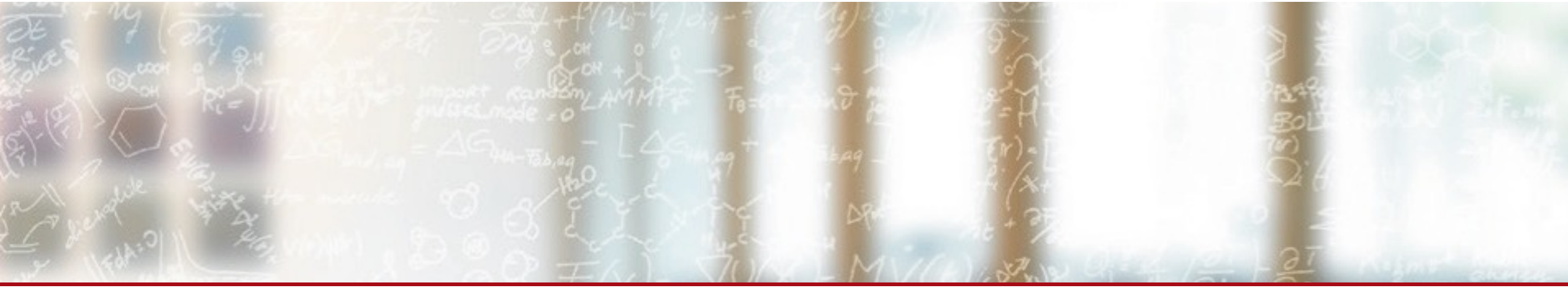
CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich



Resource management

About ownerhsip, raw pointers, smart pointers and guidelines

Advanced C++

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

October 12, 2021

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

- 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.”

Resources can be very different:

- Memory allocation
- 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(=nightmare) a race-condition.

Managing the lifetime of a resource in an object-oriented context, where objects are created, manipulated, and they interact with other parts of the program by "going around" to exchange information, easily becomes difficult.

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 (read it as impossible).

FULL CONTROL != DO IT MANUALLY

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

The language, through the compiler, is at our disposal, we can 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 and writing

READABLE, CORRECT and EFFICIENT code.

Let's start from the main "handles" we have for controlling object life.

```
class Example {
  Example();

  Example(...);

  Example(const Example&);
  Example& operator=(const Example&);

  Example(Example&&);
  Example& operator=(Example&&);

  ~Example();
}
```

With these you can define what happens

- C'tors + D'tor
  when an object is created/destroyed, i.e. deal with actual resource management (**RAII**)

- CopyC'tor + MoveC'tor (+ operator=)
  when it is passed around (copied/moved)*, i.e. deal with resource **ownership**

**note*
value vs reference semantic
not every object has the same semantic, i.e. copying an object does not always mean the same thing for every object.

Ownership

A fundamental concept introduced with RAII is the one of OWNERSHIP.

With RAII an object starts representing the ownership of the resource, so it 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.

Ownership

About ownership, it is possible to differentiate:

- **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.

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

This requires some machinery, which we will quickly see later.

Use-case with pointers

Raw Pointers

```
// HEAP
int* pointer             = new int(26);
int* pointer_array       = new int[13];

// STACK
int value = 26;
int *pointer_on_stack    = &value;
```

Are they the right tool for managing resources?

Problem: who is responsible?

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

Problem: how should it be released?

How was it allocated?

- `new` → `delete`
- `new[]` → `delete[]`
- `malloc` → `free`

Problem: burden of the management

Ok, I will take care of that…

- Remember to do it
- Do it in the correct order (e.g. dependencies between resources)

Problem: have you considered all execution paths?

