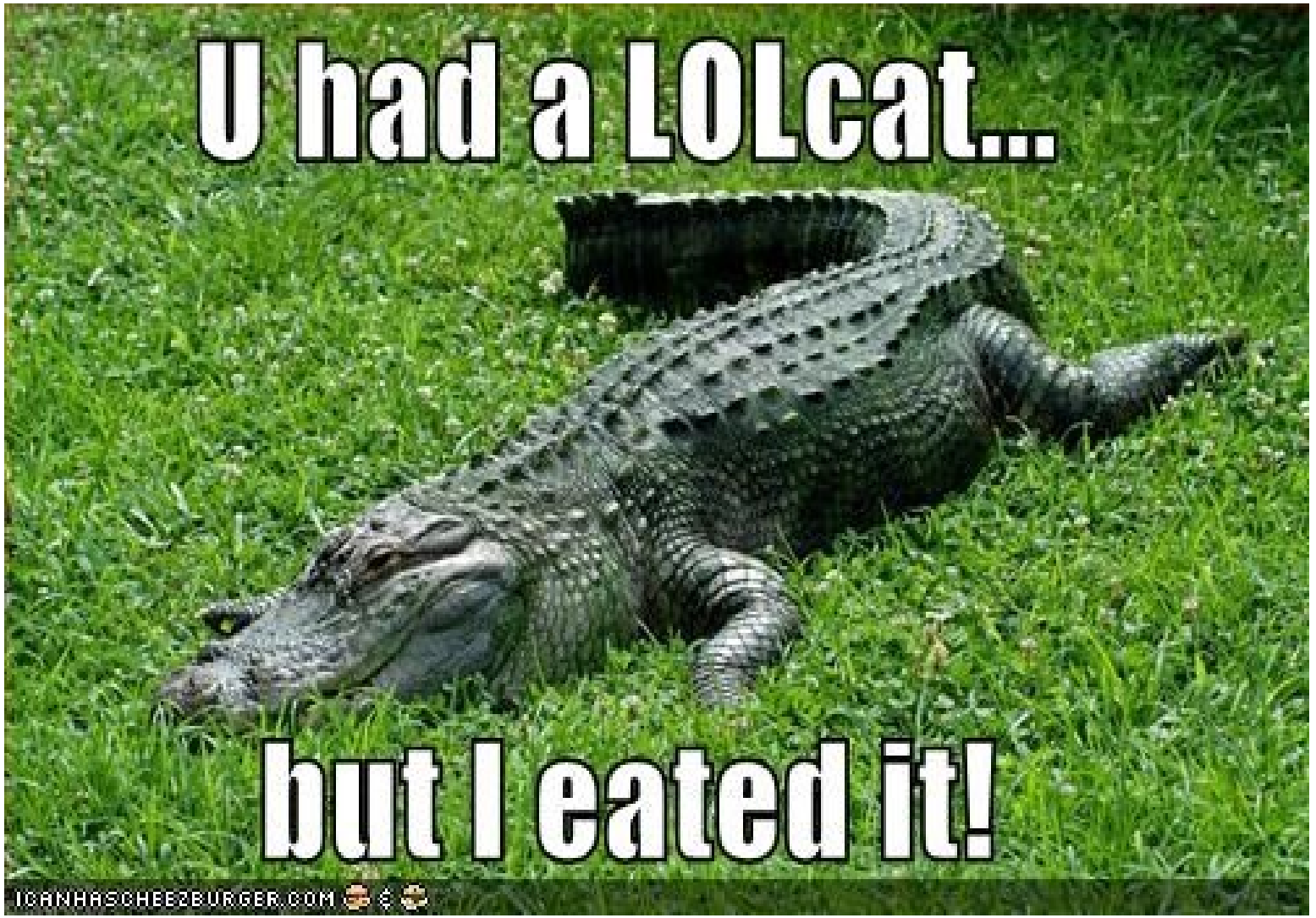
# Iterators Will Stay

## A Survey of Range Libraries

Sebastian Redl
sebastian.redl@getdesigned.at


U had a LOLcat...
but I eated it!
ICANHASCHEEZBURGER.COM

# Range Library

```
vector<int> v = { 4, 2, 5, 2, 5, 2, 1, 3 };

sort(v);

copy(equal_range(v, 2), insert<int>(cout));
```

# Range Library

```
vector<int> v = { 4, 2, 5, 2, 5, 2, 1, 3 };


sort(v);

copy(equal_range(v, 2), insert<int>(cout));


for (auto e : transformed(_1 / 2,
                  filtered(_1 % 2 == 0, v)))
{}
```

# Adapters

- Wrappers for iterators (or ranges)
- Modify primitive operations
  - