The Best Type Traits

...that C++ doesn’t have (yet)

Arthur O’Dwyer
2018-06-06

Outline

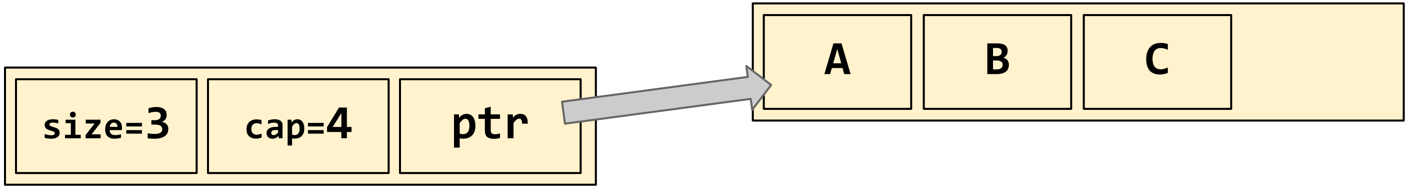
- is_trivially_relocatable<T> [3–32]
  - Motivation / Implementation / Benchmarks / Downsides
- is_trivially_comparable<T> [34–46]
  - Motivation / Implementation / Benchmarks / Downsides
  - “memcmp comparable” versus “memberwise comparable” [42–46]
- tombstone_traits<T> [48–77]
  - Motivation / Implementation
  - Benchmark design and results [67–73] / Downsides

Hey look!
Slide numbers!

Motivating “relocation”

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

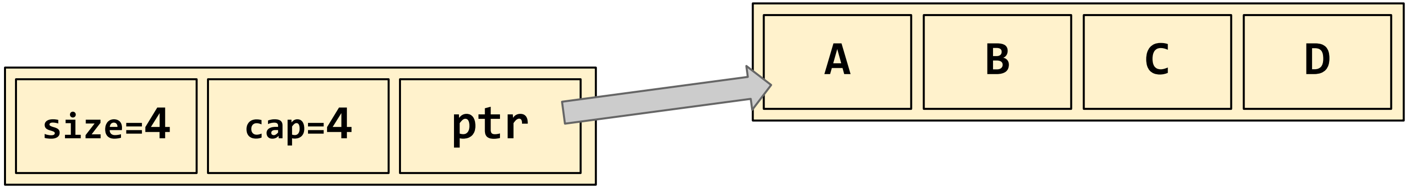
```
std::vector<T> vec { A, B, C };
```



Motivating “relocation”

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

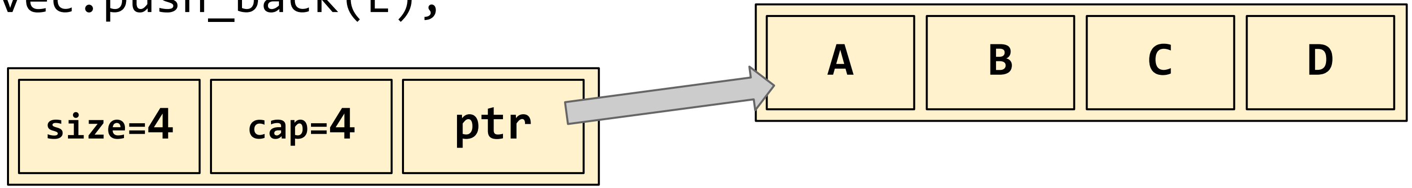
```
std::vector<T> vec { A, B, C };
vec.push_back(D);
```



Motivating “relocation”

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

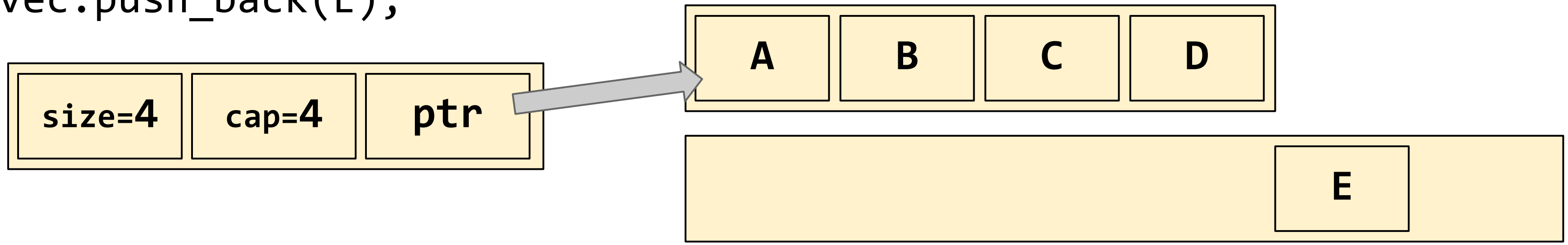
```
std::vector<T> vec { A, B, C };
vec.push_back(D);
vec.push_back(E);
```



Motivating “relocation”

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

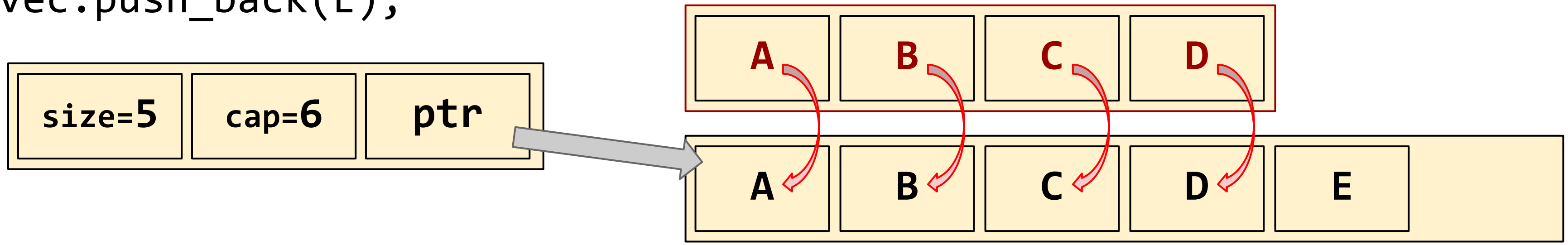
```
std::vector<T> vec { A, B, C };
vec.push_back(D);
vec.push_back(E);
```



Motivating “relocation”

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

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



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

Rules for `std::vector` reallocation

[vector.modifiers] /1:

> If an exception is thrown other than by the copy constructor, move constructor, assignment operator, or move assignment operator of `T` or by any `InputIterator` operation, there are no effects.
>
> If an exception is thrown while inserting a single element at the end and `T` is `CopyInsertable` or `is_nothrow_move_constructible_v<T>` is `true`, there are no effects.
>
> Otherwise, if an exception is thrown by the move constructor of a non-`CopyInsertable` `T`, the effects are unspecified.

