Move semantics in C++ and Rust: The case for destructive moves

Radek Vít · Follow

10 min read · Feb 10, 2021

Listen Share

For value-oriented programming languages, move semantics present a big step forward in both optimization and representing uniqueness invariants. C++ has chosen the path of non-destructive moves, where moved-from variables are still usable (albeit usually in an unspecified state). Rust, on the other hand, uses destructive moves, where the moved-from variable can no longer be used. I'll introduce both approaches in a little more detail and present some issues with non-destructive moves. Finally, I will present what C++ could have looked like with destructive moves.

Move semantics in C++ (simplified)

Value categories

In C++, each expression has not only a type, but also a *value category*. There exist three *primary* type categories, and two *mixed* type categories. Each expression has a primary type category, which determines how the language will treat it in relation to other expressions.

- An lvalue is, simply put, a variable; a memory address with a name.
- An xvalue is like an lvalue, but we declare that the resources that this variable owns may be transferred to a new owner.
- A prvalue is a temporary value without a name.
- A glvalue (mixed) is either an lvalue or an xvalue.
- An rvalue (mixed) is either an xvalue or a prvalue.

```
auto i
// `i`
// `std
// `static_cast<std::string&&>(i)` is an xvalue
// `std::string{"value categories"}` is a prvalue (pure rvalue)
```


rvalue references

As their name suggests, rvalue references are refrences that point to rvalues. With these references, we can differentiate between lvalues and rvalues, most often in constructors and assignment operators.

```
struct MyData
{
    std::string data1;
    std::string data2;

    MyData() noexcept = default;
    // this is (basically) what the compiler will generate for you
    // never write these by hand unless you're managing resources
    // copy constructor
    MyData(const MyData& other)
        : data1{other.data1}
        , data2{other.data1}
    {}
    // copy assignment
    MyData& operator=(const MyData& other) {
         data1 = other.data1;
         data2 = other.data2;
         return *this;
    }
    // move constructor
    MyData(MyData&& other) noexcept
        : data1{std::move(other.data1)}
        , data2{std::move(other.data1)}
    {}
    // move assignment
    MyData& operator=(MyData&& other) noexcept {
         data1 = std::move(other.data1);
         data2 = std::move(other.data2);
         return *this;
    }
};
```

Classes that manage resources, like `std::vector<T>`, `std::string`, will usually do the following in their move constructors: instead of allocating new memory, they will take the already allocated buffer from the rvalue they're being constructed from, and leave some valid value in its stead. Move assignment will usually simply swap the allocat[...] moved-from rvalu[...]


```
template <typename T>
class almost_vector {
    T* buffer = nullptr;
    T* data_end = nullptr;
    T* buffer_end = nullptr;
public:
    almost_vector() noexcept = default;
    almost_vector(const almost_vector& other)
    {
        // allocate buffer, copy elements
    }
    almost_vector& operator=(const almost_vector& other) {
        // allocate new buffer, copy elements
        // swap the buffers
        // deallocate the old buffer
    }
    // the move constructor will do something like this
    almost_vector(almost_vector&& other) noexcept
    {
        std::swap(buffer, other.buffer);
        std::swap(data_end, other.data_end);
        std::swap(buffer_end, other.buffer_end);
    }
    // move assignment will do something like this
    almost_vector& operator=(almost_vector&& other) noexcept {
        std::swap(buffer, other.buffer);
        std::swap(data_end, other.data_end);
        std::swap(buffer_end, other.buffer_end);
        return *this;
    }
};
```

There is a non-intuitive side to rvalue references. For example, variables that are rvalue references become lvalues when used in expressions! Also, when you write `std::move(data)`, the expression actually does nothing on its own; it is merely a cast to an rvalue reference.

```
void foo(std::string data);

void bar() {
    std::string data;
    std::string&& data_ref = std::move(data);
    foo(data); // this will copy!
```

```
    foo(std::move(data)); // this moves
}
```


There exists a third kind of reference in C++ aside from lvalue references and rvalue references: the forwarding reference. In templated functions, `T&&` becomes a forwarding reference instead of an rvalue reference, and `auto&&` is always a forwarding reference. Forwarding references preserve the value category of the expression they're initialized with, and can be preserved when passing to other functions.

```
std::string baz(std::string);

template <typename T>
struct Templated {
    // t is an rvalue reference
    void foo(T&& t) {}
    // u is a forwarding reference
    template <typename U>
    void bar(U&& u) {
        // forward to another function
        // x is a forwarding reference
        auto&& x = baz(std::forward<U>(u));
    }
};
```

`std::move`

`std::move` is a utility function in the standard library that lets us mark lvalues as xvalues. It doesn't hide any compiler magic, as its implementation is just a `static_cast` to an rvalue reference.

```
void foo (T&&);

void bar() {
    T value;
    // this is a noop
    std::move(value);
    foo(std::move(value));
    // this is the same thing, only cryptic
    T value2;
    foo(static_cast<T&&>(value2));
}
```

Moved-from states

The variables we move from are still usable after the move in C++. The variables' destructors es, and users may

The C++ standard library chooses to keep the moved-from variables in a valid, but unspecified state; this means that we can reuse the variable, we just cannot rely on its contents.

For user-declared types, the only real requirement is that the destructor on a moved-from variable must run without causing any issues for the rest of the program. Any invariants of the type may be broken and calling any functions on them can cause undefined behavior, it is just a matter of convention (and convenience) that we usually don't do these things.

Move semantics in Rust

In Rust, *all types* are movable, and all move operations amount to a *bit copy* of the original data to its new location. Unlike C++, moving is the default operation, and we explicitly have to call a function to copy them.

Rust move operations are also *destructive*. After we move from a variable (even potentially), that variable becomes unusable in code.

```
#[derive(Clone)]
struct MyData {
    boxed_uint: Box<u64>,
    data: String,
}

fn foo(_data: MyData) {
    // do something with _data
}
fn bar() {
    let data = MyData{
        boxed_uint: Box::new(42),
        data: "".to_owned()
    };
    foo(data.clone()); // we copy here
    if random_bool() {
        foo(data); // we move here
    }
    // foo(data); // ERROR: use of moved value
}
```

The reason why bit copies are always enough for a move operation in Rust is that Rust does r

ving rules in Rust ma ss you reach for raw pointers and `unsafe`). Such structs would require their move operations to adjust these references after moving the resources from the original object, but without them, there is no real need to execute arbitrary code on moves.

```
// you cannot do this in Rust

// C++
struct SelfReferential {
    std::array<char, 1'000> data;
    char* cursor = nullptr;
    SelfReferential(): data{{}}, cursor{&data[0]} noexcept {}
    SelfReferential(SelfReferential&& other)
        : data{other.data}
        , cursor{&(data[0]) + (other.cursor - &(other.data[0]))}
    {}
    // copy constructor, assignment operators omitted
};
```

The Clone and Copy traits

For copy operations, Rust has the `Clone` trait. Structs implementing this trait take a reference and create a new value from it.

```
#[derive(Clone)]
struct MyData {
    boxed_uint: Box<u64>,
    data: String,
}

/* derive(Clone) will generate
   something semantically identical to this
impl Clone for MyData {
    #[inline]
    fn clone(&self) -> MyData {
        MyData {
            boxed_uint: self.boxed_uint.clone(),
            data: self.data.clone(),
        }
    }
}
*/
```

There exist types where copying by default is desirable (like integers, bools, floats, tuples of in … n the `Copy` trait, which … y convention, only types that are inexpensive to copy are marked with this trait.

Where non-destructive moves fail

Weaker invariants for resource management

In C++, raw pointers can have many different meanings: they can represent

1. Nothing (`nullptr`)
2. An address of a single object in owned dynamically allocated memory
3. An address of a single object in non-owned memory
4. An address of an array of objects in owned dynamically allocated memory
5. An address of an array of objects in non-owned memory

Because of this semantic ambiguity, references are usually preferred in modern C++, because they always point to one valid object, where we always know that we don’t own it (both are possible to break, but breaking the first assumption is undefined behavior and breaking the second breaks every reasonable C++ convention). There exist alternatives for other scenarios from this list as well.

Where C++ has been able to improve this situation in non-owning contexts, it still has the same billion dollar mistake ingrained in its core smart pointers: both `unique_ptr` and `shared_ptr` can be `nullptr` .

With non-destructive moves, this is a neccessity. There exists no other real option for a moved-from state other than `nullptr` for smart pointers: If they kept the original pointer in them, `unique_ptr` would free the same memory twice, and `shared_ptr` would have more references than it tracks. If they assigned a random address, we would access (and `delete`) random memory. Finally, an explicit marker for moved-from states would be exactly nullptr, but slower.

Thanks to destructive moves, Rust’s smart pointers (`Box` , the counterpart of `unique_ptr` and `Arc` , the equivalent of `shared_ptr`) *always* hold dynamically allocated memory. This invariant lets us prevent many possible errors at compile time instead of relying on conventions (like never passing `nullptr` smart pointers) or runtime checks everywhere. For situations where we actually want nullable pointers, w `<Arc<T>>`, where we a d have nice built-in ways of handling those situations).

Non-destructive move operations may fail (if you consider OOM errors recoverable by default)

For at least some container implementations in C++, moved-from objects *require* memory allocations. This means that at least in some cases, calling the move constructor is not an infallible operation.

With destructive moves (or by treating OOM errors as unrecoverable), C++ could realistically mandate that all of its move constructors are `noexcept`. While there theoretically exist other potential failures when moving objects with arbitrary code, I haven't seen any convincing examples where types with other kinds of move errors are worth complicating the language over.

Move semantics become complicated

As we saw in the C++ overview of move semantics, non-destructive moves bring with them heaps of complexity: we add an entire new value category, two more kinds of references, and we all of a sudden have at least 5 ways to pass an argument to any function (by value, by pointer, by reference, by const reference, and by rvalue reference; not counting arrays and optional values), where all of them have valid usecases. We have to care and know about moved-from states, and we introduce potential failures for move operations.

Containers become complicated

Containers in the C++ standard library provide exception guarantees (if an operation fails in the middle of its execution, the container will be left in a valid state) and strong exception guarantees (if an operation fails in the middle of its execution, the container will be left in an identical state to what it was originally). If we consider `std::vector`'s `push_back`, the container must

1. Potentially increase the size of the buffer to fit the new element

2. Move or copy the new element in its new place.

To achieve strong exception guarantees when increasing its size and copying elements, `push_back` will

1. Allocate a new buffer

2. Copy all elements into the new buffer

3. Swap th

4. Free the memory of the old buffer

Done this way, if any of the copy operations fail, the container still has its original buffer with all of its elements. For fallible move operations, achieving strong exception guarantees is impossible this way:

1. Allocate a new buffer

2. Move all the elements into the new buffer

3. In the middle, a move operation fails

4. We can't move the already moved objects back, because that could fail too

For this reason, only vectors containing objects with `noexcept` move constructors will use move semantics when resizing their internal buffers. If you forget to mark your move constructors `noexcept` , you lose a lot of the optimization you thought you were getting by implementing them.

C++ with destructive moves

I will present a rough outline of what C++ might have looked like with destructive moves, and how it could have avoided some of C++'s current problems.

```
void foo(std::string x);

void bar() {
    std::string data {"Important stuff"};
    if (random_bool()) {
        foo(move data);
    } else {
        // do nothing
    }
    // ERROR: cannot use potentially moved-from variable
    // foo(data);
    // data's destructor will run if it hasn't been moved from here
}
```

