Value Semantics and Range Algorithms

Efficiency and Composability

Chandler Carruth chandlerc@gmail.com

Functional and Composable Range Algorithms

(or some other, better title...)

Some background...

What are Ranges?

In C++11 we are left to guess what a range is by squinting at [iterator.ranges] and the specification from range-based for loops.

- Can call **`std::begin`** and **`std::end`** on a range to get begin and end iterators
- Can pass them to a range based for loop

What are Ranges?

With C++14, we almost got this baked into the library with range-based constructors to containers.

Fundamentally:
a begin and end iterator

(for now)

Fundamentally: a begin and end iterator

(for now)

(sorry Eric)

What aren’t Ranges?
What can’t they do?

This is a big open question…

We should debate this more in Sebastian’s talk later this week...

What aren't Ranges?

A replacement for iterators

(sorry Andrei)

What does an output range look like?

I don’t have all the answers

(and I hope that’s OK…)

What are Containers?

Mutable, owning ranges.

- Mutability of the *range*, not the *elements*
- **`std::array`** and **`int[N]`** are *not containers*
  - And it turns out they make life really hard...

What are value-semantics?

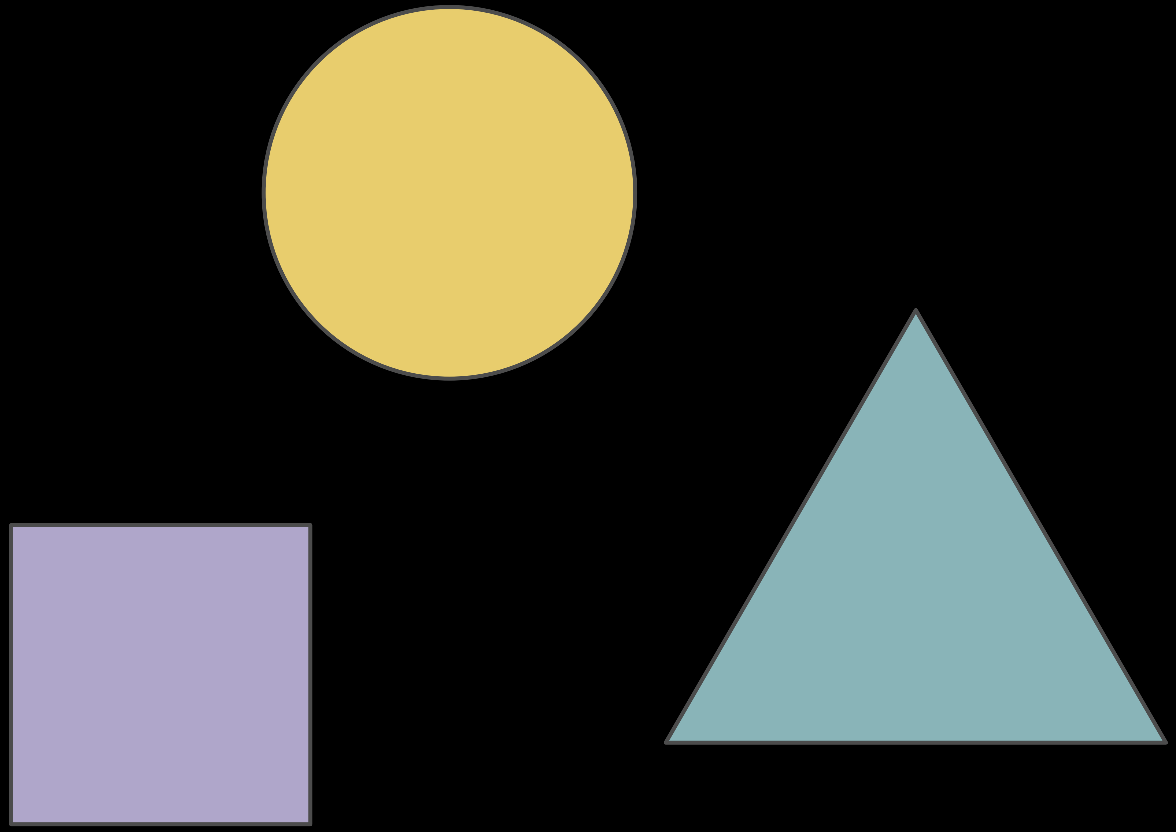
The term *value-semantics* means subtly different things to different people. I’m not trying to argue, just give a working definition here.

What are value-semantics?

A type has value-semantics when it is “regular”.

- Simple behavioral model:
  - copy, move, assign, compare
  - copies compare equal, move preserves identity

What are value-semantics?