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

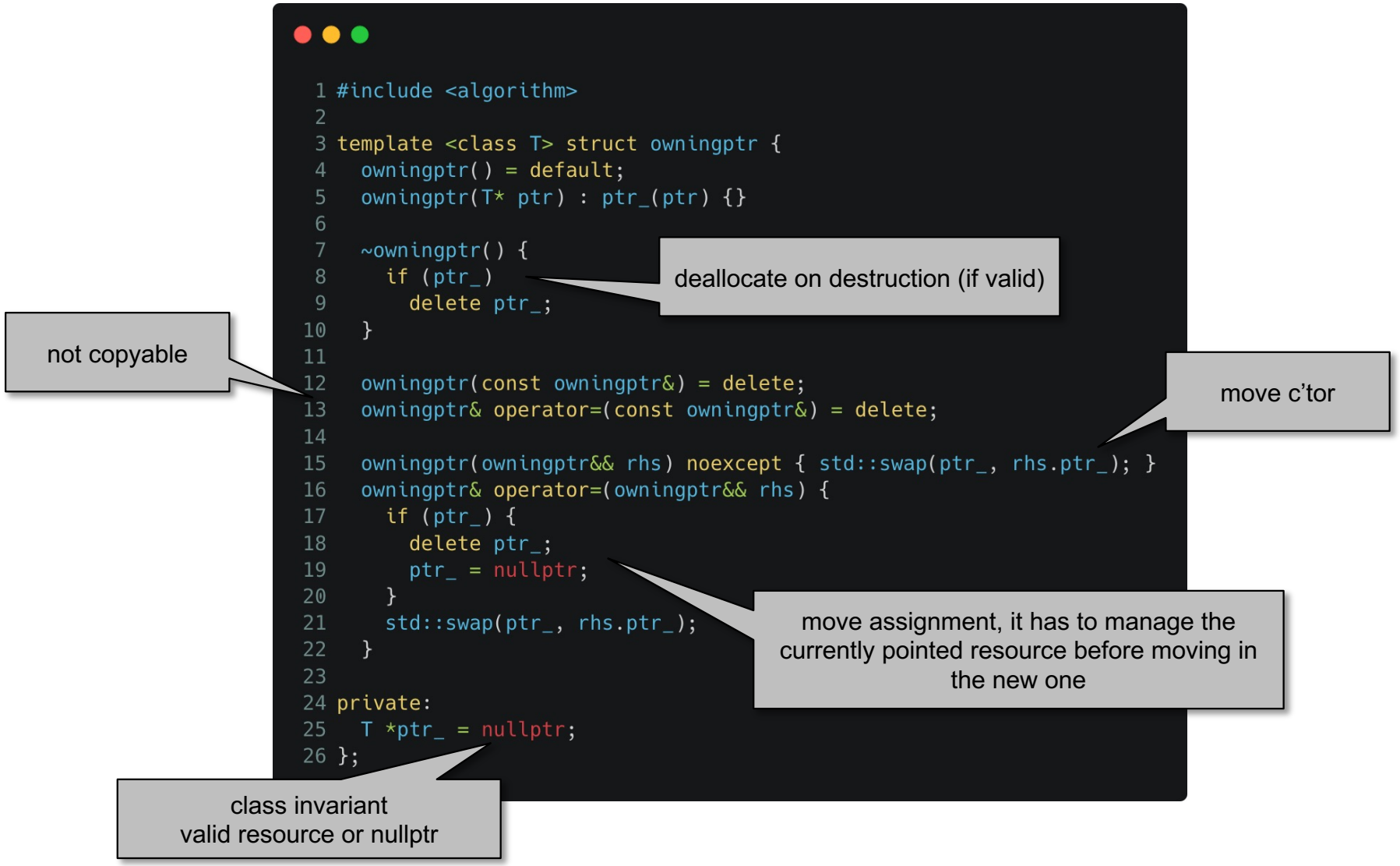
Problem: … even exceptions?

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

Raw pointers and references do not express ownership.

What if we could have an object which uses the RAII technique and that allow us to avoid all these problems…

An exclusive ownership pointer



STL Smart Pointers

STL provides the solution in the `<memory>` header

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

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_ptr`s 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)