Rules for `std::vector` reallocation

[vector.modifiers] /1:

> Only `T`'s copy constructor, move constructor, copy assignment operator, or move assignment operator of `T` are allowed to throw exceptions that screw up the vector's contents. In all other cases we must give the strong guarantee. (This includes if the constructor of `T` selected by `emplace` throws!)
>
> In fact, if we're inserting at the end, and `T` is either noexcept move-constructible, ***or*** copy-constructible, we must ***still*** give the strong guarantee.
>
> If we're inserting in the middle and/or if `T`'s copy constructor is deleted, we are permitted to give merely the basic guarantee.

Rules for `std::vector` reallocation

In other words:

- If `T` has a noexcept move constructor, we'll relocate `T` using its move constructor (and then its destructor).
- Else, if `T` has a copy constructor, we'll use its copy constructor (and then its destructor).
- Else, if `T` has a throwing move constructor, we'll use its move constructor (and then its destructor).
- Else, `T` is an immobile type such as `lock_guard`. We can refuse to compile this case.

- In the reallocating case, we never use any assignment operator at all.
- In the non-reallocating case, we use 1 move-construct on the rightmost element, then K move-assignments, and finally 1 move-construct of the new element.

Rules for `std::vector` reallocation

In other words:

- If `T` has a noexcept move constructor, we'll relocate `T` using its move constructor (and then its destructor).
- Else, if `T` has a copy constructor, we'll u… …n its destructor).
- Else, if `T` has a throwing move construc… …or (and then its destructor).
- Else, `T` is an immobile type such as `loc`… …pile this case.

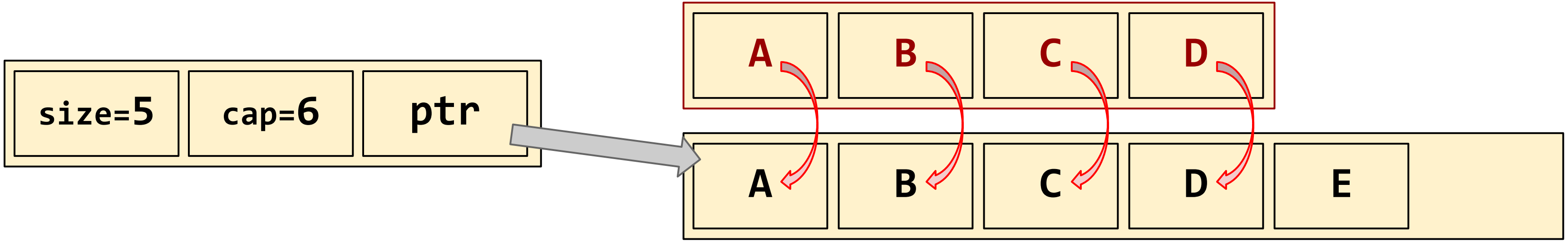
This is the only case that matters in the real world. Let's talk more about it.

- In the reallocating case, we never use any assignment operator at all.
- In the non-reallocating case, we use 1 move-construct on the rightmost element, then K move-assignments, and finally 1 move-construct of the new element.

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

Implementations are actually smart enough to optimize the following code to use `memmove`:

https://godbolt.org/g/RoAqgZ

```
void reallocate(std::vector<int*>& vec) {
    vec.reserve(100);
}
```

The red arrows indicate “clever” specializations inside libstdc++.
vector::reserve ➡ vector::_M_allocate_and_copy
➡ __uninitialized_copy_a ➡ uninitialized_copy
➡ std::copy ➡ __builtin_memmove

Relocating non-trivial types

Implementations are ***not currently*** smart enough to optimize the following code into `memmove`:

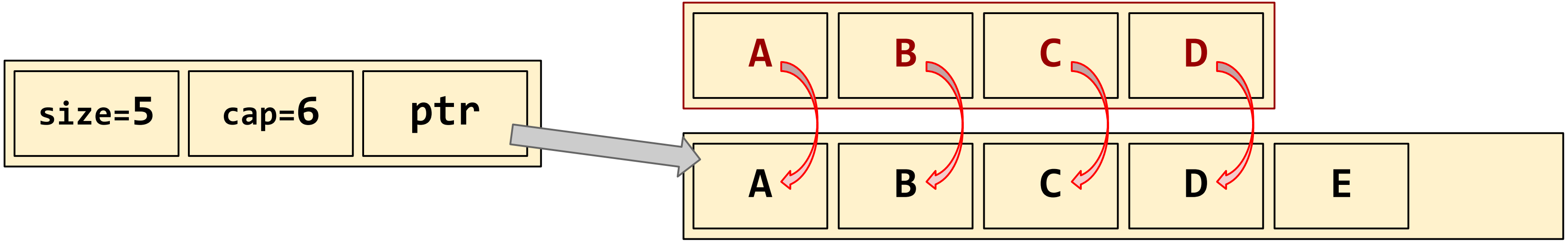
https://godbolt.org/g/RoAqgZ

```
void reallocate(std::vector<std::unique_ptr<int>>& vec) {
    vec.reserve(100);
}
```

The red arrows indicate “clever” specializations inside libstdc++.
vector::reserve ➡ vector::_M_allocate_and_copy
➡ __uninitialized_copy_a ➡ uninitialized_copy
➡ _Construct, _Destroy

Relocating non-trivial types

However, in principle, ***can't*** we implement the "relocation" of objects `A`, `B`, `C`, `D` here with a simple `memcpy`? `unique_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***.

Prior Art

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***.

- Qt calls it `Q_MOVABLE_TYPE`.
- EA’s EASTL calls it `has_trivial_relocate`.
- Bloomberg’s BSL calls it `IsBitwiseMovable`.
- Facebook’s Folly calls it `IsRelocatable`.

This list is taken from Denis Bider’s P0023 “Relocator: Efficiently Moving Types.”

Algorithm `uninitialized_relocate`

```
template<class It, class FwdIt>
FwdIt uninitialized_relocate(It first, It last, FwdIt dest)
{
    using T = typename std::iterator_traits<FwdIt>::value_type;
    std::allocator<T> alloc;
    return __uninitialized_relocate_a(first, last, dest, alloc);
}

template<class It, class FwdIt, class Alloc>
FwdIt __uninitialized_relocate_a(It first, It last, FwdIt dest, Alloc& a)
{
    using T = typename std::iterator_traits<FwdIt>::value_type;
    static_assert(is_same_v<T, typename std::allocator_traits<Alloc>::value_type>);
    while (first != last) {
        std::allocator_traits<Alloc>::construct(a, std::addressof(*dest), std::move(*first));
        std::allocator_traits<Alloc>::destroy(a, std::addressof(*first));
        ++first;
        ++dest;
    }
    return dest;
}
```

Optimizing `uninitialized_relocate`

```
template<class It, class FwdIt, class Alloc, class T = stdx::iterator_value_type_t<FwdIt>>
FwdIt __uninitialized_relocate_a(It first, It last, FwdIt dest, Alloc& a)
{
    constexpr bool is_simple_memcpy =
        std::is_same_v<T, stdx::iterator_value_type_t<It>> &&
        stdx::is_contiguous_iterator_v<It> &&
        stdx::is_contiguous_iterator_v<FwdIt> &&
        stdx::allocator_traits<Alloc>::template has_trivial_construct_and_destroy_v<T> &&
        stdx::is_trivially_relocatable_v<T>;
    if constexpr (is_simple_memcpy) {
        auto count = (last - first);
        __builtin_memcpy(std::addressof(*dest), std::addressof(*first), count * sizeof(T));
        return (dest + count);
    } else {
        while (first != last) {
            std::allocator_traits<Alloc>::construct(a, std::addressof(*dest), std::move(*first));
            std::allocator_traits<Alloc>::destroy(a, std::addressof(*first));
            ++first; ++dest;
        }
        return dest;
    }
}
```

By the way: convenience helpers

```
namespace stdx {

    template<class It> using iterator_value_type_t =
        typename std::iterator_traits<It>::value_type;

    template<class It> using iterator_category_t =
        typename std::iterator_traits<It>::iterator_category;

    template<class It> struct is_random_access_iterator :
        std::is_base_of<std::random_access_iterator_tag,
                        stdx::iterator_category_t<It>> {};

} // namespace stdx
```

If you don’t have these in your codebase, you’re missing out.

By the way: convenience helpers

See also: slides 60–65 of my CppCon 2017 talk “A Soupçon of SFINAE.”

```
namespace stdx {
    template<size_t K> struct priority_tag : priority_tag<K-1> {};
    template<> struct priority_tag<0> {};
} // namespace stdx
```

If you don’t have these in your codebase, you’re missing out.

Two new customization points

```
namespace stdx {
    // TODO: customization points are hard!
    // libstdc++ already effectively has both of these, but as
    // internal helpers, not well-known customization points.

    template<class It> struct is_contiguous_iterator :
        std::is_pointer<It> {};

    template<class T> struct is_trivially_relocatable :
        std::bool_constant<
            std::is_trivially_move_constructible_v<T> &&
            std::is_trivially_destructible_v<T>
        > {};

} // namespace stdx
```

We won't pursue contiguous iterators any further than this.

Two new customization points

We assume the vendor will specialize `is_trivially_relocatable` for most library types.
In fact, since there are so *many* class types involved, and writing template specializations sucks, we'll create a quick way to "opt in" via a class member.

```
template<class T> auto helper(priority_tag<1>) ->
    std::bool_constant<T::is_trivially_relocatable::value>;

template<class T> auto helper(priority_tag<0>) ->
    std::bool_constant<
        std::is_trivially_move_constructible_v<T> &&
        std::is_trivially_destructible_v<T>
    >;

template<class T> struct is_trivially_relocatable :
    decltype(helper<T>(priority_tag<1>{}));
```

Opting in to trivial relocatability

```
template<class T, class D = std::default_delete<T>>
class unique_ptr {
public:
    using deleter_type = D;
    using pointer = std::remove_reference_t<D>::pointer;

    using is_trivially_relocatable = std::bool_constant<
        std::is_trivially_relocatable_v<pointer> &&
        std::is_trivially_relocatable_v<deleter_type>
    >;

    // ...
};
```

reserve calls uninitialized_relocate

```
void vector::reserve(size_type cap) {
    if (cap > capacity()) {
        detail::vector_reallocator<vector> reallocator(m_allocator(), cap);
        if (m_data()) {
            T *new_begin = static_cast<T *>(reallocator.data());
            if constexpr (is_nothrow_move_constructible_v<T>) {
                // the move cannot fail, so we can destroy the original elts as we go
                __uninitialized_relocate_a(begin(), end(), new_begin, m_allocator());
            } else if constexpr (is_copy_constructible_v<T>) {
                // cannot destroy any of the original elts until we know the copy has succeeded
                __uninitialized_copy_a(begin(), end(), new_begin, m_allocator());
                __destroy_a(begin(), end(), m_allocator());
            } else {
                __uninitialized_move_a(begin(), end(), new_begin, m_allocator());
                __destroy_a(begin(), end(), m_allocator());
            }
        }
        reallocator.swap_into_place(this);
    }
}
```

Technically untrue; nothrow move-constructible doesn't mean nothrow MoveInsertable!
But all vendors get this wrong today.
LWG 2461.

reserve calls uninitialized_relocate

```
void vector::reserve(size_type cap) {
    if (cap > capacity()) {
        detail::vector_reallocator<vector> reallocator(m_allocator(), cap);
        if (m_data()) {
            T *new_begin = static_cast<T *>(reallocator.data());
            if constexpr (can_simply_relocate) {
                // the move cannot fail, so we can destroy the original elts as we go
                __uninitialized_relocate_a(begin(), end(), new_begin, m_allocator());
            } else if constexpr (is_copy_constructible_v<T>) {
                // cannot destroy any of the original elts until we know the copy has succeeded
                __uninitialized_copy_a(begin(), end(), new_begin, m_allocator());
                __destroy_a(begin(), end(), m_allocator());
            } else {
                __uninitialized_move_a(begin(), end(), new_begin, m_allocator());
                __destroy_a(begin(), end(), m_allocator());
            }
        }
        reallocator.swap_into_place(this);
    }
}
```

When is it safe to relocate in lieu of move?

```
template <class T, class Alloc>
class vector {
    using Alloc_traits = typename allocator_traits<Alloc>::template rebind_traits<T>;

    static constexpr bool can_simply_relocate =
        is_nothrow_move_constructible_v<T> || (
            is_trivially_relocatable_v<T> &&
            Alloc_traits::template has_trivial_construct_and_destroy_v<T>
        );

    // ...
}
```

Post-LWG 2461, this should say

```
Alloc_traits::template has_trivial_construct_and_destroy_v<T> && (
    is_nothrow_move_constructible_v<T> ||
    (is_trivially_destructible_v<T> && !is_copy_constructible_v<T>) ||
    is_trivially_relocatable_v<T>
);
```

T's own move-constructibility doesn't matter a whit, if the ***allocator*** isn't trivial.

Implementing `allocator_traits::htcad`

```
template <class T, class Alloc>
class vector {
    using Alloc_traits = typename allocator_traits<Alloc>::template rebind_traits<T>;

    static constexpr bool can_simply_relocate =
        is_nothrow_move_constructible_v<T> || (
            is_trivially_relocatable_v<T> &&
            Alloc_traits::template has_trivial_construct_and_destroy_v<T>
        );

    // ...
}
```

We’ve seen several places where libstdc++ has hard-coded this already. Example:

```
template<typename _InputIterator, typename _ForwardIterator, typename _Tp>
   inline _ForwardIterator
   __uninitialized_copy_a(_InputIterator __first, _InputIterator __last,
                          _ForwardIterator __result, allocator<_Tp>&)
   { return std::uninitialized_copy(__first, __last, __result); }
```

Implementing `allocator_traits::htcad`

In other words, libstdc++ “knows” this much:

```
template<class T> auto htcad(std::allocator<T>&, priority_tag<1>)
    -> true_type;
template<class Alloc> auto htcad(Alloc&, priority_tag<0>)
   -> false_type;


template<class Alloc>
struct allocator_traits {
    // ...

    template<class T> using has_trivial_construct_and_destroy =
        decltype(detail::htcad(declval<Alloc&>(), priority_tag<1>{}));
    template<class T> static constexpr bool has_trivial_construct_and_destroy_v =
        has_trivial_construct_and_destroy<T>{};
}
```

We’re going to generalize it.

