

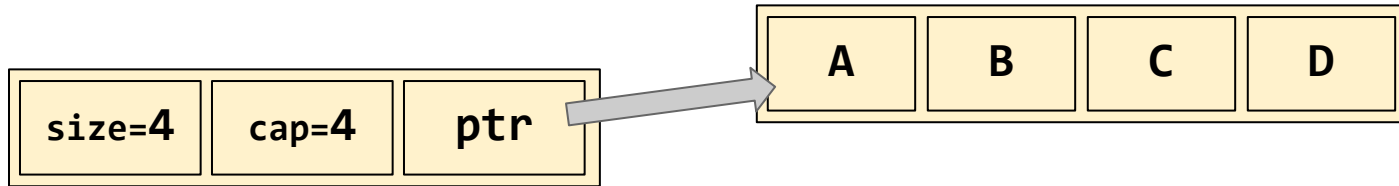
Trivially Relocatable

Arthur O'Dwyer
2018-09-25

Motivating “relocation”

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

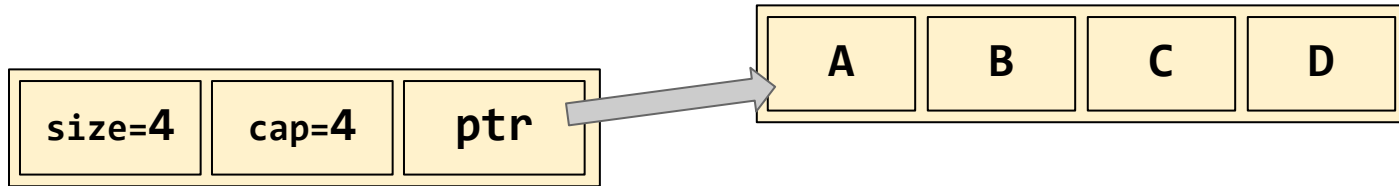
```
std::vector<T> vec { A, B, C, D };
```



Motivating “relocation”

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

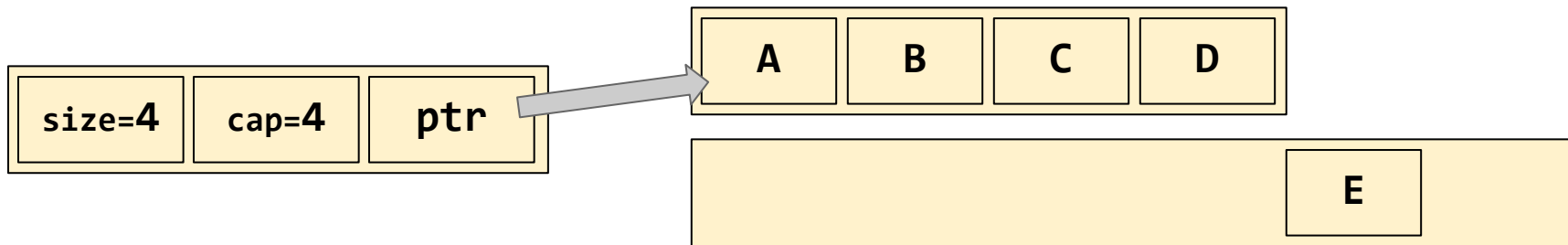
```
std::vector<T> vec { A, B, C, D };  
vec.push_back(E);
```



Motivating “relocation”

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

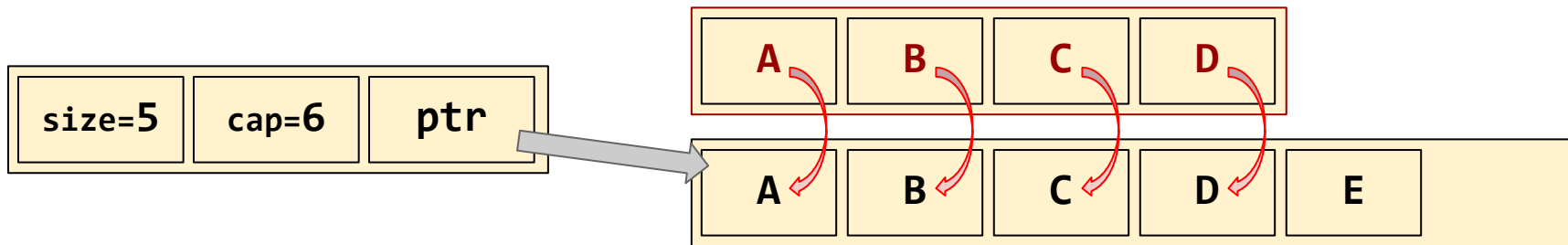
```
std::vector<T> vec { A, B, C, D };  
vec.push_back(E);
```



Motivating “relocation”

