See: std::expected
- *Basically an improved std::optional*
- *C++23 only :/*

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

std::expected<int,std::string> parse_char(char c) {
    using namespace std::string_literals;
    if (c < '0' || c > '9')
        return std::unexpected{"character is not a number: "s + c};
    return c - '0';
}

auto inner() {
    auto first = parse_char('5');
    auto second = parse_char('a');
    if(!first.has_value()) return 0;
    return first.value() + second.value();
}

int main() {
    try {
        inner();
    } catch(std::bad_expected_access<std::string> &err) {
        std::cout << err.error() << std::endl;
    }
}
```

Templates

And Now For Something Completely Different

- Often: Write code that is type-agnostic
- e.g., sorting
  - Need sizeof (for pointer arithmetic)
  - Comparison
- C approach: copy-paste code
- C++ approach: let compiler replace types
  - Create class or function *template*
  - Compiler creates new element each time using template
  - Called *template instantiation*
  - Rust name: *Monomorphization*

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

template<typename T>
auto bubble_sort(std::vector<T> input) {
    if (input.size() < 2) return input;
    for (auto i = input.size(); i > 1; i -= 1) {
        for (auto j = 0; j < i - 1; ++j) {
            if (input[j] > input[j + 1])
                std::swap(input[j], input[j + 1]);
        }
    }
    return input;
}

int main() {
    auto vec1 = std::vector
        {5, 4, 3, 2, 1, 6, 7, 8, 9};
    auto vec2 = std::vector
        {5.5, 4.5, 3.5, 2.5, 1.5, 6.5, 7.5, 8.5, 9.5};
    std::cout << (typeid(vec1) == typeid(vec2)) << std::endl;
    vec1 = bubble_sort(vec1);
    vec2 = bubble_sort(vec2);
    for(auto el : vec1) std::cout << el << ", ";
    std::cout << std::endl;
    for(auto el : vec2) std::cout << el << ", ";
    std::cout << std::endl;
}
```

Generic Types

Writing Libraries, Properly

- Often: Write code that is type-agnostic
- Extremely often: data structures (i.e., containers)
  - Contain more objects, handle them
  - list, vector, ...: completely data agnostic
  - tree, hashmap: require some data functions (concepts, eventually)
- Even specialized code (Quadtree, bitsets, ...) can decide on data width
- Almost everything can be a template!
- Decision guide: template if actually required, else keep refactorable

```cpp
#include <iostream>
#include <memory>

template<typename T>
struct list {
    std::unique_ptr<list<T>> next;
    T data;
    void append(T data) {
        auto ptr = this;
        while(ptr->next)
            ptr = ptr->next.get();
        ptr->next = std::make_unique<list<T>>(std::move(data));
    }
    list() = default;
    list(T data) : list {nullptr, data} {}
private:
    list(std::nullptr_t, T data) : next {nullptr}, data{data} {}
};
int main() {
    list<int> l{0};
    for (auto i : {1, 2, 3, 4})
        l.append(i);
    auto ptr = &l;
    while(true) {
        std::cout << ptr->data << ", ";
        if (ptr->next)
            ptr = ptr->next.get();
        else break;
    }
    std::cout << std::endl;
}
```

Recap

What we did today

- Today:
  - Exceptions
  - Exception performance 🚴
  - Exception alternatives
  - Template introduction
- Next Week:
  - Template in depth