Implementing `allocator_traits::htcad`

Generalized!

```
template<class Alloc, class T> auto htcad(Alloc& a, T *p, priority_tag<3>)
   -> typename Alloc::template has_trivial_construct_and_destroy<T>;
template<class Alloc, class T> auto htcad(Alloc& a, T *p, priority_tag<2>)
   -> decltype(void(a.destroy(p)), false_type{});
template<class Alloc, class T> auto htcad(Alloc& a, T *p, priority_tag<1>)
   -> decltype(void(a.construct(p, declval<T&&>())), false_type{});
template<class Alloc, class T> auto htcad(Alloc& a, T *p, priority_tag<0>)
   -> true_type;

template<class Alloc>
struct allocator_traits {
    // ...

    template<class T> using has_trivial_construct_and_destroy =
        decltype(detail::htcad<Alloc, T>(declval<Alloc&>(), nullptr,
priority_tag<3>{}));
}
```

TLDR / dependency graph

This dude...	normally uses this slow approach...	but if this trait is set appropriately...	it can use this faster approach...
std::vector reallocation	__uninit_copy_a + __destroy_a	is_nothrow_ move_constructible	__uninit_move_a + __destroy_a
std::vector reallocation	__uninit_move_a + __destroy_a	HTCAD && is_trivially_relocatable	__uninit_relocate_a
__uninit_relocate_a	Allocator::construct + Allocator::destroy in a loop	HTCAD	uninitialized_relocate
uninitialized_relocate	move + destroy in a loop	is_trivially_relocatable	__builtin_memcpy

Benchmark results

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

struct R : std::unique_ptr<int> {
    using is_trivially_relocatable = std::true_type;
};

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

std::vector<NR>	26µs ±110ns
std::vector<R>	26µs ±260ns
std'::vector<NR>	26µs ±180ns
std'::vector<R>	9µs ±60ns

The first 2 rows are libc++.
The final 2 rows are libc++ with my patch applied.

Compiler support required?

Consider:

```
struct Relocatable { using is_trivially_relocatable = std::true_type; };

struct AlsoRelocatable {
    Relocatable data_;
};

struct NotActuallyRelocatable : public Relocatable {
    Relocatable *ptr_to_myself_ = this;
};
```

Seems like we might need cooperation from the compiler in deducing the “trivial relocatability” of an arbitrary class type. Otherwise, we will have tedious false negatives for classes containing only relocatable ***members***, and dangerous false positives for classes that ***inherit*** an incorrect setting of their `is_trivially_relocatable` member!

Outline

- `is_trivially_relocatable<T>` [3–32]
  - Motivation / Implementation / Benchmarks / Downsides
- `is_trivially_comparable<T>` [34–46]
  - Motivation / Implementation / Benchmarks / Downsides
  - “memcmp comparable” versus “memberwise comparable” [42–46]
- `tombstone_traits<T>` [48–77]
  - Motivation / Implementation
  - Benchmark design and results [67–73] / Downsides

Motivating “comparison”

Comparison is one of the fundamental Stepanov operations.
When we make a copy of an object,

```
T b(a);
```

it is important that the copy be equal to the original — that’s what it means to be a “copy.”
And it is important that when things ***are*** equal, they ***compare*** equal.

```
assert( b == a );
```

`std::any` is one example of a type where copies ***are*** equal, but do not ***compare*** equal, in that `std::any` does not provide comparison operators.

Motivating "comparison"

When `T` is trivially copyable, we know we can make copies via `memcpy`.

```
T a;
T b(a);  // as-if by memcpy
```

If `a` is a trivially copyable object, and `b` is `memcmp`-equal to `a`,
then `b` is a ***copy*** of `a`; therefore `b` should ***compare*** equal to `a`.

But the reverse is not true!

For example, `float` is trivially copyable, but it has two different zeroes.
+0 and –0 compare unequal with `memcmp`, but compare equal with `operator==`.

Speed up `vector::operator==`

Any time we are lexicographically comparing a long contiguous sequence of trivially comparable objects, we can do it object-by-object, or we can do it all at once with `memcmp`.

```
template <class T, class A>
inline bool operator== (const vector<T, A>& x, const vector<T, A>& y)
{
    return x.size() == y.size() && std::equal(x.begin(), x.end(), y.begin());
}

template <class T, class A>
inline bool operator< (const vector<T, A>& x, const vector<T, A>& y)
{
    return std::lexicographical_compare(x.begin(), x.end(), y.begin(), y.end());
}
```

big (endian) caveat

Speed up vector::operator==

```
template<class It1, class It2, class BinaryPred>
bool__equal(It1 first1, It1 last1, It2 first2, BinaryPred pred, true_type) {
    auto length = std::distance(first1, last1) * sizeof(iterator_value_type_t<It1>);
    return __builtin_memcmp(std::addressof(*first1), std::addressof(*last1), length) ==
0;
}

template<class It1, class It2, class BinaryPred>
bool equal(It1 first1, It1 last1, It2 first2, BinaryPred pred) {
    using T = iterator_value_type_t<It1>;
    using U = iterator_value_type_t<It2>;
    using __compare_via_memcmp = integral_constant<bool,
        __is_contiguous_iterator_v<It1> && __is_contiguous_iterator_v<It2> &&
        is_same_v<remove_const_t<T>, remove_const_t<U>> &&
        is_trivially_comparable_v<T> &&
        __is_simple_equality_v<T, remove_cvref_t<BinaryPred>>
    >;

    return__equal(first1, last1, first2, pred, __compare_via_memcmp{});
}
```

Benchmark results

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

struct T : std::unique_ptr<int> {
    using is_trivially_comparable = std::true_type;
};

template<class VectorT, class Compare>
void test(benchmark::State& state) {
    int M = state.range(0);
    VectorT v1(M), v2(M);
    static bool b;
    for (auto _ : state) {
        b = Compare{}(v1, v2);
        benchmark::DoNotOptimize(b);
        benchmark::DoNotOptimize(v1);
        benchmark::DoNotOptimize(v2);
    }
}

