Trivially Relocatable

Arthur O’Dwyer
2019-05-09

PART 1:

- What is relocation (move+destroy)?
  - Applications for reliable detection of trivial relocatability [5–44]
- Prior art (Folly, EASTL, BSL) [45–55]
  - And why it's fragile and error-prone [56–68]
- P1144 definition [69–90]

PART 2:

- Versus past and current proposals [91–98, 99–103, 104–109, 110–114]
- Versus "persistent memory" [115–127]
- Open questions [128–129]

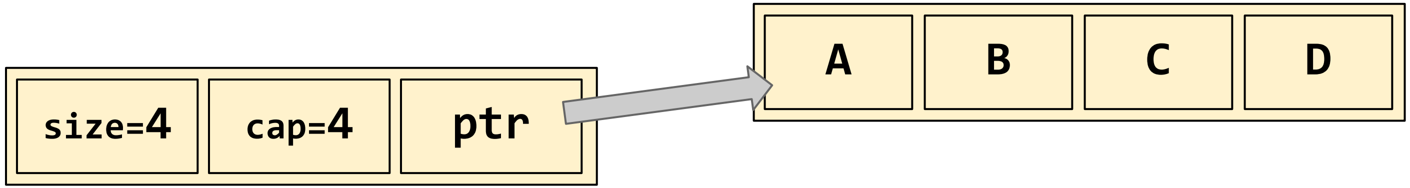
BONUS SLIDES [130–156]

Hey look!
Slide numbers!

Motivating “relocation”

Consider what happens when we resize a `std::vector<T>`.

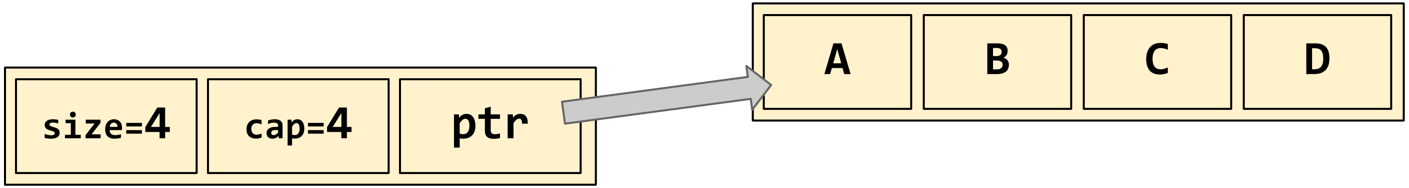
```
std::vector<T> vec { A, B, C, D };
```



Motivating “relocation”

Consider what happens when we resize a `std::vector<T>`.

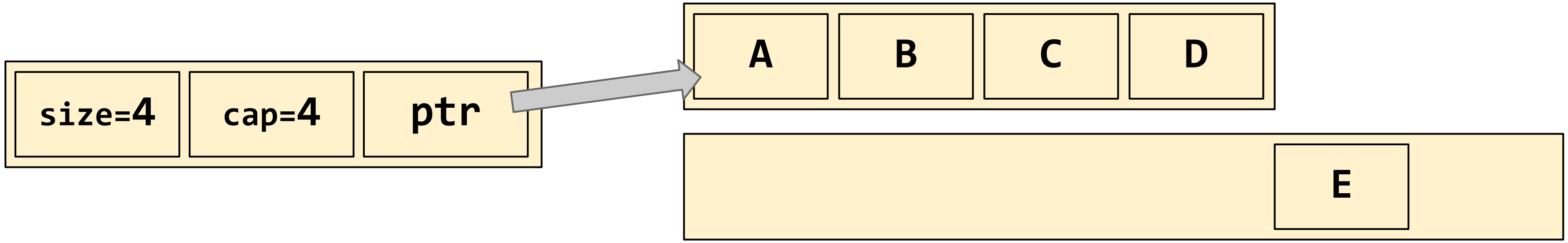
```
std::vector<T> vec { A, B, C, D };
vec.push_back(E);
```



Motivating “relocation”

Consider what happens when we resize a `std::vector<T>`.

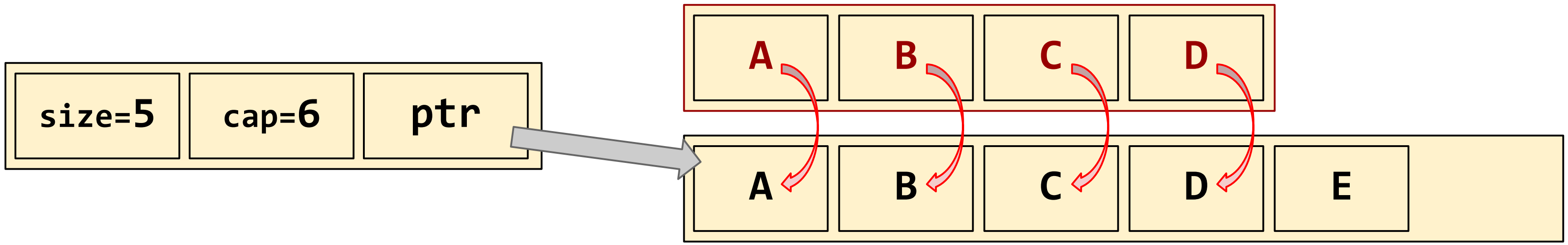
```
std::vector<T> vec { A, B, C, D };
vec.push_back(E);
```



Motivating “relocation”

Consider what happens when we resize a `std::vector<T>`.

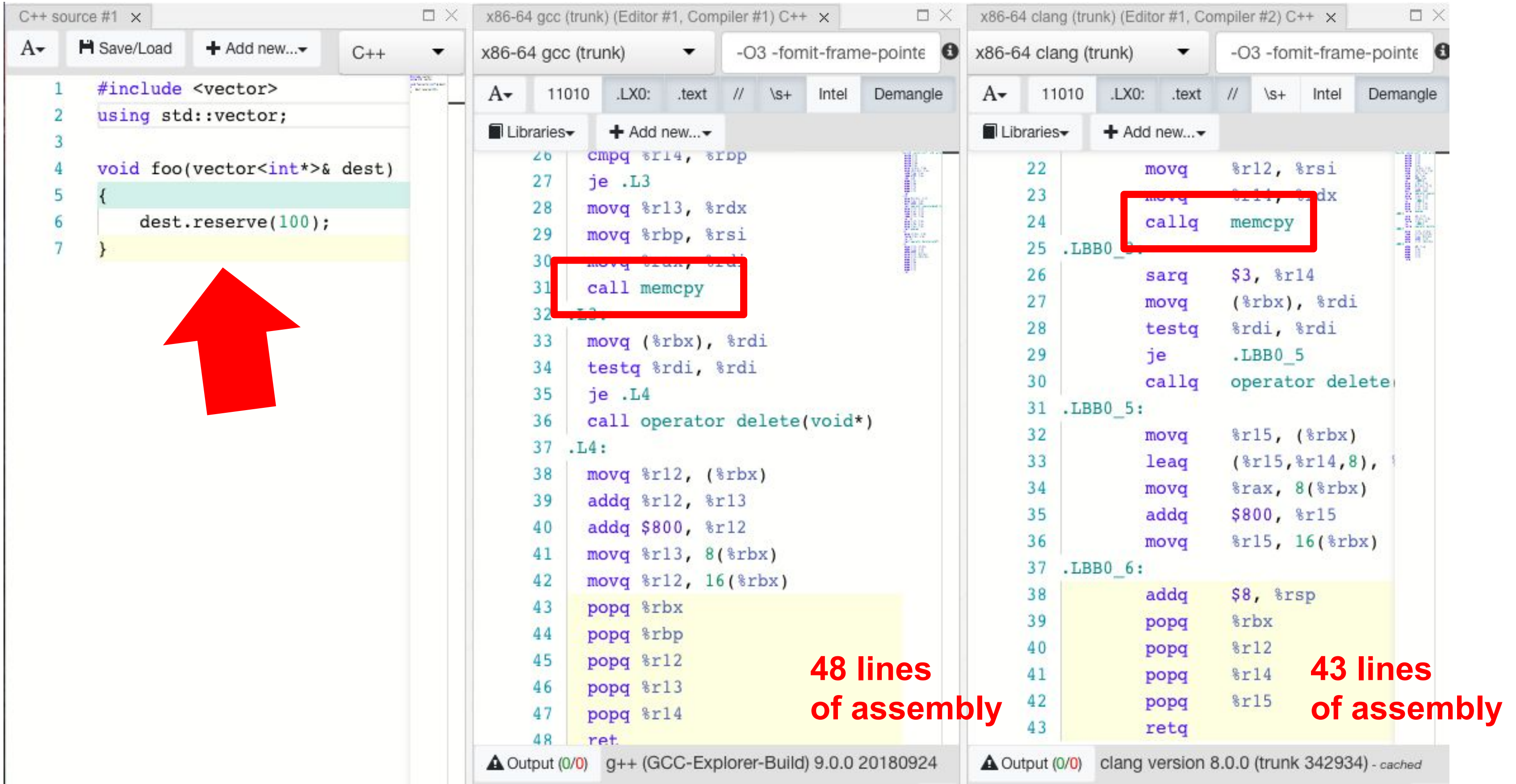
```
std::vector<T> vec { A, B, C, D };
vec.push_back(E);
```



The “relocation” of objects `A`, `B`, `C`, `D` involves 4 calls to the move-constructor, followed by 4 calls to the destructor.

https://godbolt.org/z/wxVF4C

Relocating trivially copyable `int*`



https://godbolt.org/z/qVeOnc

Relocating non-trivial shared_ptr

A▾ Save/Load + Add new... C++

```cpp
#include <memory>
#include <vector>
using std::shared_ptr;
using std::vector;

using P = shared_ptr<int>;

void foo(vector<P>& dest)
{
    dest.reserve(100);
}
```

x86-64 gcc (trunk) -O3 -fomit-frame-pointe

A▾ 11010 .LX0: .text // \s+ Intel Demangle

Libraries▾ + Add new...▾

```
  jne .L34
.L16:
  testq %r15, %r15
  je .L17
  movl $-1, %eax
  lock xaddl %eax, 12(%rbx)
.L18:
  cmpl $1, %eax
  jne .L12
  movq (%rbx), %rax
  movq %rcx, 8(%rsp)
  movq %rbx, %rdi
  movq 24(%rax), %rdx
  cmpq $std::_Sp_counted_base<(
  jne .L19
  call *8(%rax)
  movq 8(%rsp), %rcx
  addq $16, %rbp
  cmpq %rbp, %rcx
  jne .L8
.L32:
  movq 0(%r13), %rbp
```

138 lines of assembly

x86-64 clang (trunk) -O3 -fomit-frame-pointe

A▾ 11010 .LX0: .text // \s+ Intel Demangle

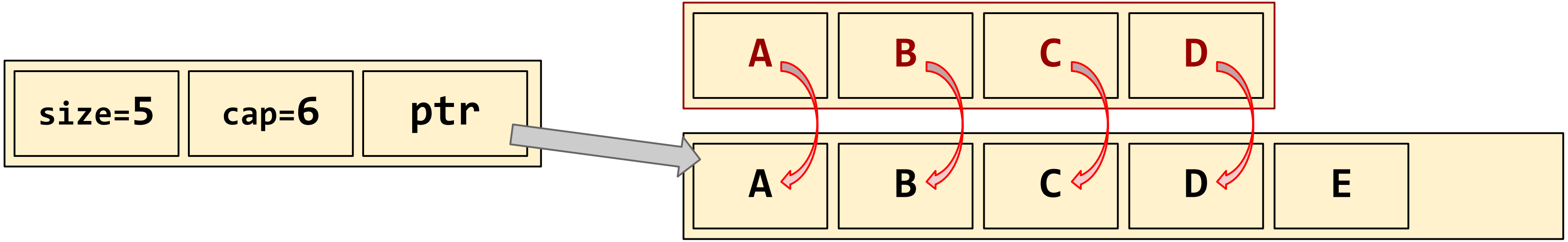
Libraries▾ + Add new...▾

```
        je      .LBB1_24
        movq    %r13, 8(%rsp)
        movq    %r15, 16(%rsp)
        movl    $__pthread_key_c
.LBB1_12:
        movq    8(%rbx), %r13
        testq   %r13, %r13
        je      .LBB1_22
        testq   %r15, %r15
        je      .LBB1_15
        movl    $-1, %eax
        lock            xaddl
        cmpl    $1, %eax
        je      .LBB1_17
        jmp     .LBB1_22
.LBB1_15:
        movl    8(%r13), %eax
        leal    -1(%rax), %ecx
        movl    %ecx, 8(%r13)
        cmpl    $1, %eax
        jne     .LBB1_22
.LBB1_17:
```

159 lines of assembly

Relocating non-trivial types

In principle, we ***can*** implement the “relocation” of objects `A`, `B`, `C`, `D` here with a simple memcpy. `shared_ptr`’s move constructor is non-trivial, and its destructor is also non-trivial, but if we always call them together, the ***result*** is tantamount to memcpy.



The operation of “calling the move-constructor and the destructor together in pairs” is known as ***relocation***.
A type whose relocation operation is tantamount to memcpy is ***trivially relocatable***.

Not all types are trivially relocatable

A type whose relocation operation is ***not*** tantamount to memcpy is called ***non-trivially relocatable***.

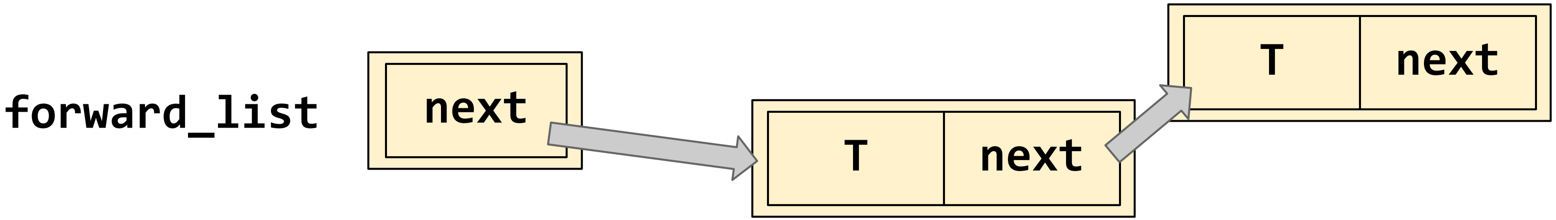
Although many everyday types are trivially relocatable, there do exist non-trivially relocatable types.

Some of them are pretty common. For example, libc++’s `std::list`.

Let’s compare libc++’s `std::forward_list<int>` and `std::list<int>`.

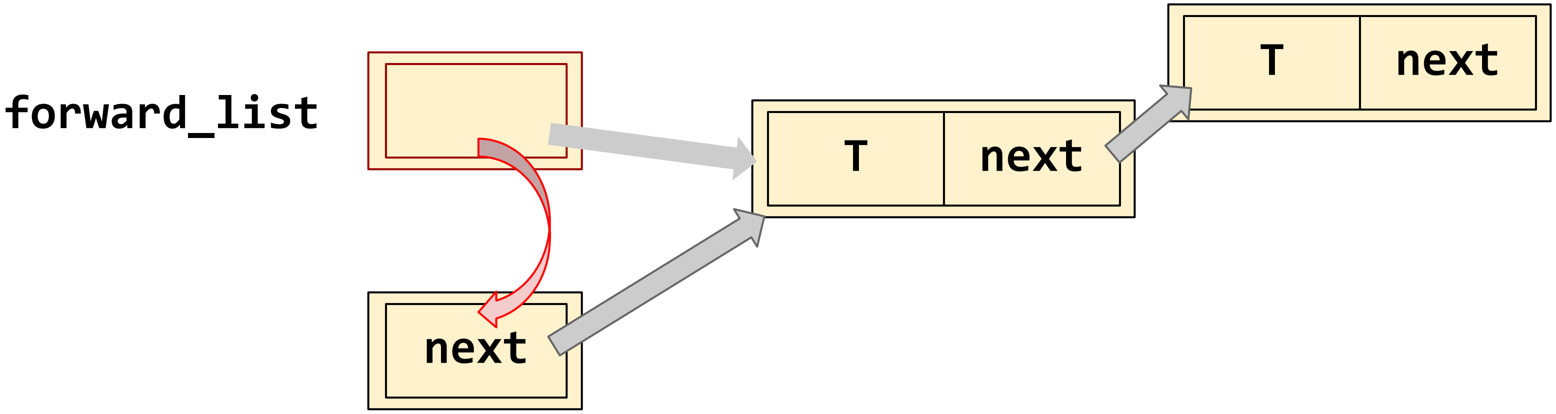
Not all types are trivially relocatable

libc++'s `std::forward_list<T>` (with `std::allocator`) is trivially relocatable.



Not all types are trivially relocatable

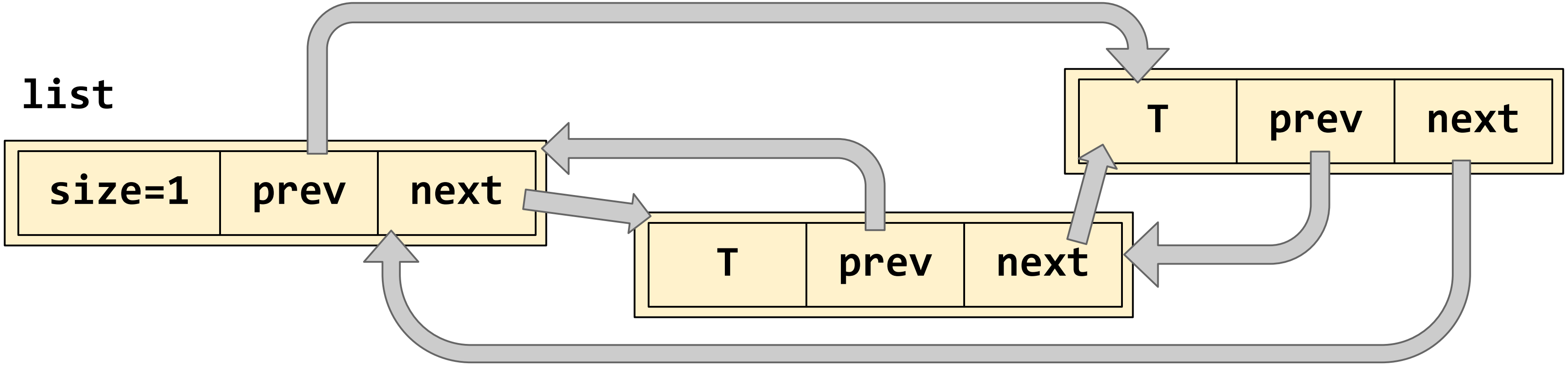
libc++'s `std::forward_list<T>` (with `std::allocator`) is trivially relocatable.



: The memory may still hold that bit-pattern, but the C++ object's lifetime is already over.

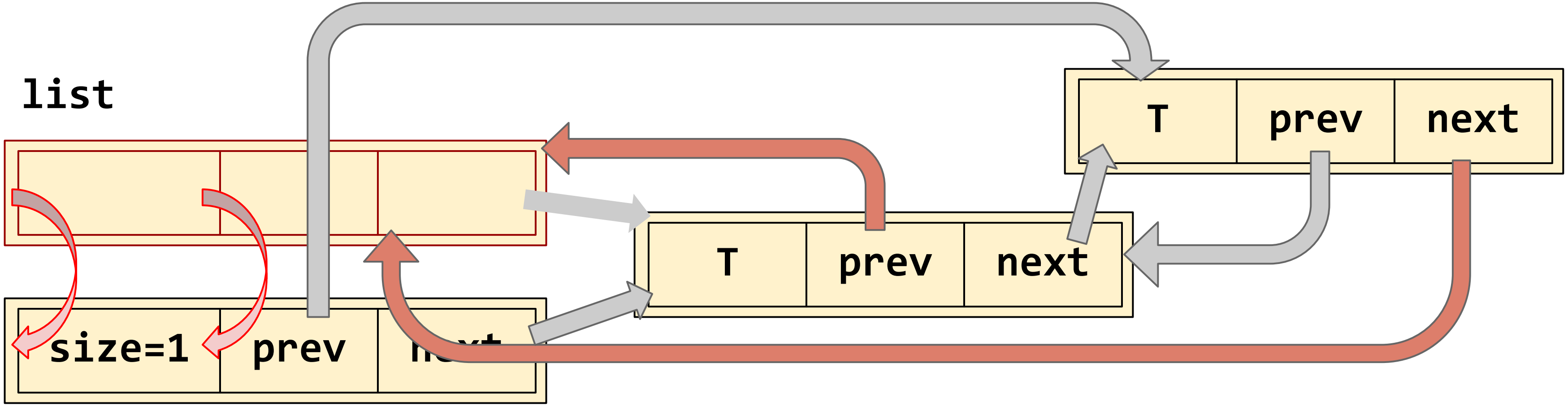
Not all types are trivially relocatable

libc++'s `std::list<T>` is non-trivially relocatable.



Not all types are trivially relocatable

libc++'s `std::list<T>` is non-trivially relocatable.



Relocating the `list` object requires some "fixup" beyond just a simple memcpy.

Not all types are trivially relocatable

If we can reliably distinguish trivially relocatable types from non-trivially relocatable types, then we can use trivial `memcpy` for the former and fall back to non-trivial move+destroy for the latter.

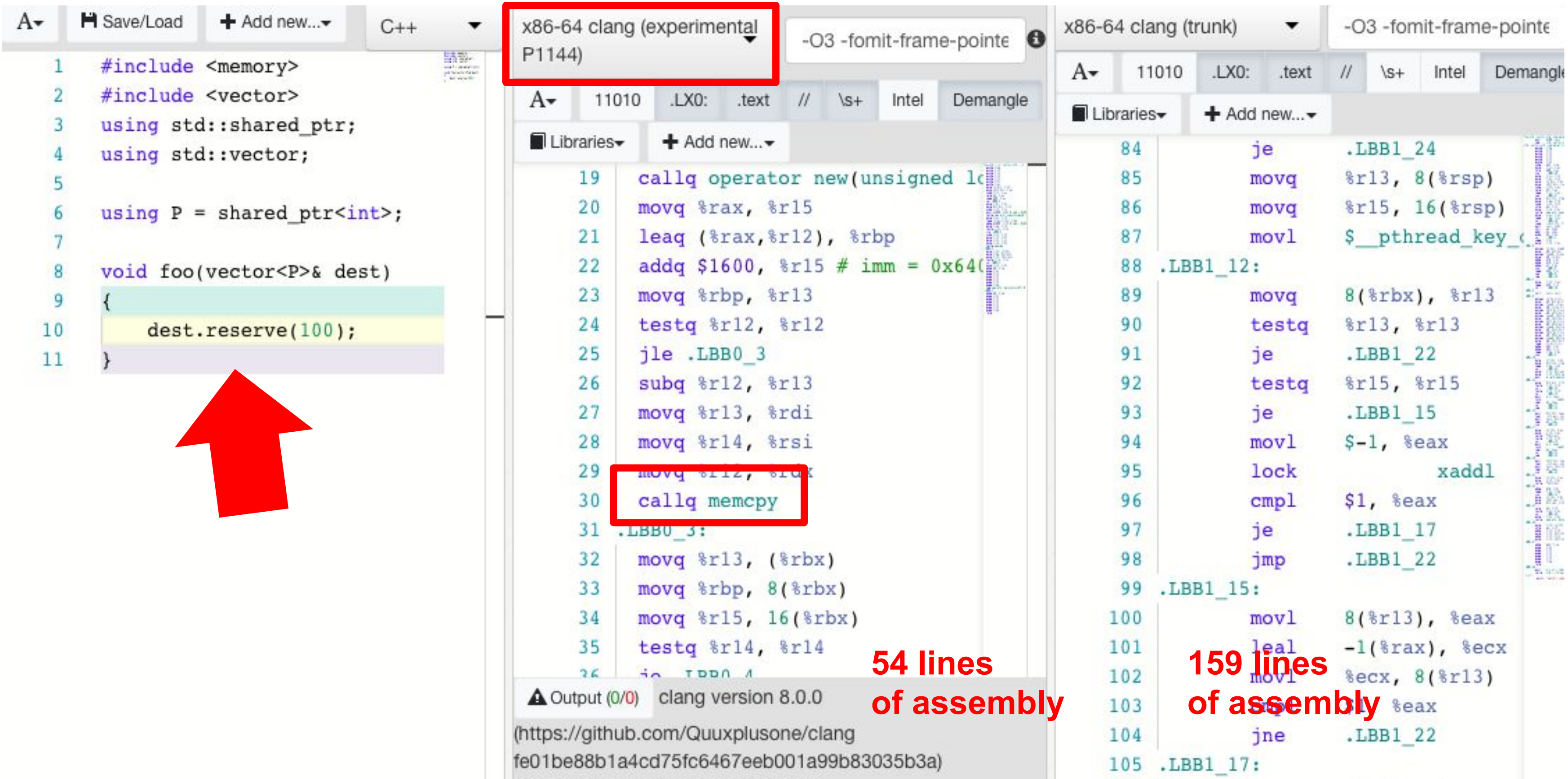
Reallocating a vector of `std::forward_list<T>`? Use `memcpy`!

Reallocating a vector of `std::list<T>`? Use move+destroy in a loop.

Let's teach libc++ that `shared_ptr` is trivially relocatable, and see what improvement we get on our "vector of `shared_ptr`" example.

https://p1144.godbolt.org/z/-TBVP6

Trivially relocatable shared_ptr

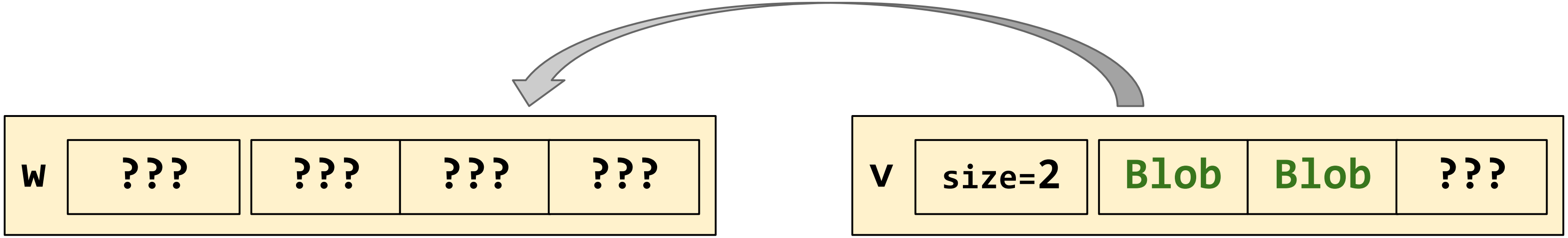



BUT WAIT!
THERE'S MORE!

Application #2 for relocatability

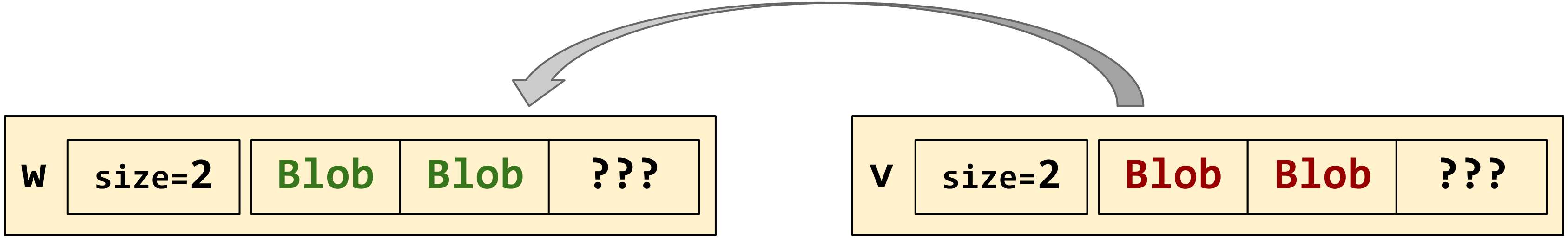
A reliable way of detecting "trivial relocatability" permits optimizing the move operations of `fixed_capacity_vector`:

```
fixed_capacity_vector<Blob, 3> v = { ... };
auto w = std::move(v);
```



Application #2 for relocatability

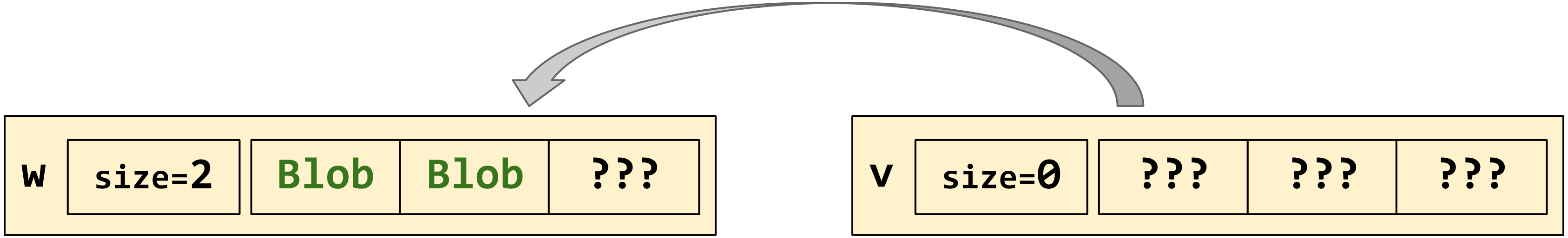
```
fixed_capacity_vector(fixed_capacity_vector&& rhs) {
    uninitialized_move(rhs.begin(), rhs.end(), begin());
    size_ = rhs.size_;
}
```



MOVING IS
INEFFICIENT AND BAD

Application #2 for relocatability

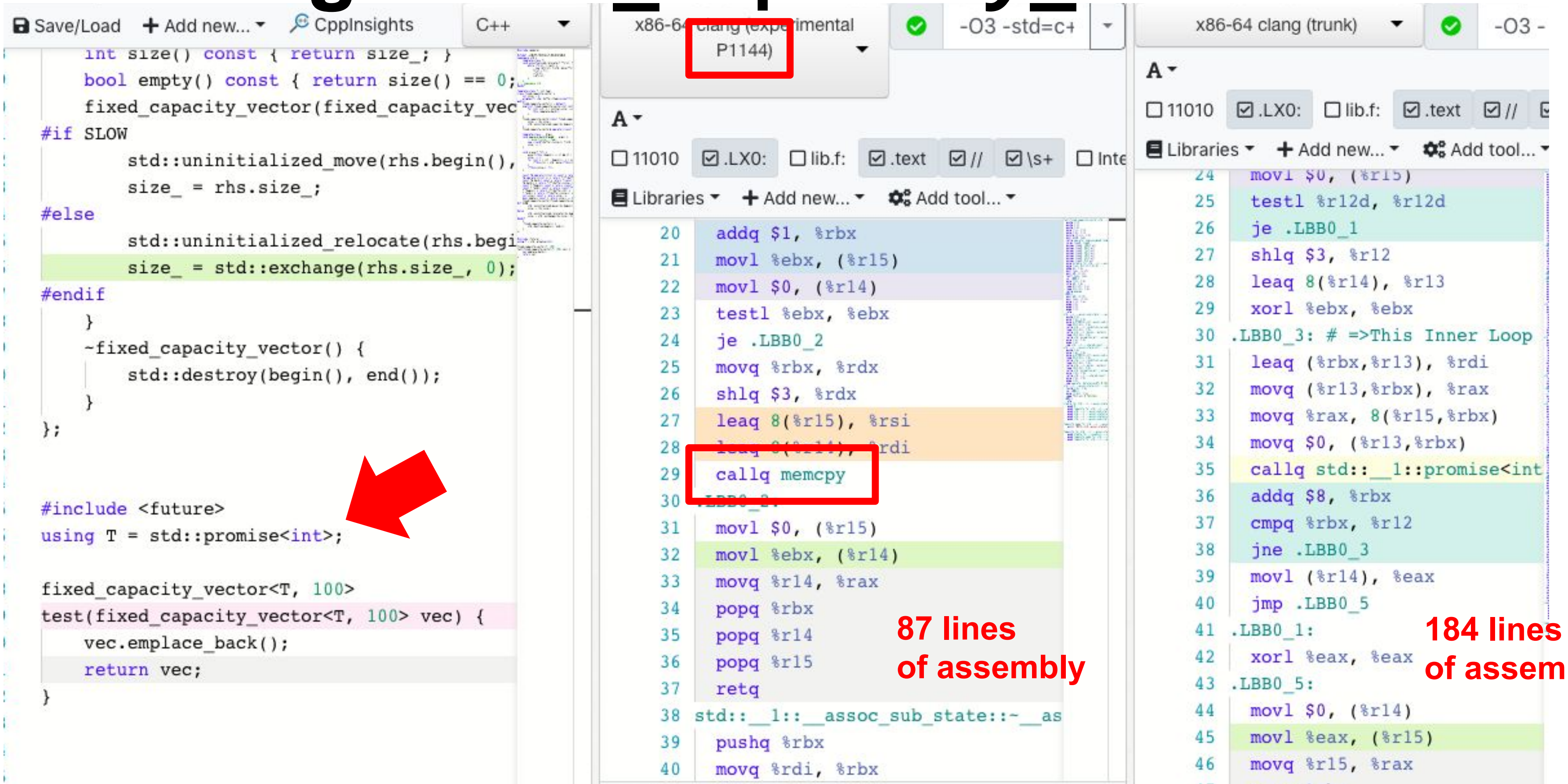
```
fixed_capacity_vector(fixed_capacity_vector&& rhs) {
    uninitialized_relocate(rhs.begin(), rhs.end(), begin());
    size_ = std::exchange(rhs.size_, 0);
}
```



RELOCATING IS
EFFICIENT AND GOOD

https://p1144.godbolt.org/z/jSqEUf

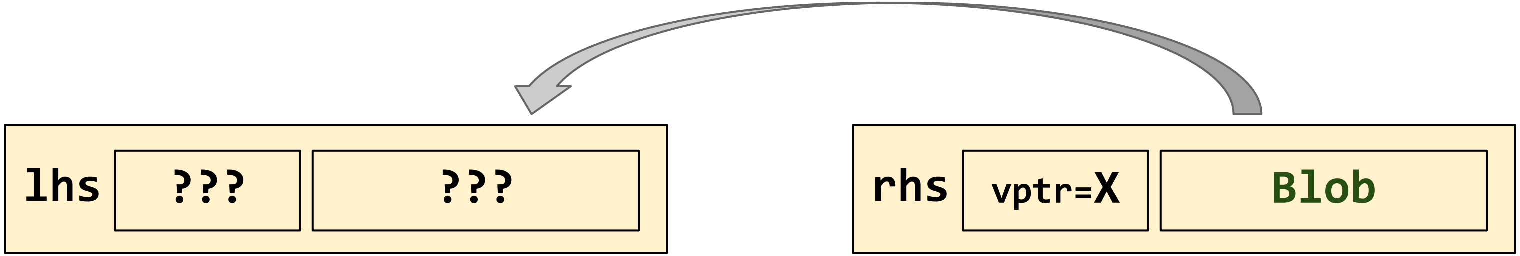
Moving fixed_capacity_vector



Application #3 for relocatability

Consider the move-constructor of `std::function`.

What options do we have, to get the data from point `rhs` to point `lhs`?

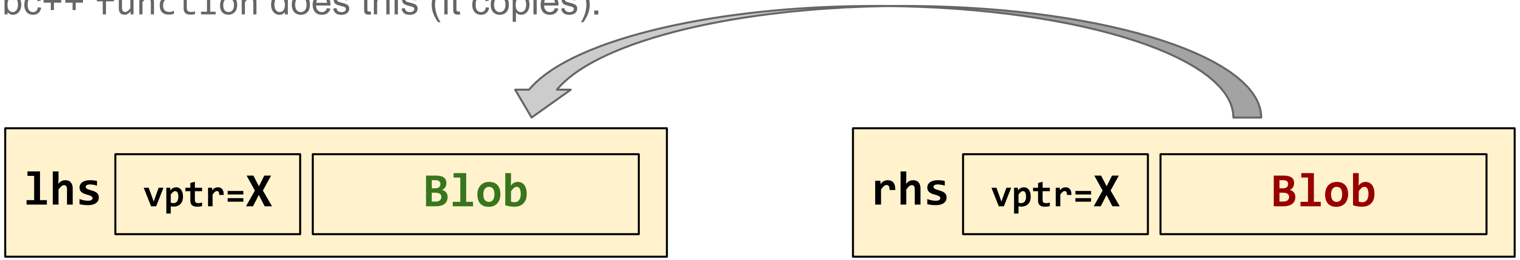


Application #3 for relocatability

```
function(function&& rhs) {
    rhs.vptr_->move(rhs_->storage_, this->storage_);
}
```

We could just move (or copy) the data, leaving `rhs` engaged.

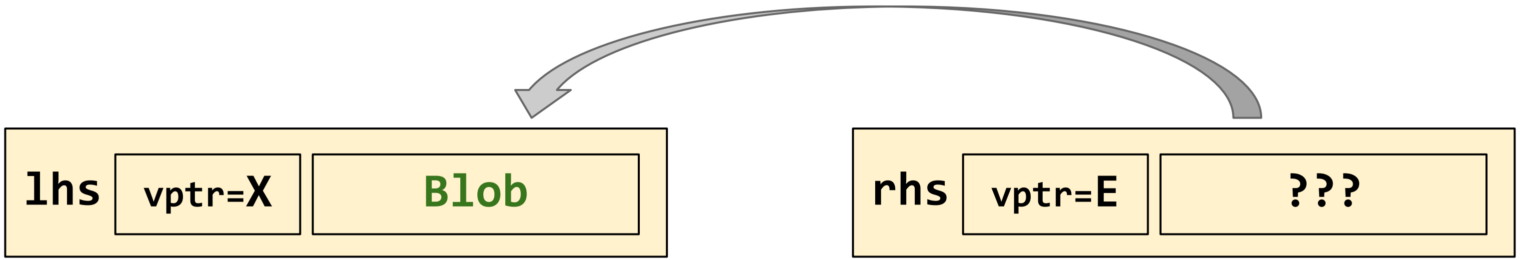
libc++ `function` does this (it copies).



**MOVING IS
INEFFICIENT AND BAD**

Application #3 for relocatability

```
function(function&& rhs) {
    rhs.vptr_->move(rhs_->storage_, this->storage_);
    rhs.vptr_->destroy(rhs->storage_);
    this->vptr_ = std::exchange(rhs.vptr_, &empty_vtable);
}
```

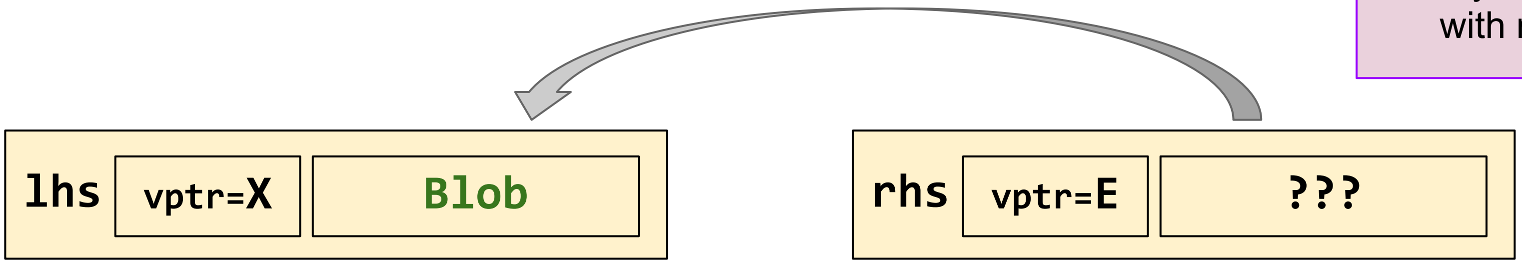


MOVING IS *STILL* INEFFICIENT AND BAD

Application #3 for relocatability

```
function(function&& rhs) {
    rhs.vptr_->relocate(rhs_->storage_, this->storage_);
    this->vptr_ = std::exchange(rhs.vptr_, &empty_vtable);
}
```

Now you're thinking with relocate!



RELOCATING IS
EFFICIENT AND GOOD

Now we can do something cool...

Look at this line:

```
rhs.vptr_->relocate(rhs->storage_, this->storage_);
```

When we “type-erase” `T` into a `relocate` function, we generally have to do

```
[](void *src, void *dest) {
    T& from = *(T*)src;
    ::new (dest) T(std::move(from));
    from.~T();
}
```

Different code for each different `T`. Lots of linker symbols.

Now we can do something cool...

But a reliable way of detecting "**trivial** relocatability" permits deduplicating the codepaths for all trivially relocatable `T`s of a given size!

```
rhs.vptr_->relocate(rhs->storage_, this->storage_);
```

When we “type-erase” a trivially relocatable `T`, we can just use

```
[](void *src, void *dest) {
    memcpy(dest, src, sizeof(T));
}
```

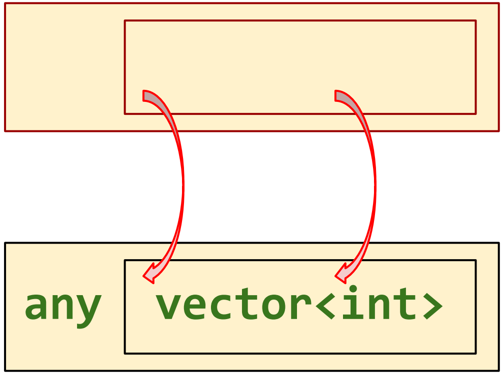
which means we can use the ***same single piece of code*** for `double`, `int*`, `std::unique_ptr<int>`, and so on. Less work for the linker!

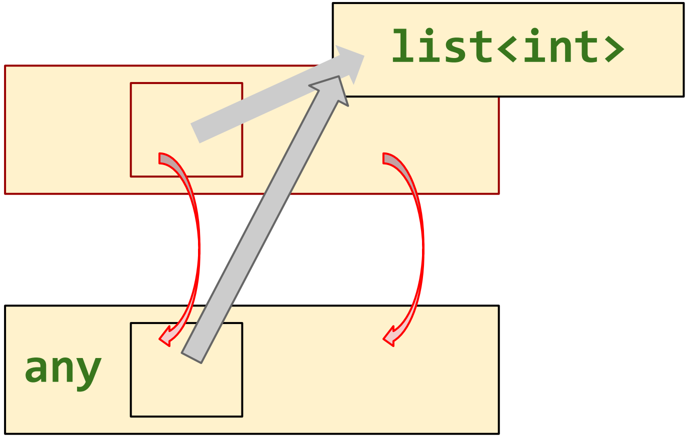
Application #4 for relocatability

With a reliable way of detecting "trivial relocatability,"
we can **avoid SBO for wrappees that are not trivially relocatable.**
Then the wrapper itself becomes trivially relocatable!

Application #4 for relocatability

With a reliable way of detecting "trivial relocatability,"
we can **avoid using our SBO for wrappees that are not trivially relocatable.**
Then the wrapper itself becomes trivially relocatable!





Application #5 for relocatability

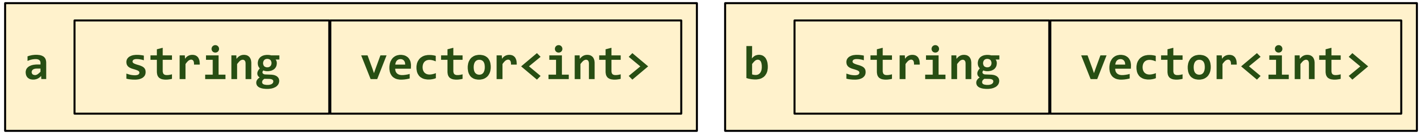
- Consider this naïve user-defined class type.
  It has several members, all of trivially relocatable types.
- I call it “naïve” because it follows the Rule of Zero
  and does not define a custom ADL `swap` overload.
- I wish all my code were naïve!

```
struct Pair {
    std::string first;
    std::vector<int> second;
};

Pair a, b;
std::swap(a, b);
```

Application #5 for relocatability

```
struct Pair {
    std::string first;
    std::vector<int> second;
};

Pair a, b;
std::swap(a, b);
```

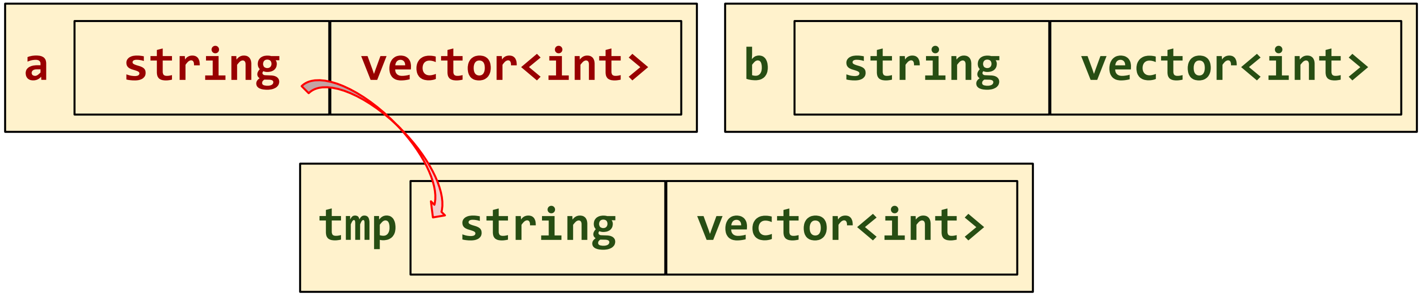
```
template<class T>
void swap(T& a, T& b) {
    T temp = std::move(a);
    a = std::move(b);
    b = std::move(temp);
}
```



Application #5 for relocatability

```
struct Pair {
    std::string first;
    std::vector<int> second;
};

Pair a, b;
std::swap(a, b);
```

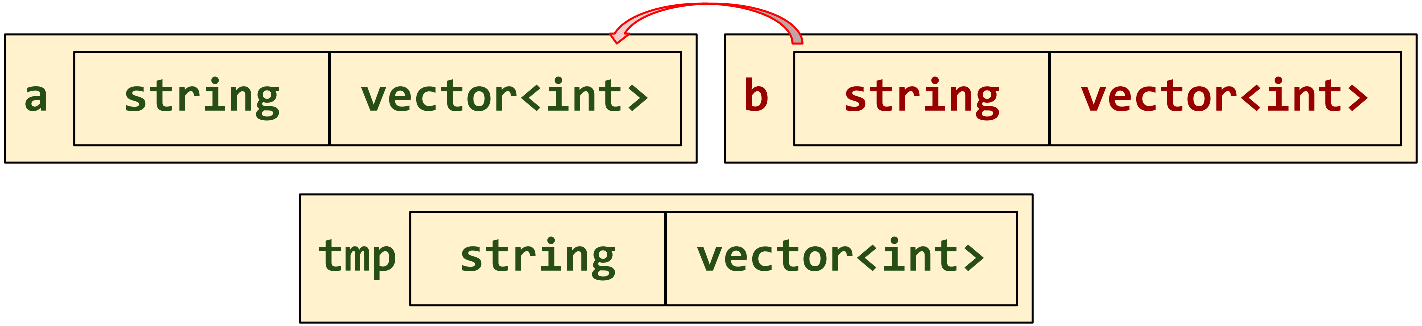
```
template<class T>
void swap(T& a, T& b) {
    T temp = std::move(a);
    a = std::move(b);
    b = std::move(temp);
}
```



Application #5 for relocatability

```
struct Pair {
    std::string first;
    std::vector<int> second;
};

Pair a, b;
std::swap(a, b);
```

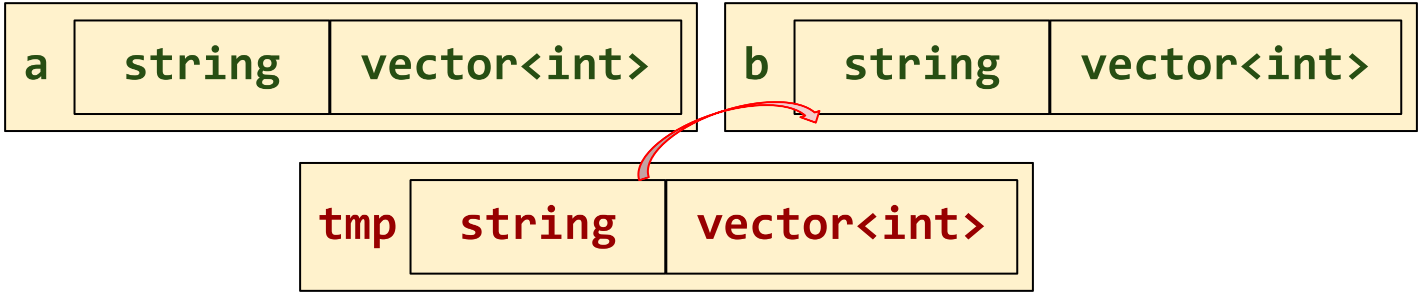
```
template<class T>
void swap(T& a, T& b) {
    T temp = std::move(a);
    a = std::move(b);
    b = std::move(temp);
}
```



Application #5 for relocatability

```
struct Pair {
    std::string first;
    std::vector<int> second;
};

Pair a, b;
std::swap(a, b);
```

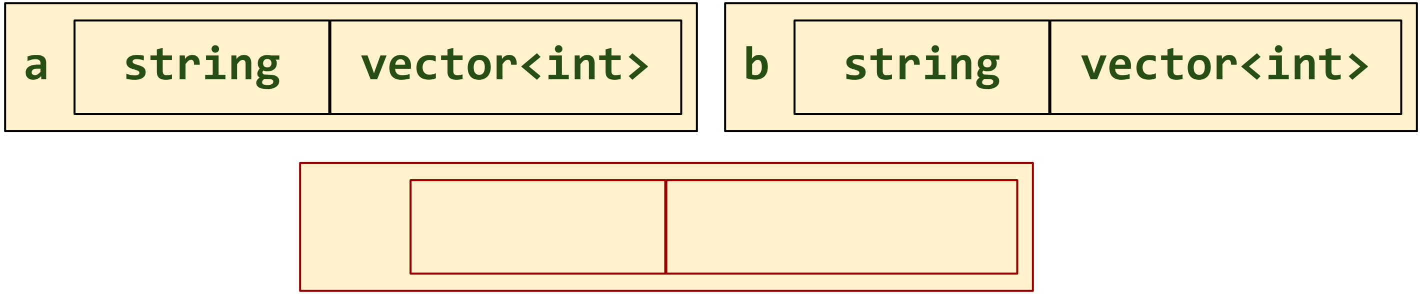
```
template<class T>
void swap(T& a, T& b) {
    T temp = std::move(a);
    a = std::move(b);
    b = std::move(temp);
}
```



Application #5 for relocatability

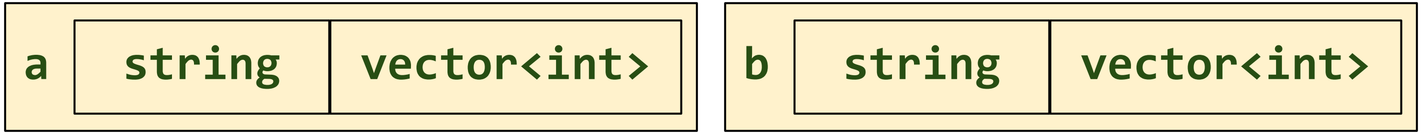
```
struct Pair {
    std::string first;
    std::vector<int> second;
};

Pair a, b;
std::swap(a, b);
```

```
template<class T>
void swap(T& a, T& b) {
    T temp = std::move(a);
    a = std::move(b);
    b = std::move(temp);
}
```



But what *should* happen?

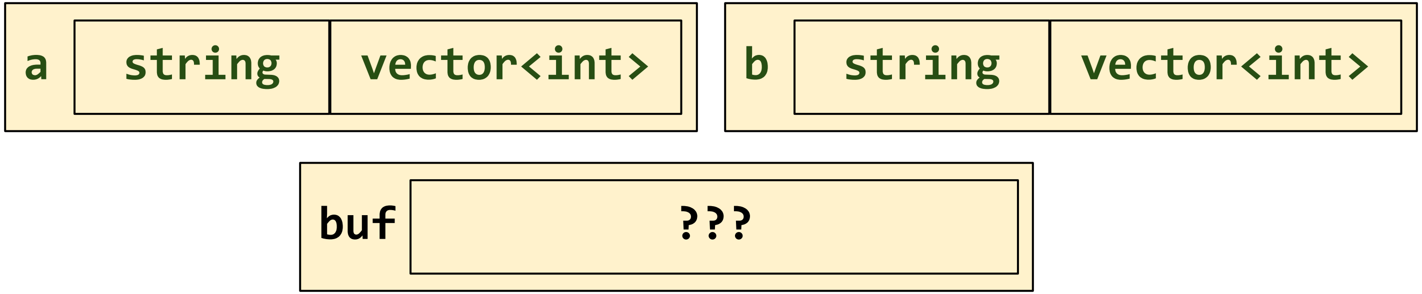
```
struct Pair {
    std::string first;
    std::vector<int> second;
};

Pair a, b;
std::swap(a, b);
```



But what *should* happen?

```
struct Pair {
    std::string first;
    std::vector<int> second;
};

Pair a, b;
std::swap(a, b);
```

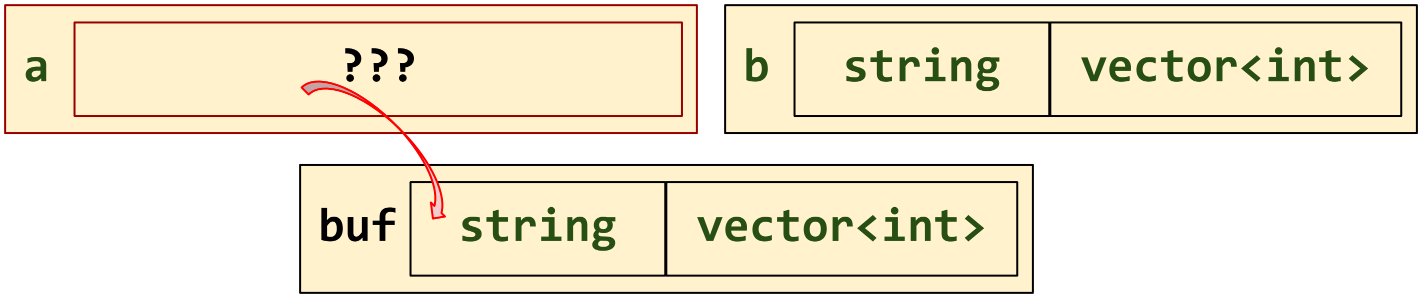
```
template<class T>
void swap(T& a, T& b) {
    alignas(T) char buf[sizeof(T)];
    std::relocate_at(&a, (T*)buf);
    std::relocate_at(&b, &a);
    std::relocate_at((T*)buf, &b);
}
```



But what *should* happen?

```
struct Pair {
    std::string first;
    std::vector<int> second;
};

Pair a, b;
std::swap(a, b);
```

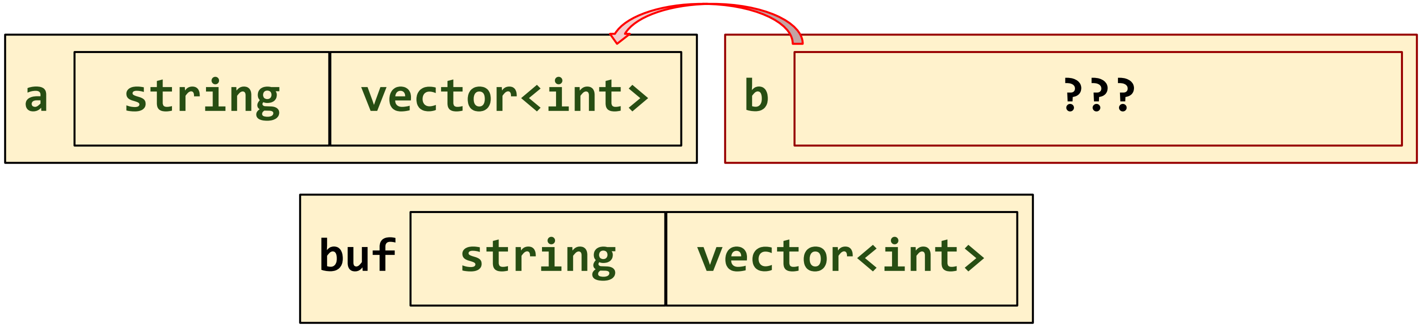
```
template<class T>
void swap(T& a, T& b) {
    alignas(T) char buf[sizeof(T)];
    std::relocate_at(&a, (T*)buf);
    std::relocate_at(&b, &a);
    std::relocate_at((T*)buf, &b);
}
```



But what *should* happen?

```
struct Pair {
    std::string first;
    std::vector<int> second;
};

Pair a, b;
std::swap(a, b);
```

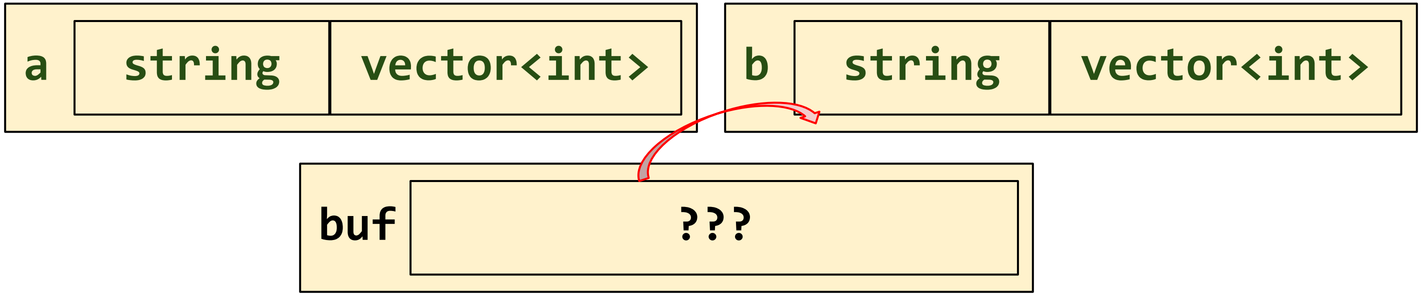
```
template<class T>
void swap(T& a, T& b) {
    alignas(T) char buf[sizeof(T)];
    std::relocate_at(&a, (T*)buf);
    std::relocate_at(&b, &a);
    std::relocate_at((T*)buf, &b);
}
```



But what *should* happen?

```
struct Pair {
    std::string first;
    std::vector<int> second;
};

Pair a, b;
std::swap(a, b);
```

```
template<class T>
void swap(T& a, T& b) {
    alignas(T) char buf[sizeof(T)];
    std::relocate_at(&a, (T*)buf);
    std::relocate_at(&b, &a);
    std::relocate_at((T*)buf, &b);
}
```



Comparison of `swap` approaches

```
template<class T>
void swap(T& a, T& b) {
    T temp = std::move(a);
    a = std::move(b);
    b = std::move(temp);
}
```

- 4 invocations of 3 different operations
- Usually, ***none*** of the 3 are trivial

```
template<class T>
void swap(T& a, T& b) {
   alignas(T) char buf[sizeof(T)];
   std::relocate_at(&a, (T*)buf);
   std::relocate_at(&b, &a);
   std::relocate_at((T*)buf, &b);
}
```

- Just 3 invocations of 1 operation
- ...which ***is usually trivial***

https://p1144.godbolt.org/z/SdL6FE

Benchmarking `std::rotate`

A ▾ Save/Load + Add new... ▾ CppInsights C++

```cpp
#include <algorithm>
#include <string>
#include <vector>

struct Pair {
    int key;
    std::string first;
    std::vector<int> second;
};

void test(Pair *first, int n, int k) {
    std::rotate(first, first + n, first + k)
}
```

x86-64 clang (experimental P1144) -O3 -std=c+

A ▾ ☐ 11010 ☑ .LX0: ☐ lib.f: ☑ .text ☑ // ☑ \s+ ☐ Inte

Libraries ▾ + Add new... ▾ Add tool... ▾

```
movups %xmm1, 16(%rdi)
movups %xmm0, (%rdi)
movq -8(%rsp), %rax
movq %rax, 48(%rcx)
movaps -56(%rsp), %xmm0
movaps -40(%rsp), %xmm1
movaps -24(%rsp), %xmm2
movups %xmm2, 32(%rcx)
movups %xmm1, 16(%rcx)
movups %xmm0, (%rcx)
leaq 56(%rdi), %rsi
addq $56, %rcx
cmpq %r8, %rcx
jne .LBB0_13
movq %r11, %rcx
movq %rsi, %rdi
cmpq %rsi, %r11
jne .LBB0_10
.LBB0_12:
retq
```

131 lines of assembly

x86-64 clang (trunk) -O3 -

A ▾ ☐ 11010 ☑ .LX0: ☐ lib.f: ☑ .text ☑ // ☑ \

Libraries ▾ + Add new... ▾ Add tool... ▾

```
movq %r15, 24(%r14)
movq $0, 15(%rsp)
movq $0, 8(%rsp)
movq 32(%r14), %rdi
testq %rdi, %rdi
je .LBB1_8
callq operator delete(void*
movups %xmm0, (%rbp)
movq $0, 16(%rbp)
.LBB1_8:
movaps 32(%rsp), %xmm0 # 16
movups %xmm0, 32(%r14)
movq %r12, 48(%r14)
addq $56, %rsp
popq %rbx
popq %r12
popq %r13
popq %r14
popq %r15
popq %rbp
retq
```

176 lines of assembly

Rewriting `swap` for an enormous class of types affects codegen for all `swap`-based algorithms. My libc++ doesn't change a single line of `std::rotate`!

So by this point you’re probably thinking...


GIVE IT TO ME

Prior Art

Folly, BSL, EASTL

Prior art in Electronic Arts EASTL

https://github.com/electronicarts/EASTL/blob/master/doc/EASTL%20Best%20Practices.html

If the user has a class that is relocatable (i.e. can safely use `memcpy` to copy values), the user can use the `EASTL_DECLARE_TRIVIAL_RELOCATE` declaration to tell the compiler ... This will automatically significantly speed up some containers and algorithms that use that class.

```
EASTL_DECLARE_TRIVIAL_RELOCATE(Widget);

vector<Widget> wVector;
wVector.erase(wVector.begin()); // This operation will use memcpy.
```

`has_trivial_relocate` is ... very useful in allowing for the generation of optimized object moving operations. It is similar to the `is_pod` type trait, but goes further and allows non-POD classes to be categorized as relocatable. Such categorization is something that no compiler can do, as only the user can know if it is such. Thus `EASTL_DECLARE_TRIVIAL_RELOCATE` is provided to allow the user to give the compiler a hint.

Prior art in Bloomberg BSL

`bslmf::IsBitwiseMoveable` ... allows generic code to determine whether `TYPE` can be destructively moved using `memcpy`. Given a pointer `p1` to an object of type `TYPE`, and a pointer `p2` of the same type pointing to allocated but uninitialized storage, a destructive move from `p1` to `p2` comprises the following pair of operations:

```
new ((void*) p2) TYPE(*p1);  // OR new ((void*) p2) TYPE(std::move(*p1));
p1->~TYPE();
```

An object of type `TYPE` is ***bitwise moveable*** if the above operation can be replaced by the following operation without affecting correctness:

```
std::memcpy(p2, p1, sizeof(TYPE));
```

Prior art in Bloomberg BSL

```
class Pair {
    bsl::string first;
    bsl::vector<int> second;
  public:
    BSLMF_NESTED_TRAIT_DECLARATION(Pair,
                                   bslmf::IsBitwiseMoveable);
};

static_assert(bslmf::IsBitwiseMoveable<Pair>::value);
```

Folly has by far
the most quotable
explanation of
relocation semantics.

Prior art in Facebook Folly

https://github.com/facebook/folly/blob/master/folly/docs/FBVector.md#object-relocation

C++’s assumption of non-relocatable values hurts everybody for the benefit of a few questionable designs. The issue is that moving a C++ object "by the book" entails (a) creating a new copy from the existing value; (b) destroying the old value. This is quite vexing and violates common sense. Consider this hypothetical conversation between Captain Picard and an incredulous alien:

> Incredulous Alien: "So, this teleporter, how does it work?"
>
> Picard: "It beams people and arbitrary matter from one place to another."
>
> Incredulous Alien: "Hmmm... is it safe?"
>
> Picard: "Yes, but earlier models were a hassle. They'd clone the person to another location. Then the teleporting chief would have to shoot the original. Ask O’Brien, he was an intern during those times. A bloody mess, that's what it was."

Prior art in Facebook Folly

- I’m going to pick on Folly here. I’m going to show you its bugs.
- This is **not** because Folly is poor-quality code!
- It’s because Folly is highly **comprehensible** code.

> **There are two ways of constructing a software design: One way is to make it so simple that there are *obviously no deficiencies*, and the other way is to make it so complicated that there are *no obvious deficiencies*. The first method is far more difficult.**
>
> **—C.A.R. Hoare**

Prior art in Facebook Folly

- I'm going to pick on Folly here. I'm going to show you its bugs.
- This is **not** because Folly is poor-quality code!
- It's because Folly is highly **comprehensible** code.

> **There are two ways of constructing a software design: One way is to make it so simple that *its deficiencies are obvious*, and the other way is to make it so complicated that *its deficiencies are not obvious*. The first method is far more difficult.**
>
> **—with apologies to C.A.R. Hoare**

Prior art in Facebook Folly

https://github.com/facebook/folly/blob/master/folly/docs/FBVector.md#object-relocation

`folly::fbvector<Widget>` will not compile unless you provide a “warrant” that `Widget` is relocatable.

Here’s what a warrant looks like, according to the Folly docs:

```
namespace folly {
    struct IsRelocatable<Widget> : std::true_type {};
}
```

Two downsides of explicit warrants

With explicit warrants, you need to know ***how*** to write one.

In each of our three case studies, this requires

- using a library-specific macro, and/or
- knowing the C++ syntax for explicit template specialization.

Did you notice that the Folly docs actually get the syntax wrong?

With explicit warrants, you need to know ***when*** to write one.

- You have to know whether your type actually *is* trivially relocatable.

Even the developers get it wrong!

Get the warrant wrong

https://github.com/facebook/folly/issues/889

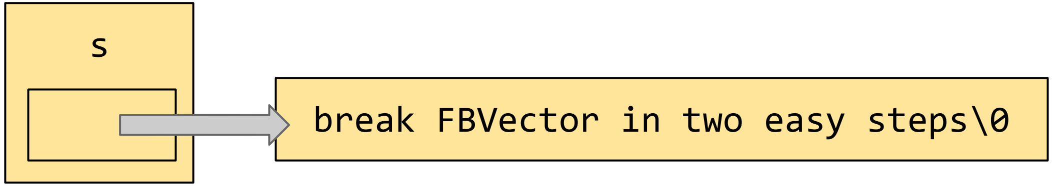
Folly provides these warrants in their "Traits.h" header:

```
#ifndef _MSC_VER
FOLLY_ASSUME_FBVECTOR_COMPATIBLE_2(std::vector)
FOLLY_ASSUME_FBVECTOR_COMPATIBLE_2(std::deque)
FOLLY_ASSUME_FBVECTOR_COMPATIBLE_2(std::unique_ptr)
FOLLY_ASSUME_FBVECTOR_COMPATIBLE_1(std::shared_ptr)
FOLLY_ASSUME_FBVECTOR_COMPATIBLE_1(std::function)
#endif
```

Unfortunately, if using libc++, the warrant for `std::function` is simply a lie;
& the warrant for `std::unique_ptr<T, D>` can cause problems for some `D`.

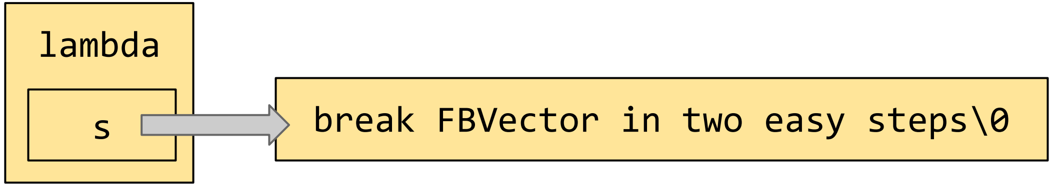
How to break FBVector

```
#include <folly/FBVector.h>

int main() {
    std::string s = "break FBVector in two easy steps";
    folly::fbvector<std::function<int()>> v;
    v.push_back([s]() { std::cout << s << std::endl; });
    v.reserve(v.capacity() + 1);
    v[0]();
}
```



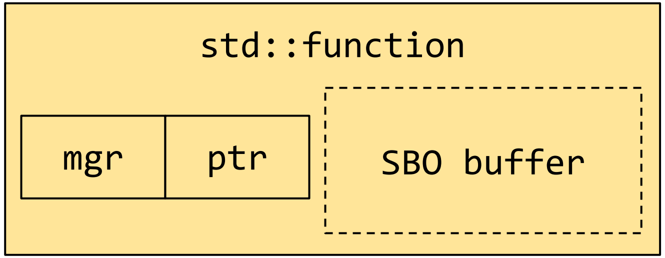
How to break FBVector

```
#include <folly/FBVector.h>

int main() {
    std::string s = "break FBVector in two easy steps";
    folly::fbvector<std::function<int()>> v;
    v.push_back([s]() { std::cout << s << std::endl; });
    v.reserve(v.capacity() + 1);
    v[0]();
}
```



How to break FBVector

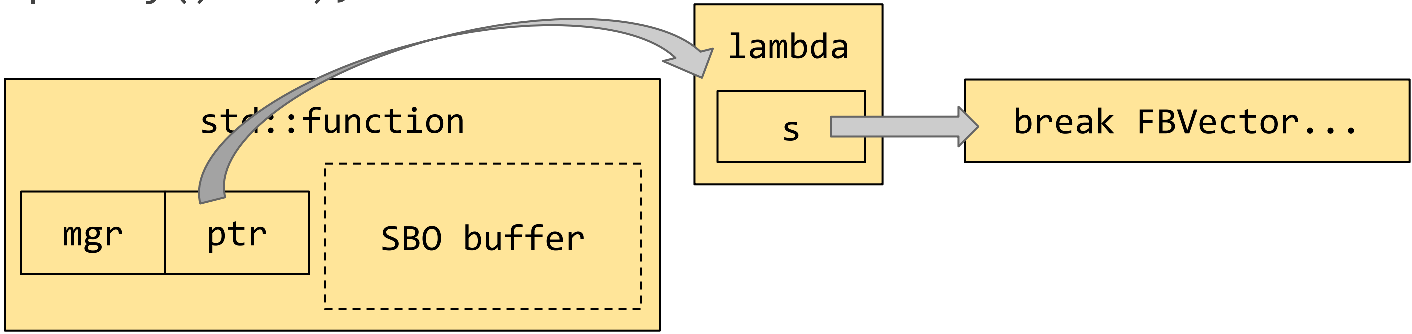
```
#include <folly/FBVector.h>

int main() {
    std::string s = "break FBVector in two easy steps";
    folly::fbvector<std::function<int()>> v;
    v.push_back([s]() { std::cout << s << std::endl; });
    v.reserve(v.capacity() + 1);
    v[0]();
}
```



libc++'s `std::function` has a pointer to a "manager" function (basically a manual vtable), a pointer to the wrapped data, and also a buffer for the Small Buffer Optimization.

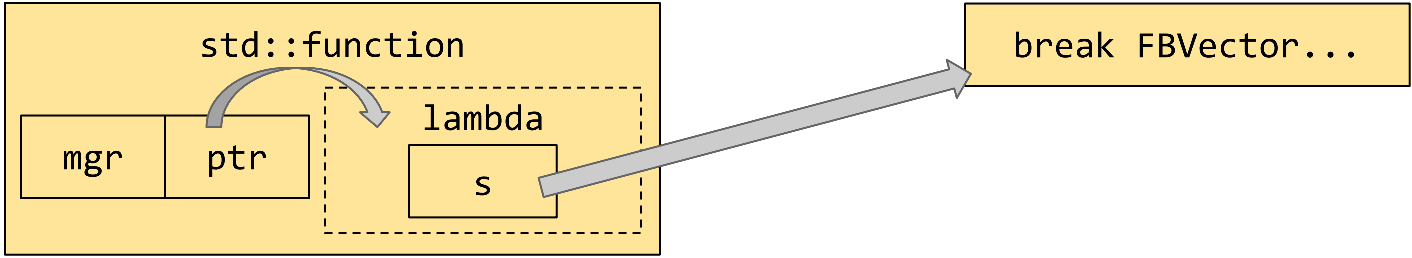
How to break FBVector

```
#include <folly/FBVector.h>

int main() {
    std::string s = "break FBVector in two easy steps";
    folly::fbvector<std::function<int()>> v;
    v.push_back([s]() { std::cout << s << std::endl; });
    v.reserve(v.capacity() + 1);
    v[0]();
}
```



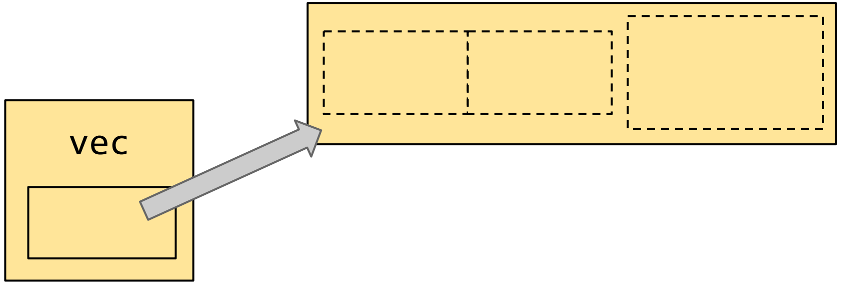
How to break FBVector

```
#include <folly/FBVector.h>

int main() {
    std::string s = "break FBVector in two easy steps";
    folly::fbvector<std::function<int()>> v;
    v.push_back([s]() { std::cout << s << std::endl; });
    v.reserve(v.capacity() + 1);
    v[0]();
}
```



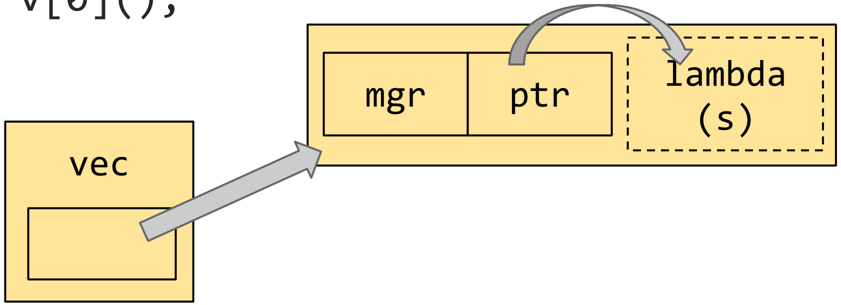
How to break FBVector

```
#include <folly/FBVector.h>

int main() {
    std::string s = "break FBVector in two easy steps";
    folly::fbvector<std::function<int()>> v;
    v.push_back([s]() { std::cout << s << std::endl; });
    v.reserve(v.capacity() + 1);
    v[0]();
}
```



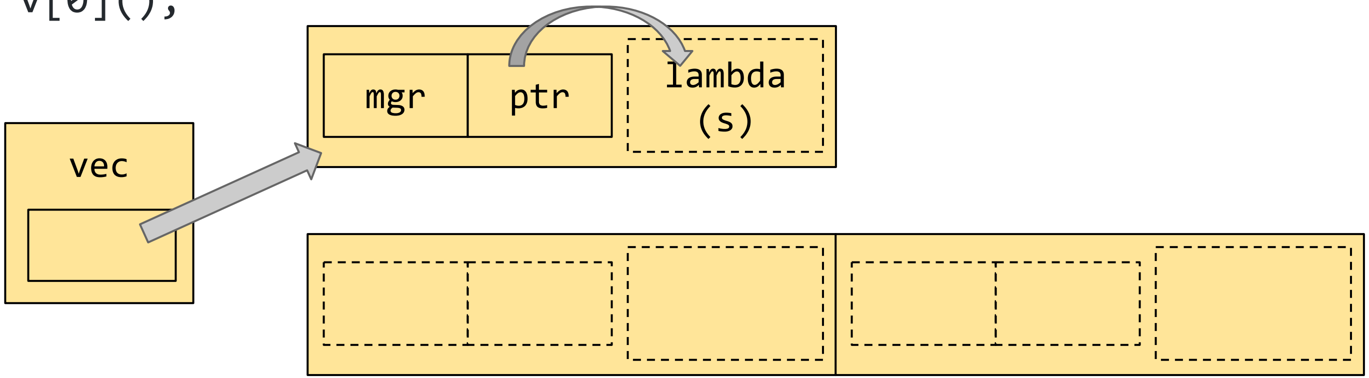
How to break FBVector

```
#include <folly/FBVector.h>

int main() {
    std::string s = "break FBVector in two easy steps";
    folly::fbvector<std::function<int()>> v;
    v.push_back([s]() { std::cout << s << std::endl; });
    v.reserve(v.capacity() + 1);
    v[0]();
}
```



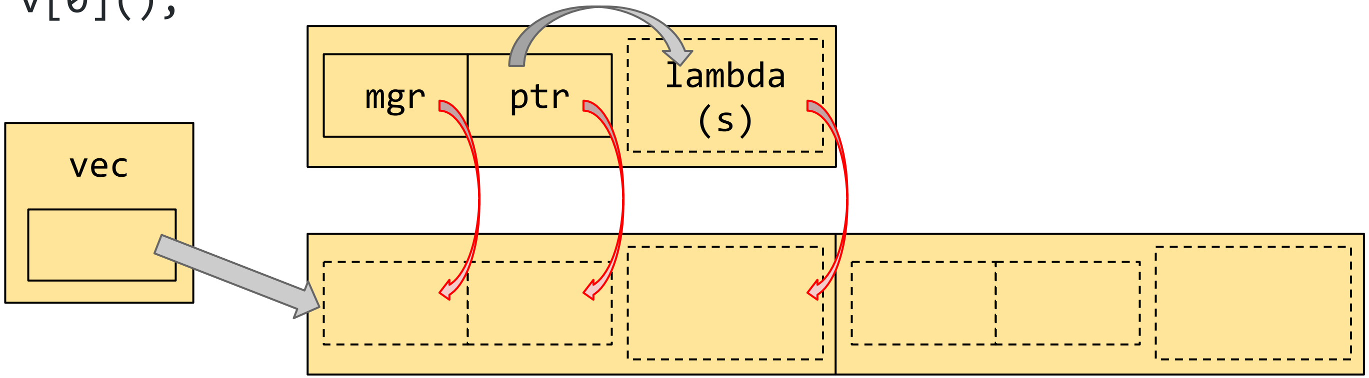
How to break FBVector

```
#include <folly/FBVector.h>

int main() {
    std::string s = "break FBVector in two easy steps";
    folly::fbvector<std::function<int()>> v;
    v.push_back([s]() { std::cout << s << std::endl; });
    v.reserve(v.capacity() + 1);
    v[0]();
}
```



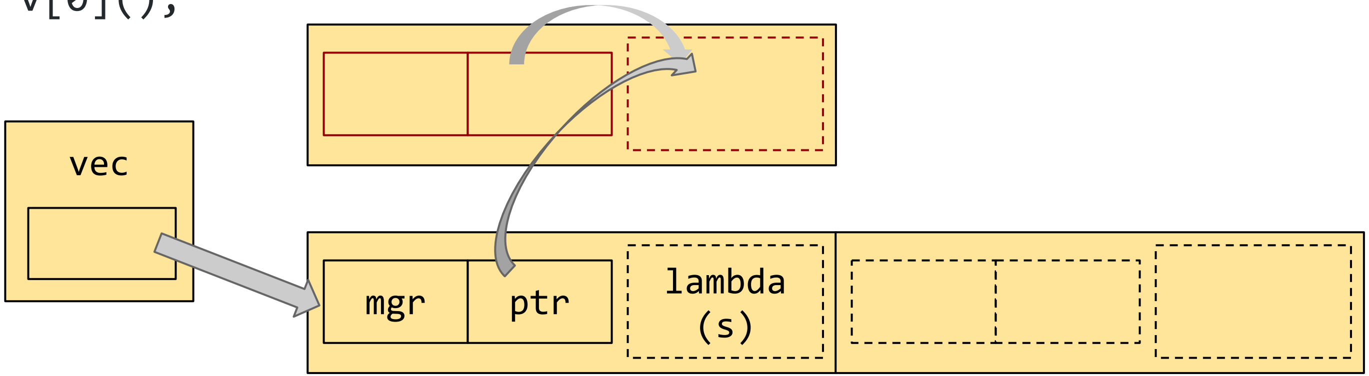
How to break FBVector

```
#include <folly/FBVector.h>

int main() {
    std::string s = "break FBVector in two easy steps";
    folly::fbvector<std::function<int()>> v;
    v.push_back([s]() { std::cout << s << std::endl; });
    v.reserve(v.capacity() + 1);
    v[0]();
}
```



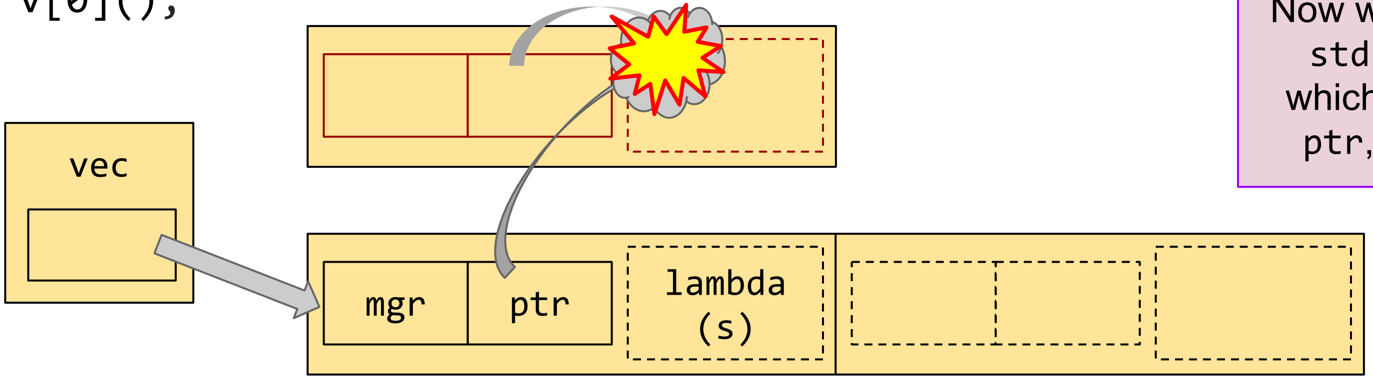
How to break FBVector

```
#include <folly/FBVector.h>

int main() {
    std::string s = "break FBVector in two easy steps";
    folly::fbvector<std::function<int()>> v;
    v.push_back([s]() { std::cout << s << std::endl; });
    v.reserve(v.capacity() + 1);
    v[0]();
}
```



How to break FBVector

```
#include <folly/FBVector.h>

int main() {
    std::string s = "break FBVector in two easy steps";
    folly::fbvector<std::function<int()>> v;
    v.push_back([s]() { std::cout << s << std::endl; });
    v.reserve(v.capacity() + 1);
    v[0]();
}
```



Now we try to call the `std::function`, which dereferences `ptr`, and... boom.

Hand-written warrants are bug-prone

- `folly::fbvector` relies on hand-written warrants
  - written by the Folly developers (who get it wrong)
  - written by the client developer (who will get it wrong)
- Hand-written warrants are usually wrong, because they require that the ***developer*** know internal details of the ***library vendor***'s implementation.
- And they don't scale.

Hand-written warrants don’t scale

```
#include <folly/FBVector.h>

struct Pair {
    std::string first;
    std::vector<int> second;
};

folly::fbvector<Pair> v;  // ERROR: no warrant
```

Folly “helpfully” prevents `fbvector<Pair>` from compiling. EASTL and BSL let it compile and just don’t do the optimization — so this code is silently pessimized.

Adding warrants is ***too much work*** for the developer.

Enter P1144:
“Object Relocation in terms of Move plus Destroy”

P1144 proposal in a nutshell

- Targets C++2b (“C++23”)
- Combination of core-language feature, type traits, library algorithms
- By design, ***preserves correctness*** of all C++17 code
- By design, ***preserves conformance*** of all C++17 library implementations (!!)

P1144 requires vendors to implement a few simple algorithms and type-traits, but does not require an ABI break. For example, libstdc++’s `std::string` is not trivially relocatable. This is 100% okay according to P1144.

What matters is that we have a reliable way to ***detect*** its trivial relocatability.

P1144 proposal in a nutshell

- “Trivially relocatable” becomes a term of art, similar to “trivially copyable,” “trivially destructible,” etc.
- Just like those properties, the compiler exposes a built-in which is wrapped up by the STL into a type trait: `std::is_trivially_relocatable<T>`.
- Just like those properties, the compiler automatically propagates trivial relocatability to every Rule-of-Zero class, according to the trivial relocatability of its bases and members.
- All non-Rule-of-Zero classes are assumed to be non-trivially relocatable. Expert users can explicitly warrant the property via a class attribute: `[[trivially_relocatable]]`.

P1144 is implemented on Godbolt

- Use the C++ compiler “x86-64 clang (experimental P1144)”
- This is built from stock `llvm`...
- ...with a branch of the `clang` front-end that implements `[[trivially_relocatable]]` and `__is_trivially_relocatable(T)`...
- ...and a branch of libc++ that implements P1144’s new library features.

P1144 is implemented on Godbolt

- Use the C++ compiler “x86-64 clang (experimental P1144)”
- This is built from stock `llvm`...
- ...with a branch of the `clang` front-end that implements `[[trivially_relocatable]]` and `__is_trivially_relocatable(T)`...
- ...and a branch of libc++ that implements P1144’s new library features.
- And trivially relocatable `std::any`.

P1144 is implemented on Godbolt

- Use the C++ compiler “x86-64 clang (experimental P1144)”
- This is built from stock `llvm`...
- ...with a branch of the `clang` front-end that implements `[[trivially_relocatable]]` and `__is_trivially_relocatable(T)`...
- ...and a branch of libc++ that implements P1144’s new library features.
- And trivially relocatable `std::any`.
- And trivially copyable `std::vector<bool>::iterator`.

P1144 is implemented on Godbolt

- And the entire `<memory_resource>` header.
- And CTAD deduction guides for all associative and unordered containers.
- And `std::priority_queue::replace_top()`.
- And container adaptors which are conditionally trivially destructible.
- And `string_view` support in `std::regex` (thanks to Mark de Wever!)

All this stuff is maintained in the `trivially-relocatable` branch of `github.com/Quuxplusone/libcxx`.

You already know the entire feature

All of the optimizations I’ve shown can be implemented invisibly to the user-programmer. The proof is the Godbolt links/screenshots you’ve already seen.

Okay, there are two more little wrinkles:

- `[[trivially_relocatable(bool)]]`
- `std::relocate_at` and `std::uninitialized_relocate`

Conditional trivial relocatability

```
template<class T>
struct ZeroWrap {
    T t;
};

using R = std::vector<int>;
using NR = std::list<int>;

static_assert(std::is_trivially_relocatable_v<ZeroWrap<R>>);
static_assert(!std::is_trivially_relocatable_v<ZeroWrap<NR>>);
```

Follow the Rule of Zero and you get conditional trivial relocatability for free. Here's an example.

Conditional trivial relocatability

```
template<class T>
struct ExpertWrap {
  std::pair<int*, T> p;
  friend void swap(ExpertWrap& a, ExpertWrap& b) noexcept { std::swap(a.p, b.p); }
  ExpertWrap() { p.first = new int; }
  ExpertWrap(ExpertWrap&& rhs) noexcept : p(rhs.p) { rhs.p.first = nullptr; }
  auto& operator=(ExpertWrap rhs) noexcept { swap(*this, rhs); }
  ~ExpertWrap() { delete p.first; }
};

using R = std::vector<int>;
using NR = std::list<int>;

static_assert(!std::is_trivially_relocatable_v<ExpertWrap<R>>);
static_assert(!std::is_trivially_relocatable_v<ExpertWrap<NR>>);
```

Break the Rule of Zero and the compiler will conservatively assume that your type is ***not*** trivially relocatable. Remember our design goals!

Conditional trivial relocatability

```
template<class T>
struct [[trivially_relocatable]] ExpertWrap {
  std::pair<int*, T> p;
  friend void swap(ExpertWrap& a, ExpertWrap& b) noexcept { std::swap(a.p, b.p); }
  ExpertWrap() { p.first = new int; }
  ExpertWrap(ExpertWrap&& rhs) noexcept : p(rhs.p) { rhs.p.first = nullptr; }
  auto& operator=(ExpertWrap rhs) noexcept { swap(*this, rhs); }
  ~ExpertWrap() { delete p.first; }
};

using R = std::vector<int>;
using NR = std::list<int>;

static_assert(std::is_trivially_relocatable_v<ExpertWrap<R>>);
static_assert(std::is_trivially_relocatable_v<ExpertWrap<NR>>);
```

As with Folly, BSL, or EASTL, if you give the wrong warrant, your code will be wrong. P1144 reduces the ***number*** of times you use the chainsaw; it doesn't change the ***danger level*** of the chainsaw.

Conditional trivial relocatability (1)

```
template<class T, bool TrivReloc = false>
struct ExpertImpl {
  std::pair<int*, T> p;
  friend void swap(ExpertImpl& a, ExpertImpl& b) noexcept { std::swap(a.p, b.p); }
  ExpertWrap() { p.first = new int; }
  ExpertWrap(ExpertImpl&& rhs) noexcept : p(rhs.p) { rhs.p.first = nullptr; }
  auto& operator=(ExpertImpl rhs) noexcept { swap(*this, rhs); }
  ~ExpertImpl() { delete p.first; }
};

template<class T>
struct [[trivially_relocatable]] ExpertImpl<T, true> {
  cut and paste exactly the same code as above
};

template<class T>
struct ExpertWrap : private ExpertImpl<T, std::is_trivially_relocatable_v<T>> {
  this top-level class now follows the Rule of Zero
};
```

P1144R2 supported only this metaprogramming approach.

Conditional trivial relocatability (2)

```
template<class T>
struct [[trivially_relocatable(std::is_trivially_relocatable_v<T>)]] ExpertWrap {
  std::pair<int*, T> p;
  friend void swap(ExpertWrap& a, ExpertWrap& b) noexcept { std::swap(a.p, b.p); }
  ExpertWrap() { p.first = new int; }
  ExpertWrap(ExpertWrap&& rhs) noexcept : p(rhs.p) { rhs.p.first = nullptr; }
  auto& operator=(ExpertWrap rhs) noexcept { swap(*this, rhs); }
  ~ExpertWrap() { delete p.first; }
};
```

```
using R = std::vector<int>;
using NR = std::list<int>;
```

In P1144R3, I've decided that this syntax removes enough library complexity to be worth the added language complexity.

```
static_assert(std::is_trivially_relocatable_v<ExpertWrap<R>>);
static_assert(!std::is_trivially_relocatable_v<ExpertWrap<NR>>);
```

Conditional trivial relocatability (3)

```
template<class T>
struct [[maybe_trivially_relocatable]] ExpertWrap {
  std::pair<int*, T> p;
  friend void swap(ExpertWrap& a, ExpertWrap& b) noexcept { std::swap(a.p, b.p); }
  ExpertWrap() { p.first = new int; }
  ExpertWrap(ExpertWrap&& rhs) noexcept : p(rhs.p) { rhs.p.first = nullptr; }
  auto& operator=(ExpertWrap rhs) noexcept { swap(*this, rhs); }
  ~ExpertWrap() { delete p.first; }
};
```

```
using R = std::vector<int>;
using NR = std::list<int>;
```

John McCall suggested this version.
Here `ExpertWrap` is trivially relocatable
if and only if all its bases and members are.

```
static_assert(std::is_trivially_relocatable_v<ExpertWrap<R>>);
static_assert(!std::is_trivially_relocatable_v<ExpertWrap<NR>>);
```

Isn’t this all undefined behavior?

- It’s undefined behavior to `memcpy` bits and bytes between objects of non-trivially copyable types.
- P1144R1 says let’s change that. It is now ***legal*** to `memcpy` between objects of non-trivially copyable (but trivially relocatable) types. Problem solved!
- But doesn’t that permit the user to “make a copy” of a `unique_ptr` via `memcpy`? Copying should be legal only if we can extract a promise from the user not to look at the source object’s value afterward.

Isn’t this all undefined behavior?

- P1144R3 adds a library algorithm `std::relocate_at(from, to)`. This algorithm is documented to begin the lifetime of its destination, like `std::construct_at`, and end the lifetime of its source, like `std::destroy_at`.
- For non-trivially relocatable types, this is just move-construct plus destroy.
- For trivially relocatable types, this *can* be implemented as a `memcpy` surrounded by some compiler magic to begin and end lifetimes appropriately.
- In practice, that “compiler magic” is a no-op.

Similar to `std::bless`

P0593 `std::bless` works in a similar way.

```
struct ListNode {
    ListNode *next;
    T t;
};
void example(void *raw_bytes) {
    std::bless(raw_bytes, sizeof(ListNode));
    ListNode *node = (ListNode *)raw_bytes;
    ::new (&node->t) T();  // There’s no T there...
    node->next = nullptr;  // ...but there’s a next ptr?
}
```

Easy implementation of `std::bless`

```
void example(void *raw_bytes) {
    std::bless(raw_bytes, sizeof(ListNode));
    ListNode *node = (ListNode *)raw_bytes;
    ::new (&node->t) T();
    node->next = nullptr;
}
```

When the compiler sees this line...

```
void bless(void *p, size_t n);
```

...it has no idea what the body of `bless` looks like, because that's an out-of-line library function.

Easy implementation of `std::bless`

```
void example(void *raw_bytes) {
    std::bless(raw_bytes, sizeof(ListNode));
    ListNode *node = (ListNode *)raw_bytes;
    ::new (&node->t) T();
    node->next = nullptr;
}
```

```
void bless(void *p, size_t n) {
    ::new (p) ListNode{};
    ((ListNode*)p)->t.~T();
}
```

For all the compiler knows, the body of bless might look like ***this!*** So it must optimize accordingly.

Easy implementation of `relocate_at`

```
T *relocate_at(T *src, T *dest) {
  if constexpr (std::is_trivially_relocatable_v<T>) {
    ::new (dest) T(std::move(*src));
    src->~T();
  } else {
    __triv_reloc_at(dest, src, sizeof(T));
  }
  return dest;
}

void __triv_reloc_at(void*, void*, size_t);
```

When the compiler sees this line...

...it has no idea what the body of `__triv_reloc_at` looks like.

Easy implementation of `relocate_at`

```
void __triv_reloc_at(void*, void*, size_t) {
    if (you_are_relocating_a_string) {
        ::new (dest) std::string(std::move(*src));
        src->~basic_string();
    } else if (you_are_relocating_a_vector) {
        ::new (dest) std::vector<int>(std::move(*src));
        src->~vector();
    } else if (.........
    ......
}
```

The compiler must imagine that the body of `__triv_reloc_at` ***might*** look like this! The compiler is therefore not allowed to make any optimizations that would be incorrect in such a world.

Meanwhile, in our actual world, the linker symbol `__triv_reloc_at` is just an alias for `memcpy`.

End of Part 1
Time Check
Questions?

P1144 relocation is not...

Not Pablo Halpern's N4158

Can I define my own "relocation" operation, to be used by `vector::resize` and so on, which is not quite `memcpy` but also more efficient than move-plus-destroy?

My P1144 says, definitively, ***no***.

`std::relocate_at(from, to)` is a standard library algorithm, like `std::destroy_at`.

Pablo's N4158 says, definitively, ***yes***.

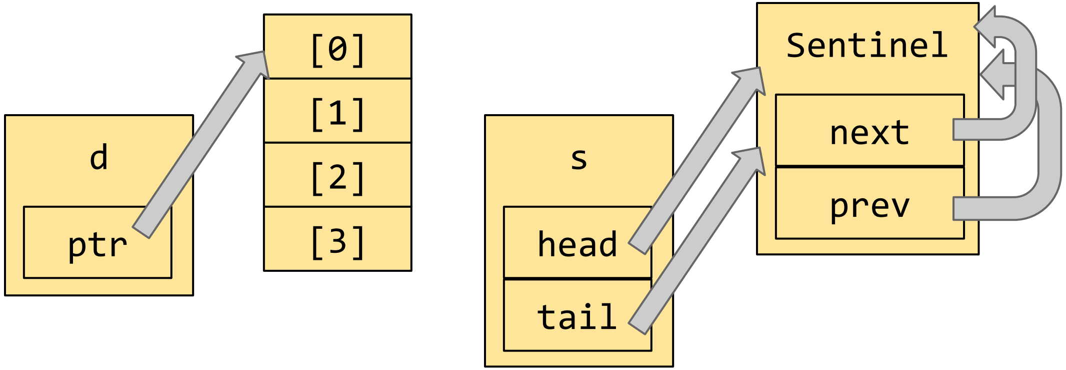
`uninitialized_destructive_move(from, to)` is an ADL customization point, like `swap`.

Not Pablo Halpern's N4158

N4158 supports some use-cases that P1144 does not.

Consider MSVC's `std::list<T>` or GCC's `std::deque<T>`. They have a "never-completely-null" invariant:

```
std::deque<int> d;
std::list<int> s;
```



They are trivially relocatable
but ***not*** nothrow-move-constructible.

Not Pablo Halpern’s N4158

Vice versa, GCC’s `std::list<T>` is nothrow-move-constructible but ***not*** trivially relocatable.

```
struct Pair {
    std::deque<int> d;
    std::list<int> s;
};

// Assuming libstdc++, then...
static_assert(!std::is_trivially_relocatable_v<Pair>);
static_assert(!std::is_nothrow_move_constructible_v<Pair>);
```

Not Pablo Halpern's N4158

```
struct Pair {
    std::deque<int> d;
    std::list<int> s;
};

static_assert(!std::is_trivially_relocatable_v<Pair>);
static_assert(!std::is_nothrow_move_constructible_v<Pair>);
```

P1144 says: game over, that's all you get.

N4158 says: You can customize the ADL customization point to relocate `d` (trivially) and then relocate `s` (non-trivially but still nothrow-ly). Thus, `Pair` can be made *nothrow-relocatable*.

Not Pablo Halpern's N4158

```
struct Pair {
    std::deque<int> d;
    std::list<int> s;
    friend void uninitialized_destructive_move(
                              Pair *from, Pair *to) noexcept {
        using std::uninitialized_destructive_move;
        uninitialized_destructive_move(&from->d, &to->d);
        uninitialized_destructive_move(&from->s, &to->s);
    }
};

static_assert(!std::is_nothrow_move_constructible_v<Pair>);
static_assert(std::is_nothrow_destructive_movable_v<Pair>);
```

Not Pablo Halpern’s N4158

```
struct Pair {
    std::string first;
    std::vector<int> second;
};

// N4158 says you must do this. Danger, Will Robinson!
template<>
struct std::is_trivially_destructive_movable<Pair>
    : std::true_type {};

static_assert(std::is_trivially_destructive_movable_v<Pair>);
```

Not Pablo Halpern’s N4158

TLDR:

- N4158 could achieve “nothrow-relocatability” in more cases than P1144
- Therefore it could avoid the vector pessimization in more cases
- But N4158 was ***not simple***
- Relied on an ADL customization point
- Also relied on reopening namespace `std` to provide explicit warrants
    - `std::is_trivially_destructive_movable`
      was a specialization point like `std::is_error_code_enum`,
      not a type-trait like `std::is_trivially_destructible`

P1144 relocation is not...

Not Denis Bider’s P0023

```
struct Example {
    A a;
    B b;

    >>Example(Example& rhs) :
        >>a(rhs.a), >>b(rhs.b) {}  // or =default
};

void relocate(Example *from, Example *to) {
    ::new (to) >>Example(*from);
}
```

Not Denis Bider’s P0023

TLDR:

- P0023 adds a completely new operation: the “relocator.” (Like a constructor or a destructor, but not the same as either.)
- “Relocation” is a new core-language operation with its own core-language syntax. It is not “the same” as move plus destroy, except by convention.
- P0023 provides a higher-level `std::relocate_or_move(dest, src)` to dispatch between “relocate” (if possible) or “move+destroy,” just like the STL provides `std::move_if_noexcept` to dispatch between “move” (if possible) or “copy.”
- The compiler figures out trivial relocatability the same way it figures out trivial destructibility or trivial copy-constructibility.

Not Denis Bider’s P0023

```
struct Uniq {
    int *p = nullptr;
    Uniq() = default;
    Uniq(Uniq&& rhs) : p(rhs.p) { rhs.p = nullptr; }
    friend void swap(Uniq& a, Uniq& b) { std::swap(a.p, b.p); }
    void operator=(Uniq rhs) { swap(*this, rhs); }
    ~Uniq() { delete p; }
};

static_assert(!std::is_relocatable_v<Uniq>);
static_assert(!std::is_trivially_relocatable_v<Uniq>);
```

Not Denis Bider’s P0023

```
struct Uniq {
    int *p = nullptr;
    Uniq() = default;
    Uniq(Uniq&& rhs) : p(rhs.p) { rhs.p = nullptr; }
    friend void swap(Uniq& a, Uniq& b) { std::swap(a.p, b.p); }
    void operator=(Uniq rhs) { swap(*this, rhs); }
    ~Uniq() { delete p; }
    >>Uniq(Uniq&) = default;
};

static_assert(std::is_relocatable_v<Uniq>);
static_assert(std::is_trivially_relocatable_v<Uniq>);
```

P1144 relocation
is not...

Not Niall Douglas's P1029

```
struct Uniq {
    int *p = nullptr;
    Uniq() = default;
    Uniq(Uniq&& rhs)
        : p(rhs.p) { rhs.p = nullptr; }
    friend void swap(Uniq& a, Uniq& b) { std::swap(a.p, b.p); }
    void operator=(Uniq rhs) { swap(*this, rhs); }
    ~Uniq() { delete p; }
};

static_assert(!std::is_move_construction_relocating_v<Uniq>);
```

Not Niall Douglas's P1029

```
struct Uniq {
    int *p = nullptr;
    Uniq() = default;
    [[move_relocates]] Uniq(Uniq&& rhs)
        : p(rhs.p) { rhs.p = nullptr; }
    friend void swap(Uniq& a, Uniq& b) { std::swap(a.p, b.p); }
    void operator=(Uniq rhs) { swap(*this, rhs); }
    ~Uniq() { delete p; }
};

static_assert(std::is_move_construction_relocating_v<Uniq>);
```

Not Niall Douglas's P1029

My impression is that P1029 is extremely confused.

P1029 adds an attribute `[[move_relocates]]`. You put it on the move-constructor (only), in order to indicate a relationship between the move-constructor, the destructor, and the default constructor.

The relationship is:

> "Move-constructing `d` from `s`" is tantamount to "`memcpy`ing `s` to `d`, and then setting `s`'s bit-pattern to the bit-pattern of a default-constructed T."

Furthermore (although it's not clearly explained in the paper):

> "Destroying a `T` in the default-constructed state has no side effects." &/or "Default-constructing a `T` has no side effects."

Not Niall Douglas's P1029

Furthermore, maybe (it's not clear in the paper):

> "No operation on `T` cares where the `T` is physically located in memory."

P1029 uses this last property to claim that any `T` which has "relocating move-construction" can therefore be passed in CPU registers because it is no longer important to match up the constructors and destructors.

```
struct Widget {
  Widget() {
    printf("A Widget constructed at %p...\n", (void*)this);
  }
  ~Widget() {
    printf("...might validly be destroyed at %p.\n", (void*)this);
  }
};
```

Not Niall Douglas's P1029

TLDR:

- P1029 proposes a property warranted by an attribute, just like P1144.
- P1029's property is very complex. It involves a "default-constructed state." Types which aren't default-constructible cannot be "move-relocating."
- P1029 deliberately, explicitly, has ABI implications. "Move-relocating" class types are supposed to be passed in registers. For example, a move-relocating `std::unique_ptr<int>` would be returned directly in `%rax`, instead of being returned on the stack.
- Thus, P1029 forces vendors to choose: status quo (no benefit for users), or ABI break (pass-by-register `unique_ptr`)? There's no way to indicate "My type is trivially relocatable but I don't want the trivial ABI."

P1144 relocation is not...

Not [[clang::trivial_abi]]

```
struct [[clang::trivial_abi]] Uniq {
    int *p = nullptr;
    Uniq() = default;
    Uniq(Uniq&& rhs)
        : p(rhs.p) { rhs.p = nullptr; }
    friend void swap(Uniq& a, Uniq& b) { std::swap(a.p, b.p); }
    void operator=(Uniq rhs) { swap(*this, rhs); }
    ~Uniq() { delete p; }
};
```

Not [[clang::trivial_abi]]

```
struct Uniq {
    int *p = nullptr;
    et cetera
};
```

```
Uniq test_function(Uniq x) {
    *x.p = 42;
    return x;
}
```

Without [[trivial_abi]]:

```
movq %rdi, %rax
movq (%rsi), %rcx
movl $42, (%rcx)
movq %rcx, (%rdi)
movq $0, (%rsi)
retq
```

With [[trivial_abi]]:

```
movq %rdi, %rax
movl $42, (%rdi)
retq
```

I have a blog post on `[[trivial_abi]]`

- It’s a hack inserted directly in the Itanium C++ ABI
- It affects the calling convention for parameters and return values of type `T`
- Since the parameter is passed in a register, the caller can’t access it after the call. Responsibility for destroying that parameter object is transferred to the callee.
  - MSVC’s convention is also “callee-destroy.” The Itanium convention is “caller-destroy” by default, but `[[trivial_abi]]` changes that.
- `[[trivial_abi]]` does not elide or eliminate the observable side effects of move-constructors and destructors!

I have a blog post on `[[trivial_abi]]`

```
struct [[clang::trivial_abi]] A { ~A(); };
struct                        B { ~B(); };
struct [[clang::trivial_abi]] C { ~C(); };

void f(A, B, C, B) { }

void test() {
    f(A(), B(), C(), B());
}
```

Parameters of types `A` and `C` are callee-destroy; parameters of type `B` remain caller-destroy. So the params are constructed in the order `A,B,C,B` and destroyed in the order `C,A,B,B`.

P1144 relocation
is not...

Not likely to help with “persistence”

Sometimes people hear “you can relocate a whole `T` object with `memcpy`” and what they take away is “`T` doesn’t care what virtual address it’s at.”

Think `boost::interprocess` shared memory segments.