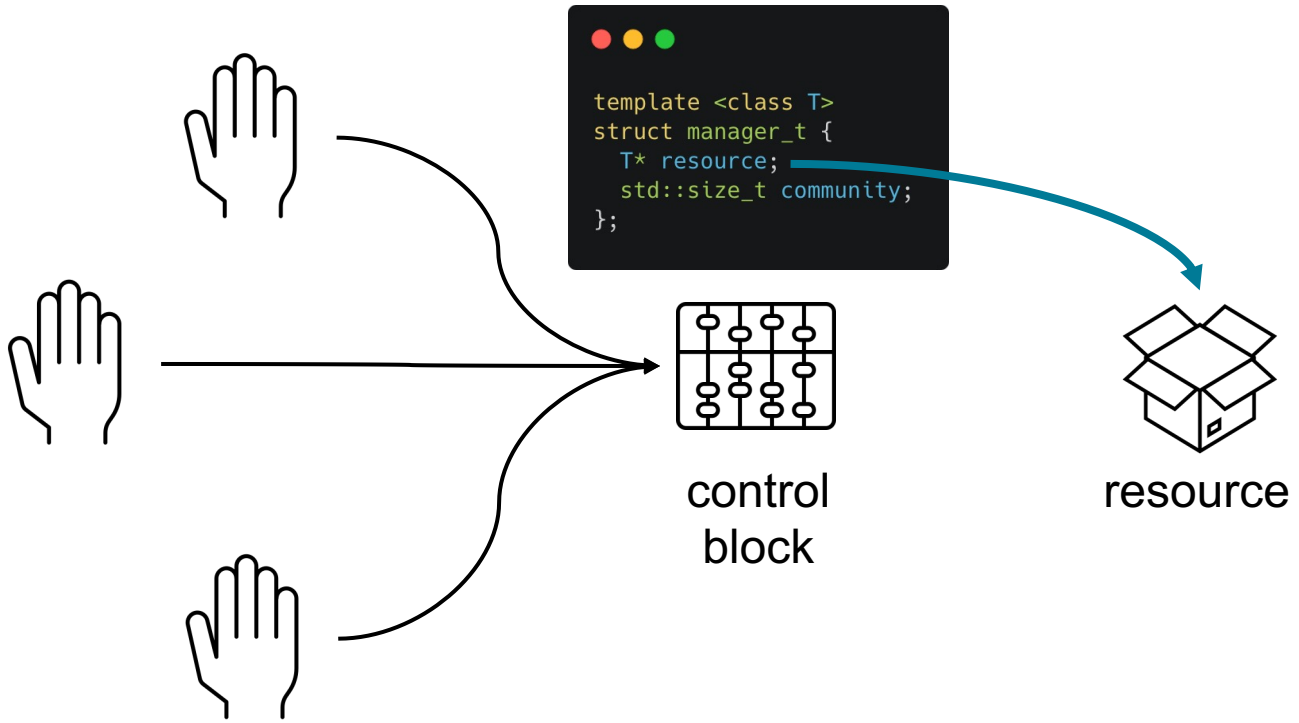
std::shared_ptr<T>

The copyability allow to extend the ownership group.

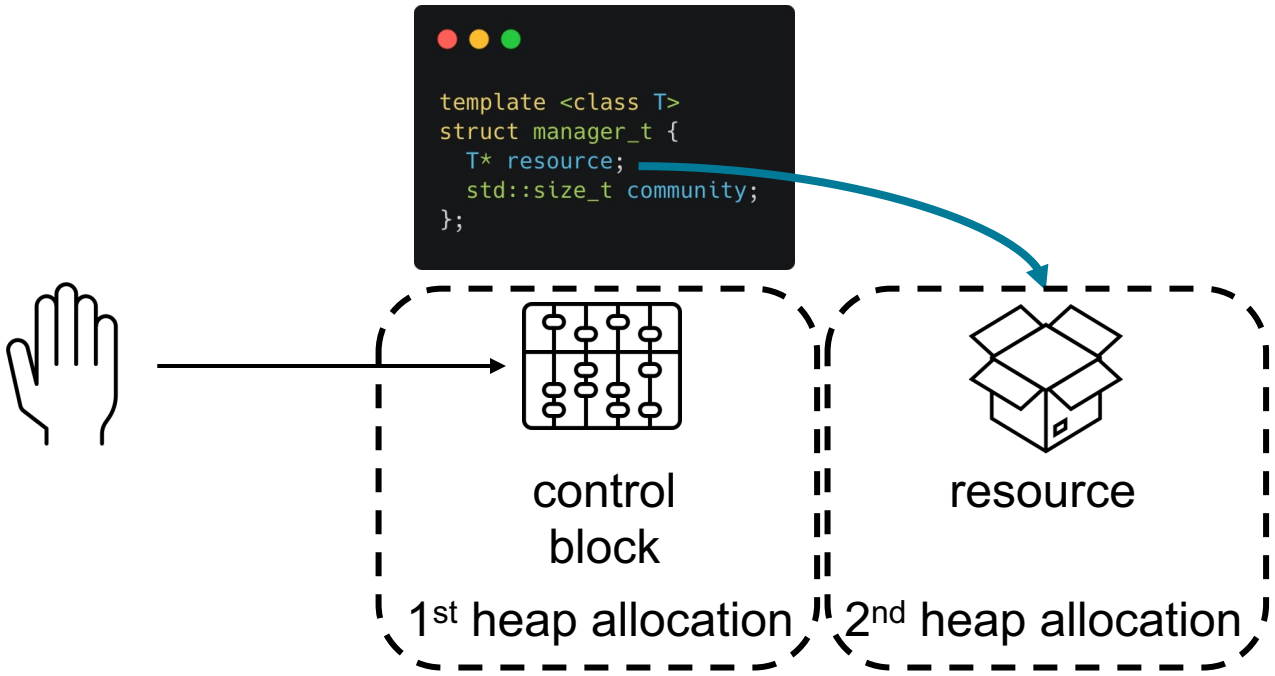
As soon as you get a new instance, the group of owners is extended.