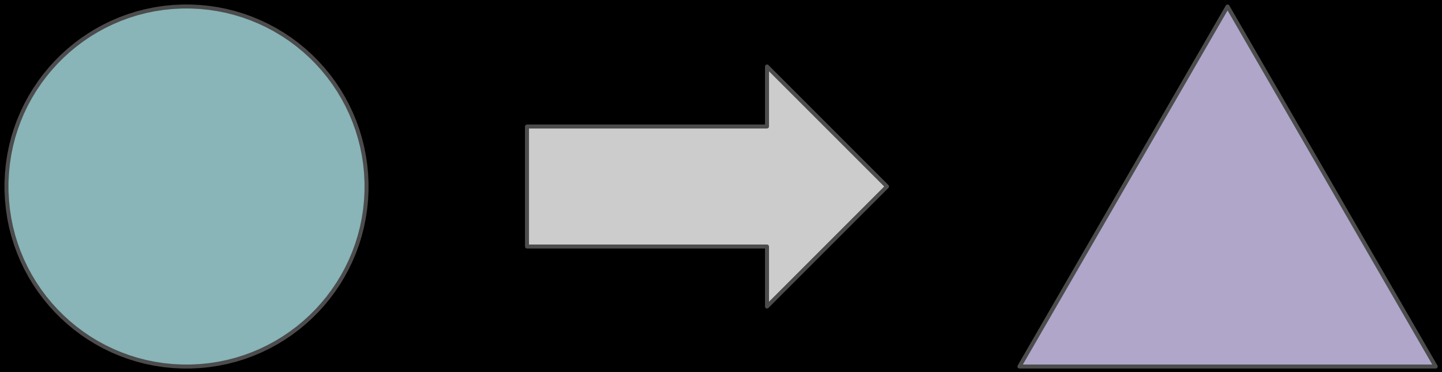


What is a value-semantic interface?

When a function interacts with your data as a value-semantic object.

- Accepts one or more values
- Returns zero or one values
- Doesn’t require calling multiple methods, passing in their results, with back-pointers to mutate the source.

What is a value-semantic interface?



What isn’t a value-semantic interface

What is a functional interface?

What is a functional interface?

… really? ;]

What is a functional interface?

Not going to dive into the formal side...

The core relevant to this discussion:

- Accepts an input parameter
- Returns the result
- No members, or in-place mutation

What is a functional interface?

This is the trivial and obvious way to satisfy my desire for an interface predicated on value semantics.

Putting it together for algorithms

- Functional - separating inputs and outputs
- Composable - because we’re using ranges
- Value Semantics - ranges are regular

And this is the point:

```
void f(const unordered_map &map_a,
       const unordered_map &map_b) {
  for (auto ab_entry : slice(
           sort(zip(transform(
                        map_a,
                         [](auto e) { return e->first; }),
                     transform(
                         map_a,
                         [](auto e) {
                            return map_b[e->first];
                         }))),
                0, 10)) {
    // do stuff...
  }
}
```

```cpp
auto f(std::vector<T> my_vec) {
  return slice(sort(std::move(my_vec)), 0, 10);
}
```

Some algorithms already close

- {all,any,none}_of
- count
- find, find_end
- search
- is_*
- partition_point
- {lower,upper}_bound, binary_search, equal_range
- includes
- {min,max,minmax}_element
- accumulate
- inner_product

Some algorithms map easily

- transform
- shuffle
- unique
- partition[1]
- sort
- nth_element
- merge, set_*
- *_heap
- {next,prev}_permutation
- iota
- adjacent_difference
- partial_sum

[1]The interface for this one isn't obvious, but my plan is to return a tuple of ranges.

Some will need very different APIs

fill, generate, and for_each notably

- Need a good generator API
- Eric Niebler has investigated a C++ range comprehension interface

These will serve the same purpose, regardless of API differences.

Remaining algorithms?

Mostly force-composed or irrelevant:

- foo_if, foo_n, etc are force-composed
- copy, move are replaced by assignment

(yes, there are also some *really* weird outliers I have skipped…)

Enter the *range* manipulators

These don’t mutate the values, but the range itself!

- slice
- filter
- zip
- concat

```
slice(<range>, begin, end[, stride])
```

Think Python’s slice:

```
 +---+---+---+---+---+---+
 | S | l | i | c | e | ! |
 +---+---+---+---+---+---+
 0   1   2   3   4   5   6
-6  -5  -4  -3  -2  -1
```

`slice(<range>, begin, end[, stride])`

Helps flesh out algorithms we've missed:

- copy_n + algorithm becomes a slice
- reverse becomes a negative stride

Provides a path toward specialized algorithmic improvements:

- partial_sort can implement slice(sort(...), …)

`filter(<range>, [](auto x) { … })`

Only visits elements for which the the lambda returns true.