BENCHMARK(test_compare<std::vector<NT>, std::less<>>)->Arg(10'000);
BENCHMARK(test_compare<std::vector<T>, std::less<>>)->Arg(10'000);
```

std::vector<NT> <	14µs ±480ns
std::vector<T> <	14µs ±570ns
std'::vector<NT> <	14µs ±240ns
std'::vector<T> <	5µs ±90ns

The first 2 rows are libc++.
The final 2 rows are libc++
with my patch applied.

Benchmark results

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

struct T : std::unique_ptr<int> {
    using is_trivially_comparable = std::true_type;
};

template<class VectorT, class Compare>
void test(benchmark::State& state) {
    int M = state.range(0);
    VectorT v1(M), v2(M);
    static bool b;
    for (auto _ : state) {
        b = Compare{}(v1, v2);
        benchmark::DoNotOptimize(b);
        benchmark::DoNotOptimize(v1);
        benchmark::DoNotOptimize(v2);
    }
}

BENCHMARK(test_compare<std::vector<NT>, std::equal_to<>>)->Arg(10'000);
BENCHMARK(test_compare<std::vector<T>, std::equal_to<>>)->Arg(10'000);
```

std::vector<NT>	<	14µs ±480ns
std::vector<T>	<	14µs ±570ns
std'::vector<NT>	<	14µs ±240ns
std'::vector<T>	<	5µs ±90ns

std::vector<NT>	==	7µs ±200ns
std::vector<T>	==	7µs ±430ns
std'::vector<NT>	==	7µs ±210ns
std'::vector<T>	==	5µs ±100ns

Compiler support required?

Consider:

```
struct Comparable { using is_trivially_comparable = std::true_type; };

struct AlsoComparable {
    Comparable data_;
    auto operator==(const AlsoComparable& b) const { return data_ == b.data_; }
};

struct NotActuallyComparable : public Comparable {
    int non_salient_data_member_ = rand();
};
```

Seems like we might need cooperation from the compiler in deducing the "trivial comparability" of an arbitrary class type. Otherwise, we will have tedious false negatives for classes containing only comparable ***members***, and dangerous false positives for classes that ***inherit*** an incorrect setting of their `is_trivially_comparable` member!

Compiler support attained(?)

In C++2a, we are getting the ability to default comparison operators.

```
struct Comparable { using is_trivially_comparable = std::true_type; };

struct AlsoComparable {
    Comparable data_;
    std::strong_equality operator<=>(const AlsoComparable&) const = default;
};
```

Here the compiler can ***deduce*** whether `AlsoComparable` is trivially `memcmp`-comparable, based on its struct layout and the `memcmp`-comparability of its individual members. Notice that the struct layout matters! ***If there are padding bytes***, inserted between members or inserted at the end of the layout, then the type will not have unique representations and therefore will not be `memcmp`-comparable.

And only the compiler knows for sure whether there will be padding bytes.

Another problem: naming

At Jacksonville, Jeff Snyder presented P0732R0 “Class Types in Non-Type Template Parameters.” This is a really really good proposal. WG21 should adopt it. Except for one little detail...

A type is *trivially comparable* if it is:

- a scalar type for which, if `x` is an expression of that type, `x <=> x` is a valid expression of type `std::strong_ordering` or `std::strong_equality`, or
- a non-union class type for which
  - there is an accessible `operator<=>` that is defined as defaulted within the definition of the class with return type of either `std::strong_ordering` or `std::strong_equality`, and
  - all of its base classes are trivially comparable, and
  - the type of each of its non-static data members is either trivially comparable or a (possibly multi-dimensional) array thereof.

Another problem: naming

This wording describes what I would call the *memberwise-comparable* concept.
This is exactly the right concept for P0732's compile-time purposes.
Unlike my *memcmp-comparable* concept, it does not care about padding.

A type is *trivially comparable* if it is:

- a scalar type for which, if `x` is an expression of that type, `x <=> x` is a valid expression of type `std::strong_ordering` or `std::strong_equality`, or
- a non-union class type for which
  - there is an accessible `operator<=>` that is defined as defaulted within the definition of the class with return type of either `std::strong_ordering` or `std::strong_equality`, and
  - all of its base classes are trivially comparable, and
  - the type of each of its non-static data members is either trivially comparable or a (possibly multi-dimensional) array thereof.

What is the meaning of “trivial”?

memberwise-comparable

- Comparison of the whole object is semantically equivalent to a lexicographic comparison of its members.
- Pro: Useful for template non-type parameters

`memcmp`-comparable

- Comparison of the whole object is semantically equivalent to a lexicographic comparison of its run-time bytewise representation.
- Pro: Useful for std::equal and std::lexicographic_compare
- Pro: In line with the “as-if by `memcpy`” meaning of “trivially copyable”
- Con: Perhaps too close to `has_unique_object_representations`

What is the meaning of “trivial”?

Boost `interprocess::offset_ptr` is an example of a type

- where `memcmp`-equality ↛ object equality
- and object equality ↛ `memcmp`-equality

`has_unique_object_representations<T>` →

- object-equality → `memcmp`-equality

“`T` is `memcmp`-comparable” →

- object-equality ↔ `memcmp`-equality (→ `has_unique_object_representations<T>`)
- object-less-than ↔ `memcmp`-less-than
- it is implementation-defined whether `int*` falls into this category

“T is memberwise-comparable” →

- `memcmp`-equality → object-equality (because all scalar types are trivially copyable)
- `int*` always falls into this category

P0732 should not co-opt "trivial"

- I think both concepts are needed.
- I think "trivially comparable" is the **wrong** name for P0732's memberwise comparison.

RS suggests "has strong structural equality."

- I think "trivially comparable" is the **right** name for "comparison always works as-if by `memcmp`."
- Both concepts require a certain amount of compiler support.
- Probably should split up `is_memcmp_equality_comparable<T>` and `is_memcmp_lessthan_comparable<T>`.

big (endian) caveat

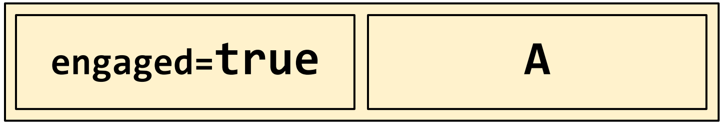
Outline

- `is_trivially_relocatable<T>` [3–32]
  - Motivation / Implementation / Benchmarks / Downsides
- `is_trivially_comparable<T>` [34–46]
  - Motivation / Implementation / Benchmarks / Downsides
  - “memcmp comparable” versus “memberwise comparable” [42–46]
- `tombstone_traits<T>` [48–77]
  - Motivation / Implementation
  - Benchmark design and results [67–73] / Downsides

Motivating “tombstones”

Consider the struct layout of C++17 `std::optional<T>`.

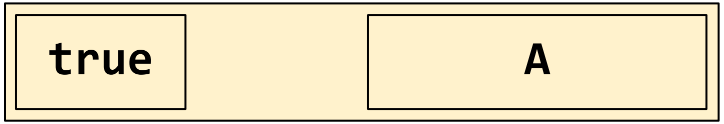
```
std::optional<T> o { std::in_place, A };
```



Motivating “tombstones”

Consider the struct layout of C++17 `std::optional<T>`.