Not likely to help with “persistence”

Address space 1	Shared memory segment	Address space 2
0x1000:	`size=2`	0x2600
0x1008:	`capacity=3`	0x2608
0x1010:	`ptr=0x1018`	0x2610
0x1018:	`v[0] = 42`	0x2618
0x1020:	`v[1] = 17`	0x2620
0x1028:	`v[2] = ??`	0x2628
0x1030:	`???`	0x2630

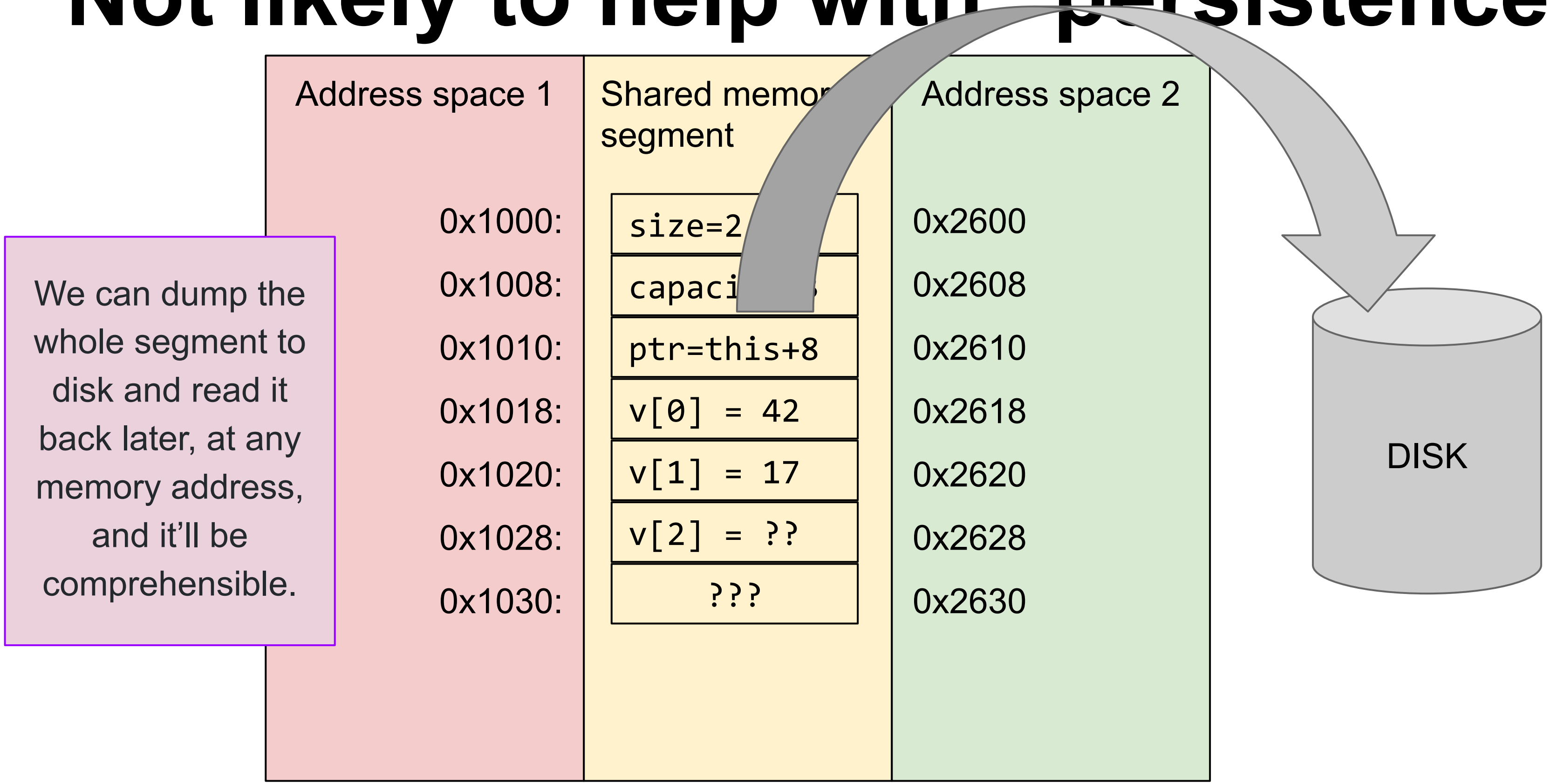
Address space 1 sees a normal `std::vector<long>`.
Address space 2 sees a wild pointer, i.e., garbage.

Not likely to help with “persistence”

Address space 1	Shared memory segment	Address space 2
0x1000:	`size=2`	0x2600
0x1008:	`capacity=3`	0x2608
0x1010:	`ptr=this+8`	0x2610
0x1018:	`v[0] = 42`	0x2618
0x1020:	`v[1] = 17`	0x2620
0x1028:	`v[2] = ??`	0x2628
0x1030:	`???`	0x2630

Boost.Interprocess fancy pointers store an **offset from `this`** rather than an absolute address. Both address spaces see a `std::vector<long, FancyAllocator>`, and are able to manipulate it.

Not likely to help with “persistence”

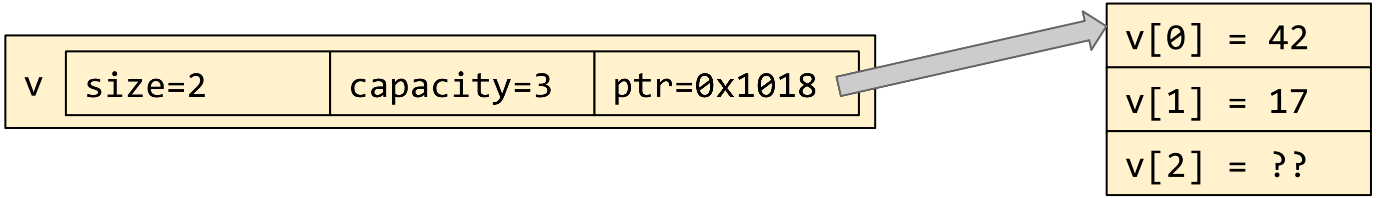


Not likely to help with “persistence”

So `is_trivially_relocatable_v<std::vector<int, FancyAllocator>>`?

No!

Go back to our familiar style of diagram.



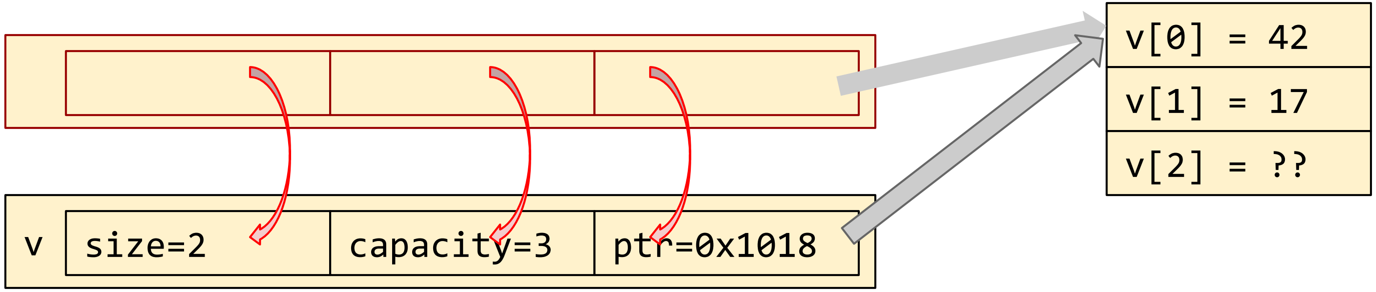
Not likely to help with “persistence”

So `is_trivially_relocatable_v<std::vector<int, FancyAllocator>>`?

Plain `vector` is trivially relocatable, because plain pointers can be `memcpy`ed.

No!

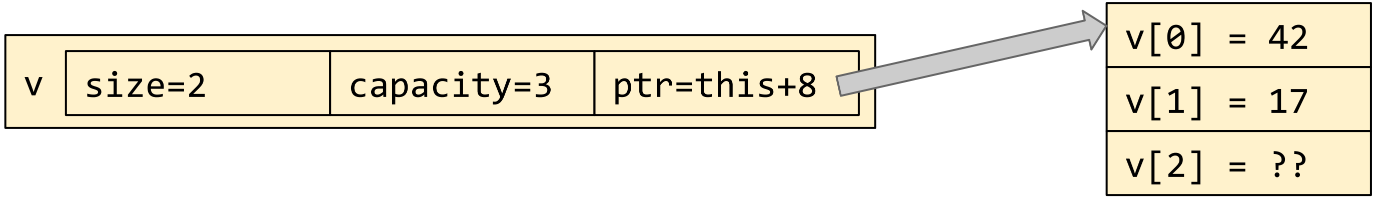
Go back to our familiar style of diagram.



Not likely to help with "persistence"

So `is_trivially_relocatable_v<std::vector<int, FancyAllocator>>`?

No!

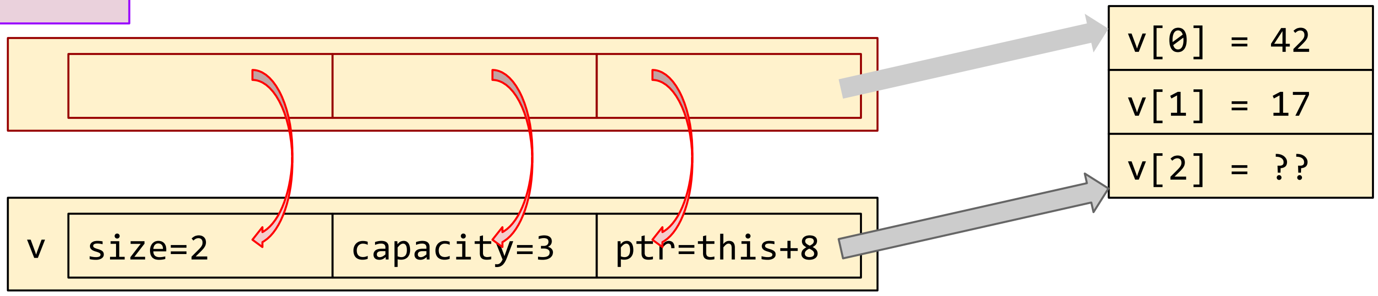


Not likely to help with “persistence”

So `is_trivially_relocatable_v<std::vector<int, FancyAllocator>>`?

This kind of `vector` is ***not*** trivially relocatable, because `memcpy` breaks its `offset_ptr`s.

No!



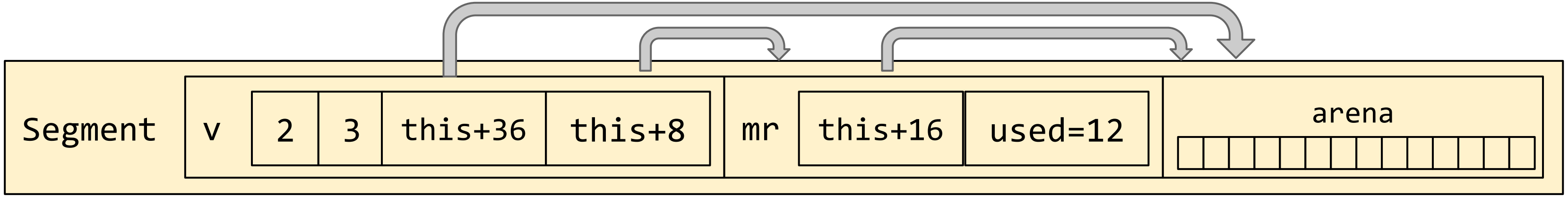
Relocation is object-level

If you want to use trivial relocation, then you want to create some `T` where `is_trivially_relocatable<T>`. You must wrap the ***whole footprint*** of your relocatable arena-segment-thing into a ***single object***.

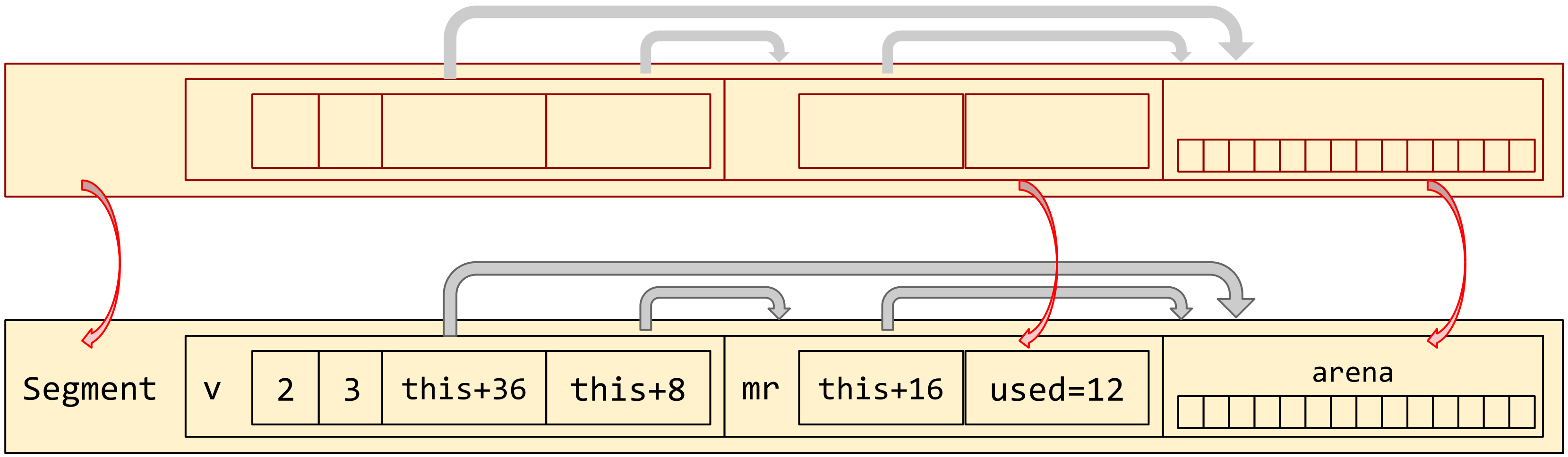
```
struct [[trivially_relocatable]] Segment {
    std::vector<int, offset_ptr_allocator<int>> v;
    offset_ptr_memoryresource myArena;
    char arena[1000];
};
```

Now you can trivially relocate a whole `Segment` from place to place within your address space:

Relocation is object-level



Relocation is object-level



Beware wrong warrants: All the stuff in the arena must ***also*** be trivially relocatable. (In this case, it's just `long`s. But what if it were `std::regex`es?)

Relocation is object-level

That was just a proof of concept. I expect `is_trivially_relocatable` will ***not*** be practically useful to people who do this kind of thing.

In practice, you ***never*** `memcpy` a giant `Segment` object, because:

- it's being shared with other threads/processes and you can't move it
- `memcpy`ing a giant object is super slow

P1144's trivial relocatability is useful for ***small*** objects that get shuffled around in memory a lot. It's not for big objects that stay in one place.

Plus, you had to write an explicit warrant for `Segment`, so I bet you got it wrong. P1144 is designed for the 99% case: Rule of Zero, no warrant, free speedup.

Open questions

- Is this what attributes are for? If not attributes, then what?

- In cases like `Segment`, `[[maybe_trivially_relocatable]]` does not suffice — we need `[[trivially_relocatable]]`. Should we provide both, anyway? Is `[[maybe_trivially_relocatable]]` “safer”?

- Technically, `swap(T& a, T& b)` cannot use “memswap” if either a or b might be a base of a non-POD class type, because there might be stuff stored in their tail padding. Every vendor gets this wrong today in `std::copy`. To what extent do we care? and if we care, to what extent does this nerf my advertised optimization of swap-based algorithms?

Open questions

- P1144 says that “relocation” is just “move + destroy.” But our `vector::reserve` optimization is actually replacing “copy + destroy” in some cases, e.g. libstdc++’s motivating `vector<deque<int>>` example.

- Our `vector::insert` and `swap` optimizations are actually replacing “move-assign-from + move-assign-to” in some cases.

  To what extent can the library assume that a trivially relocatable type behaves “normally”? Should P1144 consider `T` non-trivially-relocatable by default if it has *any* special members?

  P1144R3 actually does make this change.

End of Part 2
Questions?
Bonus slides

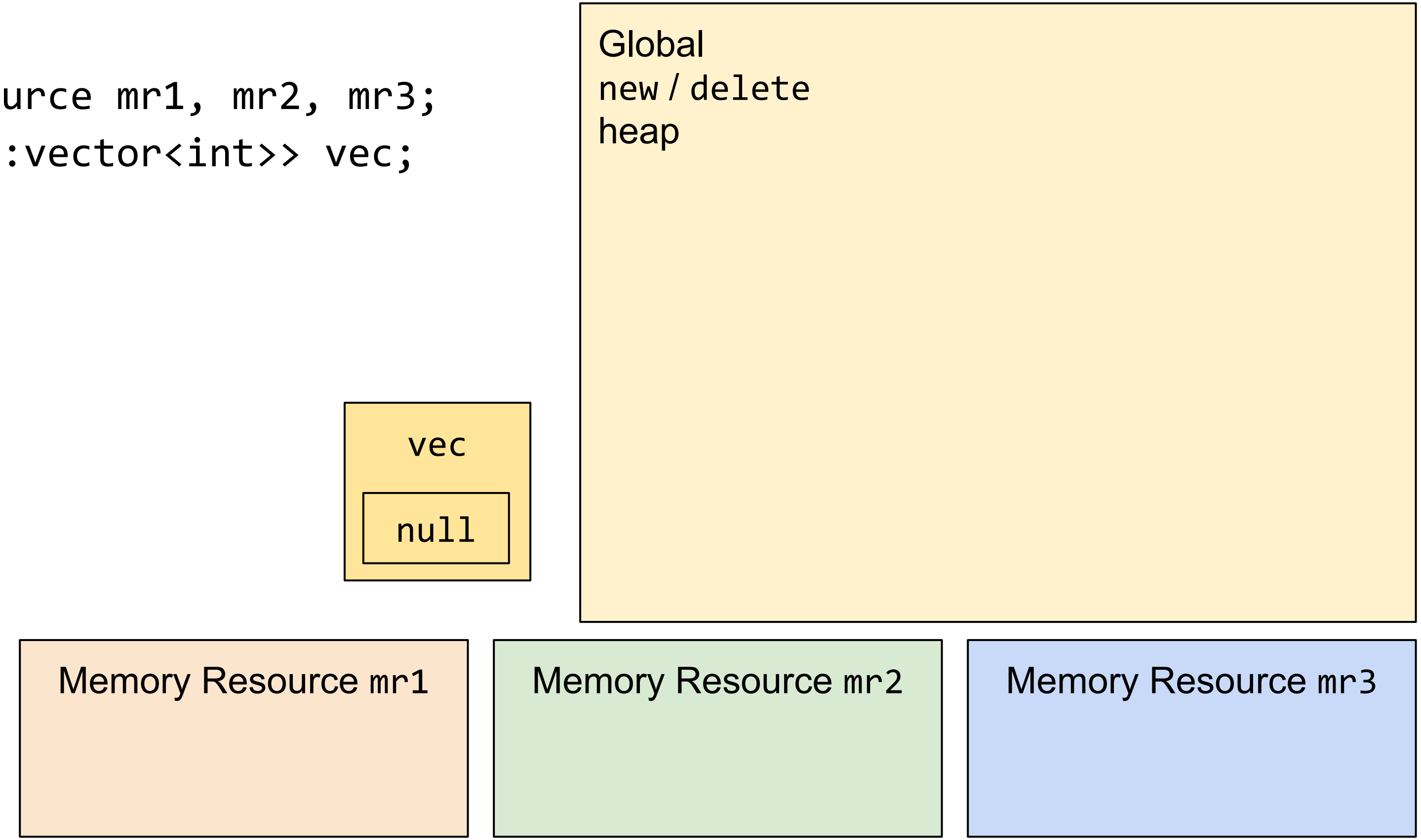
Is `pmr::vector` trivially relocatable?

Erasing from `vector<pmr::vector<int>>`

```
simple_resource mr1, mr2, mr3;
vector<pmr::vector<int>> vec;
vec.emplace_back(1, 1, &mr1);
vec.emplace_back(2, 2, &mr2);
vec.emplace_back(3, 3, &mr3);
vec.erase(vec.begin());
```

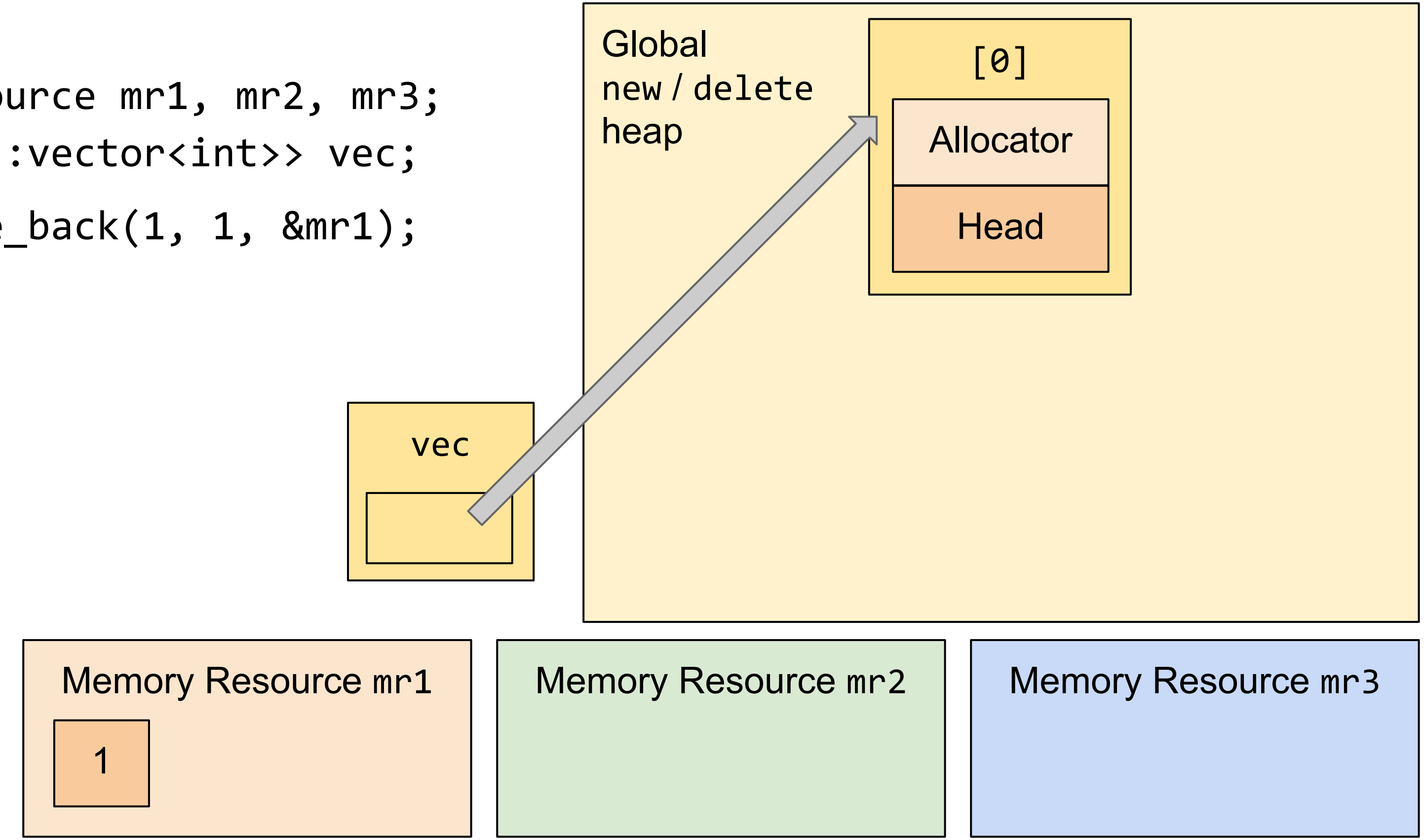
Erasing from `vector<pmr::vector<int>>`

```
simple_resource mr1, mr2, mr3;
vector<pmr::vector<int>> vec;
```



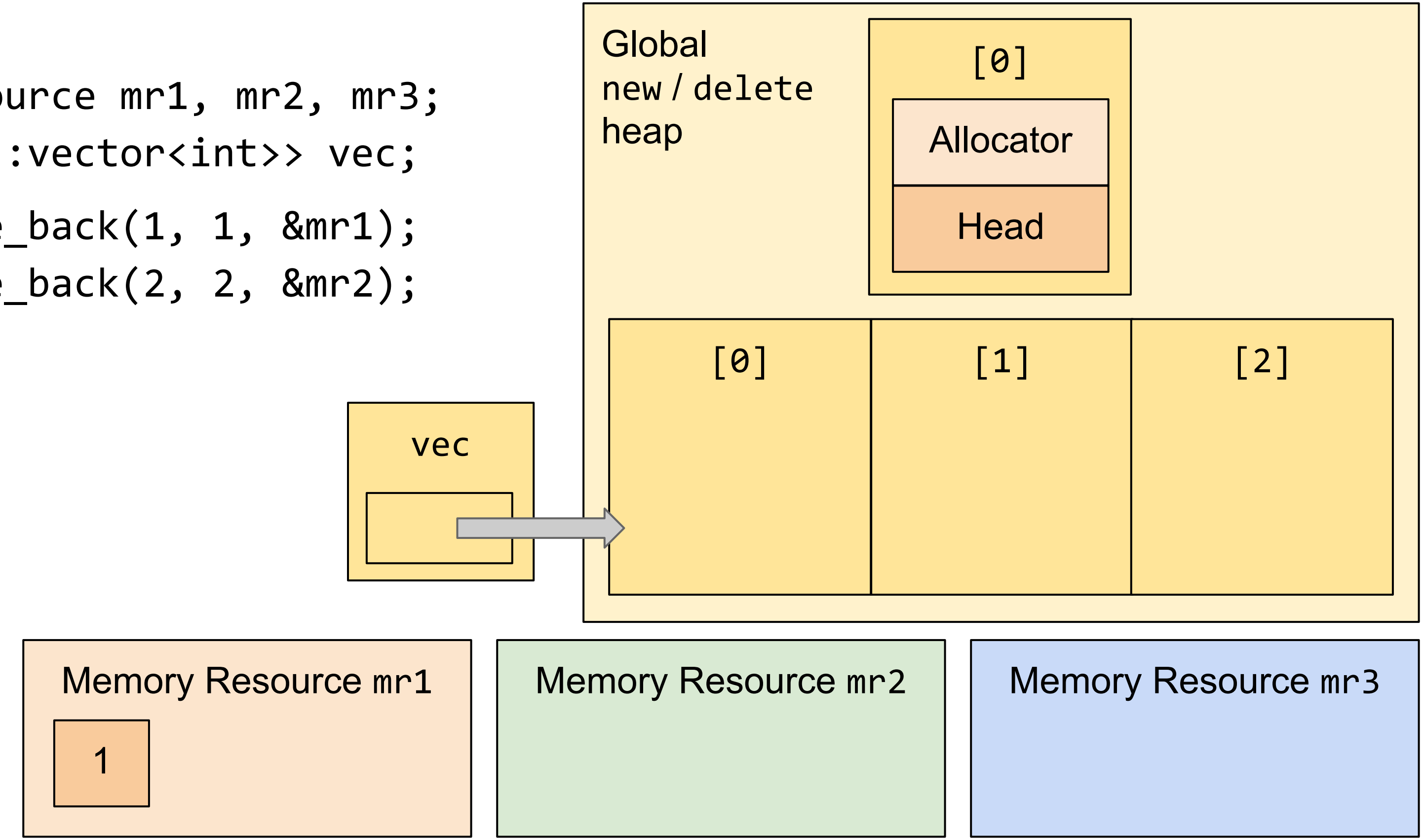
Erasing from vector<pmr::vector<int>>

```
simple_resource mr1, mr2, mr3;
vector<pmr::vector<int>> vec;
vec.emplace_back(1, 1, &mr1);
```



