



CSCS

Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich



Resource Management

About RAll, ownership, pointers and guidelines

Alberto Invernizzi, CSCS (alberto.invernizzi@cscs.ch)

C++ is an object-oriented programming language that among its main selling points has

- 🏎️ Performances
- 🛠️ Letting the user have full control over resources

Performance and **full-control** are somehow faces of the same coin: full control allows to do very clever and smart things to get best performances.





"... and with great power comes great responsibility."

Example: Memory

Memory management is an important aspects for many application, be it for

- for optimization reasons
e.g.reduce memory operations costs and overhead)
- memory limit constraints
e.g. embedded applications

This is one of the reasons why C++ is used in many industries, from Game Development to HPC.

Anywhere performance and control matters.

Indeed, C++ gives you all the knobs to manage the memory: when to allocate, when to deallocate, how much to allocate, ...

It's not just about memory ... it's about **RESOURCEs!**

- Memory
- File
- Socket
- Mutex
- MPI Communicator
- ...

Full control of a resource means managing it correctly by

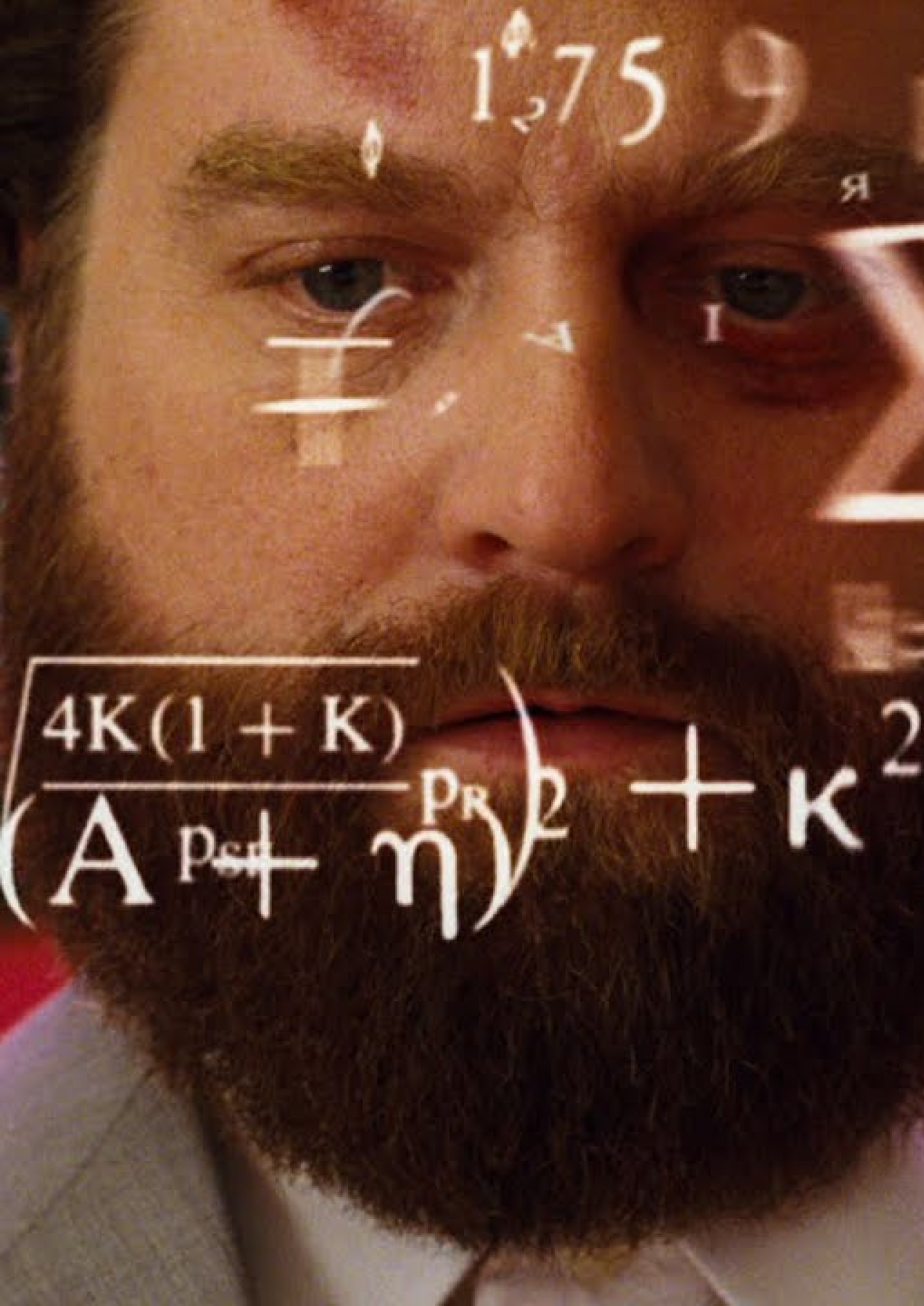
- 🤝 initializing/acquiring it
- 🚒 keeping it alive till needed
- 🙌 release it cleanly when not useful anymore.

Why should we care?

Not managing correctly resources may end up in subtle bugs...

- in the "best" case a memory leak
- in (one of) the worst cases a race-condition ✨ (=nightmare 🤪).





Managing the lifetime of a resource in an object-oriented context easily becomes difficult.

- objects are created,
- objects are manipulated
- objects are passed around to interact with other parts of the program
- ...

When the program complexity starts increasing, to ensure the correct management of these resources “manually” becomes unsustainable.

...and with concurrency it becomes even more difficult (=“**impossible**” 🤯).

Full control \neq hard to do!

Some languages address this problem using **garbage collectors** but at the expense of performances and control.

Not a solution for C++...

...but having full control does not imply having to do it manually!

The language, through the compiler, is at our disposal. We can and should leverage it at our service.

Here we are going to see what tools the language offers us and which we can and should rely on to keep things under control, aiming at

READABLE, CORRECT and **EFFICIENT** code.

RAII

RAII, which stands for *Resource Allocation Is Initialization*, is a programming technique that binds resource acquisition to **object lifetime**.

If an object follows RAII, it ensures that:

- the resource is acquired/allocated/initialized when the object is initialised
- it will be available for the lifetime of the object
- and when the object is destroyed (it goes out of scope) the binded resource will be released too.

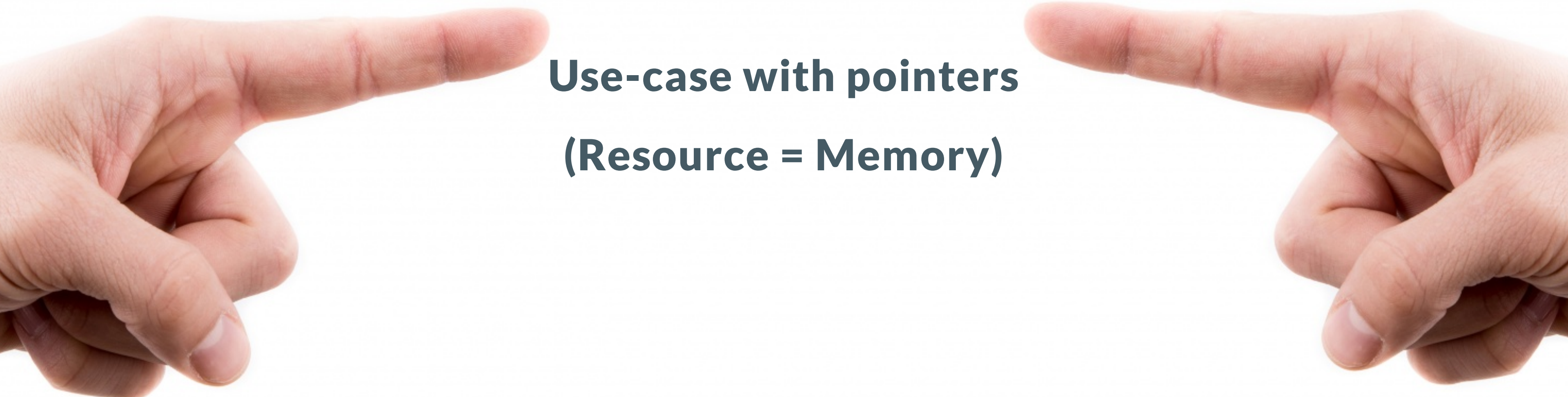
Ownership

With RAll an object starts representing the **ownership** of the resource, so object has the responsibility of the correct management.

Developer does not have anymore the "direct" responsibility of the resource, but it does not mean they don't have anymore control over it.

We delegated the hard-work of managing correctly the resource to the object and we can now reason about its ownership.

It's a higher level of control, we don't care anymore about what happens when the resource has to be created/released, we just have to think where and how long we need the resource and manipulate the object accordingly.

Two hands are shown from the left and right sides of the frame, with their index fingers pointing directly at the central text. The hands are light-skinned and the background is a plain, light color.

Use-case with pointers (Resource = Memory)

Raw Pointers

Every C and C++ developer had to overcome the obstacle of pointers...

```
void foo() {  
    int value = 26;  
    int *pointer_on_stack = &value;  
  
    int* pointer = new int(26);  
    int* buffer = new int[13];  
  
    for (int i = 0; i < 13; ++i) {  
        int val = vec[i];  
        if (val == 0)  
            return;  
    }  
  
    delete[] buffer;  
}
```

But are they the right tool for managing resources?
(i.e. resource = memory in this case)

Problem: Who is in charge?