Operator `move`

To move objects in C++, we introduce a new operator `move`. This operator would always call alue reference). lestructor would run on it. Operator `move` would not be usable on lvalue references. There would exist a variant similar to placement new in that the target of the move could be a specific memory address instead of a new temporary.

```
struct Movable {
    std::string data;
    std::string data2;

    Movable() = default;
    // default move constructor; always noexcept
    // the argument's destructor is not called after this
    Movable(Movable&& other)
        : data {move other.data}
        , data2 {move other.data2}
    {}
    // default assignment for movable types
    Movable& operator=(Movable other) {
         data = move other.data;
         data2 = move other.data1;
         // after (partially) moving from a variable's members
         // the destructor is only called for non-moved-from members
         return *this;
    }
};

struct NotMovable {
    std::string data;
    std::string data2;

    NotMovable() = default;
    // declaring a copy constructor still disables move semantics
    NotMovable(const NotMovable&) = default;
    // default assignment for non-movable types
    NotMovable& operator=(const NotMovable& other) {
        data = other.data;
        data2 = other.data2;
        return *this;
    }
};
```

Operator ref_move

Operator ref_move would be usable through lvalue references. Instead of removing the variable, it would leave unspecified data in its place. This operator would be necessary for implementing memory-handling standard functions such as `std::swap`, where we would be able to make sure the original variable ends up with a valid value by the end of the call. We would also need a variant for placing the result direc

This is what a destructive move-based `swap` function could look like:

```
// enable if T is movable
template <typename T>
void swap(T& lhs, T& rhs) noexcept {
   T temp {ref_move lhs};
   ref_move(&lhs) rhs; // place the move into lhs
   move(&rhs) temp; // place the move into rhs
}
```

rvalue references

In this outline, we keep rvalue and forwarding references in the language. This allows us to keep consistency with copy constructors and perfect forwarding.

If we gave up both of these and trusted the compiler to optimize away extra moves, we could use different syntax (such as `move T(T& other)`) for move constructors and do away with rvalue and forwarding references entirely.

Solving nondestructive move's issues with destructive moves

1. `std::unique_ptr` and `std::shared_ptr` are never `nullptr`. They always hold dynamically allocated memory. For nullable managed pointers, we have `std::optional<std::unique_ptr<T>>`. Similarly, other classes that manage resources would be allowed to always hold non-null valid values.

2. We never need to allocate memory for moved-from objects. Move operations are always `noexcept` and cannot fail.

3. We only have two value categories: lvalues and rvalues. There exist no moved-from states. Overall, move semantics become less complicated.