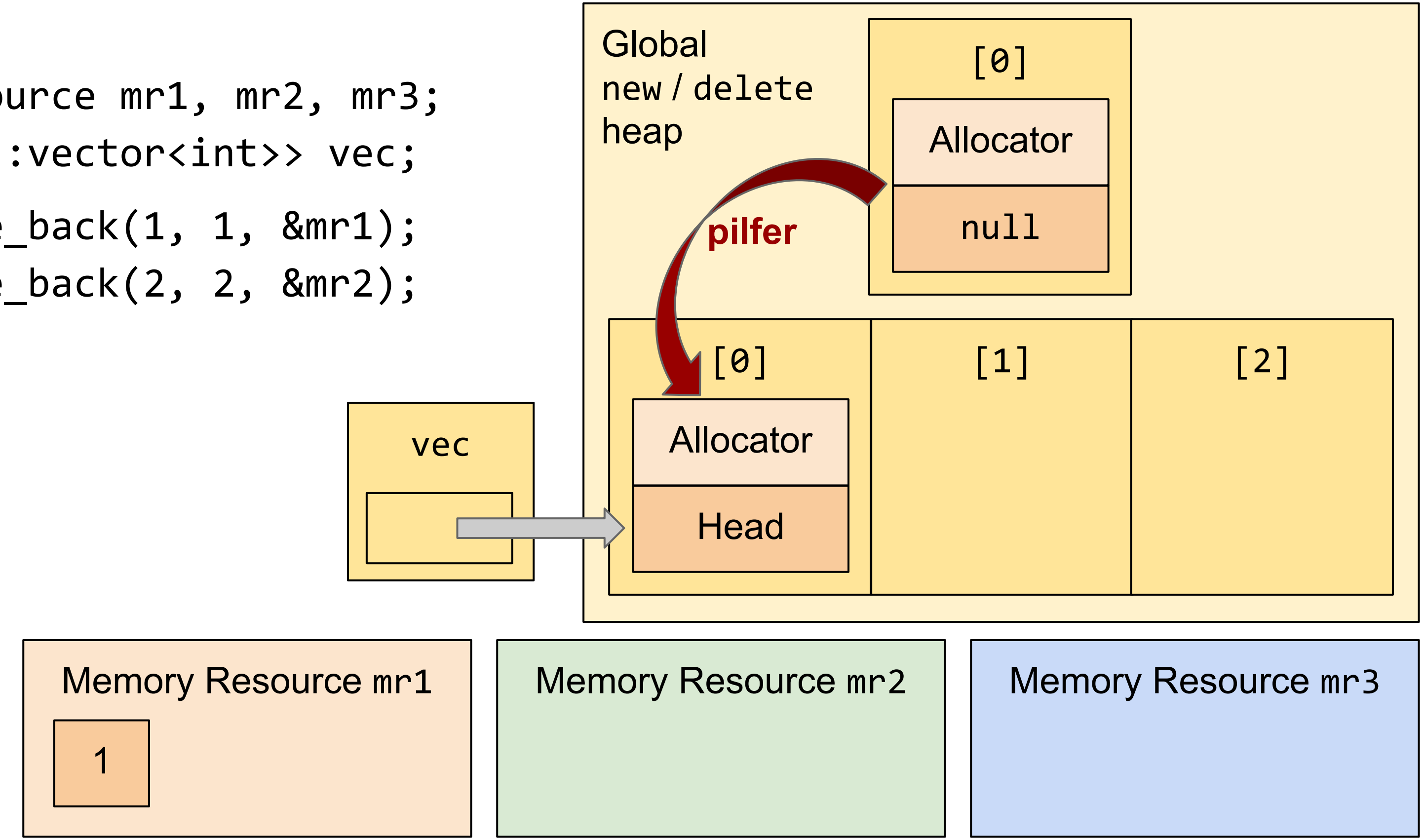
Erasing from vector<pmr::vector<int>>

```
simple_resource mr1, mr2, mr3;
vector<pmr::vector<int>> vec;
vec.emplace_back(1, 1, &mr1);
vec.emplace_back(2, 2, &mr2);
```



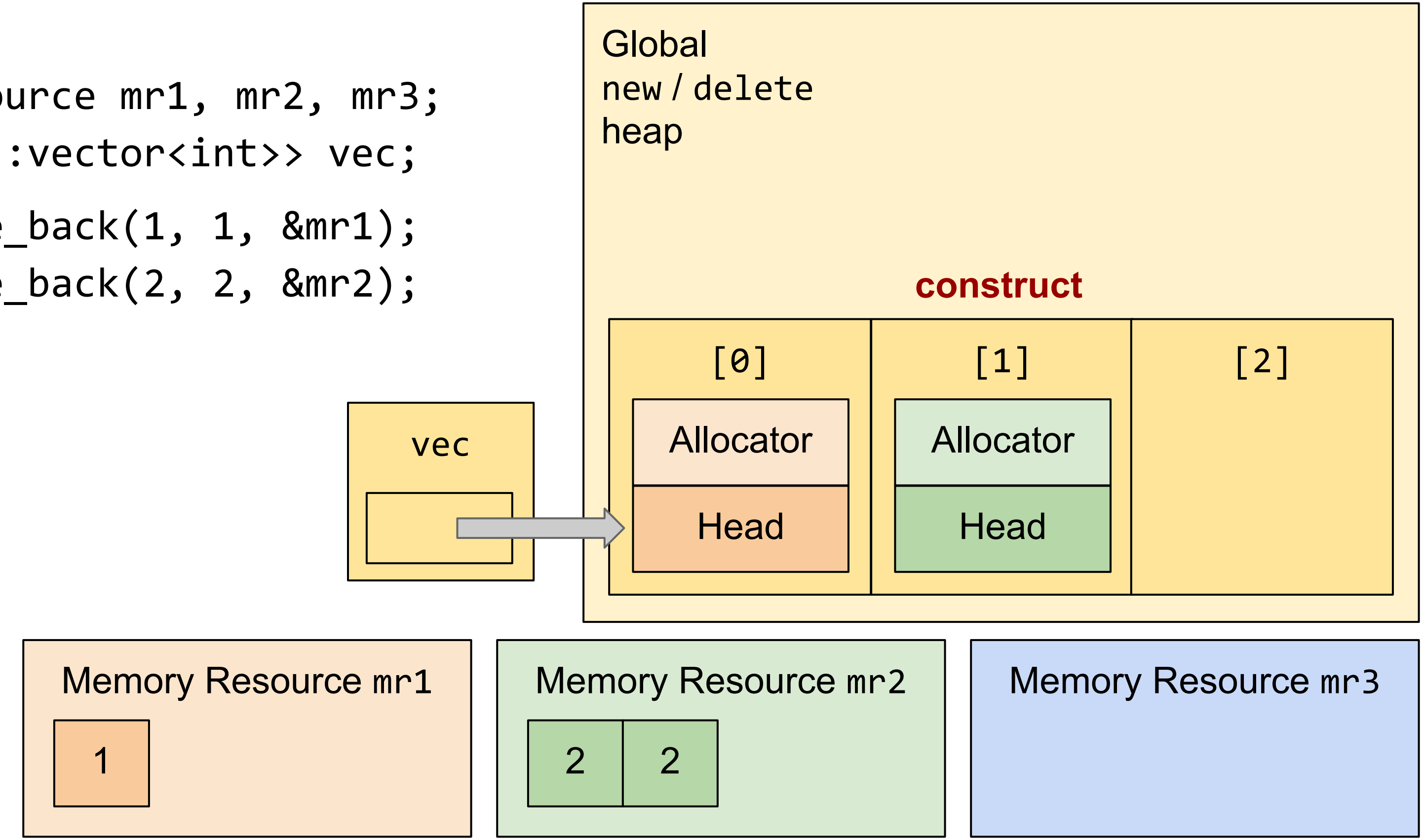
Erasing from vector<pmr::vector<int>>

```
simple_resource mr1, mr2, mr3;
vector<pmr::vector<int>> vec;
vec.emplace_back(1, 1, &mr1);
vec.emplace_back(2, 2, &mr2);
```



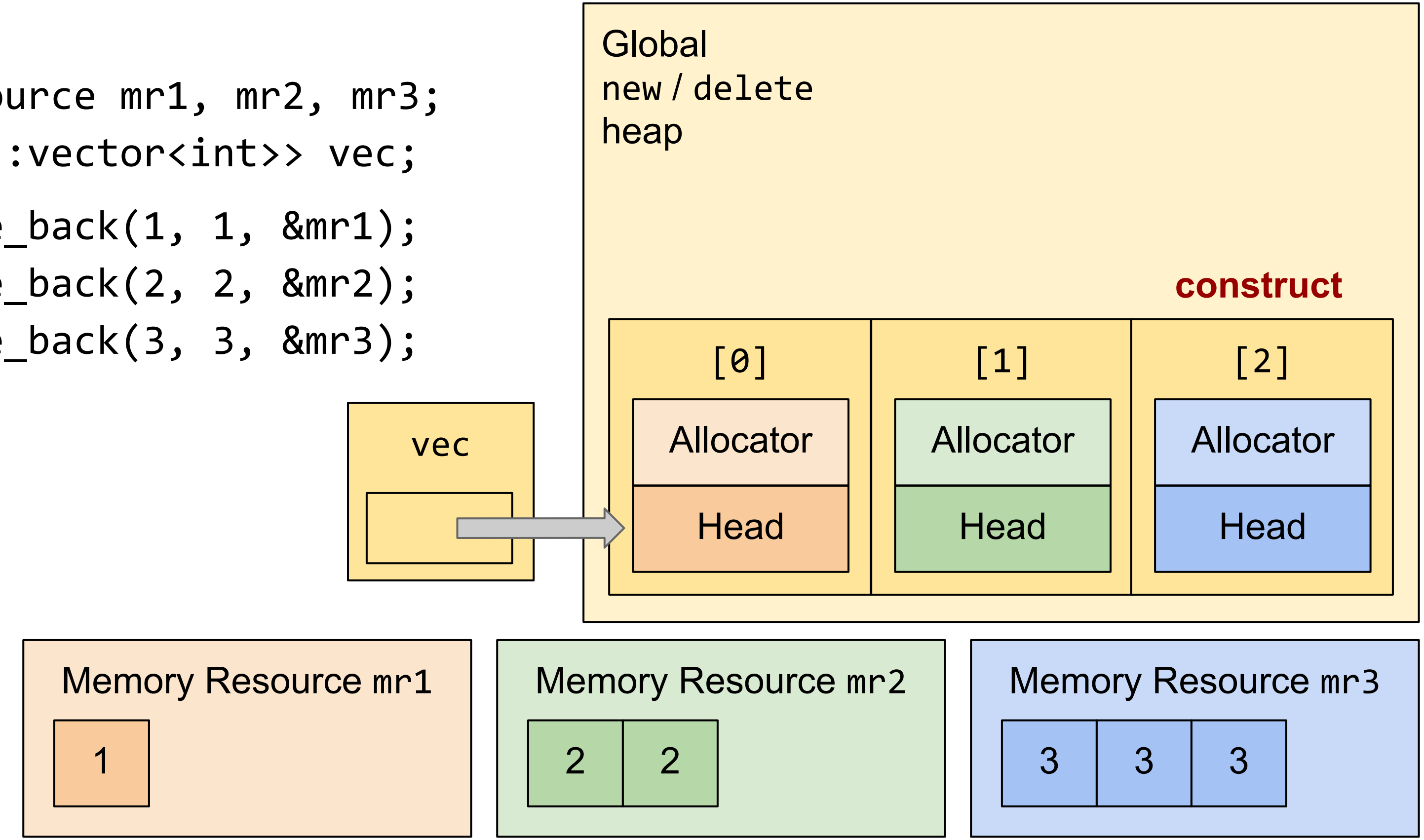
Erasing from vector<pmr::vector<int>>

```
simple_resource mr1, mr2, mr3;
vector<pmr::vector<int>> vec;
vec.emplace_back(1, 1, &mr1);
vec.emplace_back(2, 2, &mr2);
```



Erasing from vector<pmr::vector<int>>

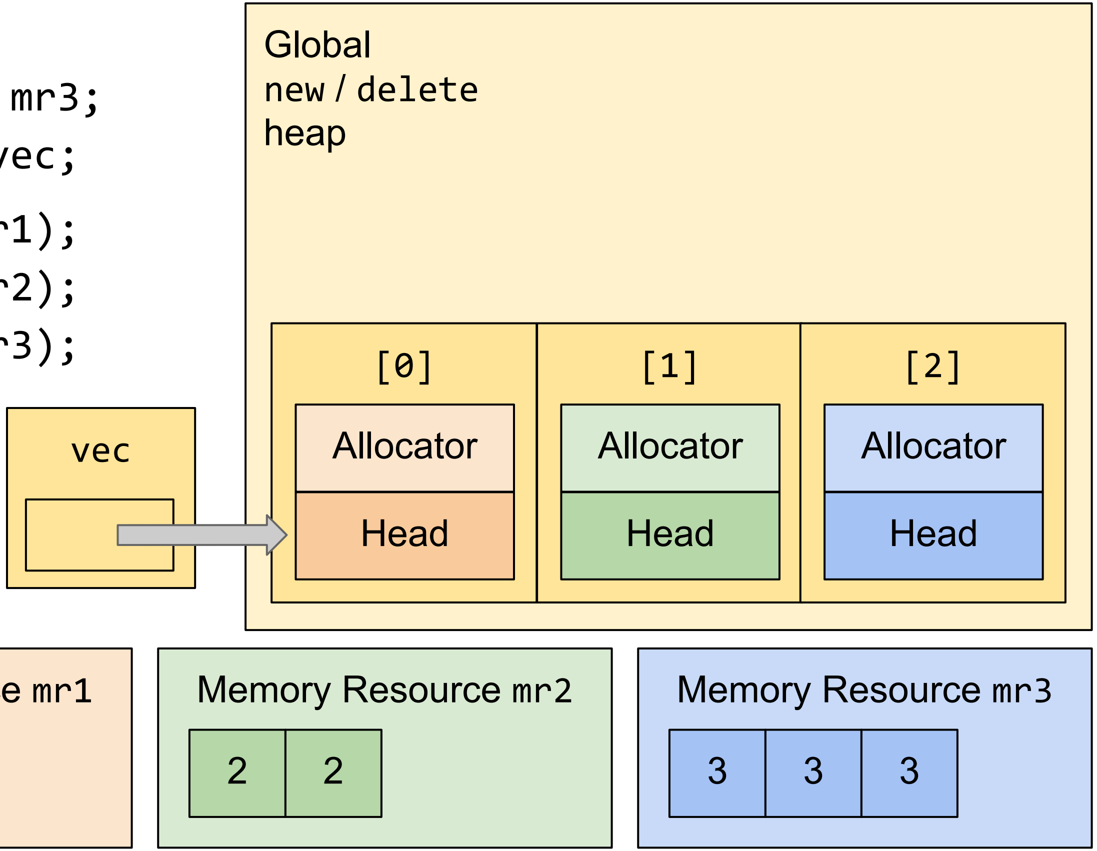
```
simple_resource mr1, mr2, mr3;
vector<pmr::vector<int>> vec;
vec.emplace_back(1, 1, &mr1);
vec.emplace_back(2, 2, &mr2);
vec.emplace_back(3, 3, &mr3);
```



Erasing from vector<pmr::vector<int>>

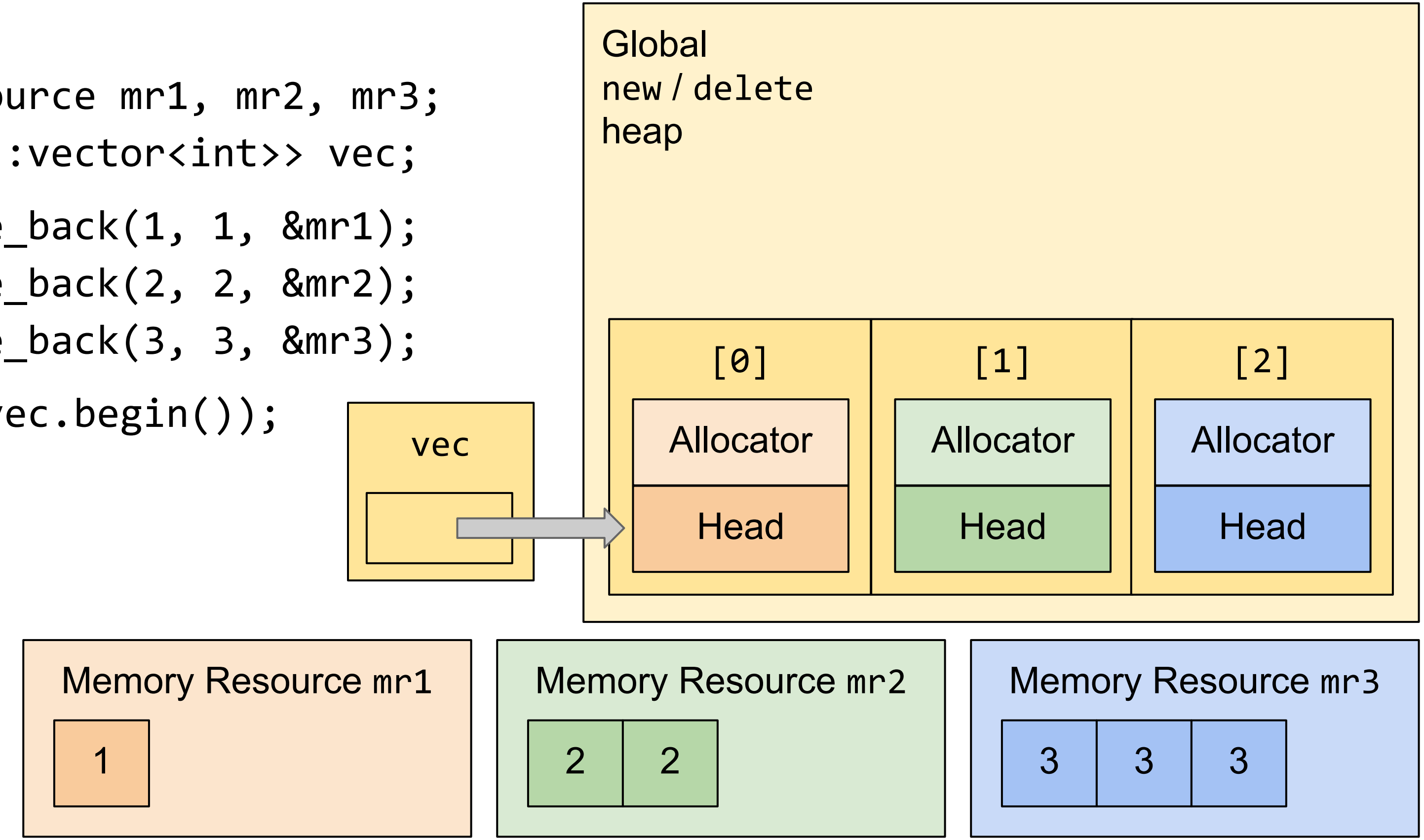
```
simple_resource mr1, mr2, mr3;
vector<pmr::vector<int>> vec;
vec.emplace_back(1, 1, &mr1);
vec.emplace_back(2, 2, &mr2);
vec.emplace_back(3, 3, &mr3);
```

Now here comes the fun part.



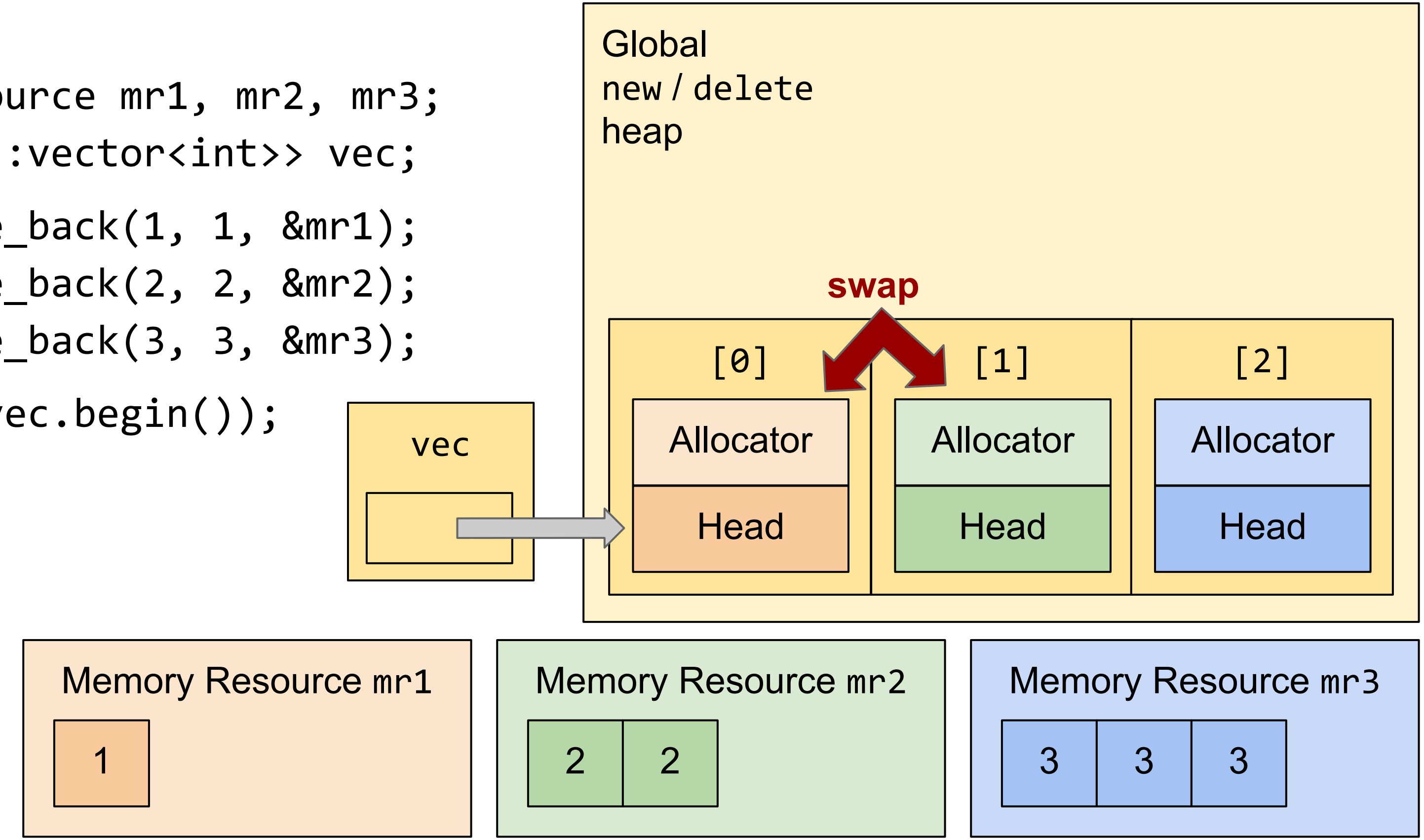
Erasing from vector<pmr::vector<int>>

```
simple_resource mr1, mr2, mr3;
vector<pmr::vector<int>> vec;

vec.emplace_back(1, 1, &mr1);
vec.emplace_back(2, 2, &mr2);
vec.emplace_back(3, 3, &mr3);

vec.erase(vec.begin());
```



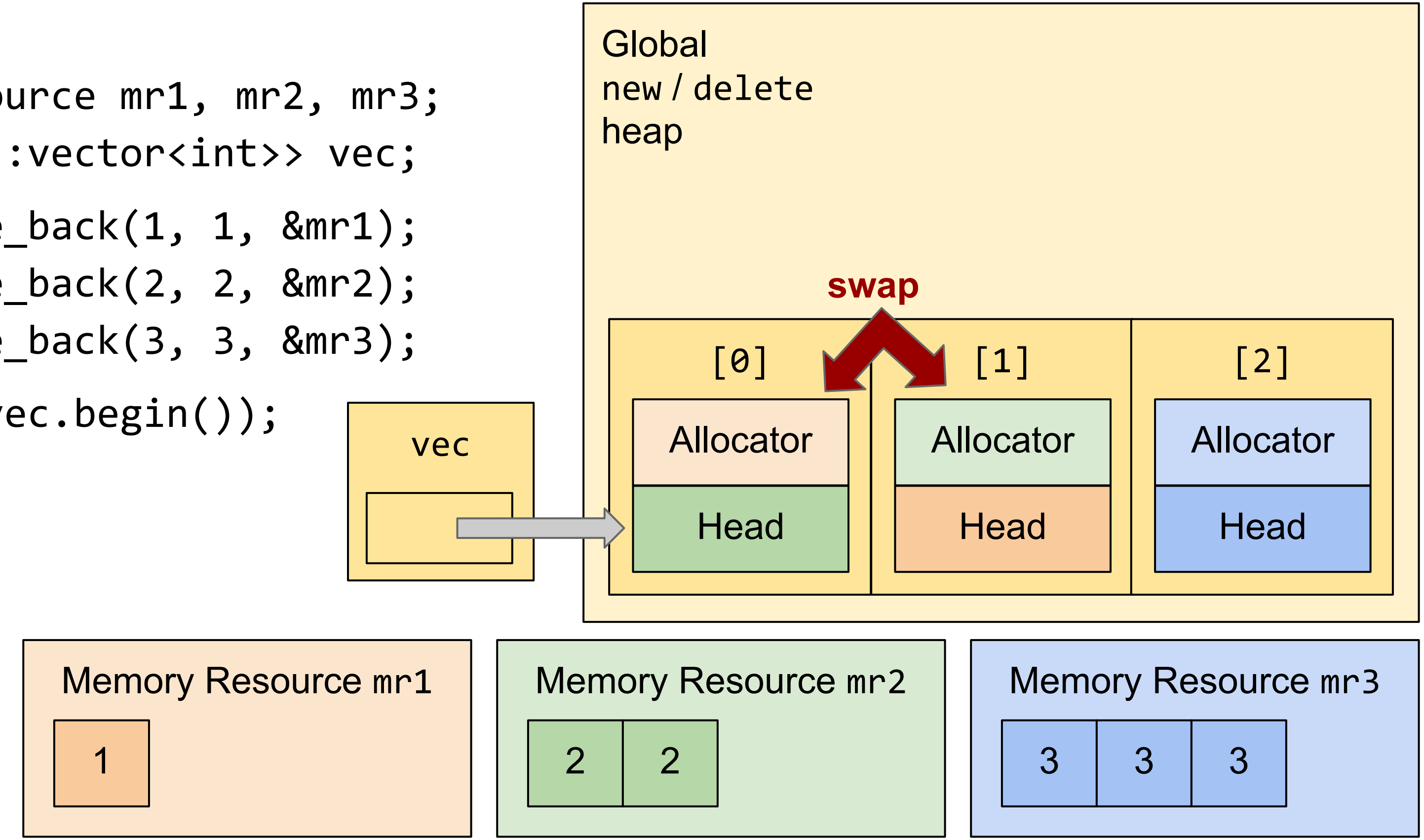
If we implement erase via `std::swap`

```
simple_resource mr1, mr2, mr3;
vector<pmr::vector<int>> vec;

vec.emplace_back(1, 1, &mr1);
vec.emplace_back(2, 2, &mr2);
vec.emplace_back(3, 3, &mr3);

vec.erase(vec.begin());
```