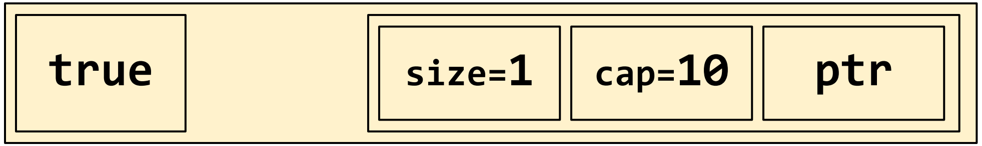
```
std::optional<T> o { std::in_place, A };
```



Motivating “tombstones”

Consider the struct layout of C++17 `std::optional<T>`.

```
std::optional<std::string> o { std::in_place, "A" };
```

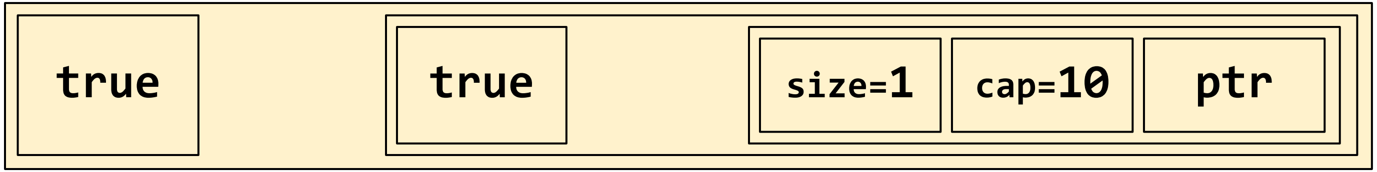


Toolchain	sizeof `string`	sizeof `opt<string>`
libstdc++	32	40
libc++	24	32
MSVC	32	40

Motivating “tombstones”

Consider the struct layout of C++17 std::optional<T>.

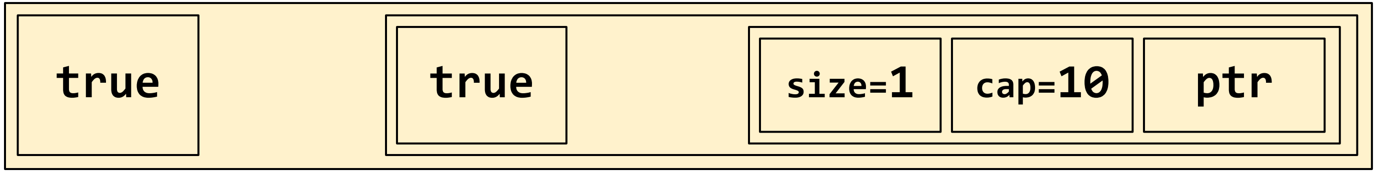
```
std::optional<std::string> o { std::in_place, "A" };
```



Toolchain	sizeof string	sizeof opt<string>	sizeof opt<opt<string>>
libstdc++	32	40	48
libc++	24	32	40
MSVC	32	40	48

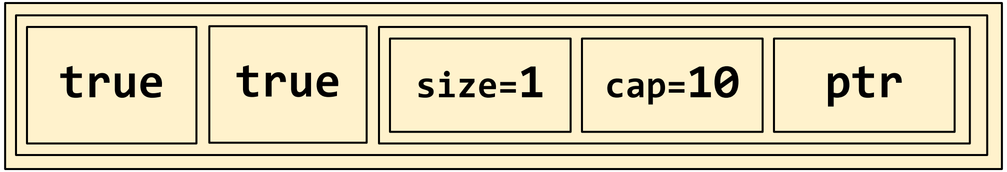
Motivating “tombstones”

Wouldn’t it be nice if `std::optional<std::string>` could be the same size as `std::string` itself? It’s not like we’re short of spare bits...



Motivating “tombstones”

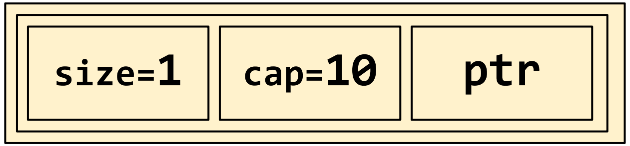
Wouldn’t it be nice if `std::optional<std::string>` could be the same size as `std::string` itself? It’s not like we’re short of spare bits...



Idea number one: Take the `engaged` member of the outer optional and cuddle it into the padding bytes of the inner optional.
This works, but it is very limited. It won’t get us to our goal.
`std::string` has no padding bytes to exploit.

Motivating “tombstones”

Wouldn’t it be nice if `std::optional<std::string>` could be the same size as `std::string` itself? It’s not like we’re short of spare bits...

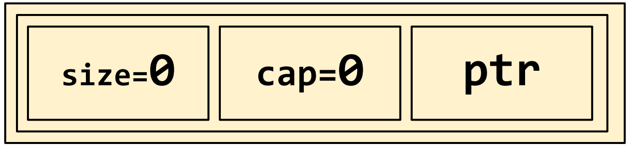


Idea number two: Give the string datatype itself some knowledge of an “invalid” or “tombstone” state.
In this case, our “invalid” state is `(size > cap)`. This bit-pattern will never ever be produced by a valid `std::string` object.
Therefore, `optional<std::string>` can use this bit-pattern to represent “disengaged”!

Motivating “tombstones”

We can now represent `opt<opt<opt<...<opt<string>...>` in exactly the same number of bytes as `string`!

```
opt<opt<opt<opt<string>>>> o = string("");
```



This `optional` holds a valid (empty) `string`.

Motivating “tombstones”

We can now represent `opt<opt<opt<...<opt<string>...>` in exactly the same number of bytes as `string`!

```
opt<opt<opt<opt<string>>>> o = opt<string>(nullopt);
```

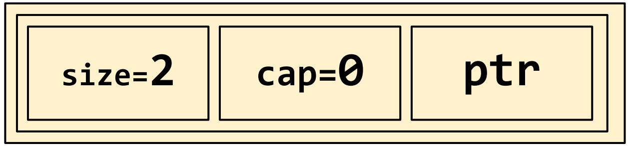
size=1	cap=0	ptr

Notice that this bit-pattern is *not* a `string`. The `optional` is disengaged; it does not hold a `string` at the moment. We are not *allowed* to construct a `string` inside a disengaged `optional`!
This is merely a *bit-pattern distinguishable from* that of any actual `string`.

Motivating “tombstones”

We can now represent `opt<opt<opt<...<opt<string>...>` in exactly the same number of bytes as `string`!

```
opt<opt<opt<opt<string>>>> o = opt<opt<string>>(nullopt);
```

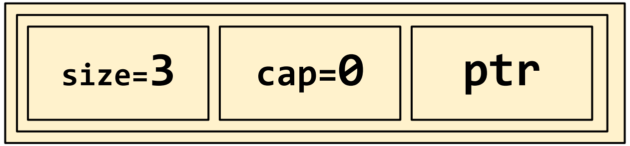


Notice that this bit-pattern is *not* a `string`. The `optional` is disengaged; it does not hold a `string` at the moment. We are not *allowed* to construct a `string` inside a disengaged `optional`!
This is merely a *bit-pattern distinguishable from* that of any actual `string`.