Even without looking at the documentation, a reasonable expectation is that what it returns is a pointer to a memory allocated by the function.

```
gsl_multifit_fsolver* gsl_multifit_fsolver_alloc(  
    const gsl_multifit_fsolver_type * T,  
    size_t n,  
    size_t p);
```

- is it up to me to deallocate it and keep it alive, right?
- and `T`? should it be kept alive till `multifit_solver` is in use, correct?

Problem: how should it be released?

How was it allocated?

- `new` -> `delete`
- `new[]` -> `delete[]`

Called by ***delete***-expressions to deallocate storage previously allocated **for a single object**.
The **behavior** [...] **is undefined** unless:

- *ptr* is a **null pointer** or
- *is a pointer previously obtained from* [...] `operator new(std::size_t)` or `operator new(std::size_t, std::nothrow_t)`

Called by ***delete[]***-expressions to deallocate storage previously allocated **for an array of objects**.
The **behavior** [...] **is undefined** unless:

- *ptr* is a **null pointer** or
- *is a pointer previously obtained from* [...] `operator new[](std::size_t)` or `operator new[](std::size_t, std::nothrow_t)`

(source: https://en.cppreference.com/w/cpp/memory/new/operator_delete)

Problem: burden of the management

1. Remember to do it
it's not about being too lazy, it's more about cognitive load
2. Do it in the correct order
e.g. track dependencies between resources, is it deterministic?

```
#include <algorithm>
#include <iostream>
#include <random>

int main() {
    constexpr std::size_t N = 5;
    int* buffer = new int[N];

    std::mt19937 engine;
    std::uniform_int_distribution<int> uniform_dist(1, 10);
    std::generate(buffer, buffer + N, [&]() { return uniform_dist(engine);});

    int* min_value = std::min_element(buffer, buffer + N);
    int* max_value = std::max_element(buffer, buffer + N);

    delete[] buffer;

    std::cout << *min_value << ":" << *max_value << "\n";
}
```

Possible output:

```
0:12296208
```


Have you considered all execution paths?

If a function has multiple return statements, you may have to care about it multiple times...

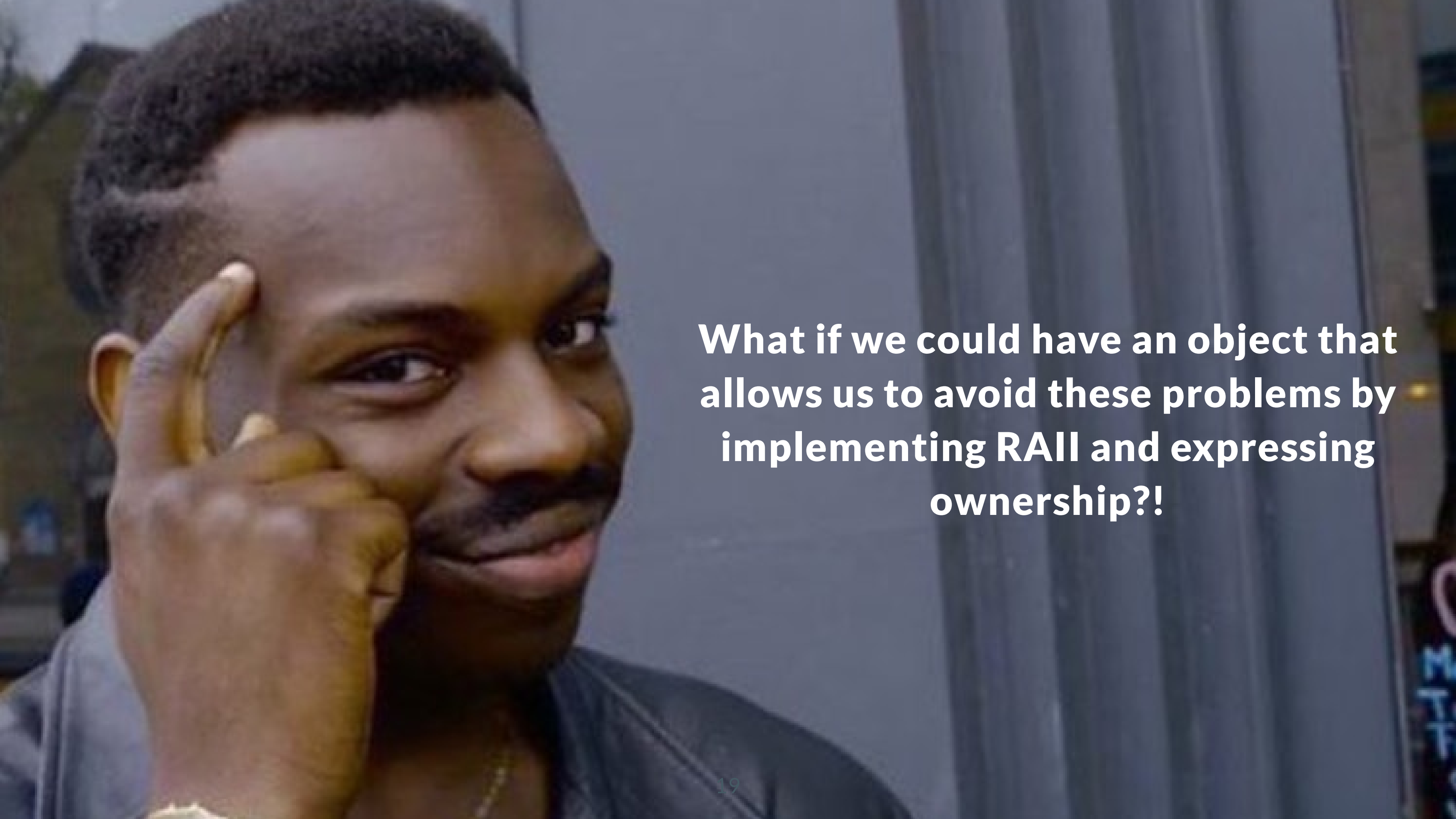
```
bool foo(int a, int b) {  
    int buffer = new int[10];  
  
    // ... (using buffer)  
  
    if (a == 0) {  
        return false;  
    }  
  
    // ... (using buffer)  
  
    delete[] buffer;  
    return true;  
}
```

... even exceptions?

In case of an exception not managed, it becomes impossible to manage release correctly...

```
float foo(int a, int b) {  
    int buffer = new int[10];  
  
    // ... (using buffer)  
  
    // possibly throwing operation...  
    float result = a / b;  
  
    // if previous instruction throws...  
    // ...nobody is going to release buffer  
    delete[] buffer;  
    return result;  
}
```

Raw pointers
do NOT follow RAII
and do NOT express ownership.

A close-up photograph of a Black man with short, dark hair and a mustache. He is looking slightly to his right with a thoughtful expression, his hand raised to his chin with his index finger pointing upwards. He is wearing a dark blue shirt. The background is a blurred outdoor setting with a grey wall and some foliage.

**What if we could have an object that
allows us to avoid these problems by
implementing RAI and expressing
ownership?!**

Object Lifetime - C'tor and D'tor

RAII binds a resource to object lifetime.

Let's see what are the main handles we have on object lifetime.

```
{  
    LessRawPtr a;      // c'tor is called  
  
    // ...  
}                      // d'tor is called
```

The language gives us the handle to the moment when an object starts its lifetime through its **constructor**!

And what happens when it goes out of scope? It gets destroyed...and the language gives us the chance to customize what happens at destruction time through its **destructor**!

Object Lifetime

```
struct LessRawPtr {  
    LessRawPtr() = default;  
    LessRawPtr(int* ptr) : ptr_(ptr) {}  
    ~LessRawPtr() { if (ptr_) delete ptr_; }  
  
private:  
    int* ptr_ = nullptr;  
};
```

- **C'tors** what to do when an object is created
 - (default) no resource managed by the object
 - (custom) bind a resource to the object
- **D'tor** what to do when an object is destroyed
 - if object is bound to a resource, release it

The real magic ✨ resides in the d'tor part. It gets called as soon as an object lifetime ends:

- it goes out of scope (e.g. block, expression, ...)
- stack unwinding, i.e. when an exception is uncaught

We are binding a resource with an object on the *stack*, so **we are transitively giving properties of an *object* on the stack to a resource!**

Object Lifetime in action: multiple return points

We don't have to care anymore about multiple execution paths! 🍾

```
void foo(int a, int b) {  
    int* data = new int[26];  
  
    if (...) {  
        // ...  
        delete[] data;  
        return ;  
    }  
  
    // use data again  
  
    delete[] data;  
}
```

```
void foo(int a, int b) {  
    LessRawPtr data(new int(26));  
  
    if (...) {  
        // ...  
        return ;  
    }  
  
    // use data  
}
```

Object Lifetime in action: exceptions

We don't have to care anymore about exceptions too! 🍾

```
void foo(int a, int b) {  
    int* memory = new int[26];  
  
    if (...) {  
        // ...  
        delete[] memory;  
        return ;  
    }  
  
    try {  
        a / b;  
    }  
    catch (...) {  
        delete[] memory;  
    }  
  
    // ...  
  
    delete[] memory;  
}
```

```
void foo(int a, int b) {  
    LessRawPtr memory(new int(26));    // call to c'tor  
  
    if (...) {  
        // ...  
        return ;  
    }  
  
    a / b;  
  
    // ... rest of code  
}                                     // call to d'tor
```

In case the exception is thrown, rest of code won't be executed...but the stack unwinding ensures that all objects on the stack are destroyed, so the d'tor gets called and the resource is released cleanly! 😊

RAII - Ownership

Now the lifetime of the resource is bound to the object, thanks to RAII. And what about ownership?