- Provides all of the _if variant algorithms
- Also makes filtered iteration a breeze

`zip(<range1>, …, <rangeN>)`

Produces a range of an N-ary tuple

- Easy associative container interfacing
- Can use a tuple of references for efficiency
- Can compose with strided slice to zip adjacent elements

`concat(<range1>, …, <rangeN>)`

Produces a range spanning elements across each argument range

- Easy “append” replacement

Composability! Composability!

I know, the horse is dead.

But this really is the point.

- These range operations are primarily powerful in how they can compose
- Through composition we can factor the algorithm space much better

What about efficiency?

Because copies are what make C++ slow, right?

No.

moving right along...

Sometimes, we need to copy

See the copy algorithm, it was not added idly.

- Even more prevalent with slice
  - Can often copy a small subsequence quickly
- Many data sets are small and not a problem
- Invariants are good, and so we sometimes can't and don't want to mutate data

Move semantics for algorithms

For all the cases where you don’t need to copy!

- Good model already in C++ for moves
- Very small extension to pure functional interface conceptually
- If done well we can round trip back to our own storage and simulate “in place” algos!

Implementation experience helps

To help address this concern, I have a very experimental / prototype implementation.

- Focuses on a small subset (slice, filter, sort)
- Really bad code currently (sorry)
- Illustrates move semantic technique working
- Uncovered quite a few gotchas!
- Hopefully cleaned up and open soon (LLVM)

Implementation experience helps

That said, beware… I'm an optimizer hacker.

- I could use folks' help who are better at this
- I may have made things far harder than strictly necessary…

Still it seems to work, so hopefully it serves to motivate further investigation.

Model for R-Value Algorithm

1. Return a special range containing a container of the same type as the input
2. Move the input into the internal storage
3. (Eventually) apply the algorithm in-place using existing library
    a. Defer applying the algorithm so we can match away composed patterns

```
template <typename R> struct remove_cv_and_ref {
  typedef typename std::remove_cv<
      typename std::remove_reference<R>::type>::type type;
};

template <typename R, typename P>
detail::filtered_range<
    typename remove_cv_and_ref<R>::type,
    typename std::remove_reference<P>::type,
    std::is_rvalue_reference<R &&>::value>
filter(R &&range, P &&predicate) {
  return detail::filtered_range<
      typename remove_cv_and_ref<R>::type,
      typename std::remove_reference<P>::type,
      std::is_rvalue_reference<R &&>::value>(
          std::forward<R>(range),
          std::forward<P>(predicate));
}
```

```
template <typename R, typename P>
class filtered_rvalue_range : private scratch_rvalue_range_base<R>,
                              public filtered_range_base<R, P> {
protected:
  template <typename P2>
  filtered_rvalue_range(R &&Range, P2 &&Predicate)
      : filtered_rvalue_range::scratch_rvalue_range_base(
            std::move(Range)),
        filtered_rvalue_range::filtered_range_base(
            std::begin(this->Range), std::end(this->Range),
            std::forward<P2>(Predicate)) {}

public:
  typedef typename filtered_rvalue_range::iterator iterator;

  // ...
}
```

Model for R-Value Algorithm

Also need to store the result back to a variable without copying for temporary algorithms.

- Detect conversion to a container which can be move constructed from the incoming type
  - Move directly from internal storage to result
- Detect conversion to any container which can be constructed from begin/end pair
  - Move elements (but not range)

```cpp
template <typename ResultT,
          typename std::enable_if<std::is_constructible<ResultT, R>::value,
                                  int>::type = 0>
operator ResultT() && {
  this->Range.erase(
      std::remove_if(std::begin(this->Range),
                     std::end(this->Range),
                     [&](const decltype(*std::begin(this->Range)) &x) {
                       return !this->Predicate(x);
                     }),
      std::end(this->Range));

  return ResultT(std::move(this->Range));
}

template <typename ResultT,
          typename std::enable_if<
              !std::is_constructible<ResultT, R>::value &&
                  std::is_constructible<ResultT, iterator, iterator>::value,
              int>::type = 0>
operator ResultT() && {
  return ResultT(std::make_move_iterator(this->begin()),
                 std::make_move_iterator(this->end()));
}
```

Model for R-Value Algorithm

What about auto and capturing an algorithm?