Motivating “tombstones”

We can now represent `opt<opt<opt<...<opt<string>...>` in exactly the same number of bytes as `string`!

```
opt<opt<opt<opt<string>>>> o = opt<opt<opt<string>>>(nullopt);
```

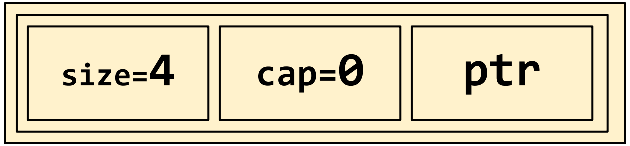


Notice that this bit-pattern is *not* a `string`. The `optional` is disengaged; it does not hold a `string` at the moment. We are not *allowed* to construct a `string` inside a disengaged `optional`!
This is merely a *bit-pattern distinguishable from* that of any actual `string`.

Motivating “tombstones”

We can now represent `opt<opt<opt<...<opt<string>...>` in exactly the same number of bytes as `string`!

```
opt<opt<opt<opt<string>>>> o = nullopt;
```



Notice that this bit-pattern is *not* a `string`. The `optional` is disengaged; it does not hold a `string` at the moment. We are not *allowed* to construct a `string` inside a disengaged `optional`!
This is merely a *bit-pattern distinguishable from* that of any actual `string`.

Design criteria

- We need cooperation from the `string` type. It must help us figure out how to create a “spare” bit-pattern that doesn’t look like a real string.
- And on the flip side, the `string` type must also help us distinguish real string bit-patterns from non-string bit-patterns.
- We want `string` to tell us about *many different* non-string bit-patterns, not just one such pattern. We should have a way to ask `string` for its number of available “spare” bit-patterns (and then recursively expose “all but one” of these patterns to our own caller).
- Of course, this approach must work for types other than `string`, too!

One new traits class

```
template<class T> struct tombstone_traits {
    static constexpr size_t spare_representations = 0;

    static constexpr size_t index(const T *) {
        return size_t(-1);
    }

    static constexpr void set_spare_representation(T *, size_t)
        = delete;
};
```

Specialization for string

```
template<> struct tombstone_traits<string> {

    static constexpr size_t spare_representations = (1 << 30);

    static size_t index(const string *p) {
        auto sz = *(size_t *)((std::byte *)p + 0);
        auto cp = *(size_t *)((std::byte *)p + 4);
        return (sz && !cp) ? (sz - 1) : size_t(-1);
    }

    static void set_spare_representation(string *p, size_t i) {
        *(size_t *)((std::byte *)p + 0) = i + 1;
        *(size_t *)((std::byte *)p + 4) = 0;
    }
};
```

Specialization for bool

```
template<> struct tombstone_traits<bool> {
    static constexpr size_t spare_representations = 254;

    static size_t index(const bool *p) {
        auto ch = *(uint8_t *)p;
        return (ch >= 2) ? (ch - 2) : size_t(-1);
    }

    static void set_spare_representation(bool *p, size_t i) {
        *(uint8_t *)p = i + 2;
    }
};
```

optional uses tombstone_traits

```cpp
template<class T, class Enable = void>
struct optional_storage {
    union { char m_dummy; T m_value; };
    bool m_has_value;
    constexpr optional_storage() noexcept
        : m_dummy(0), m_has_value(false) {}
    constexpr bool storage_has_value() const noexcept {
        return m_has_value;
    }
    void storage_reset() noexcept {
        m_value.~T();
        m_has_value = false;
    }
    template<class... Args>
    void storage_emplace(Args&&... args) {
        ::new (&m_value) T(std::forward<Args>(args)...);
        m_has_value = true;
    }
```

optional uses tombstone_traits

```
template<class T>
struct optional_storage<T, enable_if_t<tombstone_traits<T>::spare_representations >= 1>> {
   union { char m_dummy; T m_value; };
   constexpr optional_storage() noexcept {
       tombstone_traits<T>::set_spare_representation(&m_value, 0);
   }
   constexpr bool storage_has_value() const noexcept {
       return tombstone_traits<T>::index(&m_value) == size_t(-1);
   }
   void storage_reset() noexcept {
       m_value.~T();
       tombstone_traits<T>::set_spare_representation(&m_value, 0);
   }
   template<class... Args>
   void storage_emplace(Args&&... args) {
       ::new (&m_value) T(std::forward<Args>(args)...);
   }
```

Recursing on `tombstone_traits`

```
template<class T>
struct tombstone_traits<optional<T>> {
    static constexpr size_t spare_representations =
        (tombstone_traits<T>::spare_representations >= 1) ?
        (tombstone_traits<T>::spare_representations - 1) :
        tombstone_traits<bool>::spare_representations;
    static constexpr size_t index(const optional<T> *p) {
        if constexpr (tombstone_traits<T>::spare_representations >= 1) {
            auto i = tombstone_traits<T>::index(&p->m_value);
            return (i == size_t(-1) || i == 0) ? size_t(-1) : (i - 1);
        } else
            return tombstone_traits<bool>::index(&p->m_has_value);
    }
    static constexpr void set_spare_representation(optional<T> *p, size_t i) {
        if constexpr (tombstone_traits<T>::spare_representations >= 1)
            tombstone_traits<T>::set_spare_representation(&p->m_value, i + 1);
        else
            tombstone_traits<bool>::set_spare_representation(&p->m_has_value, i);
    }
};
```

Benchmark design

- A Robin Hood hash set holding elements of type `V`
    - Element = `variant<HasNeverHeldAValue, WasDeleted, V>`
    - Element = `optional<optional<V>>`
- `Tombable<K>`, a type that behaves like `std::string`
- Benchmark the (insert, erase, find) operations on a Robin Hood hash set *containing* `Tombable` strings
- Benchmark results

Benchmark design: Robin Hood hash

```
template<class T>
class robin_hood_element {
    optional<optional<T>> m_key;
    unsigned m_hash = 0;
public:
    bool has_never_held_a_value() const { return !m_key.has_value(); }
    bool holds_a_value() const { return m_key.has_value() && m_key->has_value(); }

    void clear() { m_hash = 0; m_key.reset(); }
    void set_tombstone() { m_key.emplace(); }
    void set_value(T&& key, unsigned hash) {
        m_key.emplace(std::move(key)); m_hash = hash;
    }

    unsigned& hash() { return m_hash; }
    unsigned hash() const { return m_hash; }
    T& key() { return **m_key; }
    const T& key() const { return **m_key; }
};
```

Benchmark design: `Tombable` type

```
template<int Spares> struct Tombable {
    const char *s;
    explicit Tombable(const char *t) : s(t) {}
    bool operator==(const Tombable& rhs) const noexcept { return strcmp(s, rhs.s) == 0; }
    unsigned hash() const noexcept { return bernstein_hash(s); }
};
template<int Spares> struct std::tombstone_traits<Tombable<Spares>> {
    static char special_values[Spares];
    static constexpr size_t spare_representations = Spares;
    static constexpr void set_spare_representation(Tombable<Spares> *p, size_t i) {
        *(char**)p = &special_values[i];
    }
    static constexpr size_t index(const Tombable<Spares> *p) {
        for (int i=0; i < Spares; ++i) {
            if (*(char**)p == &special_values[i])
                return i;
        }
        return size_t(-1);
    }
};
```

`Tombable<K>` is basically a very poor man's `string`. It's the size of a pointer, and has `K` different bit-patterns set aside as tombstones.

