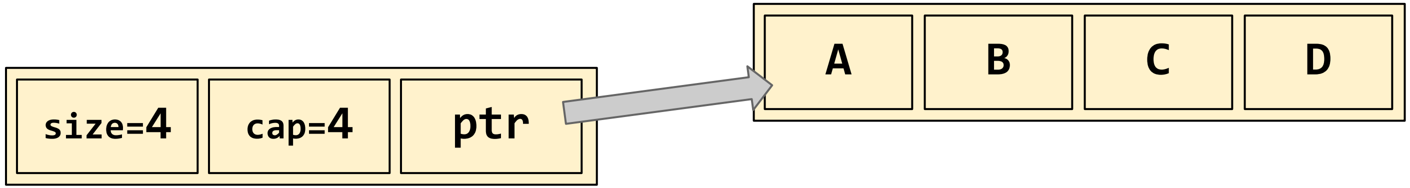
Trivially Relocatable

Arthur O’Dwyer
2018-09-25

Motivating “relocation”

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

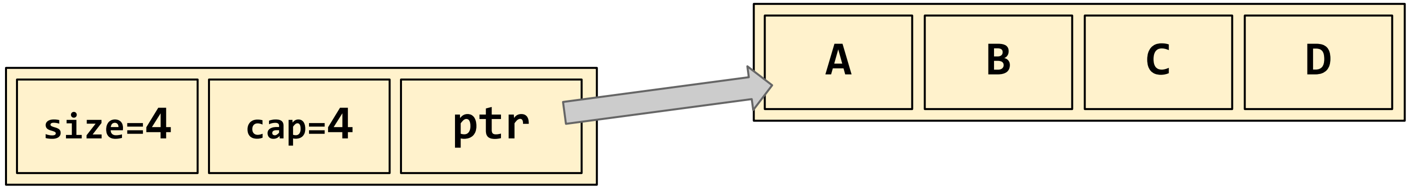
```
std::vector<T> vec { A, B, C, D };
```



Motivating “relocation”

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

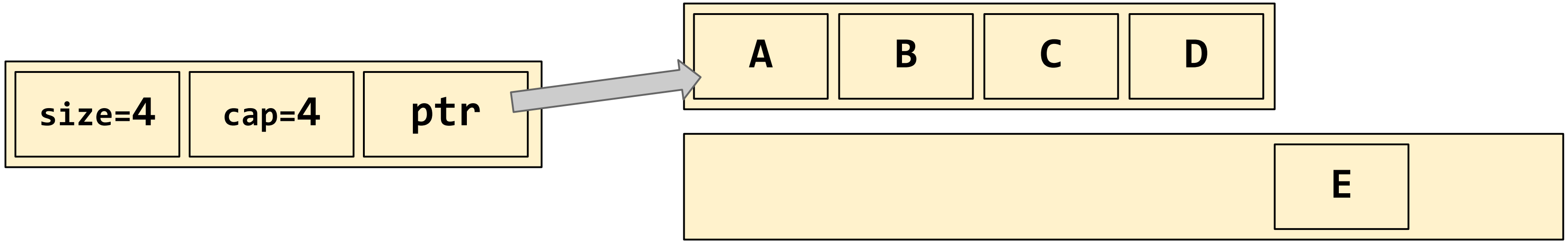
```
std::vector<T> vec { A, B, C, D };
vec.push_back(E);
```



Motivating “relocation”

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

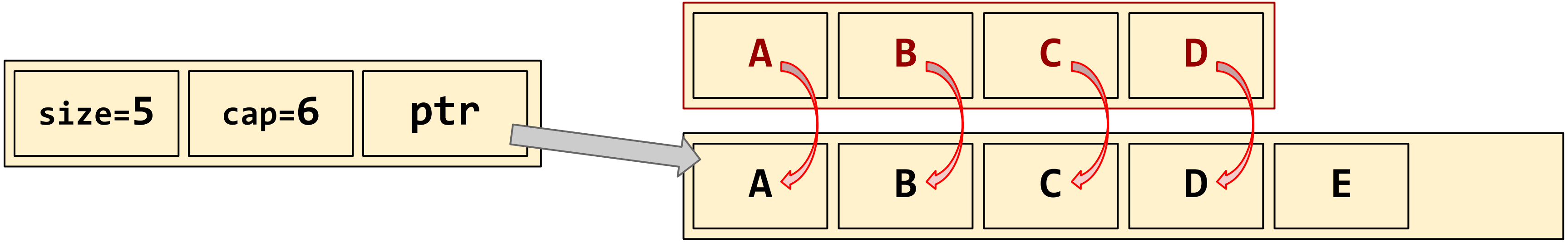
```
std::vector<T> vec { A, B, C, D };
vec.push_back(E);
```



Motivating “relocation”

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

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

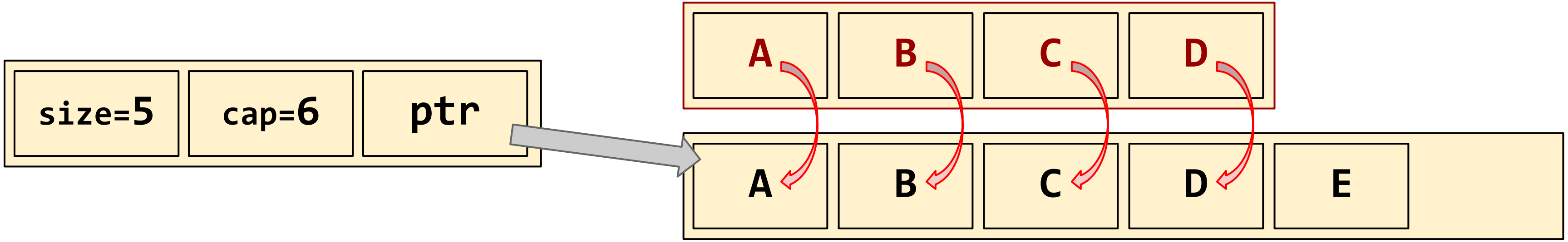


How is the “relocation” of objects A, B, C, D accomplished?

Motivating “relocation”

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

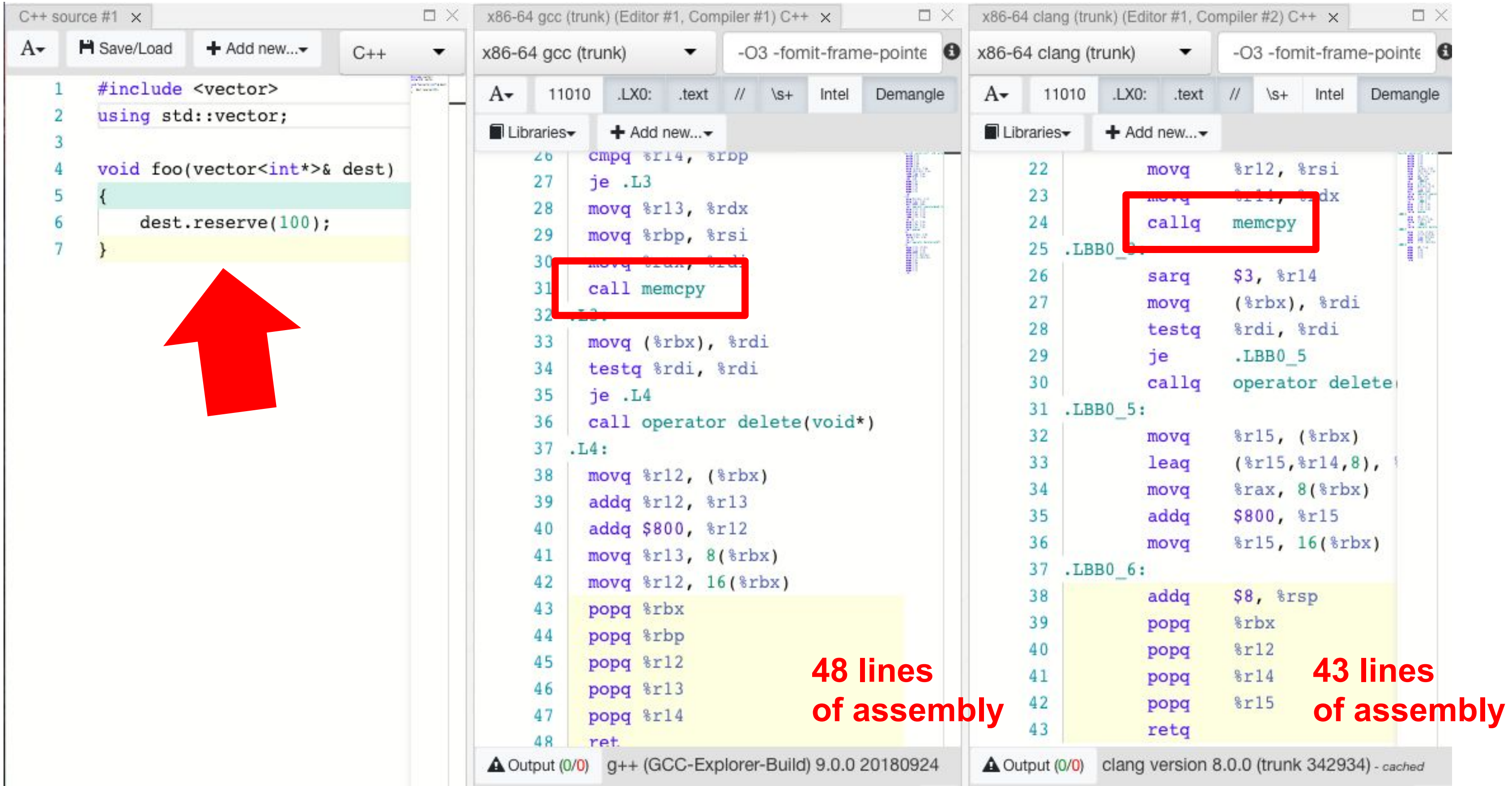
```
std::vector<T> vec { A, B, C, D };
vec.push_back(E);
```



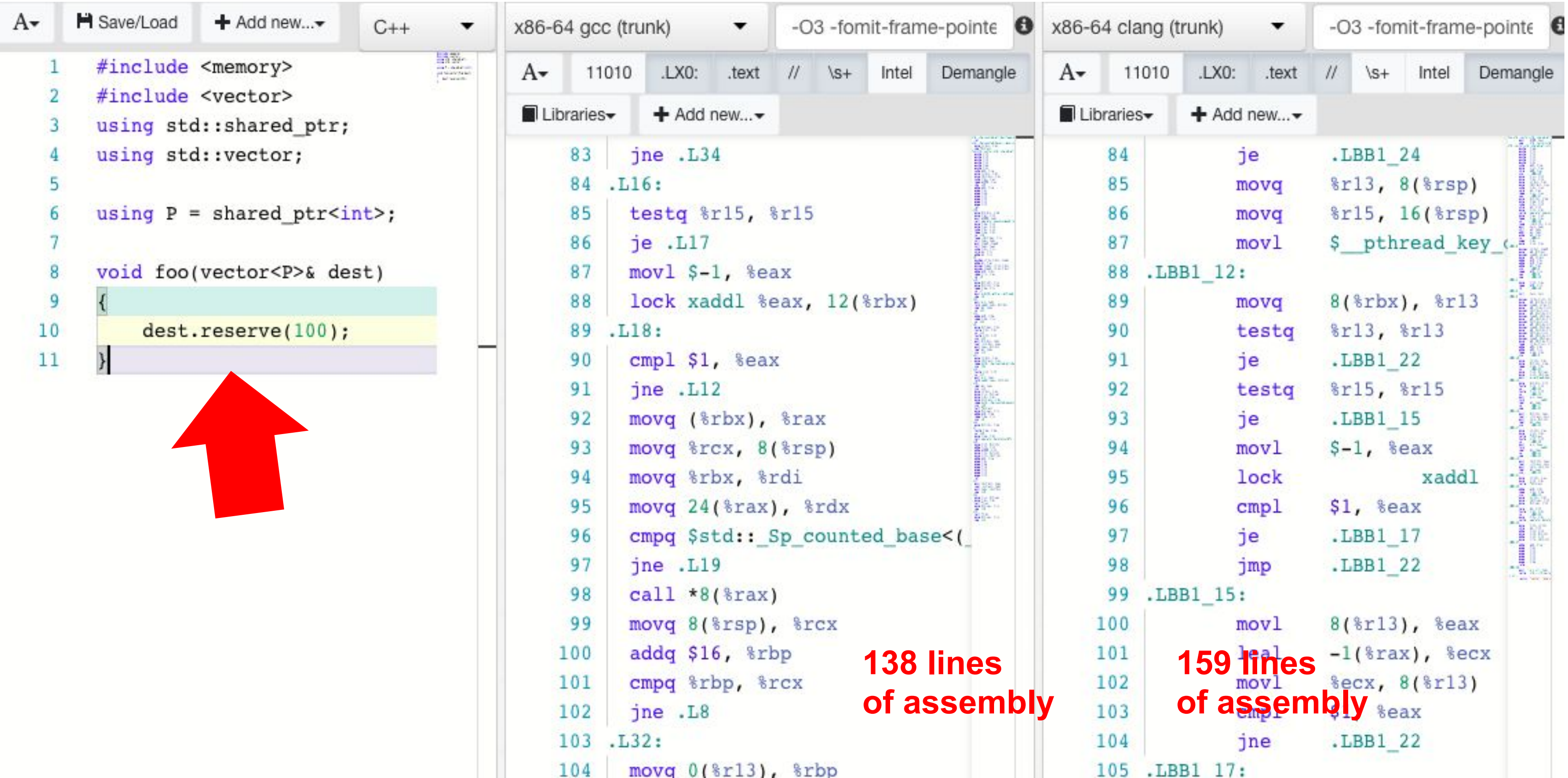
The “relocation” of objects A, B, C, D involves 4 calls to the move-constructor, followed by 4 calls to the destructor.

https://godbolt.org/z/wxVF4C

Relocating trivially copyable types

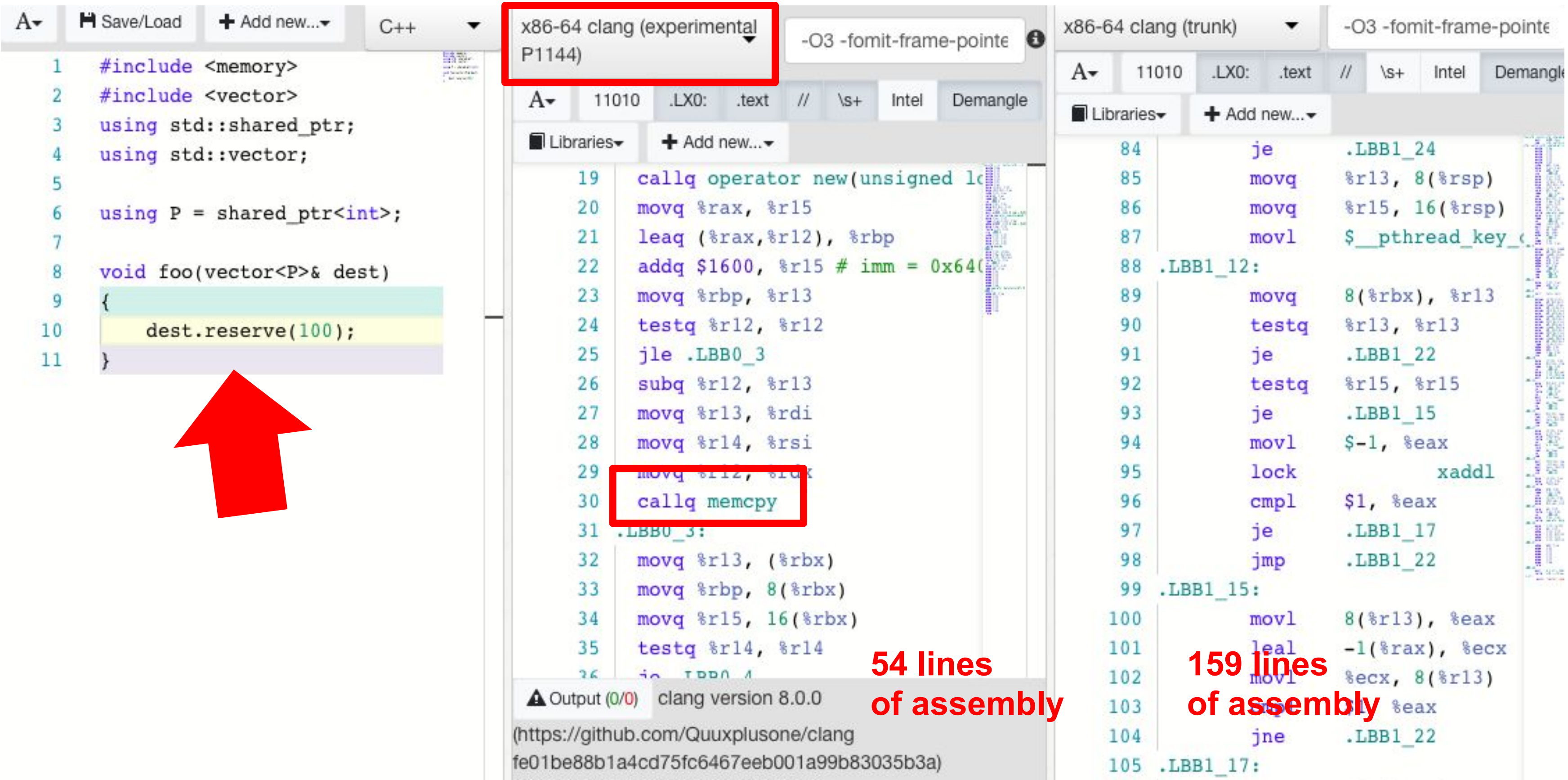


Relocating non-trivial types



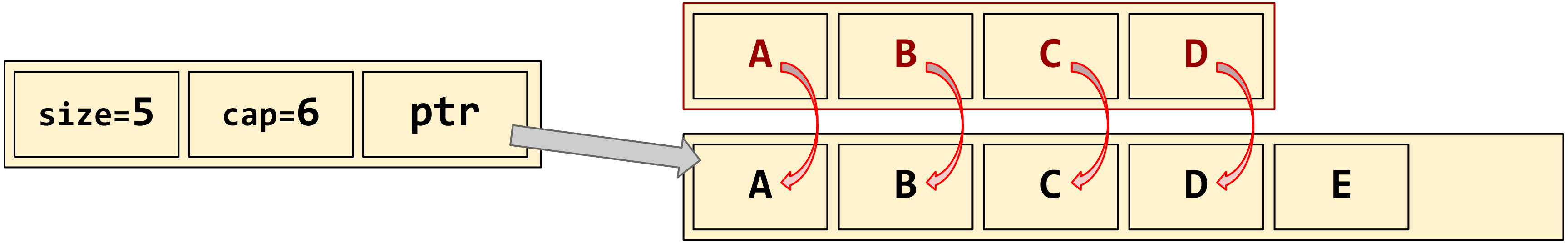
https://p1144.godbolt.org/z/-TBVP6

Then a miracle occurs...



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***.

Benchmark results

```
struct R : std::unique_ptr<int> {};

template<class VectorT>
void test_reserve(benchmark::State& state) {
    int M = state.range(0);
    VectorT v;
    for (auto _ : state) {
        state.PauseTiming(); v = VectorT(M);
        state.ResumeTiming(); v.reserve(M+1);
        benchmark::DoNotOptimize(v);
    }
}

BENCHMARK(test_reserve<std::vector<R>>)->Arg(10'000);
```

`std::vector<R>`	26µs ±260ns
`std'::vector<R>`	9µs ±60ns

The first row is vanilla libc++. The second row is libc++ with my patch applied.

Opting in to trivial relocatability

Very rarely in normal code, you might need to use the attribute.

```
struct [[trivially_relocatable]] Widget {
    Widget(Widget&&);
    ~Widget();
};
static_assert(std::is_trivially_relocatable_v<Widget>);
```

More often, it'll happen automatically, because you followed the Rule of Zero.

```
struct Gadget {
    std::string name;
    std::unique_ptr<int> p;
};
static_assert(std::is_trivially_relocatable_v<Gadget>);
```

More information

- P1144 “Object relocation in terms of move plus destroy”
  - will be in the San Diego mailing
- Blog post announcing the Godbolt compiler
  - `https://quuxplusone.github.io/blog/2018/07/18/announcing-trivially-relocatable/`
- Cpp.chat episode #40
  - `https://youtu.be/8u5Qi4FgTP8`

Other applications for relocatability

A reliable way of detecting "trivial relocatability" permits optimizing any routine that performs the moral equivalent of `realloc`, such as:

- `std::vector<R>::reserve`
- `std::vector<R>::resize`
- `std::vector<R>::emplace_back`
- `std::vector<R>::push_back`
- `std::vector<R>::insert`

Other applications for relocatability

A reliable way of detecting "trivial relocatability" permits optimizing `swap` and any algorithm that depends on `swap`:

- `std::swap`
- `std::sort`
- `std::partition`

Other applications for relocatability

A reliable way of detecting "trivial relocatability" permits deduplicating the codepaths that perform “move” of type-erased wrappers:

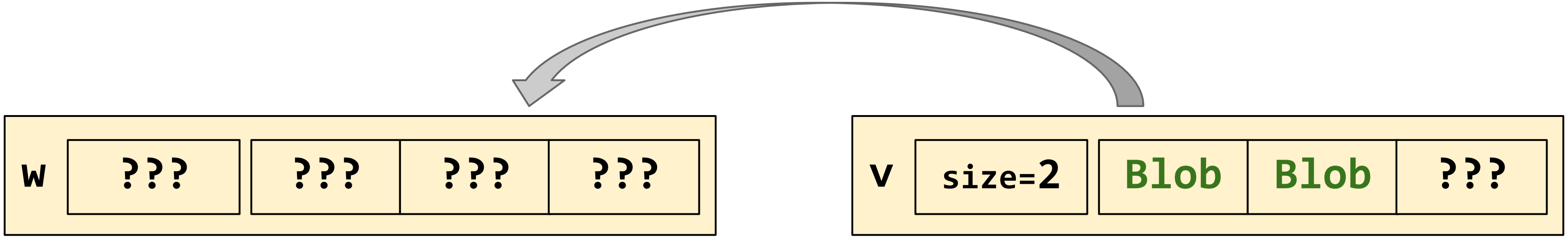
- `std::function`
- `std::any`
- `sg14::inplace_function`

Moving-out-of one of these wrappers leaves the source *wrapper* in the “disengaged” state, which means that the source *wrappee* has been moved-and-then-destroyed, i.e., relocated. We can use memcpy for this.

Other applications for relocatability

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

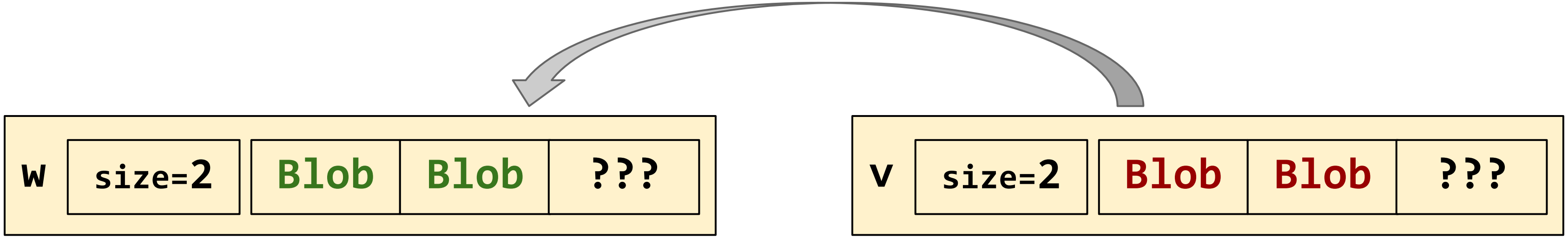
```
fixed_capacity_vector<Blob, 3> v = { ... };
auto w = std::move(v);
```