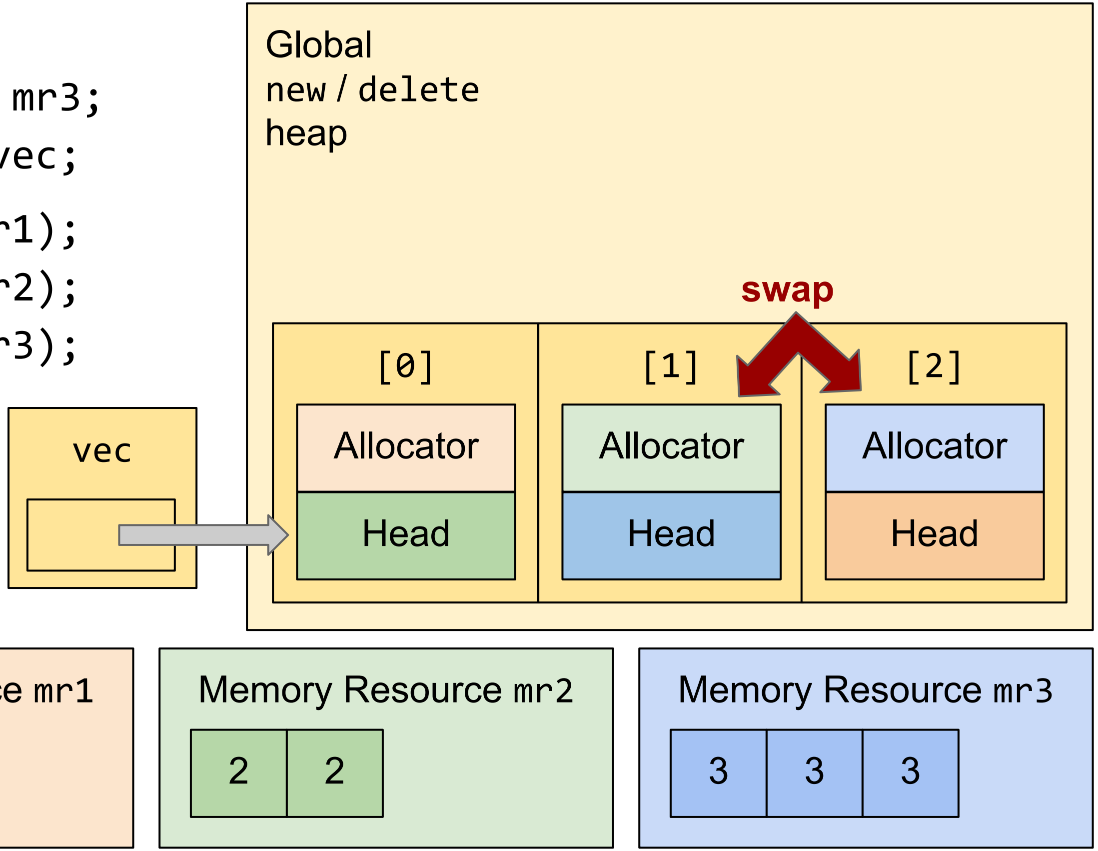
If we implement erase via `std::swap`

```
simple_resource mr1, mr2, mr3;
vector<pmr::vector<int>> vec;

vec.emplace_back(1, 1, &mr1);
vec.emplace_back(2, 2, &mr2);
vec.emplace_back(3, 3, &mr3);

vec.erase(vec.begin());
```



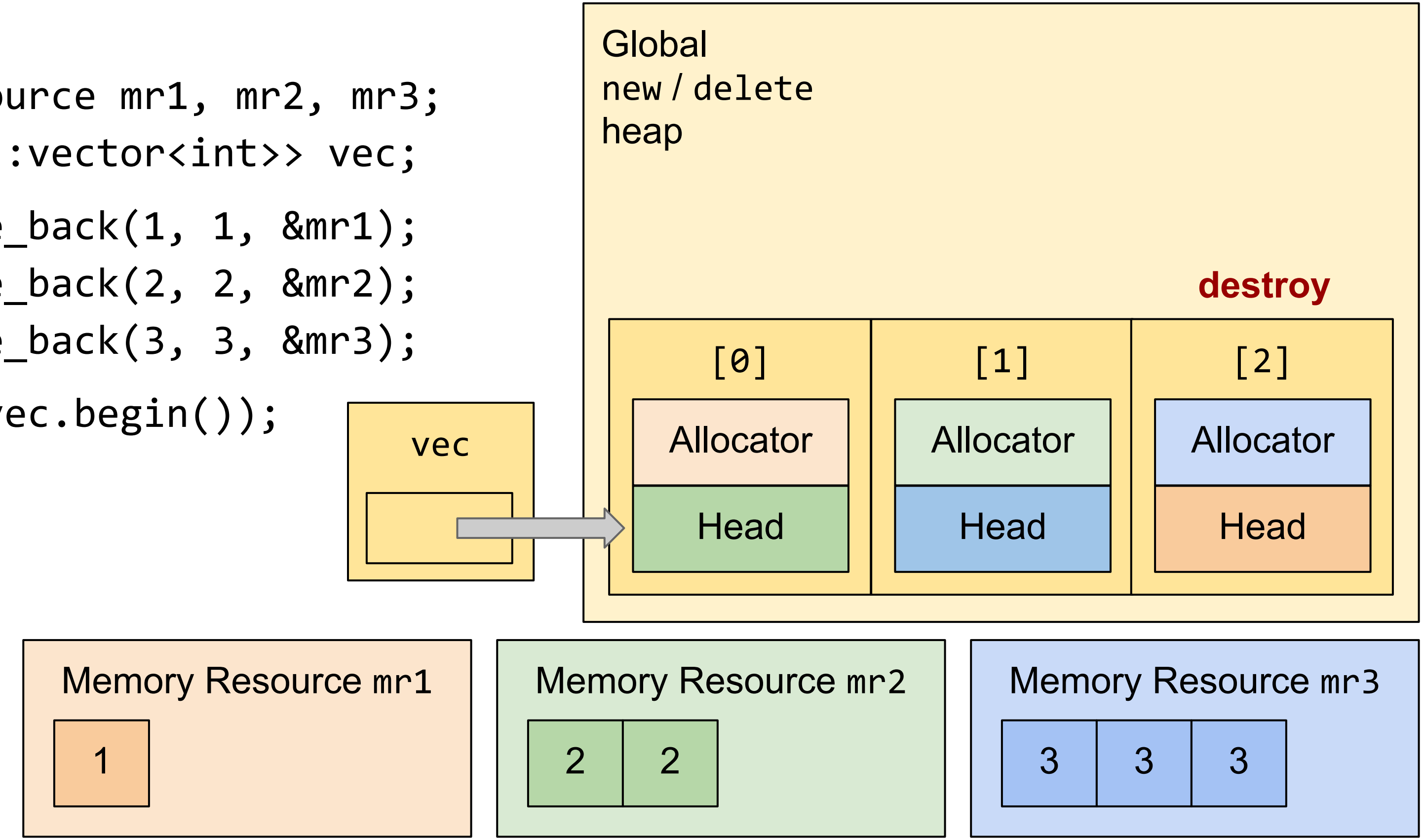
If we implement erase via `std::swap`

```
simple_resource mr1, mr2, mr3;
vector<pmr::vector<int>> vec;

vec.emplace_back(1, 1, &mr1);
vec.emplace_back(2, 2, &mr2);
vec.emplace_back(3, 3, &mr3);

vec.erase(vec.begin());
```



If we implement erase via `std::swap`

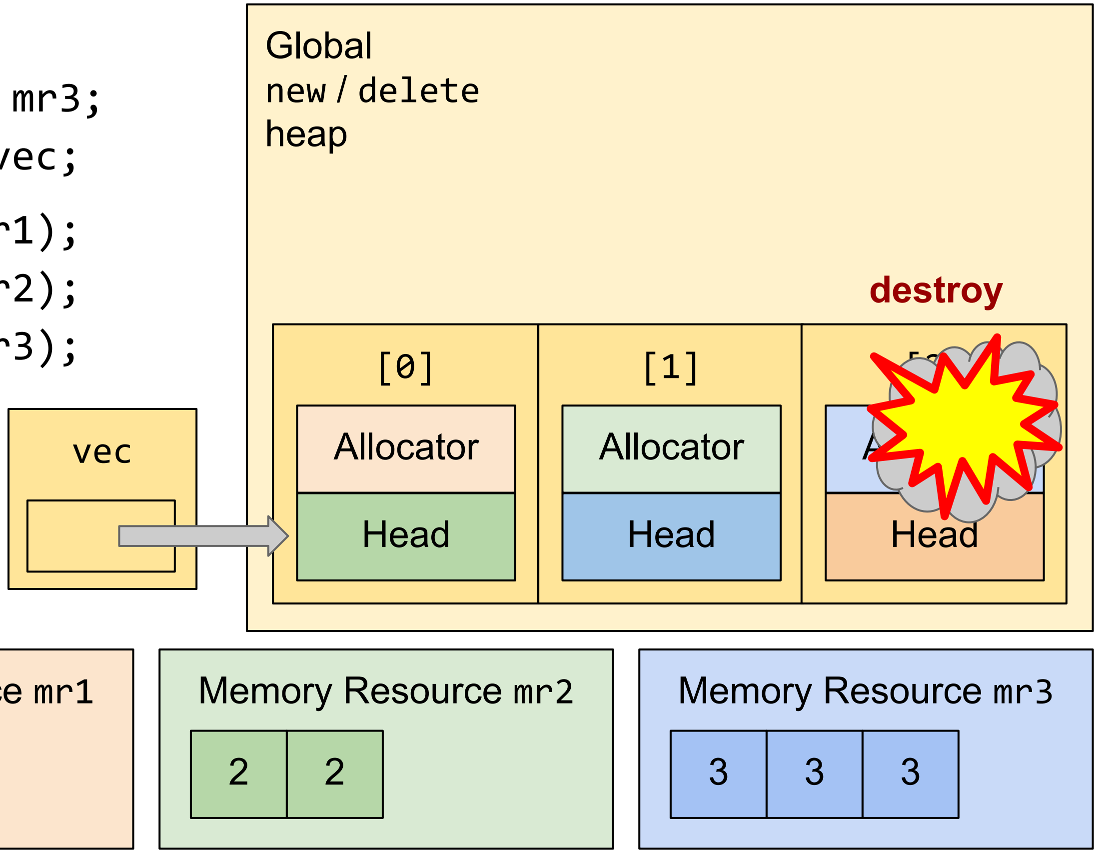
```
simple_resource mr1, mr2, mr3;
vector<pmr::vector<int>> vec;

vec.emplace_back(1, 1, &mr1);
vec.emplace_back(2, 2, &mr2);
vec.emplace_back(3, 3, &mr3);

vec.erase(vec.begin());
```



If we implement erase via std::swap

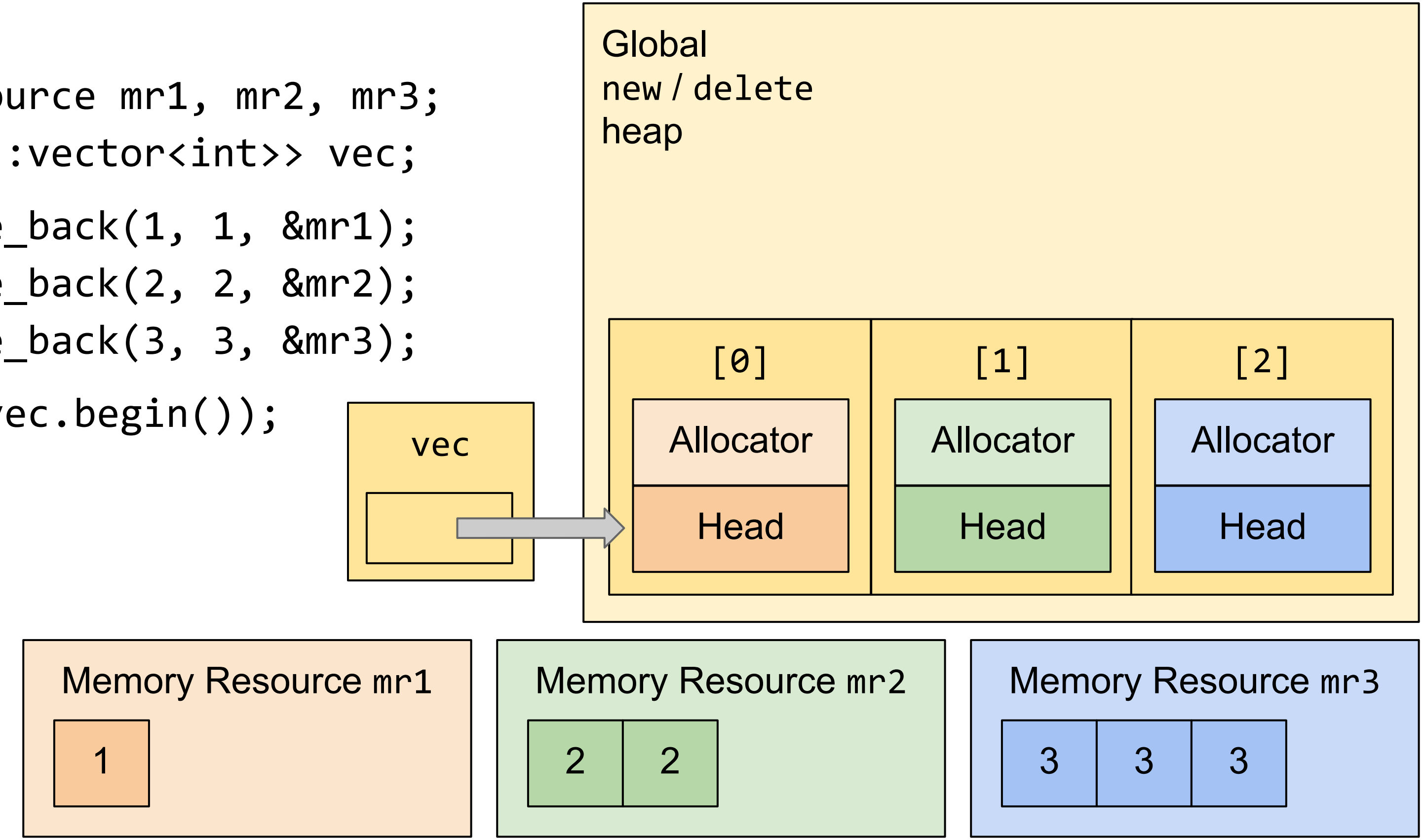
```
simple_resource mr1, mr2, mr3;
vector<pmr::vector<int>> vec;

vec.emplace_back(1, 1, &mr1);
vec.emplace_back(2, 2, &mr2);
vec.emplace_back(3, 3, &mr3);

vec.erase(vec.begin());
```

Undefined behavior, because swapping pmr::vectors with unequal allocators is UB.



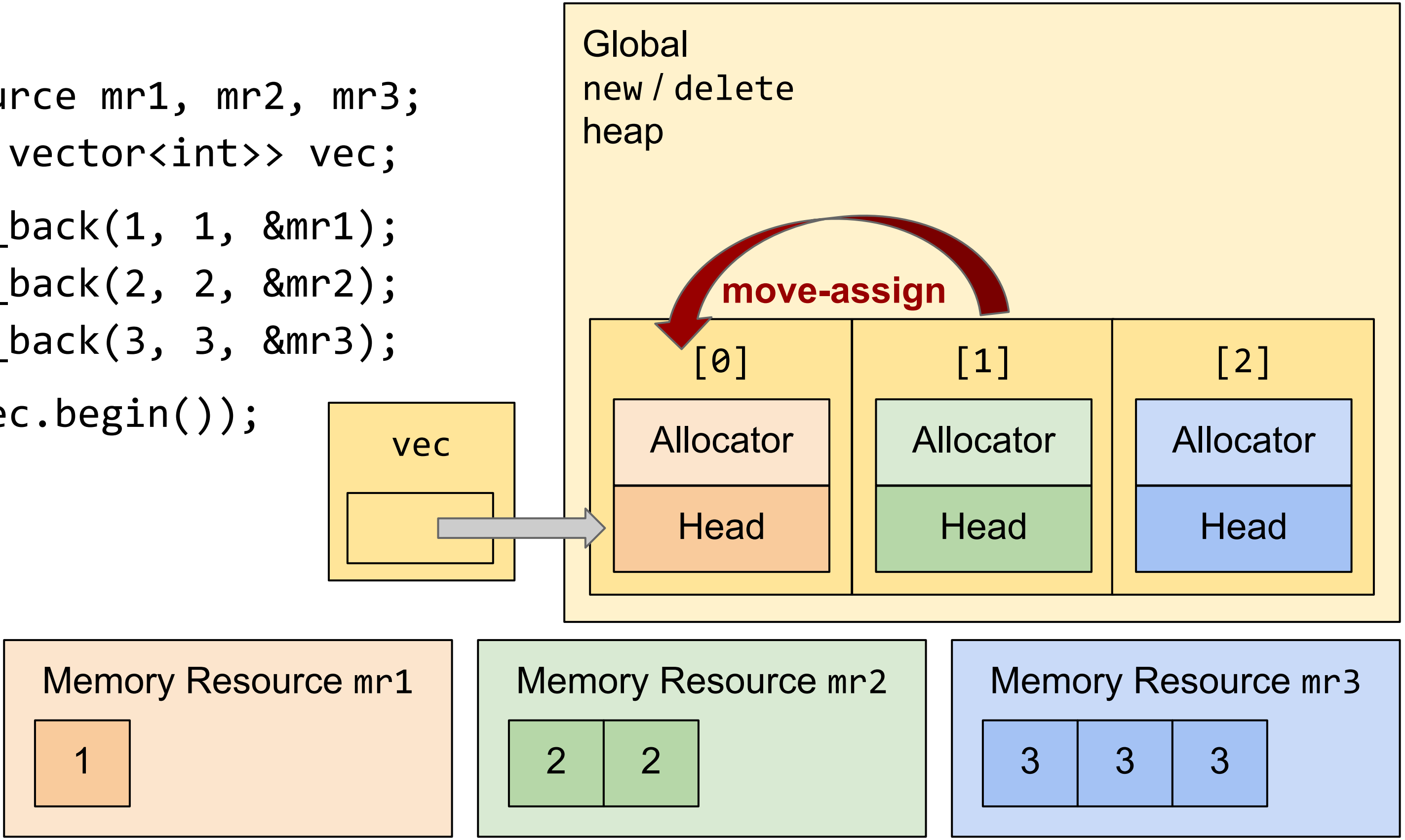
If we implement erase via move-assign

```
simple_resource mr1, mr2, mr3;
vector<pmr::vector<int>> vec;

vec.emplace_back(1, 1, &mr1);
vec.emplace_back(2, 2, &mr2);
vec.emplace_back(3, 3, &mr3);

vec.erase(vec.begin());
```



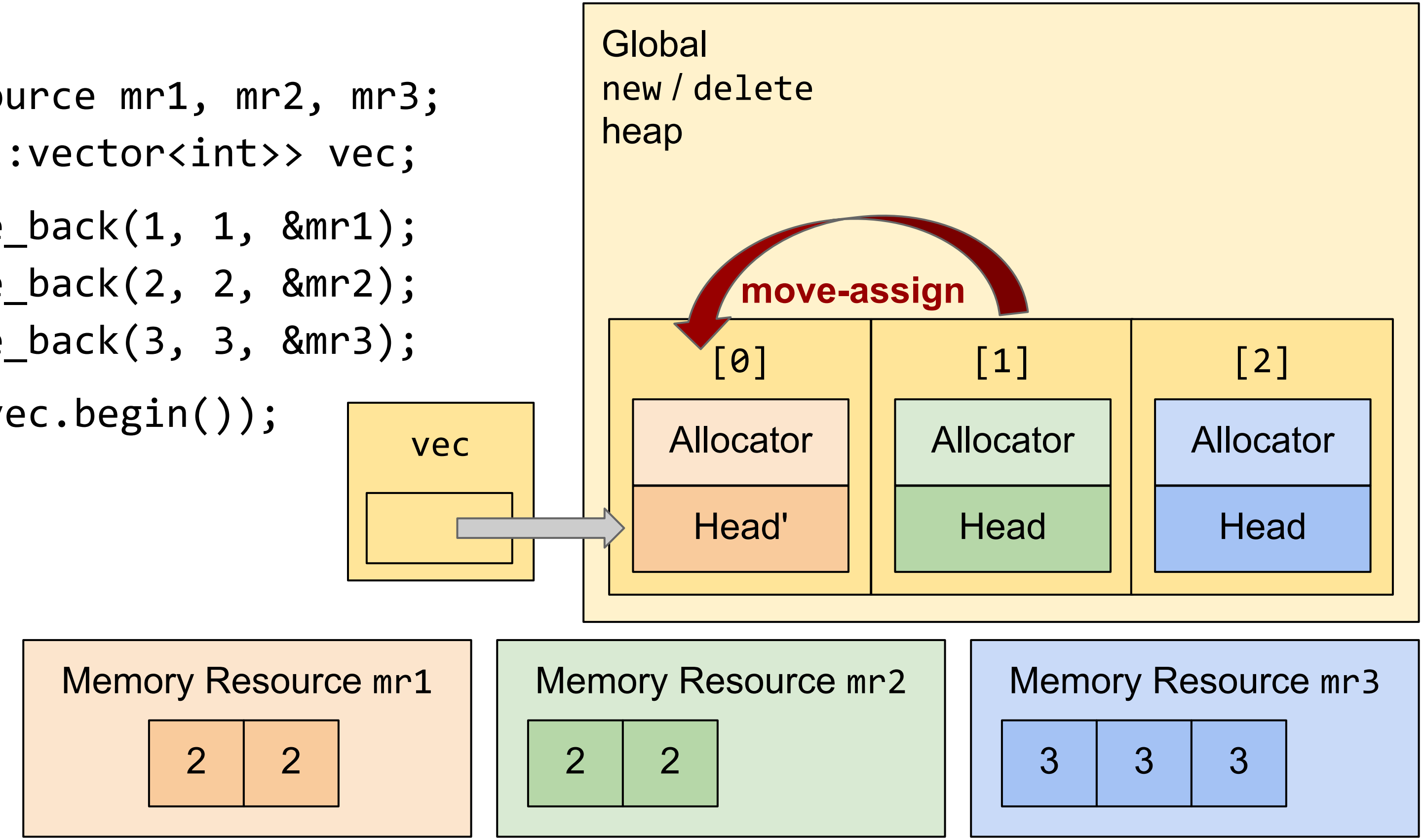
If we implement erase via move-assign

```
simple_resource mr1, mr2, mr3;
vector<pmr::vector<int>> vec;

vec.emplace_back(1, 1, &mr1);
vec.emplace_back(2, 2, &mr2);
vec.emplace_back(3, 3, &mr3);

vec.erase(vec.begin());
```



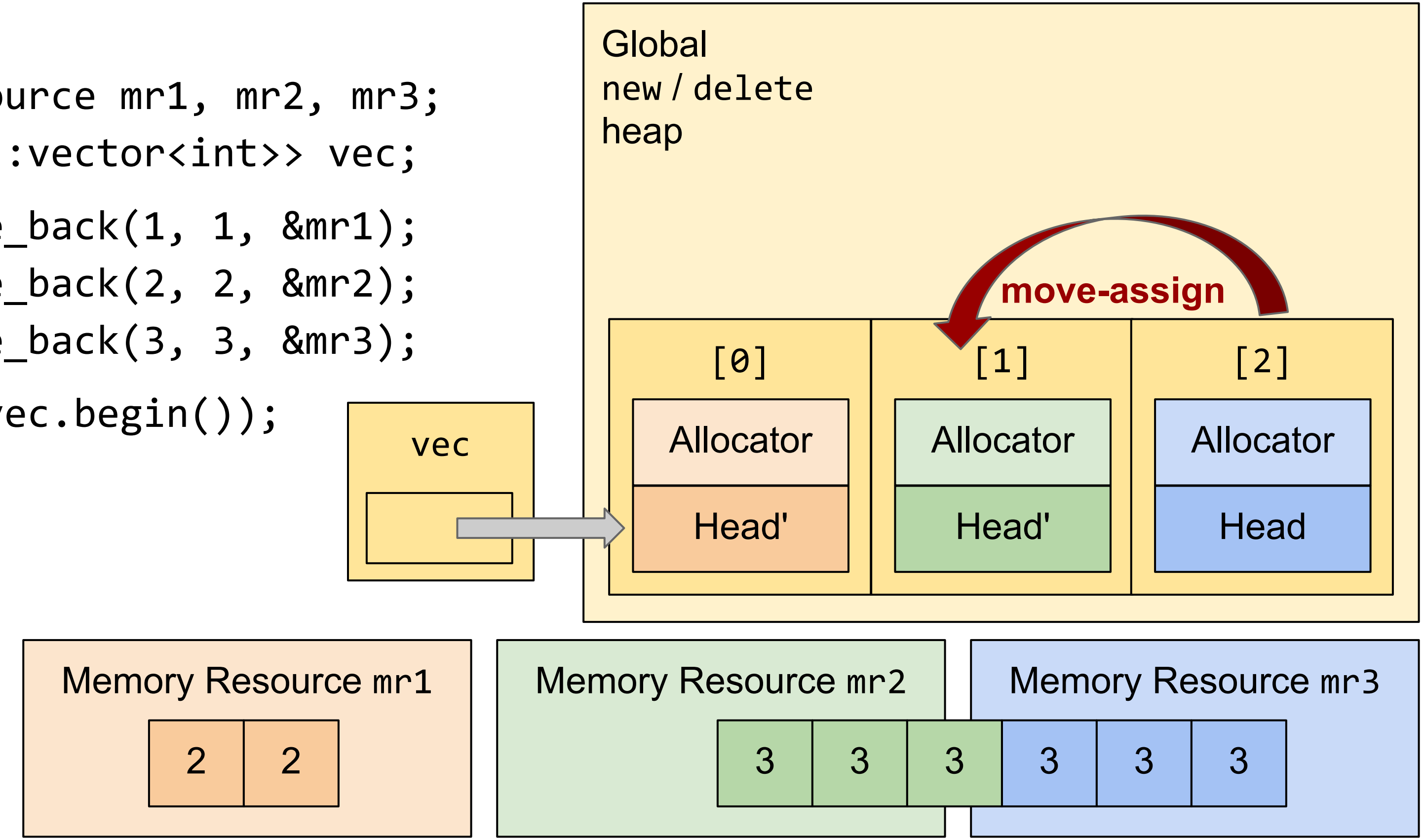
If we implement erase via move-assign

```
simple_resource mr1, mr2, mr3;
vector<pmr::vector<int>> vec;

vec.emplace_back(1, 1, &mr1);
vec.emplace_back(2, 2, &mr2);
vec.emplace_back(3, 3, &mr3);

vec.erase(vec.begin());
```