Benchmark design: Operations

```
template<class SetT,
         class T = typename SetT::value_type>
void test_set(benchmark::State& state) {
    int M = state.range(0);
    std::vector<T> to_insert, to_erase, to_find;
    for (int i=0; i < M; ++i) {
        to_insert.emplace_back(random_7_letters());
        to_erase.emplace_back(random_7_letters());
        to_find.emplace_back(random_7_letters());
    }
    for (auto _ : state) {
        SetT s;
        int x = 0;
        for (auto& r : to_insert) s.insert(r);
        for (auto& r : to_erase) erased += s.erase(r);
        for (auto& r : to_find) found += (s.find(r) != s.end());
        benchmark::DoNotOptimize(erased); benchmark::DoNotOptimize(found);
    }
}
```

We insert `M` random values into our set; then we erase `M` (different) random values; then we look up yet another `M` (different) random values.

Omitted for brevity: the code that records the high-water mark of `SetT`'s custom allocator.

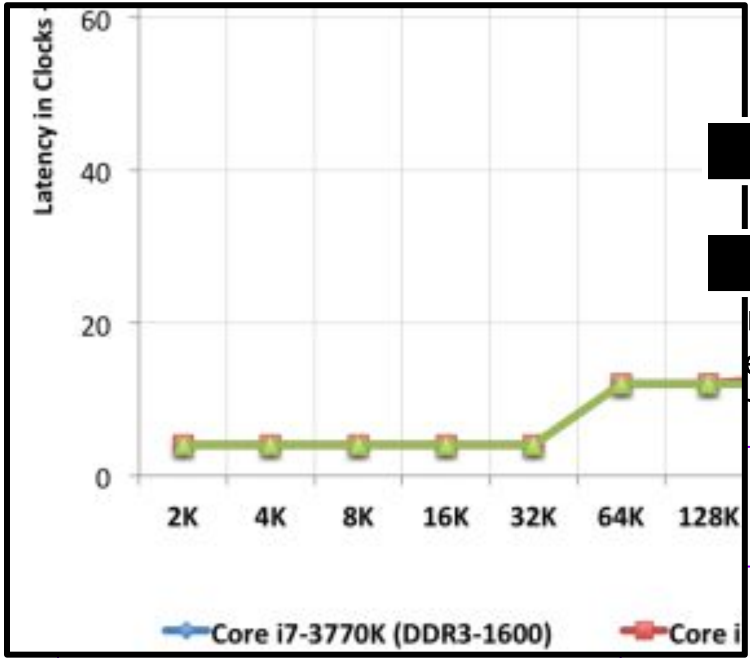
Benchmark results

Container	sizeof element before patch	sizeof element after patch	HWM before patch	HWM after patch	Running time before patch	Running time after patch
`rh_set<Tombable<0>>`	32	24	49 KB		1530µs	
`rh_set<Tombable<1>>`	32	24	49 KB		1460µs	
`rh_set<Tombable<2>>`	32	16	49 KB		1580µs	
`ska_flat_set<Tombable<0>>`	–	–	77 KB	77 KB	145µs	145µs

Benchmark results

Container	sizeof element before patch	sizeof element after patch	HWM before patch	HWM after patch	Running time before patch	Running time after patch
`rh_set<Tombable<0>>`	32	24	49 KB	37 KB	1530µs	1520µs
`rh_set<Tombable<1>>`	32	24	49 KB	37 KB	1460µs	1460µs
`rh_set<Tombable<2>>`	32	16	49 KB	25 KB	1580µs	94µs
`ska_flat_set<Tombable<0>>`	—	—	77 KB	77 KB	145µs	145µs

[redacted]



	nt e n	elemen after patch				
		24				
		24				
rh_set<Tombable<2>>	32	16	49 KB	25 KB	1580µs	94µs
ska_flat_set<Tombable<0>>	–	–	77 KB	77 KB	145µs	145µs

Now for the tradeoff

Makes optional<T>() un-constexpr

```
template<> struct tombstone_traits<bool>
{
    static void set_spare_representation(bool *p, size_t i) {
        *(uint8_t *)p = i + 2;
    }
};

template<class T> struct optional_storage<T, enable_if_t<...>>
{
   constexpr optional_storage() noexcept {
       tombstone_traits<T>::set_spare_representation(&m_value, 0);
   }
};

constexpr auto x = optional<bool>();  // No longer compiles!
```

Constexpr casts are like the Bay Bridge

```
constexpr auto foo(int *intptr) { return (void *)intptr; }  // OK!
constexpr auto foo(void *voidptr) { return (int *)voidptr; }  // Error!

error: cast from 'void *' is not allowed in a constant expression
```

Makes optional<T>() un-constexpr

```
template<> struct tombstone_traits<bool>
{
    static void set_spare_representation(bool *p, size_t i) {
        *(uint8_t *)p = i + 2;
    }
};
```

```
template<class T> struct optional_storage<T, enable_if_t<...>>
{
   constexpr optional_storage() noexcept {
       tombstone_traits<T>::set_spare_representation(&m_value, 0);
   }
};

constexpr auto x = optional<bool>();  // No longer compiles!
```

Questions?

Complete source code

- `is_trivially_relocatable:`
  https://github.com/Quuxplusone/libcxx/commit/is-relocatable
  https://github.com/Quuxplusone/libcxx/commit/34eb0b5c8f03880b
- `is_trivially_comparable:`
  https://github.com/Quuxplusone/libcxx/commit/is-trivially-comparable
  https://github.com/Quuxplusone/libcxx/commit/554689128b2dfc48
- `tombstone_traits:`
  https://github.com/Quuxplusone/libcxx/compare/master...tombstone-traits
  https://github.com/Quuxplusone/libcxx/commit/7ba5c0c217a9fbf0
  https://github.com/Quuxplusone/libcxx/commit/ea487ba07c86ba44
- `Benchmarks for all three features:`
  https://github.com/Quuxplusone/from-scratch/tree/master/cppnow2018
  https://github.com/Quuxplusone/from-scratch/tree/095b246d4dc9b88f/cppnow2018