Value semantics

Objects have values (42, 3.14, address-of-x, Tuesday, "hello").

They also have object representations (i.e., bit-patterns).

Sometimes, you can copy an object's value by copying its bit-pattern.

We call this `is_trivially_copy_assignable`. ✻

Side note: P3279

I wrote, “we call this `is_trivially_copy_assignable`.”
In fact all library vendors ***do*** optimize `std::copy` on the assumption that `is_trivially_copy_assignable` ***means*** that copy-assigning the value is tantamount to copying the object representation.

This is, technically, incorrect. https://godbolt.org/z/1EjKoqErT

```
struct Cat {};
struct Leopard : Cat {
   Leopard(int i) : spots_(i) {}
   int spots_;
   Leopard& operator=(Leopard&) = delete;
   using Cat::operator=;
};
static_assert(std::is_trivially_copyable_v<Leopard>);
static_assert(std::is_trivially_copy_assignable_v<Leopard>);
```

We will ignore this technical-incorrectness for the rest of these slides.

1. In reality, all STL implementations ignore it too.
2. P3279 proposes we should fix the traits to match reality.

Value semantics

Sometimes, you can destroy an object by “destroying” its bit-pattern.
That is, “just forget where it lives.”

We call this `is_trivially_destructible`.

Value semantics

Sometimes, you can move-construct an object by “move-constructing,” i.e., copying, its bit-pattern. We call this `is_trivially_move_constructible`.

How does this differ from `is_trivially_copy_constructible`?

Well, it depends on the ***value semantics*** that we’re replacing with memcpy. Can memcpy safely replace copy-construct? or move-construct? or either one?

This will depend on your type’s ***behavior*** when copying resp. moving.

Holistic property `is_trivially_copyable`

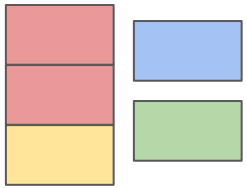
A very useful property, used by all library vendors today, is the holistic property that ***all*** value-semantic operations on `T` are tantamount to those operations on `T`'s object representation.

We call this `is_trivially_copyable`.

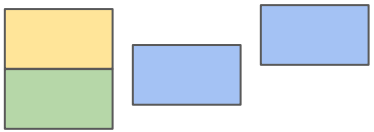
`is_trivially_copyable` means you can do any permutation/relocation/copying of `T` ***values*** simply by permuting/relocating/copying their ***object representations***.

Again, the Leopard example breaks the current Standard's trait; but we'll ignore that, just like vendors do.
P3279 is working on the fix.

Before:



After:

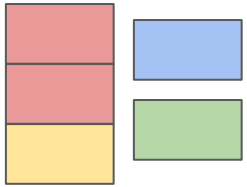


Notice that the count of  values has increased, and the count of  values has decreased. This is fine, for *trivially copyable* types.

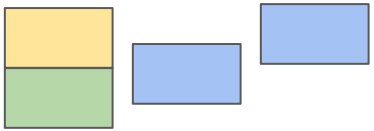
Holistic property `is_trivially_copyable`

`is_trivially_copyable` means you can do any permutation/relocation/copying of `T` ***values*** simply by permuting/relocating/copying their ***object representations***.

Before:



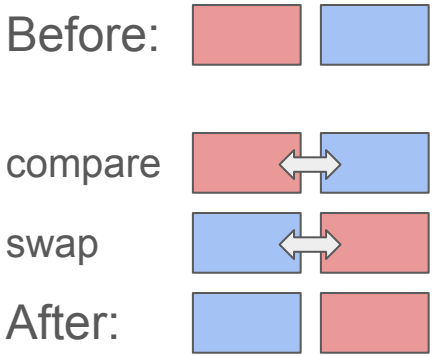
After:

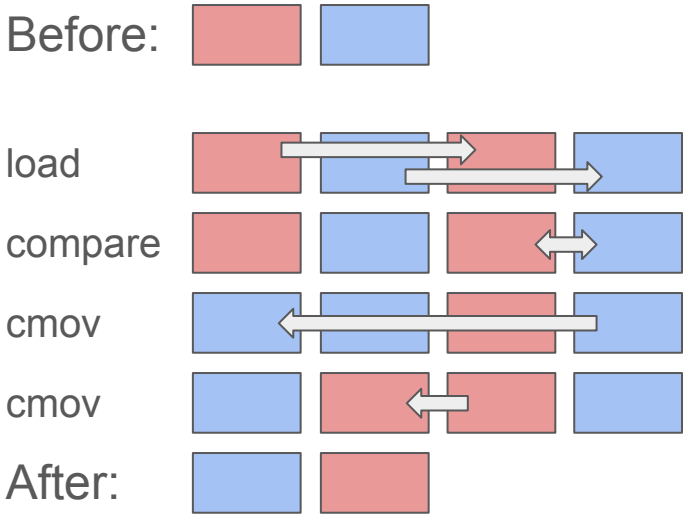


The colors here represent ***both*** values and object representations: they correspond.
We see that the green object has become blue. Was that by copy-assignment from the original blue object? or move-assignment? or copying from the yellow, destroying, and then copy-constructing from a blue object? We needn't say!

Example of `is_trivially_copyable` usage

Concretely, we can optimize `std::sort`'s helper `cond_swap` to be branchless.





✻ this comparison must *also* be trivial (libc++ has this trait)

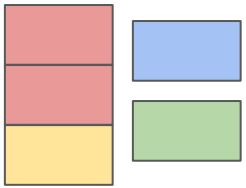
destroy

Holistic “affine” trivial value semantics

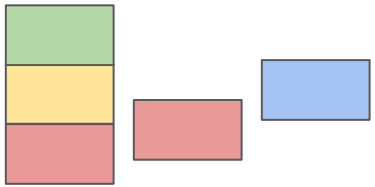
A very useful property, used by many ***third-party*** library vendors today, is the holistic property that all value-semantic operations on `T` are tantamount to those operations on `T`’s object representation ***as long as values are neither created nor destroyed***.

It means you can do any permutation/relocation (but not copying) of `T` ***values*** simply by permuting/relocating their ***object representations***.

Before:



After:



We call that property `is_trivially_relocatable`

For third-party libraries who use this holistic property, what name do they give it?

BSL calls it bslmf::IsBitwiseMoveable<T>.
Folly calls it folly::isRelocatable<T>.
Qt calls it QTypeInfo<T>::isRelocatable.
Subspace calls it sus::mem::TriviallyRelocatable<T>.
Abseil calls it absl::is_trivially_relocatable<T>.
AMC calls it amc::is_trivially_relocatable<T>.
HPX calls it hpx::experimental::is_trivially_relocatable<T>.
Parlay calls it parlay::is_trivially_relocatable<T>.
Pocketpy calls it pkpy::is_trivially_relocatable<T>.
Thrust calls it thrust::is_trivially_relocatable<T>.
U++ calls it Upp::is_trivially_relocatable<T>.

See the survey of all GitHub here:
https://quuxplusone.github.io/blog/2024/06/15/who-uses-trivial-relocation/