- **What does it mean "ownership" for an object?**

It means that an object has responsibility over the underlying resource, whatever it happens...

- **What can happen to an object?**

We can pass it around, for instance we can copy it!

- **What happens when we copy an object?**

From the language perspective, a new object is created...

- **...and what should happen from the resource perspective?**

It depends!

Does the language provide an handle for this phases of the object life?

T(const T&) and T& operator=(const T&)

- **Did we specify anything about them?** Nope.
- **What happens?** Default behavior of the language.

The language cannot know beforehand how the object should behave, so it does the most simple thing.

It implicitly defines them (= default)

- D'tor does nothing, i.e. empty body
- Copy-{C'tor, Assignment Operator}, copy by value all attributes

What does it mean in our case? `LessRawPtr` has a single attribute `ptr_`, which is a simple pointer, so it means copying the address into another object.

😬 How bad can it go?! 😬

```
struct LessRawPtr {  
    LessRawPtr(int* ptr) : ptr_(ptr) {}  
    ~LessRawPtr() { if (ptr_) delete ptr_; }  
private:  
    int* ptr_ = nullptr;  
};
```

[[SPOILER-ALERT]] really bad!

```
{
    LessRawPtr a(new int(26));    // c'tor
    {
        LessRawPtr b = a;        // copy-c'tor
    }                            // d'tor (b)
    LessRawPtr c = a;            // copy c'tor
                                // d'tor (c and a)
}
```

<https://godbolt.org/z/64bE4G3oW>

- `a` acquires the resource
- in the inner block, `b` copies `a`'s resource address, because of the default *copy c'tor*
`a` and `b` now own "together" the same resource 💣
- `b` goes out of scope so the resource gets released 🙋
- `c` will do the same that `b` did i.e. copy the address of `a`'s resource, because of the default *copy c'tor*
- both `a` and `c` believe to still own the resource (even if one does not know about the other)...
- ...but the resource has been already released! 💥

We should probably do something different when the object is copied ... **actually there is a guideline!**

Rule of Three

If a class requires either a:

- user-defined d'tor
`~LessRawPtr()`
- user-defined copy c'tor
`LessRawPtr(const LessRawPtr&)`
- user-defined copy assignment operator
`LessRawPtr& operator=(const LessRawPtr&)`

it almost certainly requires all three.



What to do? It depends!

What *copy*-{*c'tor*,*assignment*} should do depends on how the object should behave on copy (*object semantic*) with respect to the controlled resource.

It might be:

- clone (aka "deep-copy")
should it allocate another identical and independent resource and copy its value?
- not-copyable
should it just not being copiable at all? (= delete)
- something else?
there might be other possible behaviors

Whatever you want it to do, you have to define it. 🧐

Just two examples...

Clone

```
struct LessRawPtr {
    // default c'tor
    LessRawPtr() = default;
    // custom c'tor
    LessRawPtr(int* ptr) : ptr_(ptr) {}
    // d'tor
    ~LessRawPtr() {
        if (ptr_)
            delete ptr_;
    }
    // copy c'tor
    LessRawPtr(const LessRawPtr& rhs) {
        ptr_ = new int(*rhs.ptr_);
    }
    // copy assignment operator (copy-and-swap idiom)
    LessRawPtr& operator=(const LessRawPtr& rhs) {
        LessRawPtr copy = rhs;
        std::swap(copy.ptr_, this->ptr_);
        return *this;
    }
private:
    int* ptr_ = nullptr;
};
```

<https://godbolt.org/z/W5vffM7fM>

Not-copyable

```
struct LessRawPtr {
    // default c'tor
    LessRawPtr() = default;
    // custom c'tor
    LessRawPtr(int* ptr) : ptr_(ptr) {}
    // d'tor
    ~LessRawPtr() {
        if (ptr_)
            delete ptr_;
    }
    // copy c'tor
    LessRawPtr(const LessRawPtr&) = delete;
    // copy assignment
    LessRawPtr& operator=(const LessRawPtr&) = delete;
private:
    int* ptr_ = nullptr;
};
```

<https://godbolt.org/z/cPMvPd415>

First step towards a "smarter" than raw pointer

```
struct LessRawPtr {  
    // default c'tor  
    LessRawPtr() = default;  
    // custom c'tor  
    LessRawPtr(int* ptr) : ptr_(ptr) {}  
    // d'tor  
    ~LessRawPtr() {  
        if (ptr_)  
            delete ptr_;  
    }  
    // copy c'tor  
    LessRawPtr(const LessRawPtr& rhs) = delete;  
    // copy assignment operator  
    LessRawPtr& operator=(const LessRawPtr& rhs) = delete;  
private:  
    int* ptr_ = nullptr;  
};
```

- **Who is responsible?** The object itself thanks to RAII
- **How should it be released?** No worries, it is up to the object (it needs a specialization for `T[]`)
- **Burden of the management?** Again, no worries...it is up to the object (and the language)
- **All execution paths?** Yes! As soon as it goes out of scope, it will be released.
- **...even in case of exceptions?** Yes, because stack unwinding makes the objects allocated on the stack to be destroyed, so their managed resource will be released cleanly.

What do we have? An object representing ownership of a memory allocation. This last implementation is not copyable, so the ownership of the resource is exclusive and cannot be transferred in any way.

Ownership

`LessRawPtr` is really a partial implementation, to the extent that it cannot be really defined a pointer (e.g. how do I access the memory in it?!) and it would need some extensions in order to make it useful.

But it already expresses the concept of **ownership**!

It is possible to differentiate mainly two types of ownership:

- **Unique (or exclusive) ownership**
when there is exactly one object instance managing a specific resource
- **Shared ownership**
when there are more objects managing the same resource (not clones, but exactly the same resource).

What is the type of `LessRawPtr` ownership?

Let's complete the implementation of the `LessRawPtr` .

Let's make it useful!

ToDo

- Generalization for different resources/types
- Allow customization of destructor (each resource has different needs)
- Specialization for T and T[]
- Ways to access the resource
- Decide what to do about ownership
- ...



**Wait...if this is so useful and fantastic,
It is something probably existing in every C++ codebase!
Everyone should use it, no!?**

The savvy uses to say
"don't reinvent the wheel"



STL Smart Pointers

STL provides a fully-featured solution for **smart pointers**:

- `std::unique_ptr<T>` = unique ownership
- `std::shared_ptr<T>` = shared ownership
- `std::weak_ptr<T>` = shared ownership (specific use case)

Which are defined in the `<memory>` header.



STL Smart Pointers API

std::unique_ptr<T>

Member functions

(constructor)	constructs a new unique_ptr (public member function)
(destructor)	destructs the managed object if such is present (public member function)
operator=	assigns the unique_ptr (public member function)

Modifiers

release	returns a pointer to the managed object and releases the ownership (public member function)
reset	replaces the managed object (public member function)
swap	swaps the managed objects (public member function)

Observers

get	returns a pointer to the managed object (public member function)
get_deleter	returns the deleter that is used for destruction of the managed object (public member function)
operator bool	checks if there is an associated managed object (public member function)

Single-object version, unique_ptr<T>

operator* operator->	dereferences pointer to the managed object (public member function)
-------------------------	--

Array version, unique_ptr<T[]>

operator[]	provides indexed access to the managed array (public member function)
------------	--

std::shared_ptr<T>

Member functions

(constructor)	constructs new shared_ptr (public member function)
(destructor)	destructs the owned object if no more shared_ptrs link to it (public member function)
operator=	assigns the shared_ptr (public member function)

Modifiers

reset	replaces the managed object (public member function)
swap	swaps the managed objects (public member function)

Observers

get	returns the stored pointer (public member function)
operator* operator->	dereferences the stored pointer (public member function)
operator[] (C++17)	provides indexed access to the stored array (public member function)
use_count	returns the number of shared_ptr objects referring to the same managed object (public member function)
unique (until C++20)	checks whether the managed object is managed only by the current shared_ptr instance (public member function)
operator bool	checks if the stored pointer is not null (public member function)
owner_before	provides owner-based ordering of shared pointers (public member function)

std::shared_ptr<T>

```
std::shared_ptr<T>::shared_ptr
constexpr shared_ptr() noexcept; (1)
constexpr shared_ptr( std::nullptr_t ) noexcept; (2)
template< class Y >
explicit shared_ptr( Y* ptr ); (3)
template< class Y, class Deleter >
shared_ptr( Y* ptr, Deleter d ); (4)
template< class Deleter >
shared_ptr( std::nullptr_t ptr, Deleter d ); (5)
template< class Y, class Deleter, class Alloc >
shared_ptr( Y* ptr, Deleter d, Alloc alloc ); (6)
template< class Deleter, class Alloc >
shared_ptr( std::nullptr_t ptr, Deleter d, Alloc alloc ); (7)
template< class Y >
shared_ptr( const shared_ptr<Y>& r, element_type* ptr ) noexcept; (8)
template< class Y >
shared_ptr( shared_ptr<Y>&& r, element_type* ptr ) noexcept; (8)
shared_ptr( const shared_ptr& r ) noexcept; (9)
template< class Y >
shared_ptr( const shared_ptr<Y>& r ) noexcept; (9)
shared_ptr( shared_ptr&& r ) noexcept; (10)
template< class Y >
shared_ptr( shared_ptr<Y>&& r ) noexcept; (10)
template< class Y >
explicit shared_ptr( const std::weak_ptr<Y>& r ); (11)
template< class Y >
shared_ptr( std::auto_ptr<Y>&& r ); (12)
template< class Y, class Deleter >
shared_ptr( std::unique_ptr<Y, Deleter>&& r ); (13)
```

1) Constructs a `shared_ptr` which shares ownership of the object managed by `r`. If `r` manages no object, *this manages no object either. [...]