Indeed, they are aka reference counted smart pointer, which exposes how they are internally implemented.

shared_ptr<T>: the machinery

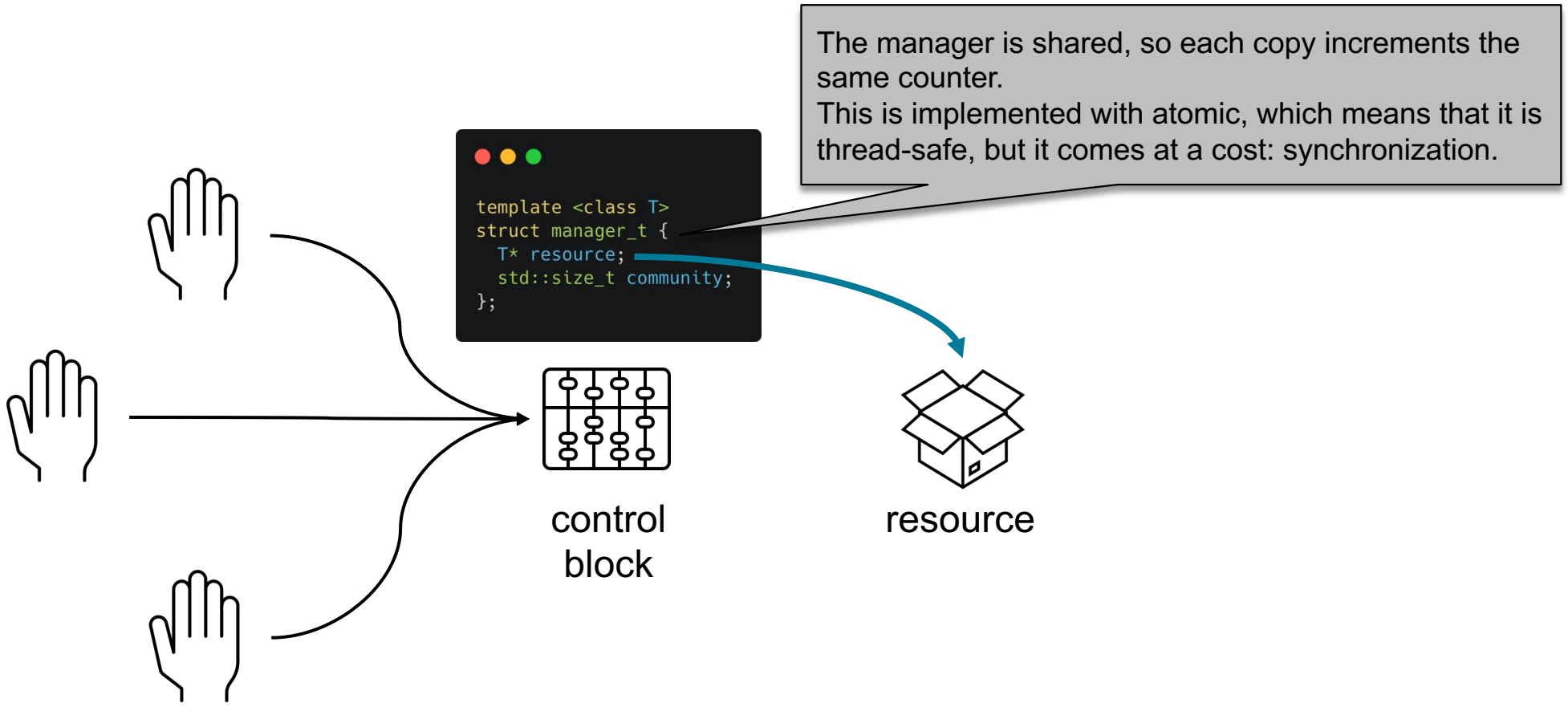


shared_ptr<T>: the costs 1/2



Use **std::make_shared** which at least allocates both all at once

shared_ptr<T>: the costs 2/2



The control block is thread safe, not the resource usage!

Smart Pointers

- Use `make_unique`/`make_shared`

```cpp
std::unique_ptr<int> unique_with_new(new int(13));
std::unique_ptr<int[]> unique_array_with_new(new int[5]);

std::unique_ptr<int> unique = std::make_unique<int>(13);
std::unique_ptr<int[]> unique_array = std::make_unique<int[]>(5);
```

- If you are not sure about ownership, use `unique_ptr`

```cpp
std::shared_ptr<int> shared(std::move(unique));
// shared = std::move(unique);
assert(not unique);
```

Raw vs Smart pointers

~~Raw vs Smart~~ Raw + Smart pointers

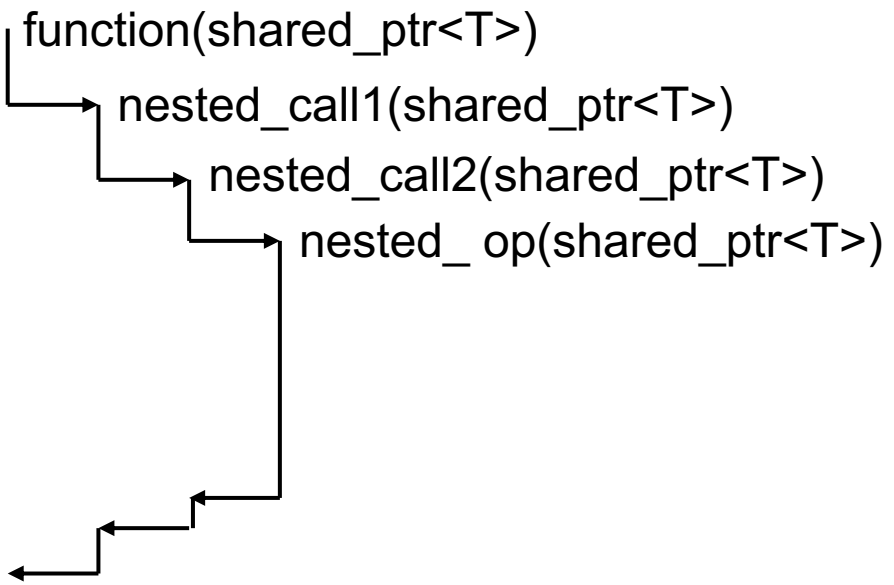
Raw pointers are really useful!

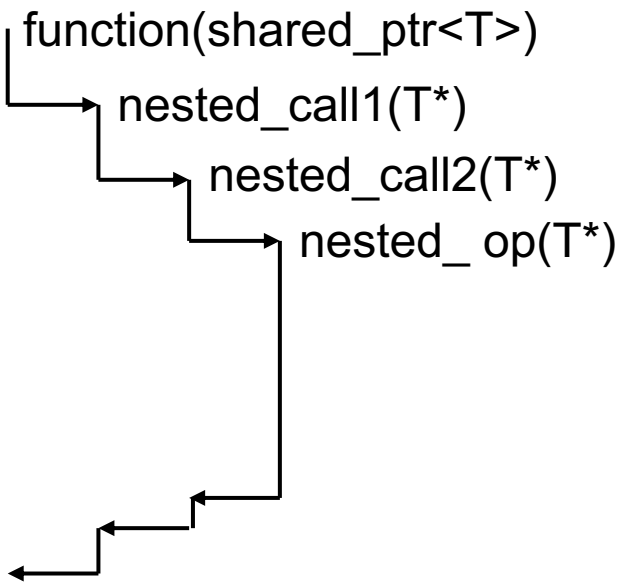
Smart pointers are not a one solution for all, raw pointers are still very useful!

The main point to keep in mind is

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

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


function(shared_ptr<T>)
nested_call1(shared_ptr<T>)
nested_call2(shared_ptr<T>)
nested_ op(shared_ptr<T>)


function(shared_ptr<T>)
nested_call1(T*)
nested_call2(T*)
nested_ op(T*)

CPP Core Guidelines 1/2

- Resource management rule summary:
  - R.1: Manage resources automatically using resource handles and RAII (Resource Acquisition Is Initialization)
  - R.2: In interfaces, use raw pointers to denote individual objects (only)
  - R.3: A raw pointer (a `T*`) is non-owning
  - R.4: A raw reference (a `T&`) is non-owning
  - R.5: Prefer scoped objects, don't heap-allocate unnecessarily
  - R.6: Avoid non-`const` global variables
- Allocation and deallocation rule summary:
  - R.10: Avoid `malloc()` and `free()`
  - R.11: Avoid calling `new` and `delete` explicitly
  - R.12: Immediately give the result of an explicit resource allocation to a manager object
  - R.13: Perform at most one explicit resource allocation in a single expression statement
  - R.14: Avoid `[]` parameters, prefer `span`
  - R.15: Always overload matched allocation/deallocation pairs

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

CPP Core Guidelines 2/2

- Smart pointer rule summary:
  - 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.24: Use `std::weak_ptr` to break cycles of `shared_ptr`s
  - R.30: Take smart pointers as parameters only to explicitly express lifetime semantics
  - R.31: If you have non-`std` smart pointers, follow the basic pattern from `std`
  - R.32: Take a `unique_ptr<widget>` parameter to express that a function assumes ownership of a `widget`
  - R.33: Take a `unique_ptr<widget>&` parameter to express that a function reseats the `widget`
  - R.34: Take a `shared_ptr<widget>` parameter to express that a function is part owner
  - R.35: Take a `shared_ptr<widget>&` parameter to express that a function might reseat the shared pointer
  - R.36: Take a `const shared_ptr<widget>&` parameter to express that it might retain a reference count to the object ???
  - R.37: Do not pass a pointer or reference obtained from an aliased smart pointer

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

Guideline Passing Parameters

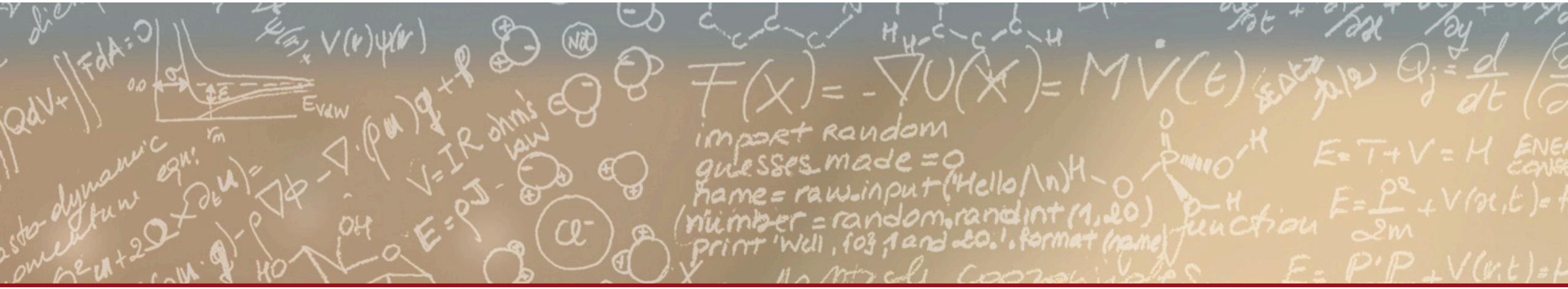
<table>
<tr><th></th><th>Cheap or impossible to copy (e.g., int, unique_ptr)</th><th>Cheap to move (e.g., vector<T>, string) or Moderate cost to move (e.g., array<vector>, BigPOD) or Don't know (e.g., unfamiliar type, template)</th><th>Expensive to move (e.g., BigPOD[], array<BigPOD>)</th></tr>
<tr><td>Out</td><td colspan="2">X f()</td><td rowspan="2"></td></tr>
<tr><td>In/Out</td><td colspan="2">f(X&)*</td></tr>
<tr><td>In</td><td rowspan="2">f(X)</td><td rowspan="2">f(const X&)</td><td rowspan="2"></td></tr>
<tr><td>In & retain "copy"</td></tr>
</table>

"Cheap" ≈ a handful of hot int copies
"Moderate cost" ≈ memcpy hot/contiguous ~1KB and no allocation

** or return unique_ptr<X>/make_shared_<X> at the cost of a dynamic allocation*

(source: https://www.youtube.com/watch?v=xnqTKD8uD64&t=3317s)

(source: https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rf-conventional)



Thank you for your attention.