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

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

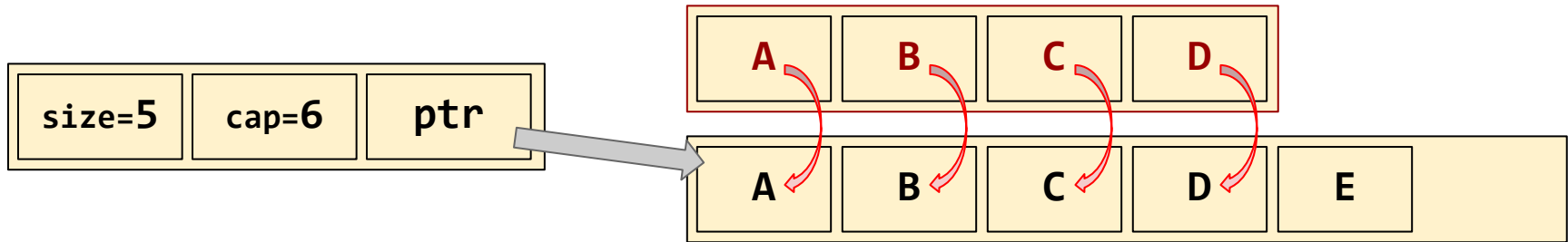


How is the “relocation” of objects A, B, C, D accomplished?

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.

Relocating trivially copyable types

C++ source #1 x

Save/Load Add new... C++

```
1 #include <vector>
2 using std::vector;
3
4 void foo(vector<int*>& dest)
5 {
6     dest.reserve(100);
7 }
```



x86-64 gcc (trunk) (Editor #1, Compiler #1) C++ x

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

Libraries Add new...

```
26 cmpq %r14, %rbp
27 je .L3
28 movq %r13, %rdx
29 movq %rbp, %rsi
30 movq %rax, %rdi
31 call memcpy
32 .L3:
33 movq (%rbx), %rdi
34 testq %rdi, %rdi
35 je .L4
36 call operator delete(void*)
37 .L4:
38 movq %r12, (%rbx)
39 addq %r12, %r13
40 addq $800, %r12
41 movq %r13, 8(%rbx)
42 movq %r12, 16(%rbx)
43 popq %rbx
44 popq %rbp
45 popq %r12
46 popq %r13
47 popq %r14
48 ret
```

48 lines
of assembly

x86-64 clang (trunk) (Editor #1, Compiler #2) C++ x

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

Libraries Add new...

```
22 movq %r12, %rsi
23 movq %r11, %rdx
24 callq memcpy
25 .LBB0_5:
26 sarg $3, %r14
27 movq (%rbx), %rdi
28 testq %rdi, %rdi
29 je .LBB0_5
30 callq operator delete
31 .LBB0_5:
32 movq %r15, (%rbx)
33 leaq (%r15,%r14,8), %r12
34 movq %rax, 8(%rbx)
35 addq $800, %r15
36 movq %r15, 16(%rbx)
37 .LBB0_6:
38 addq $8, %rsp
39 popq %rbx
40 popq %r12
41 popq %r14
42 popq %r15
43 retq
```

43 lines
of assembly


Output (0/0) g++ (GCC-Explorer-Build) 9.0.0 20180924

Output (0/0) clang version 8.0.0 (trunk 342934) - cached

Relocating non-trivial types

A- Save/Load + Add new... C++

```
1 #include <memory>
2 #include <vector>
3 using std::shared_ptr;
4 using std::vector;
5
6 using P = shared_ptr<int>;
7
8 void foo(vector<P>& dest)
9 {
10     dest.reserve(100);
11 }
```



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

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

Libraries- + Add new...

```
83 jne .L34
84 .L16:
85 testq %r15, %r15
86 je .L17
87 movl $-1, %eax
88 lock xaddl %eax, 12(%rbx)
89 .L18:
90 cmpl $1, %eax
91 jne .L12
92 movq (%rbx), %rax
93 movq %rcx, 8(%rsp)
94 movq %rbx, %rdi
95 movq 24(%rax), %rdx
96 cmpq $std::_Sp_counted_base<
97 jne .L19
98 call *8(%rax)
99 movq 8(%rsp), %rcx
100 addq $16, %rbp
101 cmpq %rbp, %rcx
102 jne .L8
103 .L32:
104 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...