Other applications 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);
```

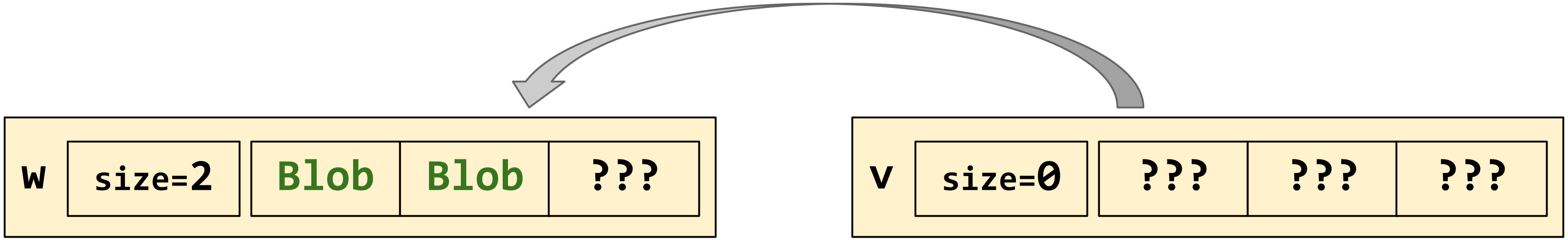


MOVING IS
INEFFICIENT AND BAD

Other applications 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);
```



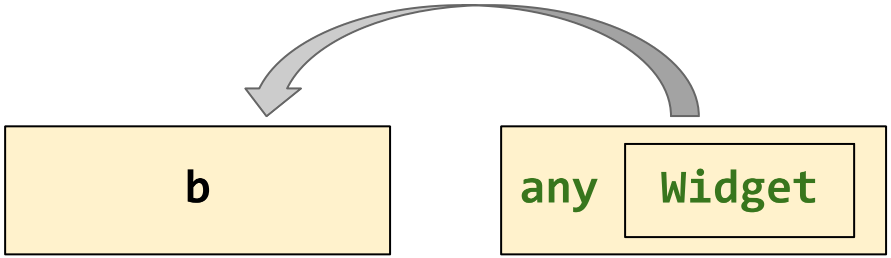
RELOCATING IS EFFICIENT AND GOOD

Other applications for relocatability

A reliable way of detecting "trivial relocatability" permits deduplicating the codepaths that perform “move” of type-erased wrappers:

- `std::function`
- `std::any`
- `sg14::inplace_function`

```
std::any a = Widget{};
auto b = std::move(a);
```

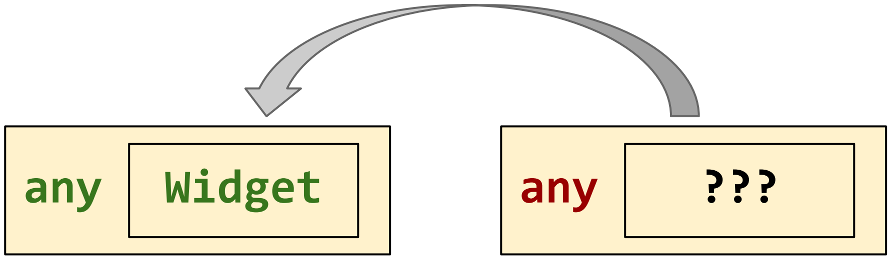


Other applications for relocatability

A reliable way of detecting "trivial relocatability" permits deduplicating the codepaths that perform “move” of type-erased wrappers:

- `std::function`
- `std::any`
- `sg14::inplace_function`

```
std::any a = Widget{};
auto b = std::move(a);
```



Other applications for relocatability

Also, if we avoid SBO for wrappees that are not trivially relocatable, then the wrapper itself becomes trivially relocatable!

- `std::function`
- `std::any`