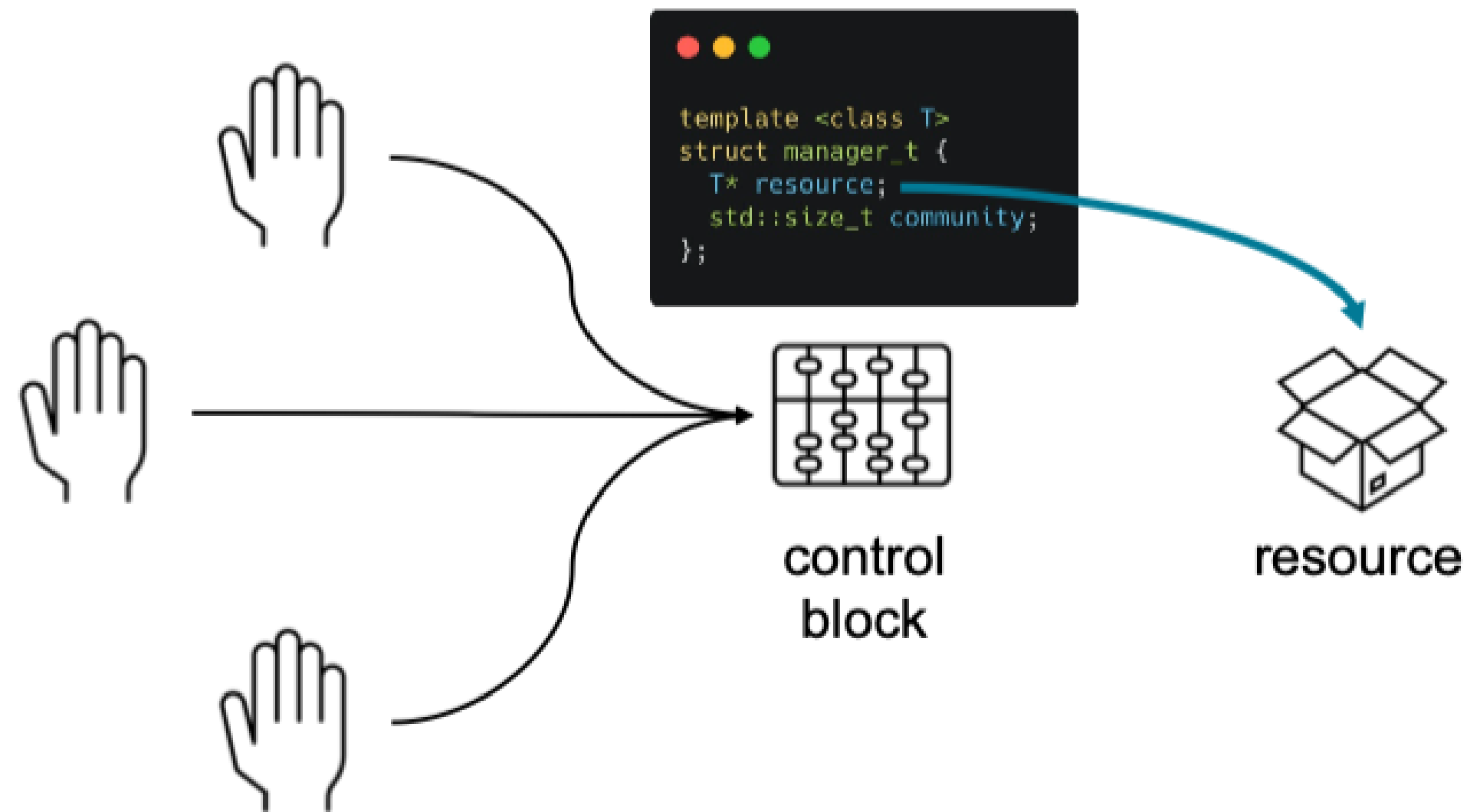
This is copyable, and the copyability for it has been implemented not to clone the resource but to "share" the resource, allowing to extend the ownership group.

As we trivially saw before, just copying the address was not enough for sharing correctly the same resource over two objects. The main problem was that once the address was copied, the two objects were not "linked" anyhow, so one didn't know if the other was still using the resource or not.

How can this information about usage be shared among multiple objects?

shared_ptr<T> : the machinery

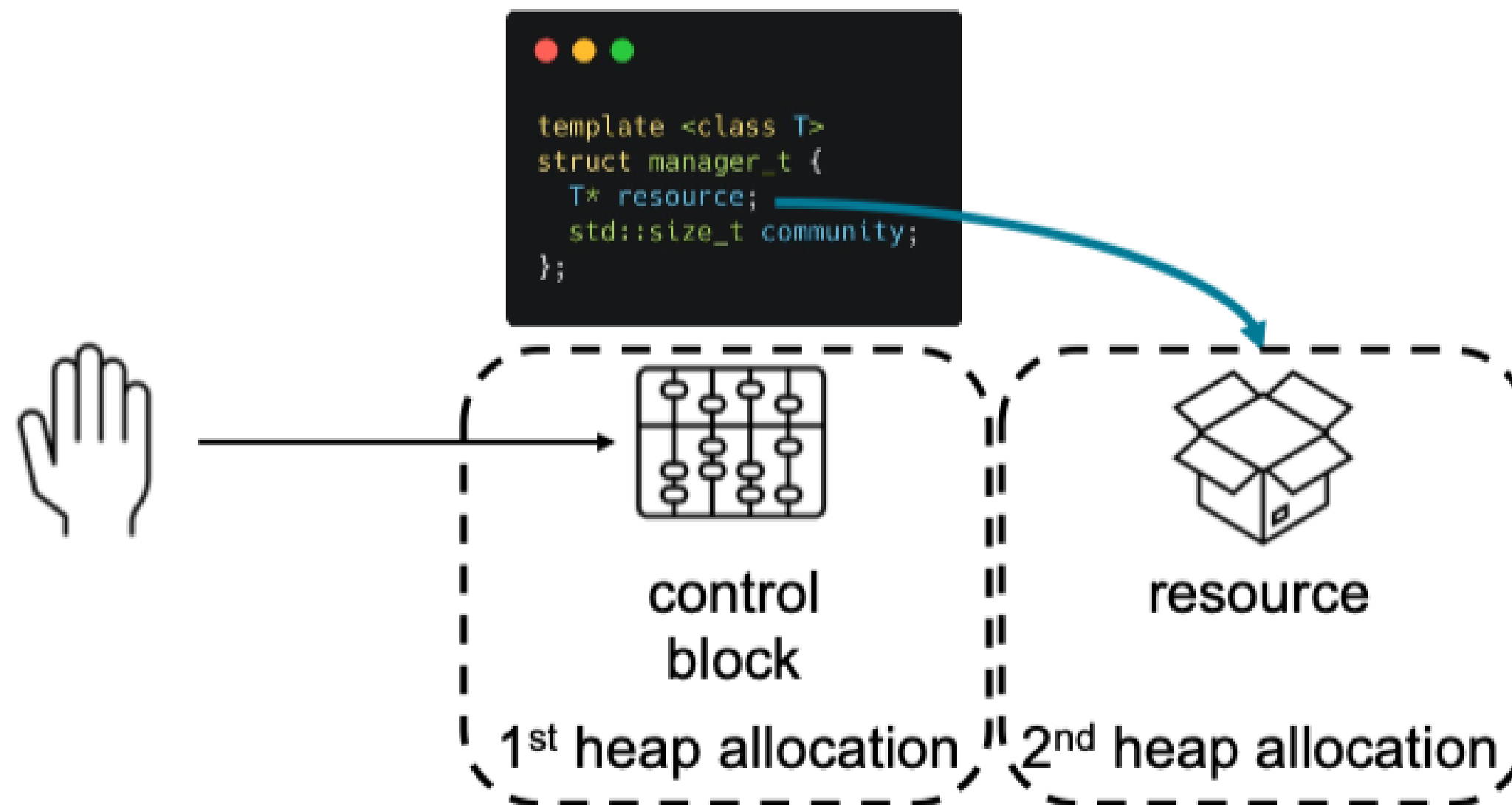
They are aka **reference counted smart pointers**, which definition already exposes their internal mechanism.



In shared ownership, the management responsibility is shared among the group, and just the last object alive, is allowed to actually destroy the resource.

shared_ptr<T> : the costs 1/2

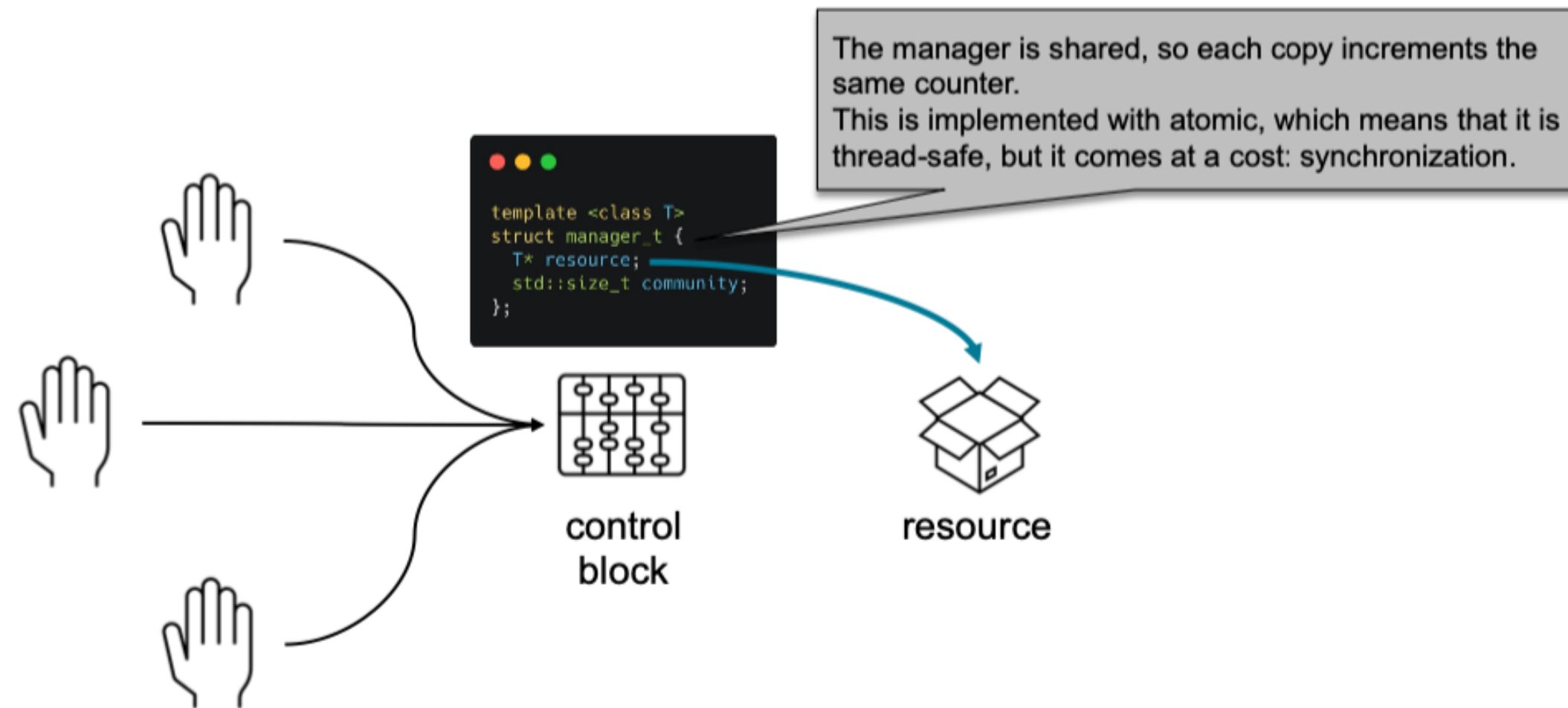
Heap allocations are not cheap 💰. Moreover having control block and resource allocated separately, might be expensive in case they end up far apart in memory.



Use `std::make_shared` which at least allocates both all at once, i.e. single allocation and close together.

shared_ptr<T> : the costs 2/2

Each time we copy the `shared_ptr`, we are working on a shared control block. This access is thread safe, so it requires a synchronization, which is expensive 💰.



*note: the control block is thread safe, not the resource usage!

Raw vs Smart pointers

~~Raw vs Smart pointers~~

Raw + Smart pointers

Raw pointers are really useful!

Smart pointers are not a one solution fits all, raw pointers are still very useful!

The main point to keep in mind is about *ownership*:

- Raw pointers (+ references) = non-owning
- Smart pointers = owning

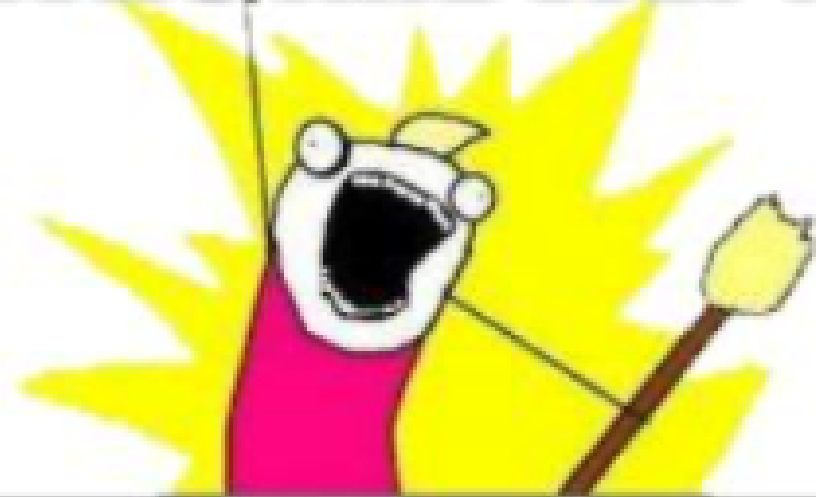
By using them correctly, you vehiculate a very important information via your API.

CPP Core Guidelines

- R.1: Manage resources automatically using resource handles and RAII
- R.3: A raw pointer (a T^*) is non-owning
- R.4: A raw reference (a $T\&$) is non-owning
- R.10: Avoid `malloc()` and `free()`
- R.11: Avoid calling `new` and `delete` explicitly
- R.20: Use `unique_ptr` or `shared_ptr` to represent ownership
- R.21: Prefer `unique_ptr` over `shared_ptr` unless you need to share ownership
- R.22: Use `make_shared()` to make `shared_ptr`s
- R.23: Use `make_unique()` to make `unique_ptr`s
- R.30: Take smart pointers as parameters only to explicitly express lifetime semantics

(source: <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#S-resource>)

WHO ARE WE??



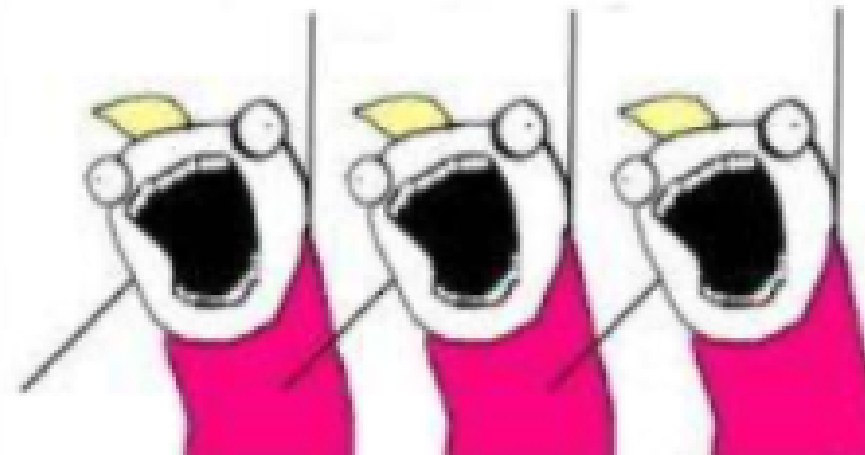
C++ PROGRAMMERS



WHAT DO WE WANT??



PERFORMANCE!




```

struct Dataset {
    Dataset() {
        std::cout << "Created dataset!\n";
    };
    ~Dataset() {
        std::cout << "Deleting dataset!\n";
    }
    Dataset(const Dataset&) {
        std::cout << "Create dataset copying GBs of data\n";
    }
    Dataset& operator=(const Dataset&) {
        std::cout << "Copying GBs of data\n";
        return *this;
    }
    void initialize() {
        std::cout << "Initialize dataset...\n";
    }
};

```

The semantic of this object is:

- Default C'tor create a dataset
- Data inside a dataset can be deep-copied

```
Dataset a;  
Dataset b;  
b = a;
```

```
Created dataset!  
Created dataset!  
Copying GBs of data  
Deleting dataset!  
Deleting dataset!
```

👍 Perfectly fine with it!

Two datasets created, one copy, two datasets destroyed.

✅ Nothing unexpected.

```
Dataset createDataset() {  
    Dataset x;  
    x.initialize();  
    return x;  
}  
  
Dataset b = createDataset();
```

```
Created dataset!  
Initialize dataset...  
Deleting dataset!
```

🤔 I would have expected:

- Two datasets created (`x` default, `b` copied)
- Two dataset destroyed

Wow! No copy?!? Indeed, there is no copy: it does not call the copy-c'tor! Thanks to **copy-elision**!

C++ IS SUPER! BEST PERFORMANCE!

Copy elision

*"(copy-elision) omits copy [...] **constructor**, resulting in zero-copy pass-by-value semantics."*

```
Dataset b = createDataset();
```

Is it a copy-constructor or a copy-assignment?

The copy assignment is a member function, so it has to be applied to an existing object.

But at this point the object does not exist yet, so even if it looks like an assignment, it is actually a constructor call, a copy-c'tor call!

Ok, from the code we expect a copy constructor call, and copy elision explicitly refers to the constructor and not to the assignment operator...

```
Dataset createDataset() {  
    Dataset x;  
    x.initialize();  
    return x;  
}  
Dataset b;  
b = createDataset();
```

```
Created dataset!  
Created dataset!  
Initialize dataset...  
Copying GBs of data  
Deleting dataset!  
Deleting dataset!
```

No copy-elision at the party, and performance are gone!

...is it so different than before?! A temporary dataset is created and, instead of using that (as in copy-elision case), it gets copied from, just before discarding it?!

C++ is stupid! Why can't it use temporary one also here?!

Why is it so stupid?!

It is not stupid, it just let you decide all details!

Remember: With C++ you have FULL CONTROL!

So here there is the plot-twist...you have control also over this specific case!

Can you spot the difference between the twos? We were talking about a "temporary"...

"deep-copy"

```
Dataset b;  
b = a;
```

"no copy-elision"

```
Dataset b;  
b = createDataset();
```

Can you see it? The main difference is that the temporary does not have a name!

Yes, C++ gives a knob also for this!

But, before seeing the handle, let's understand a bit better this question about temporaries...

In the beginning there was just LEFT and RIGHT...

Even if they are not 100% correct, these definitions are very good approximations.

- `lvalue`s can stay "**typically**" on the **left** side of `=`, and `rvalues` can "**typically**" stay on the **right**.
- `lvalue` is "**typically**" something with an **identity**, and `rvalue` has "**typically**" **no identity**

For them, the language offers two different kind of references that binds to them.

- `&` `lvalue` references
- `&&` `rvalue` references

An important detail:

`&&` (*rvalue reference*), extend lifetime of temporaries. Also `const&` (*const lvalue reference*) does.

Back to our performance problem...

Let's try to express our desired behavior in terms of `lvalues` and `rvalues`.

```
Dataset b;  
b = a;
```

```
Dataset b;  
b = createDataset();
```

Desiderata:

- we don't want to steal from an `lvalue`, because it is not a temporary and someone else might still use it
- we might want to steal from an `rvalue`, because it is a temporary and it is going to be destroyed

We used the word "steal", because a temporary object can get completely emptied. In C++ it is used the word "move", from which it originates the **move-semantic**, meaning that the "ownership" of a resource can be moved from one object to another.

What handles do we have?

Let's give another look at references that binds to `lvalues` and `rvalues`:

```
lvalue -> &  
rvalue -> &&
```

Actually, in the **copy c'tor** and in the **copy assignment operator** we use the `const&`, which is an `lvalue` reference...

```
Dataset(const Dataset&) and  
Dataset& operator=(const Dataset&)
```

What if we use an rvalue reference instead of an lvalue one?

```
Dataset(Dataset&&) and Dataset& operator=(Dataset&&)
```

We get a **move c'tor** and a **move assignment operator**!

```

struct Dataset {
    Dataset() {
        std::cout << "Created dataset!\n";
    };
    ~Dataset() {
        std::cout << "Deleting dataset!\n";
    }
    Dataset(const Dataset&) {
        std::cout << "Create dataset copying GBs of data\n";
    }
    Dataset& operator=(const Dataset&) {
        std::cout << "Copying GBs of data\n";
        return *this;
    }
    Dataset(Dataset&&) {
        std::cout << "Stole dataset\n";
    }
    Dataset& operator=(Dataset&&) {
        std::cout << "Stole dataset\n";
        return *this;
    }
    void initialize() {
        std::cout << "Initialize dataset...\n";
    }
};

```

Now `Dataset` is able to behave differently depending on the value category of the argument:

- `lvalue` -> copy c'tor or copy assignment operator
- `rvalue` -> move c'tor or move assignment operator

```
Dataset b;  
b = createDataset();
```

```
Created dataset!  
Created dataset!  
Initialize dataset...  
Stole dataset  
Deleting dataset!  
Deleting dataset!
```

Now the temporary gets stolen during the assignment.

Performance are back!

Rule of Five

Because the presence of a user-defined (or `= default` or `= delete`)

- *D'tor*
- *Copy C'tor*
- *Copy Assignment Operator*

any class for which move semantics are desirable, has to declare:

1. *D'tor*
2. *Copy c'tor*
3. *Copy assignment*
4. *Move c'tor*
5. *Move assignment*

Unlike Rule of Three, failing to implement move semantic is not an error, but a missed optimization opportunity.



This is just a part of the story...

We might want to move resources also from an `lvalue`, because we know it is going to be destroyed soon or it is not going to be used anymore.

C++ gives us an handle also for this! We can "move" ownership also from an `lvalue` with `std::move()` !

```
Dataset a;  
Dataset b;  
b = std::move(a);
```

```
Created dataset!  
Created dataset!  
Stole dataset  
Deleting dataset!  
Deleting dataset!
```

Note: When you call `std::move(a)`, after that call you cannot assume anything about `a`. It is in a *valid but unspecified state*.

`std::move` DOES NOT move

`std::move` tells that you "might move from" the object, but actually it does not move anything.

The "move" of the resources is up to the function that gets the `rvalue` reference, it might also not doing anything with it or just reading from it.

Indeed, `std::move` is just an unconditional cast from an lvalue reference to an rvalue reference!

It makes an `lvalue` appear like an `rvalue` !

Actually, the implementation is something very similar to this simplified snippet

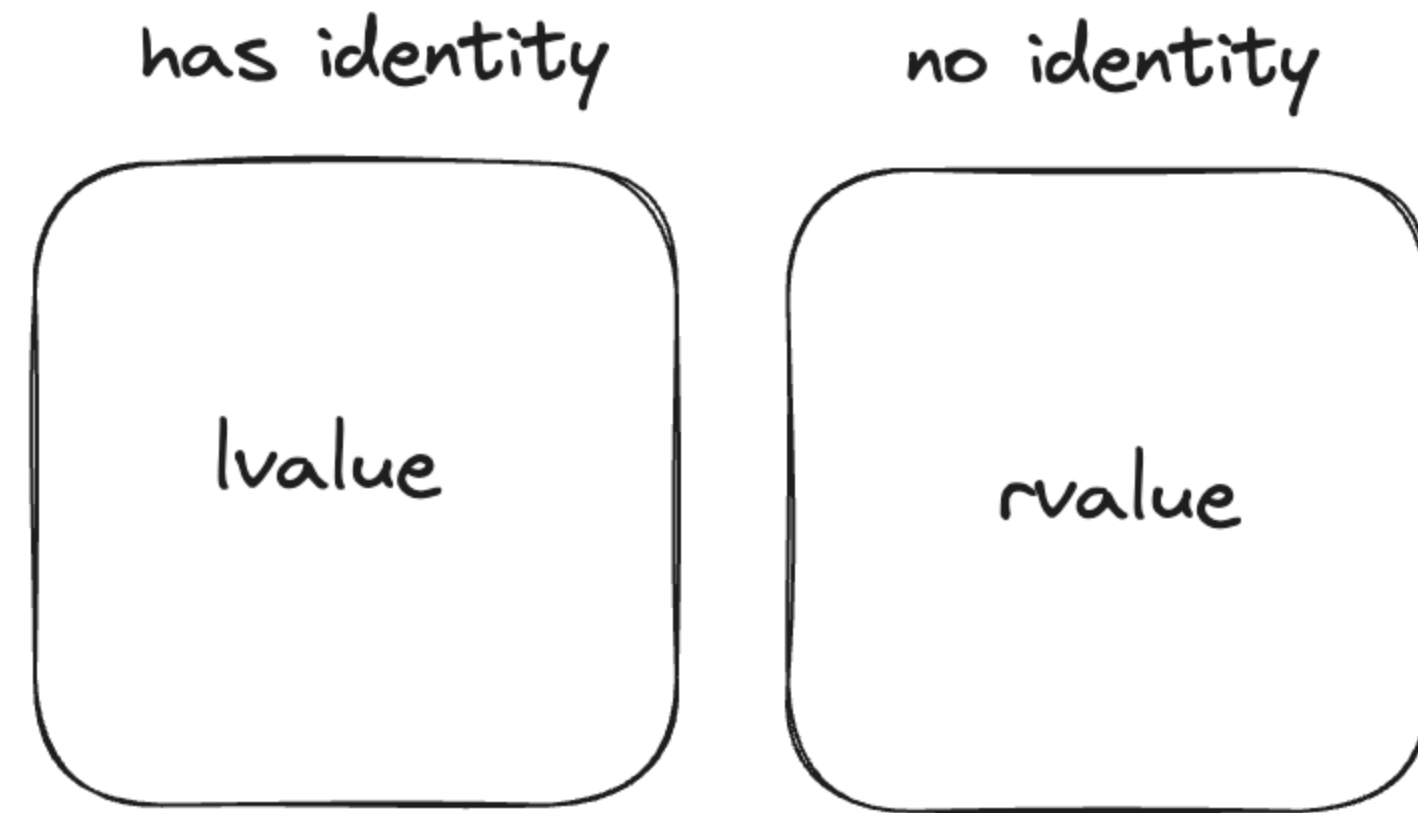
```
T&& std::move(T& lvalue) {  
    return static_cast<T&&>(lvalue);  
}
```

See actual LLVM libc++ implementation @

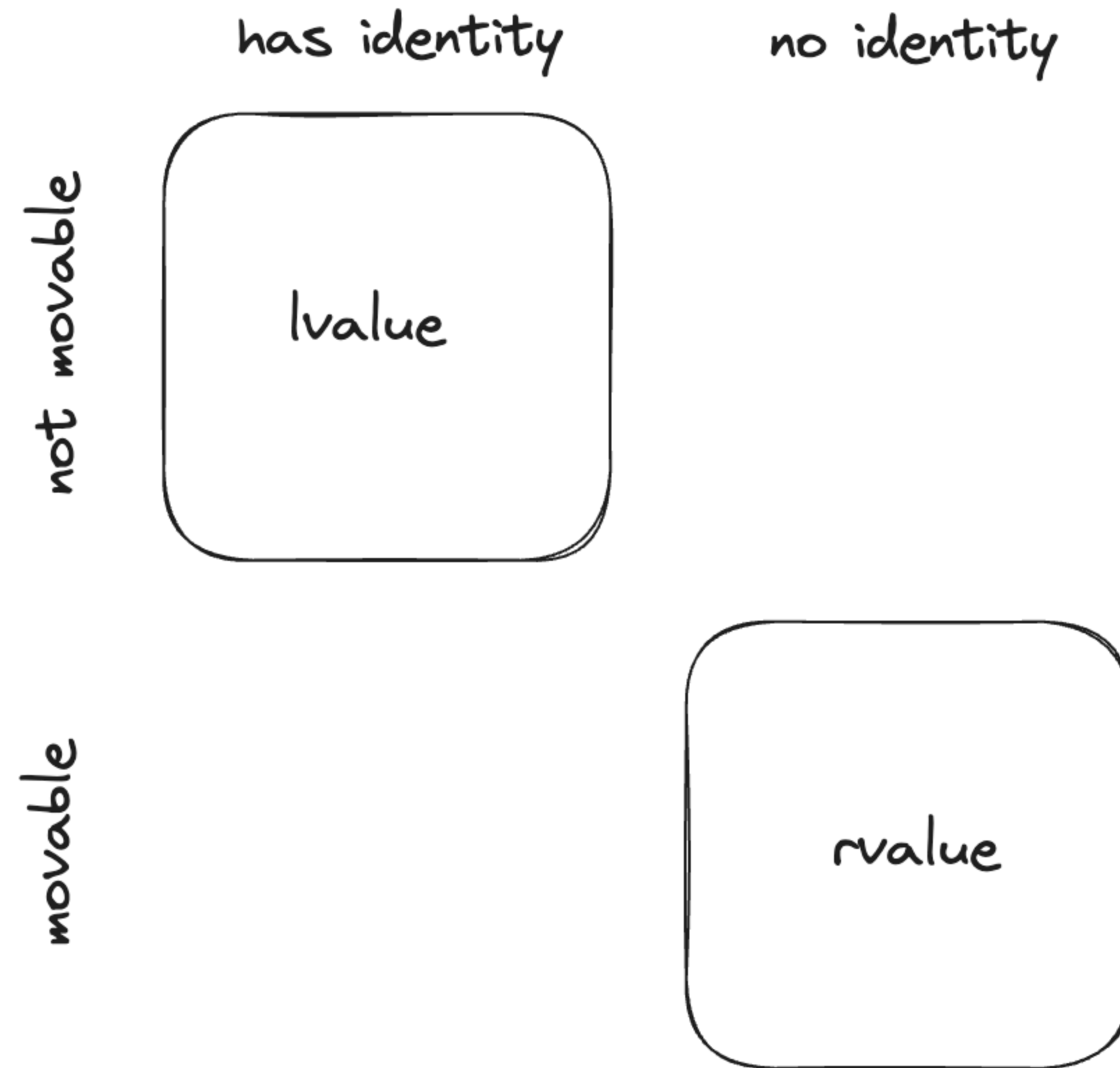
<https://github.com/llvm/llvm-project/blob/main/libc/src/support/CPP/utility/move.h>