4. Containers can always move when types are movable. Upholding strong exception guarantees becomes easier.

Cpp Rust Programming Programming Languages

Follow

Written by Radek Vít

43 Followers

More from Radek Vít

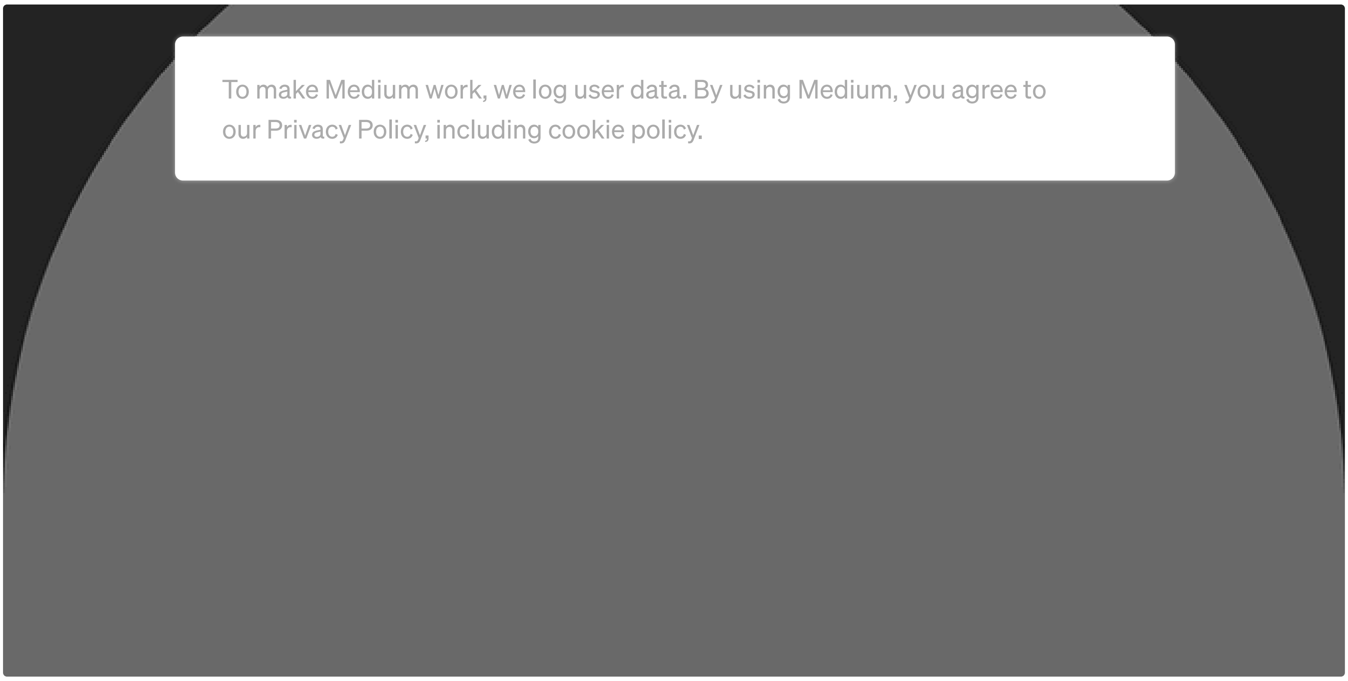
 Radek Vít

Making concurrency fearless with Rust for C++ developers

Multithreading is hard. C++ is hard too. I will demonstrate how we can make multithreading at least a little easier by avoiding some of...

6 min read · Feb 2, 2021

104 3



Radek Vít

Examples of declarative style in Rust

In most low-level programming languages, we are used to describing how what we want is achieved, rather that what we want to achieve in...

4 min read · Feb 17, 2021

80

Radek Vít

Avoiding single-threaded memory access bugs with Rust (for C++ developer

In a previous ns and invalid memory access to our code in...

5 min read · Mar 7, 2021

42 3

 Radek Vít

Writing Interfaces: Stay true in booleans

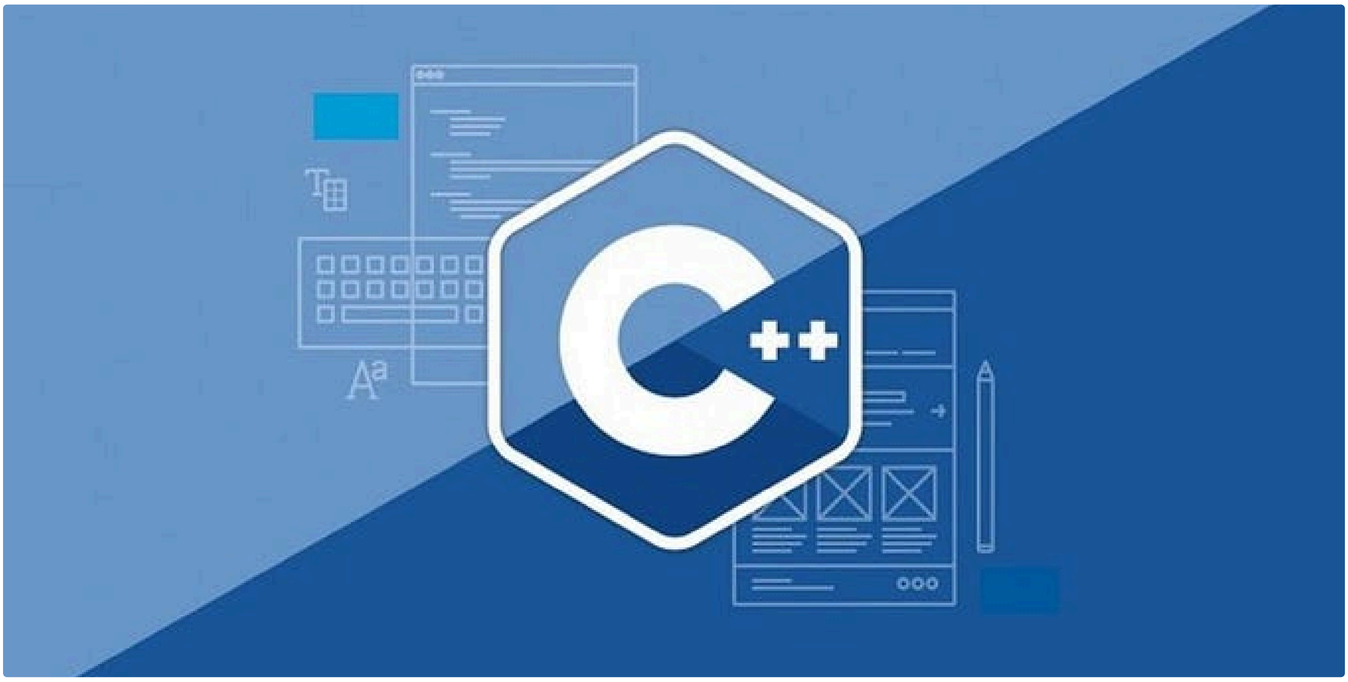
When faced with passing multiple options as arguments to functions, C++ programmers will use enum class without much hesitation. Since...