- That’s fine, we’ve moved the storage into the result
- Can even continue to have efficiency by moving back out of a variable

Model for L-Value Algorithm

Ironically, this is a bit trickier.

- A simple model: eagerly copy
  - Then can model the same as an R-Value algorithm
  - All transforms can be done in-place
- This is even required in some cases where there is no incremental and efficient algorithm
  - Notably: sort, partition, and shuffle

Model for L-Value Algorithm

But all that copying will be terribly wasteful.

Instead, we could make (some) be lazy!

- Store just enough information to produce the desired *view* of the input
- Copy only when the algorithm necessitates it
- The core range mutations are *perfect* for this!
  - slice, filter, zip, concat

RIGHT?

(this is exactly what I set out to implement)

Model for L-Value Algorithms

Yep. Except for the nightmare of complexity.

- A lazy slice is tricky if the input’s underlying iterator is bidirectional, much less an input_iterator
- Lazily filtering requires lowering from random access to bidirectional, making common operations linear time.

Model for L-Value Algorithms

- concat requires a reasonably robust variant to handle lazily walking different ranges
  - And have to use the lowest common denominator
- zip at least is easy to make lazy…

So, with all this complexity, maybe copying is the right way to go...

Model for L-Value Algorithms

Except that you can’t.

Model for L-Value Algorithms

Implementing slice for InputIterators (or InputRanges, or InputIterables) is … hard.

- Can’t increment past the start position
- Have to track the distance to the end position out-of-band
  - Wasn’t this one of the hacks with iterators that ranges were supposed to fix?

And filter is just as bad.

Model for L-Value Algorithms

You can see this problem elsewhere

- Today, sort is only applicable for random-access iterators as an algorithm
  - Doesn’t really make sense: std::list::sort?

For sort (and friends), need to pick a container

- Another need for range traits for the default
- Can cast inputs to temporaries for control

Model for L-Value Algorithms

Can upgrade to a container to cope with input iterators lacking backing store for slice, filter

- Still efficiency reasons to be lazy where we can
  - If we only iterate, strictly cheaper
  - Keeps expression templates on the table
- Primarily upgrade on conversion or composition

Careful conversion is a theme...

Need to carefully control conversion

- Per-algorithm logic necessary often
- Per-container logic necessary often
- This is only do-able with conservatively enabled conversion operators
- Completely broken by N3513
  - Glad it didn’t make it into C++14!

Still, may not be faster than today’s algorithms

But that’s not the point!

```
void f(const unordered_map &map_a, const unordered_map &map_b) {
  for (auto ab_entry : slice(
           sort(zip(transform(map_a,
                              [](auto e) { return e->first; }),
                    transform(map_a,
                              [](auto e) { return map_b[e->first]; }))),
           0, 10)) {
    // do stuff...
  }
}
```

Expression templates?

- Probably necessary to make compositional patterns provide efficiency guarantees
- Potential for significant further improvements
- But *not necessary* for this to be a success

Also one critical assumption:

Moves are *free*

Assumption: Moves are free

Not true today, sadly. But they are often close.

- Eradicate throwing moves
- Make moves elidable
  - Support move-aside, move-back with elision
  - Support identity swapping to elide moves
    - TODO: example!

This is just the start...

Some obvious next steps:

Range assignment:

- slice(x, 4, 8) = y;
- zip(x, y) = my_map;

Splice supporting ranges?

Some obvious next steps:

There are certainly more algorithms needed:

- split: range -> range-of-ranges
- join: range-of-ranges -> range
- fold[rl]: functional list primitive
- ???

Also, are there better layerings or factorings?

Questions?

OK, I lied.
Let’s talk what a range *is*.

Two iterators is insane.

Two iterators is insane.

Eric Niebler’s blog posts about this are excellent, polite, and undersell the importance of this.

Andrei gets the severity of the problem right, but takes it to an extreme (gasp).

One iterator is quite nice.

We need to have a position.

- It is fundamental.
- It is the obvious return of many operations.
- That’s OK.

The end iterator is a disaster.

See Eric’s posts:

http://ericniebler.com/2014/02/16/delimited-ranges/

http://ericniebler.com/2014/02/18/infinite-ranges/

I won’t repeat them, he does a better job. Go read them.
Done? Great.

The Iterable concept is close

http://ericniebler.com/2014/02/21/introducing-iterables/

This is close.

- It trades the end iterator for a differently typed end iterator
  - Makes sentinels much cleaner

But there is an even simpler generalization...

A range consists of an iterator and a termination predicate.

<iterator, bool(iterator)>

More Questions?