```
84 je .LBB1_24
85 movq %r13, 8(%rsp)
86 movq %r15, 16(%rsp)
87 movl $_pthread_key_
88 .LBB1_12:
89 movq 8(%rbx), %r13
90 testq %r13, %r13
91 je .LBB1_22
92 testq %r15, %r15
93 je .LBB1_15
94 movl $-1, %eax
95 lock xaddl
96 cmpl $1, %eax
97 je .LBB1_17
98 jmp .LBB1_22
99 .LBB1_15:
100 movl 8(%r13), %eax
101 leal -1(%rax), %ecx
102 movl %ecx, 8(%r13)
103 cmpl $1, %eax
104 jne .LBB1_22
105 .LBB1_17:
```

159 lines
of assembly

Then a miracle occurs...

A- Save/Load + Add new... C++ x86-64 clang (experimental P1144) -O3 -fomit-frame-pointe

```
1 #include <memory>
2 #include <vector>
3 using std::shared_ptr;
4 using std::vector;
5
6 using P = shared_ptr<int>;
7
8 void foo(vector<P>& dest)
9 {
10     dest.reserve(100);
11 }
```



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

Libraries + Add new...

```
19 callq operator new(unsigned l
20 movq %rax, %r15
21 leaq (%rax,%r12), %rbp
22 addq $1600, %r15 # imm = 0x64
23 movq %rbp, %r13
24 testq %r12, %r12
25 jle .LBB0_3
26 subq %r12, %r13
27 movq %r13, %rdi
28 movq %r14, %rsi
29 movq %r12, %rcx
30 callq memcpy
31 .LBB0_3:
32 movq %r13, (%rbx)
33 movq %rbp, 8(%rbx)
34 movq %r15, 16(%rbx)
35 testq %r14, %r14
36 je .LBB0_4
```

Output (0/0) clang version 8.0.0
(<https://github.com/Quuxplusone/clang>
fe01be88b1a4cd75fc6467eeb001a99b83035b3a)

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

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

Libraries + Add new...

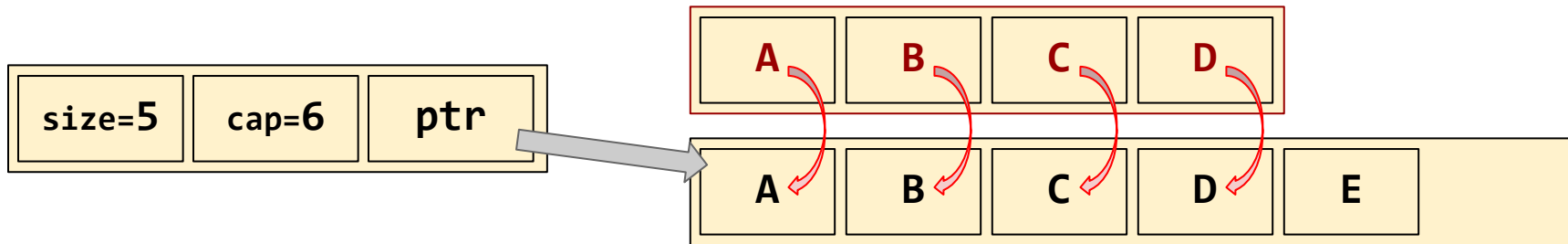
```
84 je .LBB1_24
85 movq %r13, 8(%rsp)
86 movq %r15, 16(%rsp)
87 movl $__pthread_key_c
88 .LBB1_12:
89 movq 8(%rbx), %r13
90 testq %r13, %r13
91 je .LBB1_22
92 testq %r15, %r15
93 je .LBB1_15
94 movl $-1, %eax
95 lock xaddl
96 cmpl $1, %eax
97 je .LBB1_17
98 jmp .LBB1_22
99 .LBB1_15:
100 movl 8(%r13), %eax
101 leal -1(%rax), %ecx
102 movl %ecx, 8(%r13)
103 movl %eax, %ecx
104 jne .LBB1_22
105 .LBB1_17:
```

54 lines of assembly

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**.

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);
```

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



The first row is vanilla libcpp.
The second row is libcpp 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:

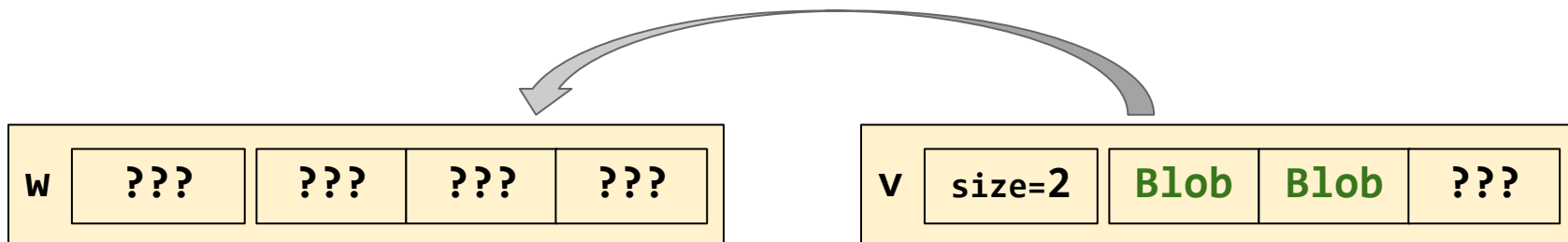
- `std::function`
- `std::any`
- `sg14::inplace_function`