3 min read · Mar 10, 2021

3

See all from Radek Vít

Recommended from Medium



Robby Seguin

Memory Leaks C++ (Unique Pointers)

It's been interesting to see different methods to reduce memory leaks and increase performances.

2 min read · Feb 7, 2024

10 1

Mayowa Obisesan

Function Declaration vs Function Definition

In C++ and other low-level languages such as Solidity, there is a commonly used term: declaring a function and defining a function. What's...

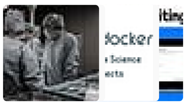
2 min read · Dec 12, 2023

7

Lists

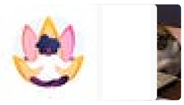
General Coding Knowledge

20 stories · 1284 saves



Coding & Development

11 stories · 644 saves

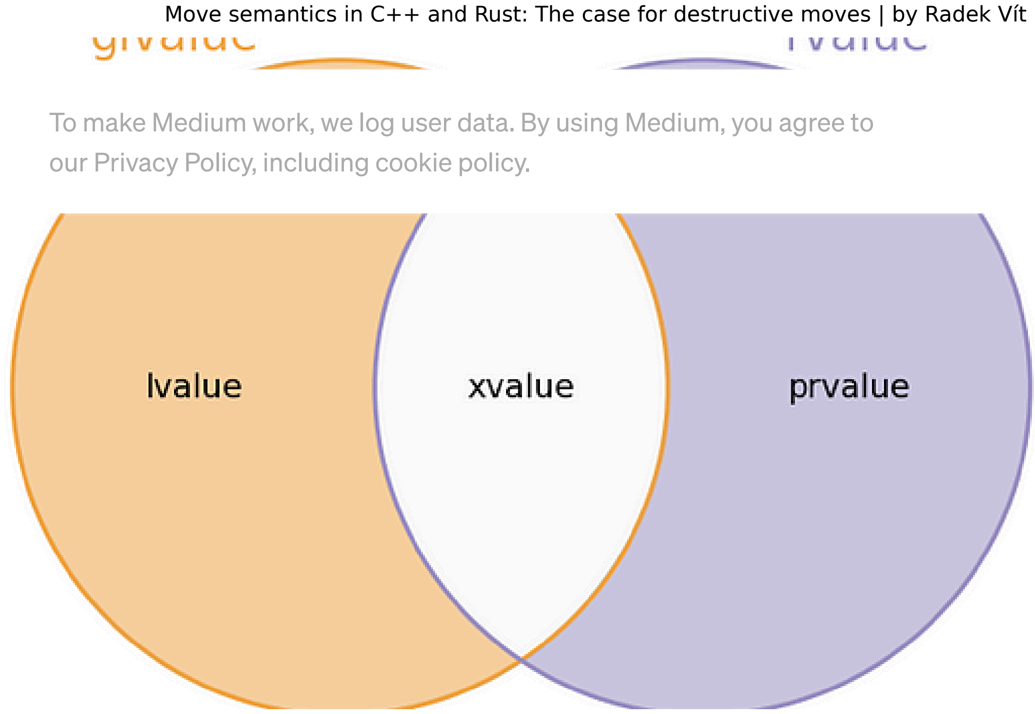


Stories to Help You Grow as a Software Developer

19 stories · 1119 saves

ChatGPT

21 stories · 672 saves



Developer Notes

lvalue & rvalue in C++

What is lvalue?