Value categories

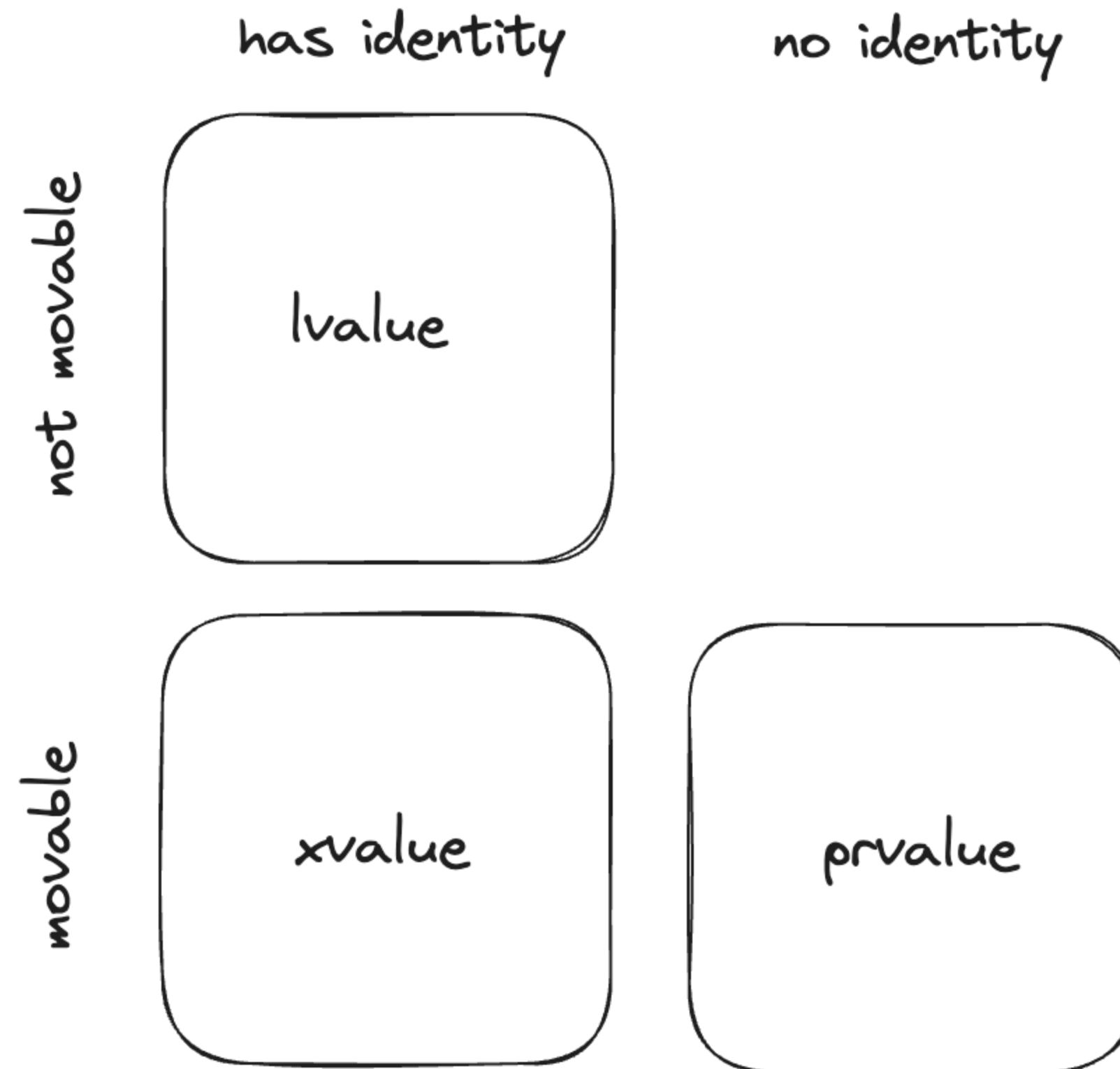
Value categories



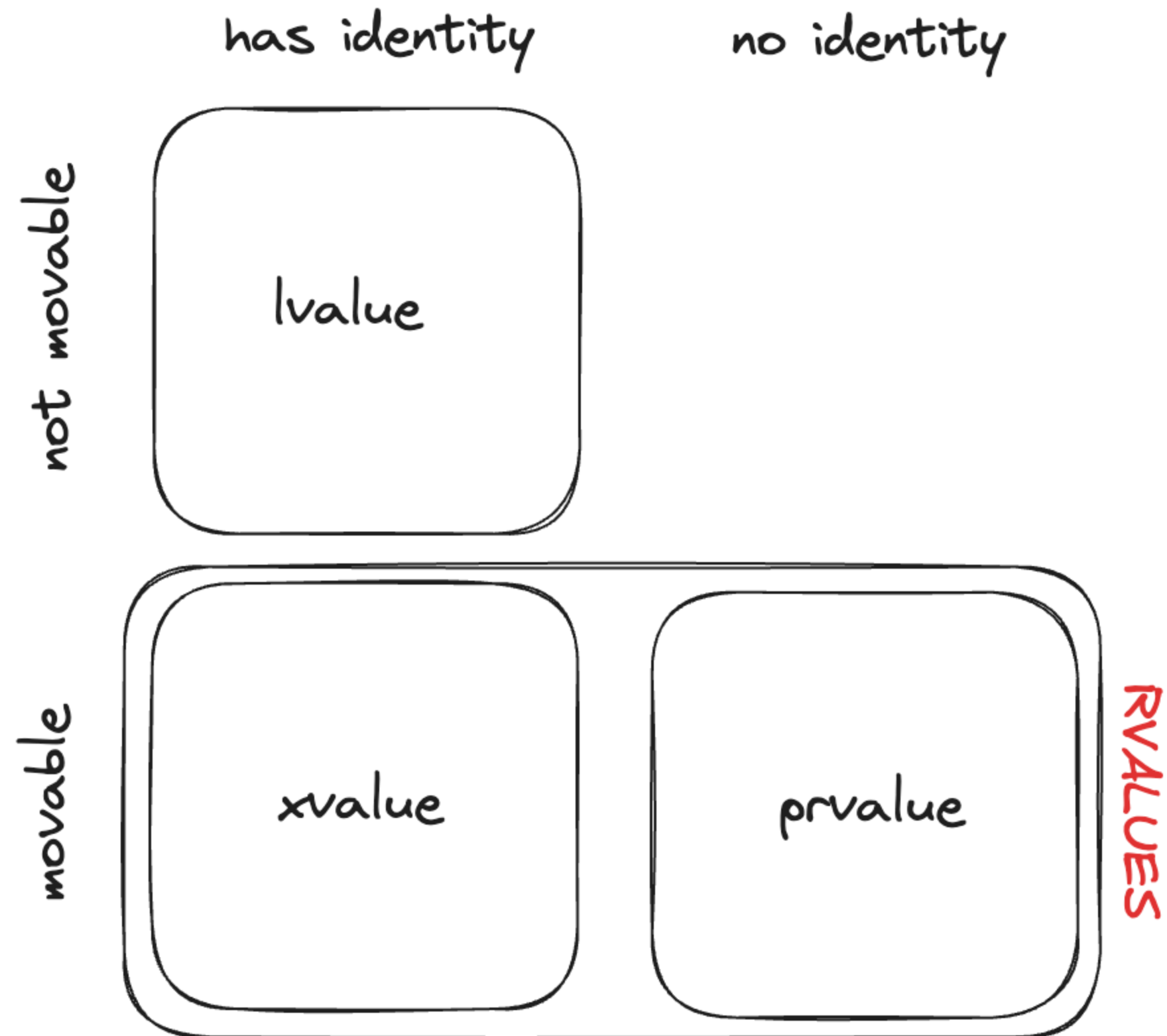
Value categories



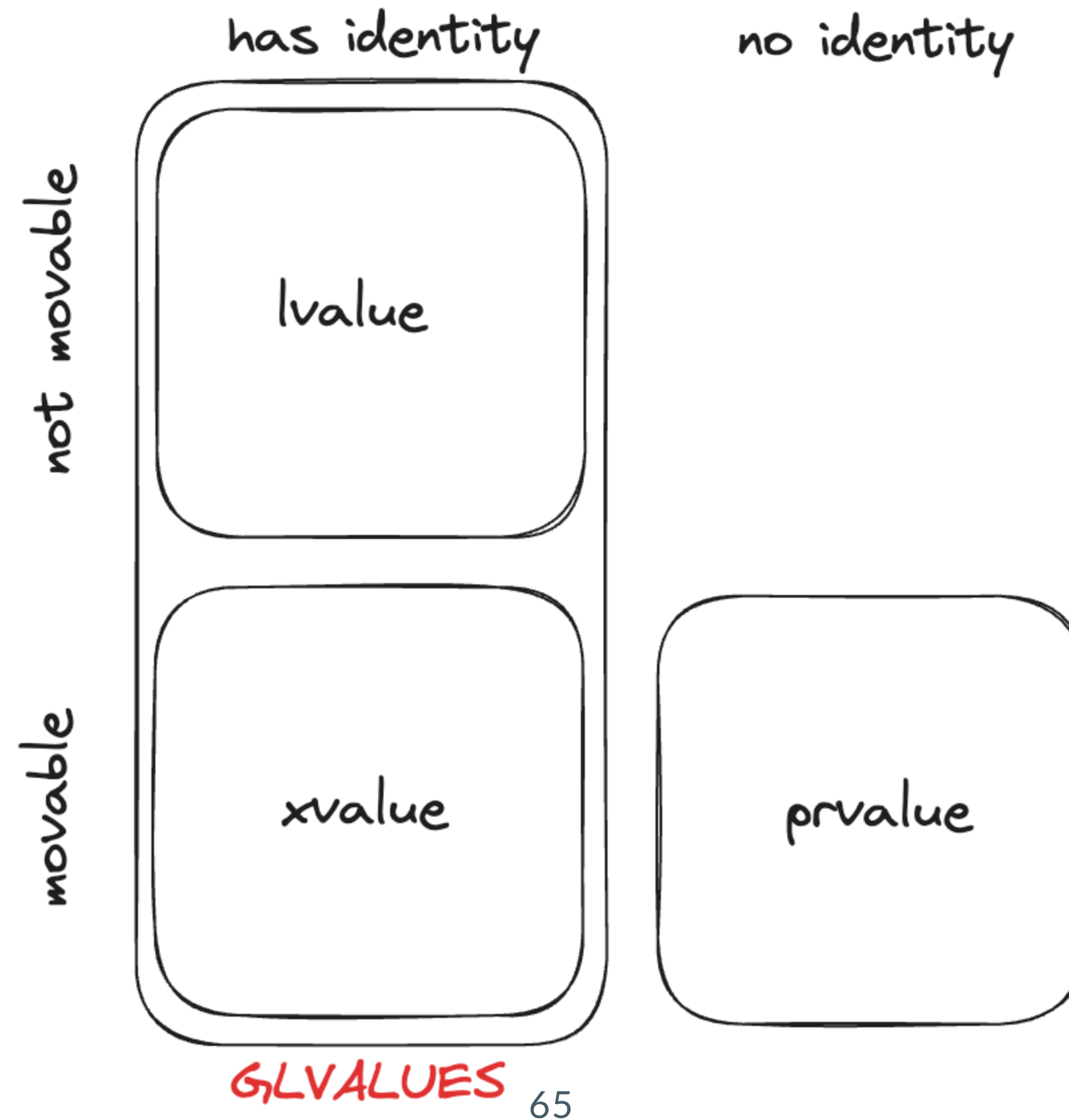
Value categories



Value categories



Value categories



Why lvalue / rvalue definitions in terms of is a good approximation?

From <https://en.cppreference.com/w/cpp/language/reference>

When a function's return type is lvalue reference, the function call expression becomes an lvalue expression:

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

char& char_number(std::string& s, std::size_t n) {
    return s.at(n);
}

int main()
{
    std::string str = "Test";
    char_number(str, 1) = 'a';
    std::cout << str << '\n';
}
```

Tast

Given RuleOf3 and RuleOf5, what's the next in the sequence?

- A. RuleOf7
- B. RuleOf0
- C. RuleOf8



Rule of Zero

Classes that have:

- custom destructors
- copy/move constructors or
- copy/move assignment operators

should deal exclusively with ownership

(which follows from the Single Responsibility Principle).

Other classes should not have custom destructors, copy/move constructors or copy/move assignment operators.

Let's say we hve C-style library

```
#include <iostream>
#include <memory>
#include <zmq.h>

int main (void) {
    void *context = zmq_ctx_new();
    void *requester = zmq_socket(context, ZMQ_REQ);

    zmq_connect(requester, "tcp://localhost:5555");
    zmq_send(requester, "Hello", 5, 0);

    zmq_close(requester);
    zmq_ctx_destroy(context);

    return 0;
}
```

We can apply RuleOf5!

```
struct ZmqContext {  
    ZmqContext(void* handle) : context_(handle) {}  
    ~ZmqContext() { zmq_ctx_destroy(context_); }  
  
    ZmqContext(const ZmqContext&) = delete;  
    ZmqContext& operator=(const ZmqContext&) = delete;  
  
    ZmqContext(ZmqContext&&) = default;  
    ZmqContext& operator=(ZmqContext&&) = default;  
private:  
    void* context_;  
};
```

```
int main () {  
    ZmqContext context = zmq_ctx_new();  
    ZmqSocket requester = zmq_socket(context, ZMQ_REQ);  
  
    zmq_connect(requester.get(), "tcp://localhost:5555");  
    zmq_send(requester.get(), "Hello", 5, 0);  
  
    return 0;  
}
```

Is it really worth it?
What about applying RuleOf0??

RuleOf0 rocks!

Actually we can use `std::unique_ptr` to express what kind of ownership we want to have for our object.

It's generic, it's reusable!

```
#include <iostream>
#include <memory>
#include <zmq.h>

struct ZmqContext {
    ZmqContext(void* handle) : context_(handle, zmq_ctx_destroy) {}
    void* get() const noexcept { return context_.get(); }

private:
    std::unique_ptr<void, decltype(&zmq_ctx_destroy)> context_;
};

struct ZmqSocket {
    ZmqSocket(void* handle) : socket_(handle, zmq_close) {}
    void* get() const noexcept { return socket_.get(); }

private:
    std::unique_ptr<void, decltype(&zmq_close)> socket_;
};
```

```
int main () {
    ZmqContext context = zmq_ctx_new();
    ZmqSocket requester = zmq_socket(context, ZMQ_REQ);

    zmq_connect(requester.get(), "tcp://localhost:5555");
    zmq_send(requester.get(), "Hello", 5, 0);

    return 0;
}
```

In the end it is just

RuleOf0 vs RuleOf5

In modern C++ (\geq C++11), where move-semantic has been introduced, RuleOf3 has practically been superseded by RuleOf5.

Moreover, as we have seen, STL provides generic objects encapsulating ownership policies, e.g. `std::unique_ptr` and `std::shared_ptr`, allowing us to completely relying on them for the management.

For this reason the guideline is

"prefer applyig RuleOf0 if possible, otherwise fallback to RuleOf5"

[C++ Core Guidelines - C.20](#)

Conclusion/Recap

- Introduction to RAll and Ownership
- RuleOfThree
- Smart Pointers
- Move semantic as an optimization chance
- RuleOfFive
- Value categories
- RuleOfZero

Q&A

Thanks

Alberto Invernizzi
Research Software Engineer @ CSCS