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`:

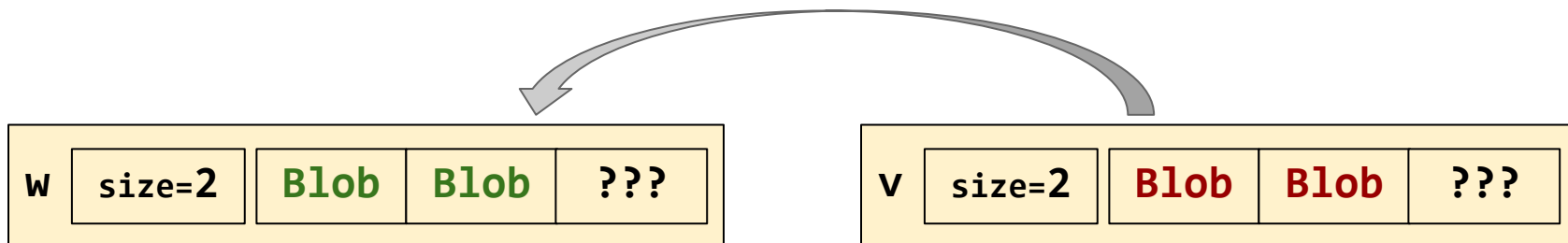
```
fixed_capacity_vector<Blob, 3> v = { ... };  
auto w = std::move(v);
```



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);
```

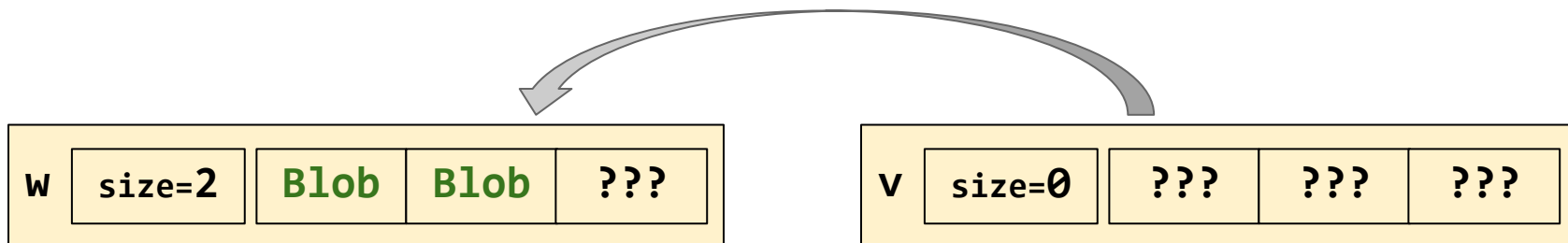


**MOVING IS
INEFFICIENT AND BAD**

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);
```



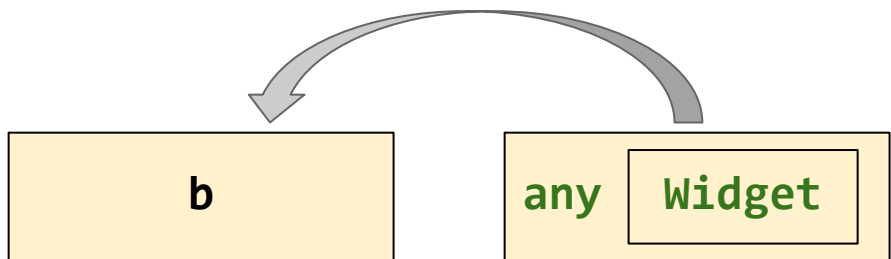
**RELOCATING IS
EFFICIENT AND GOOD**

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);
```

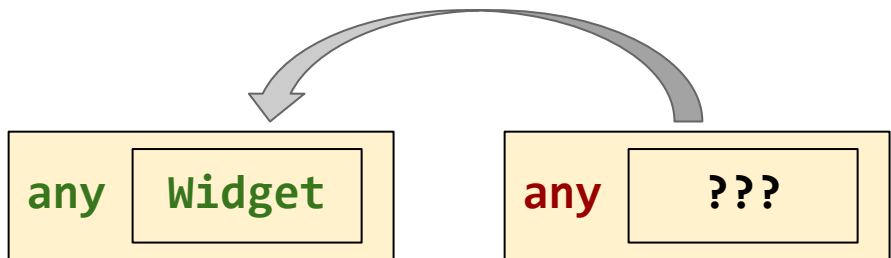


Other applications for relocatability

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`