2 min read · Dec 12, 2023

9 1

Jebinshaju

Exploring Bend: A Revolutionary Language for GPU Programming

The Bend pr… …ool
designed to

5 min read · M

7

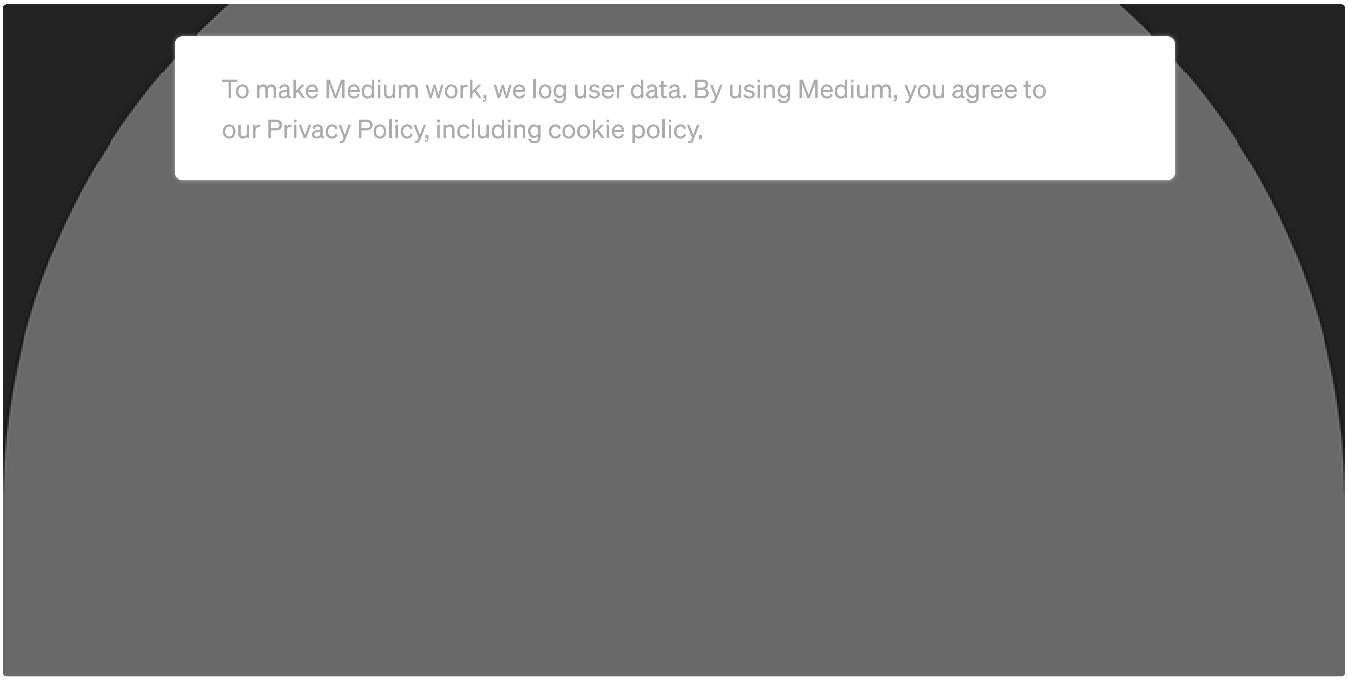
Daksh Gupta

Modern C++ Programming — Things that shouldn't be used anymore

C++ was and is a great programming language and whether we realize this or not, many of our day to day databases, scripting engines and...

6 min read · Feb 29, 2024

327 16



Vivian Aranha in Coinmonks

How Rust Programming Language is Used in Blockchain Development

Introduction

3 min read · Mar 26, 2024

See more recommendations