If we implement erase via move-assign

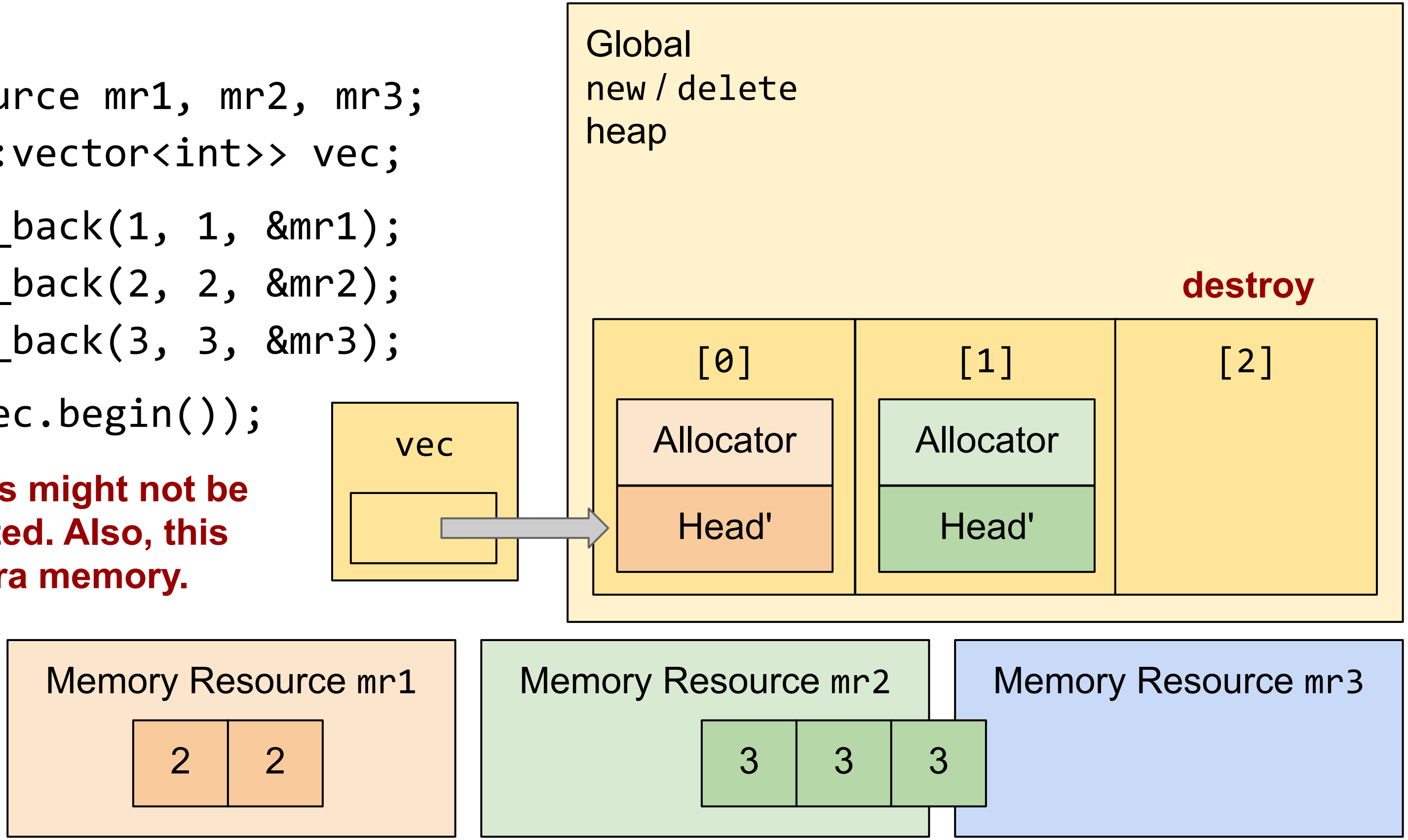
```
simple_resource mr1, mr2, mr3;
vector<pmr::vector<int>> vec;
vec.emplace_back(1, 1, &mr1);
vec.emplace_back(2, 2, &mr2);
vec.emplace_back(3, 3, &mr3);
vec.erase(vec.begin());
```



If we implement erase via move-assign

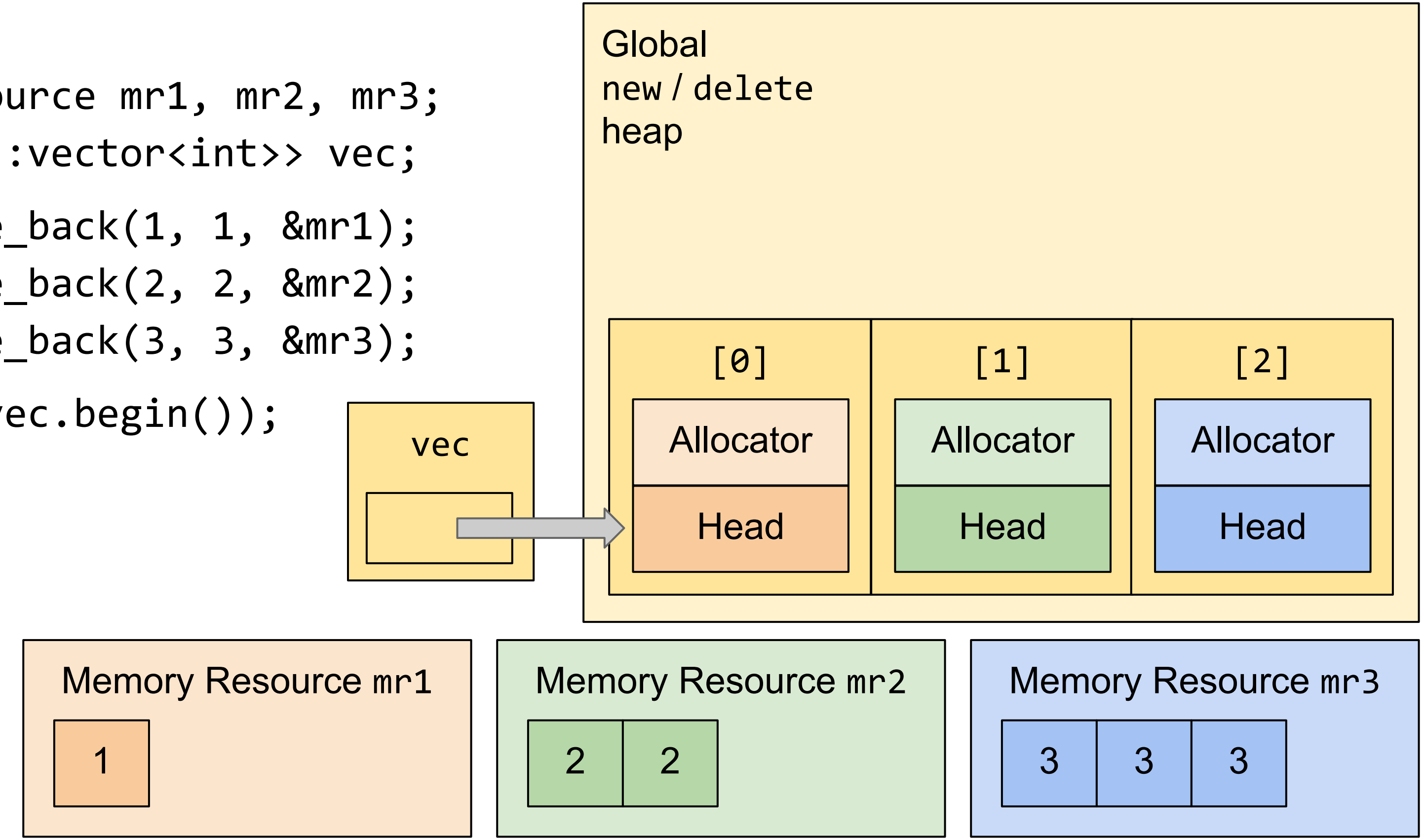
```
simple_resource mr1, mr2, mr3;
vector<pmr::vector<int>> vec;

vec.emplace_back(1, 1, &mr1);
vec.emplace_back(2, 2, &mr2);
vec.emplace_back(3, 3, &mr3);

vec.erase(vec.begin());
```

Success, but this might not be what you expected. Also, this uses a lot of extra memory.



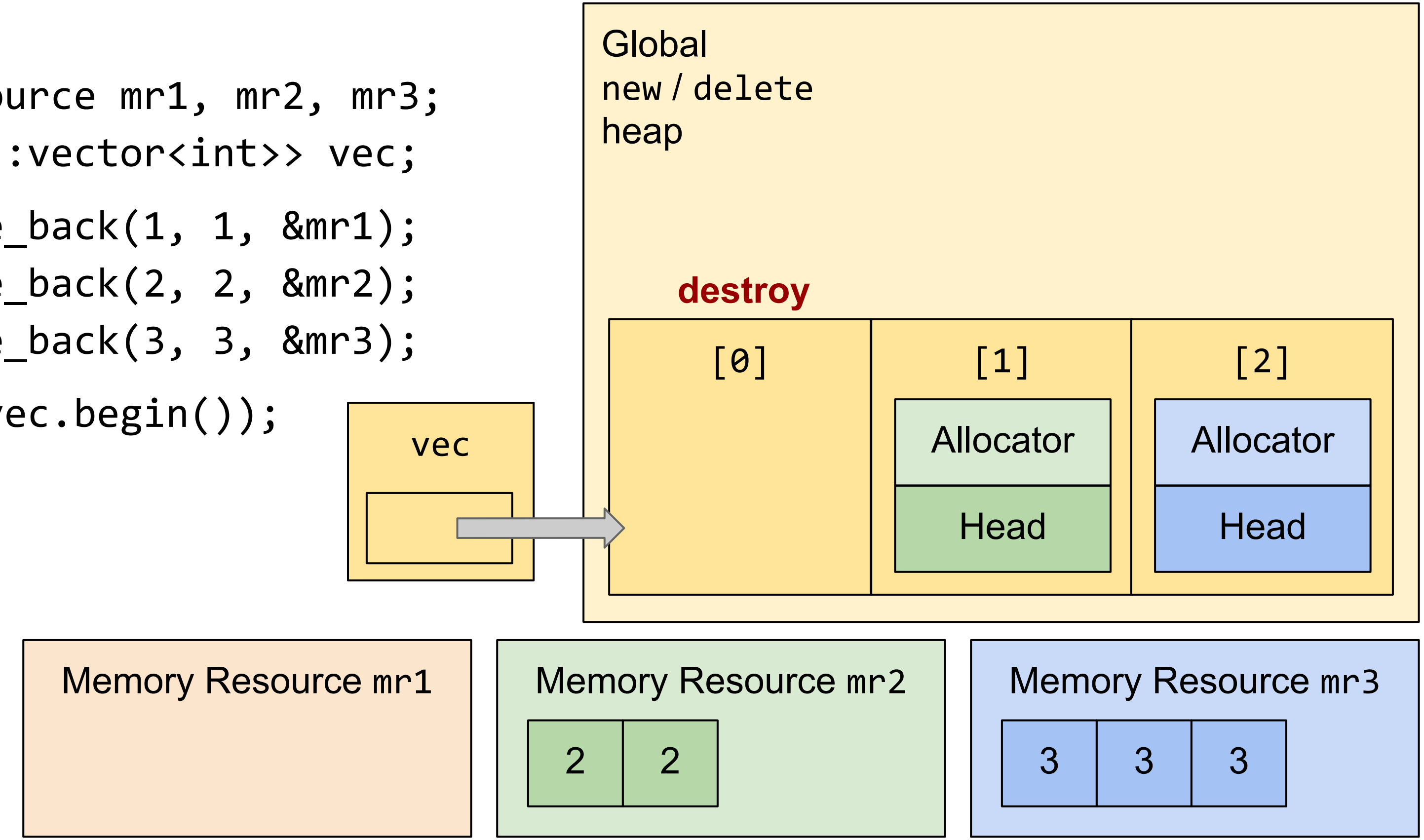
If we implement erase via move+destroy

```
simple_resource mr1, mr2, mr3;
vector<pmr::vector<int>> vec;
vec.emplace_back(1, 1, &mr1);
vec.emplace_back(2, 2, &mr2);
vec.emplace_back(3, 3, &mr3);
vec.erase(vec.begin());
```



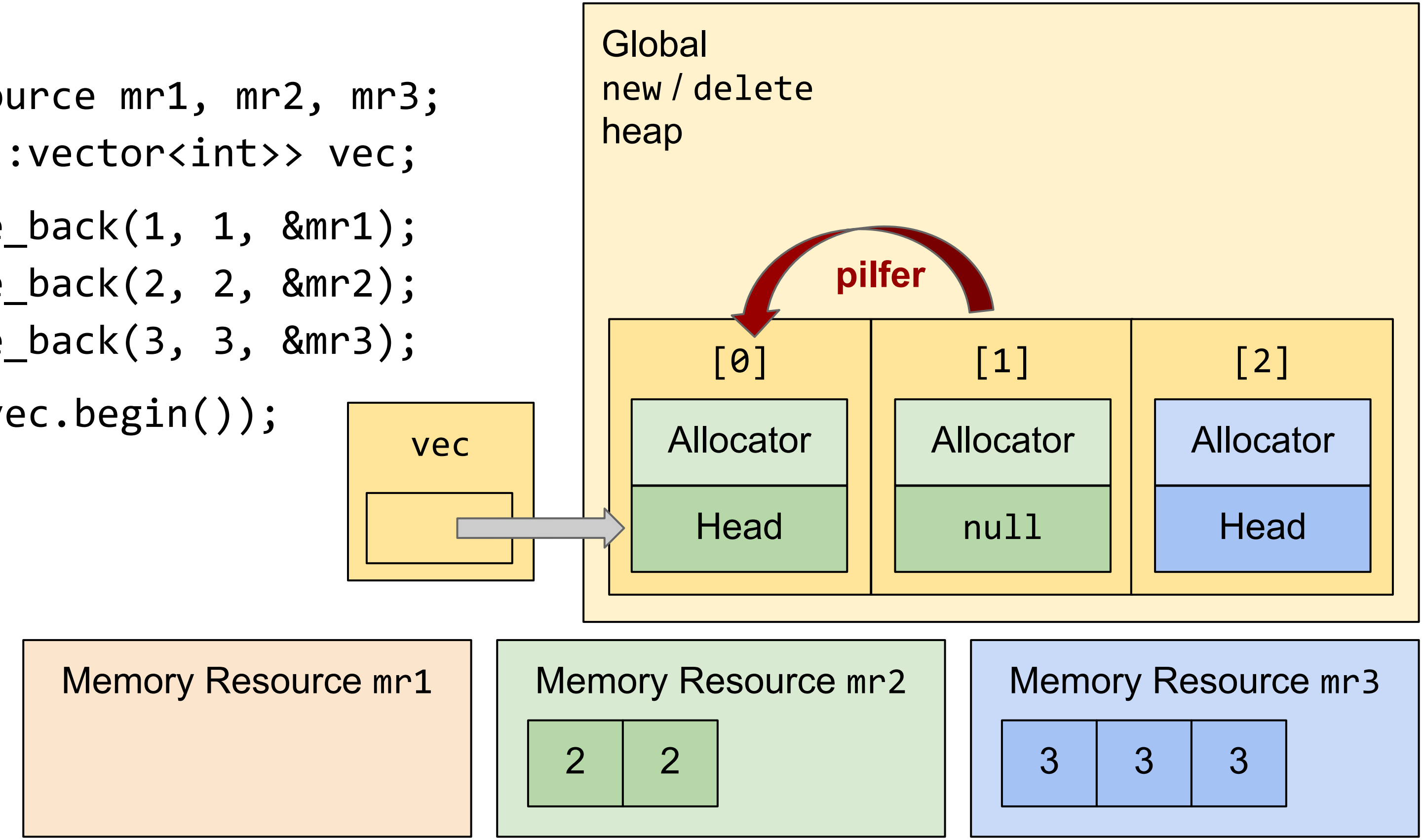
If we implement erase via move+destroy

```
simple_resource mr1, mr2, mr3;
vector<pmr::vector<int>> vec;
vec.emplace_back(1, 1, &mr1);
vec.emplace_back(2, 2, &mr2);
vec.emplace_back(3, 3, &mr3);
vec.erase(vec.begin());
```



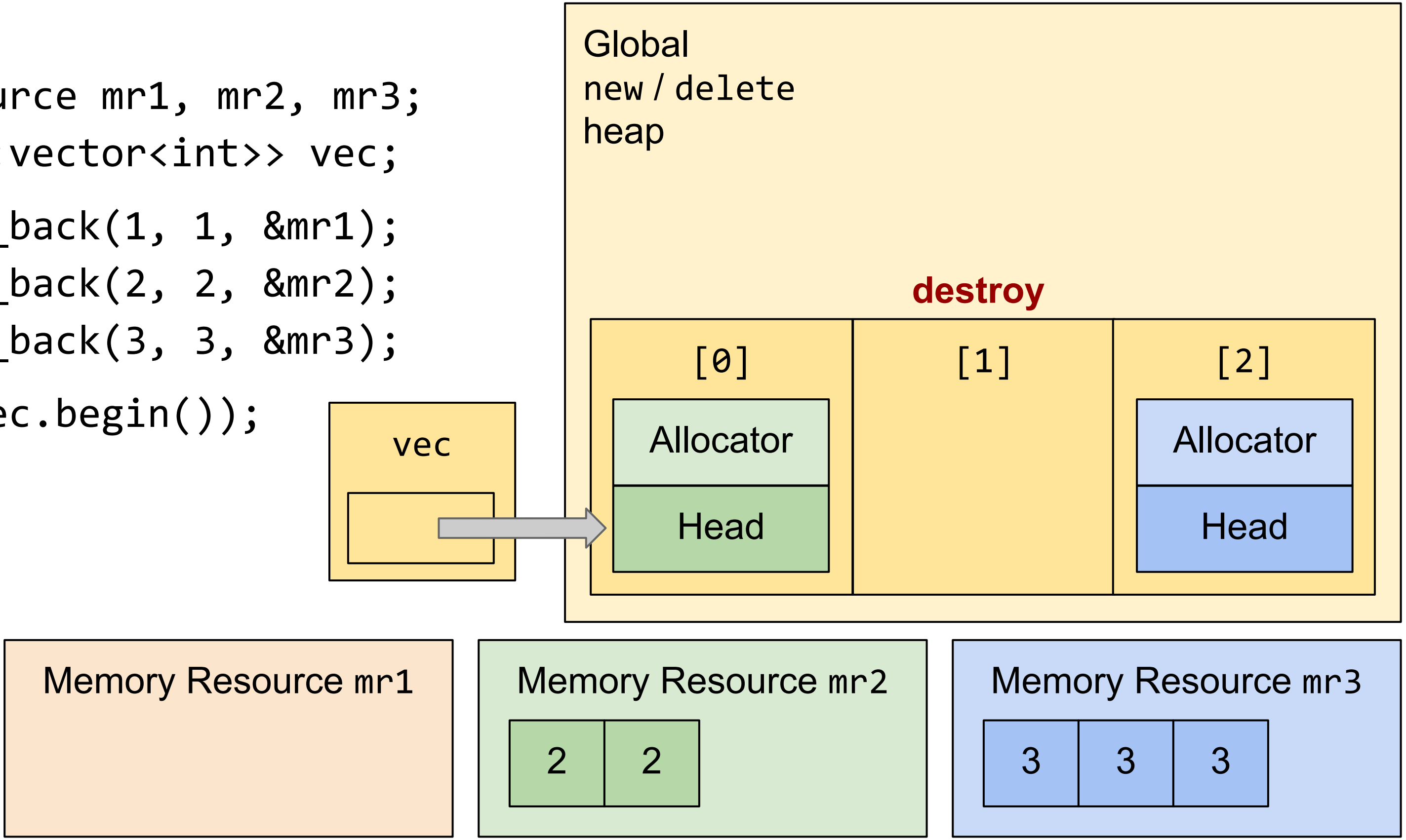
If we implement erase via move+destroy

```
simple_resource mr1, mr2, mr3;
vector<pmr::vector<int>> vec;

vec.emplace_back(1, 1, &mr1);
vec.emplace_back(2, 2, &mr2);
vec.emplace_back(3, 3, &mr3);

vec.erase(vec.begin());
```



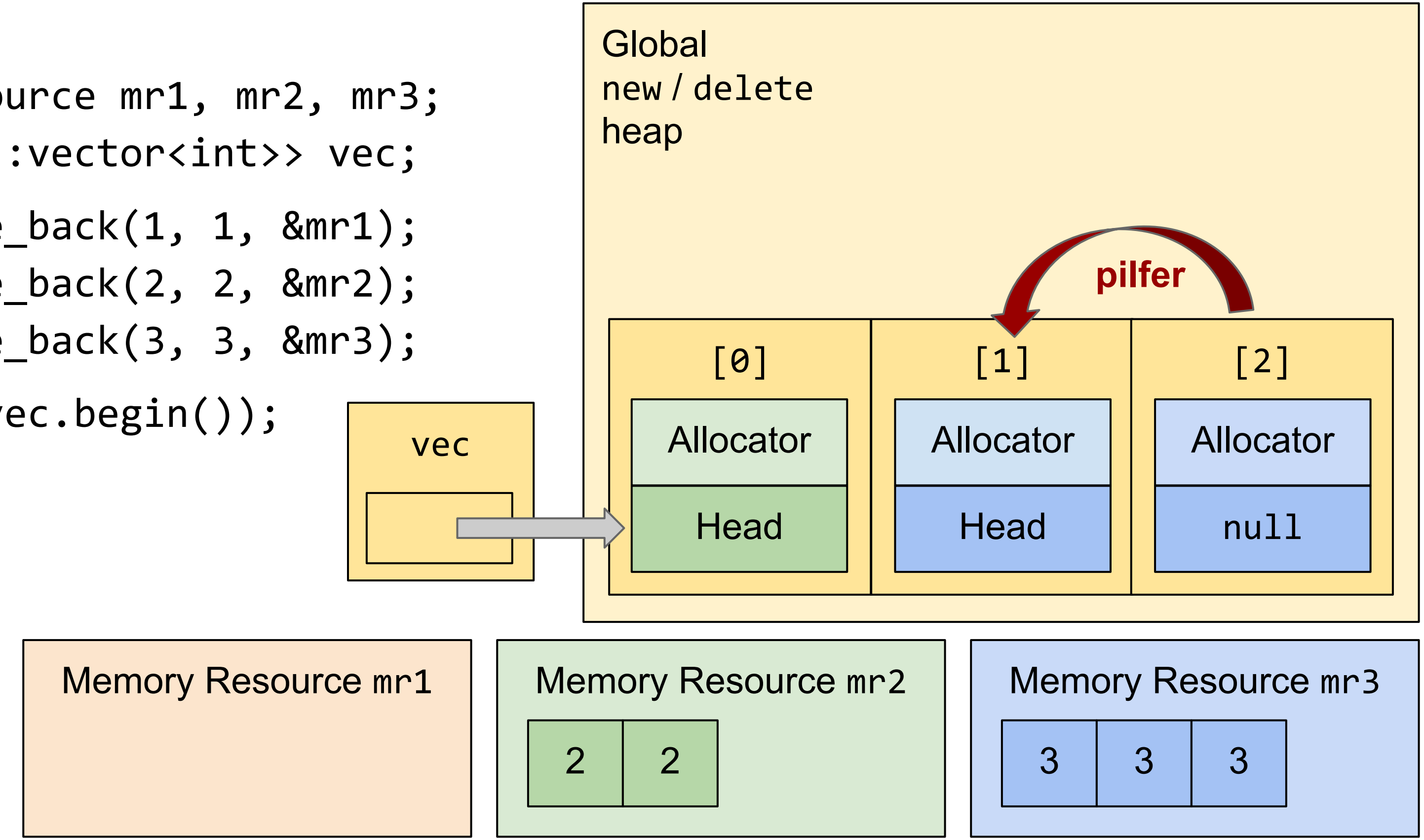
If we implement erase via move+destroy

```
simple_resource mr1, mr2, mr3;
vector<pmr::vector<int>> vec;

vec.emplace_back(1, 1, &mr1);
vec.emplace_back(2, 2, &mr2);
vec.emplace_back(3, 3, &mr3);

vec.erase(vec.begin());
```



If we implement erase via move+destroy

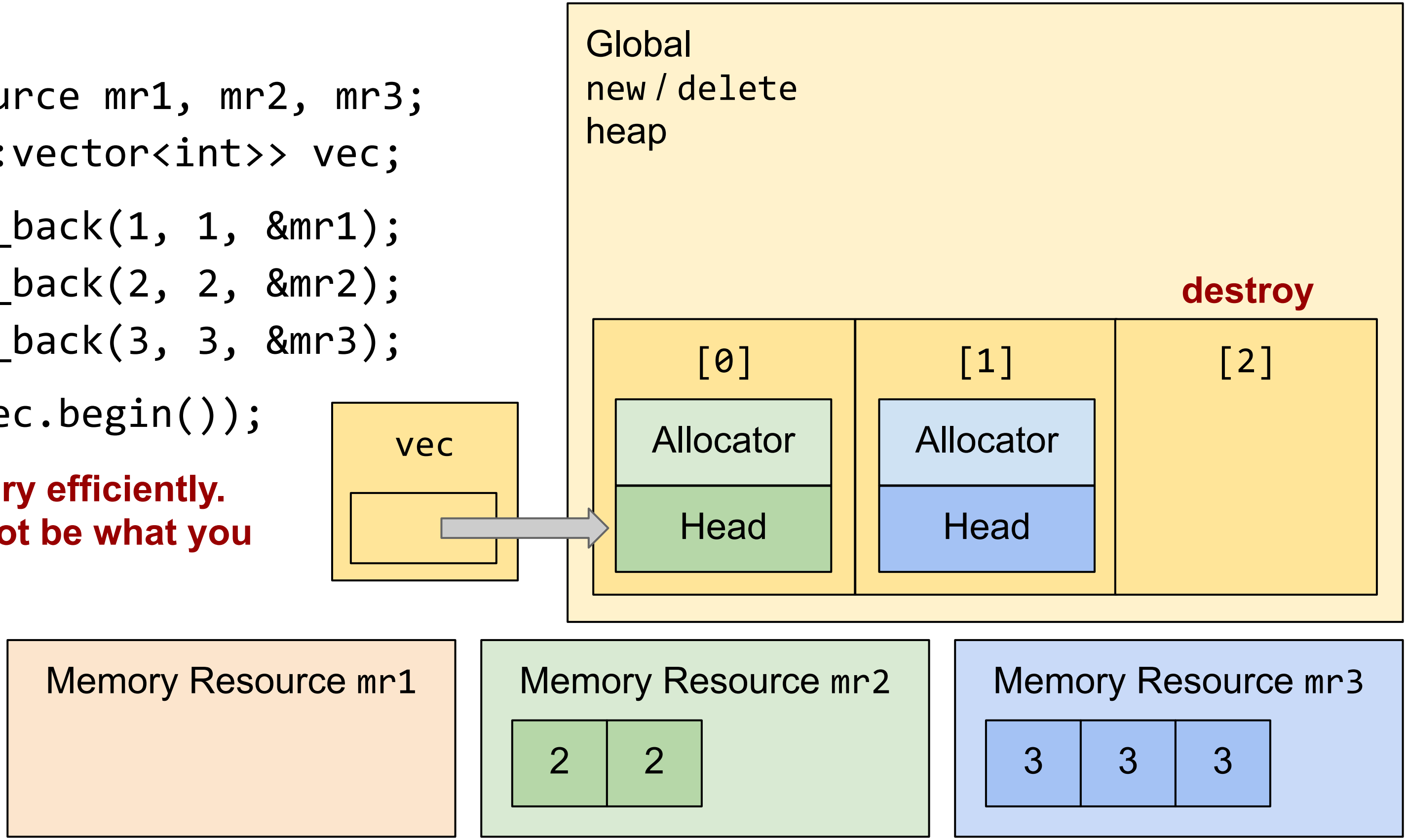
```
simple_resource mr1, mr2, mr3;
vector<pmr::vector<int>> vec;
vec.emplace_back(1, 1, &mr1);
vec.emplace_back(2, 2, &mr2);
vec.emplace_back(3, 3, &mr3);
vec.erase(vec.begin());
```



If we implement erase via move+destroy

```
simple_resource mr1, mr2, mr3;
vector<pmr::vector<int>> vec;

vec.emplace_back(1, 1, &mr1);
vec.emplace_back(2, 2, &mr2);
vec.emplace_back(3, 3, &mr3);

vec.erase(vec.begin());
```

Success, and very efficiently. But this might not be what you expected.



Compare the outcomes

```
simple_resource mr1, mr2, mr3;
vector<pmr::vector<int>> vec;

vec.emplace_back(1, 1, &mr1);
vec.emplace_back(2, 2, &mr2);
vec.emplace_back(3, 3, &mr3);

vec.erase(vec.begin());
```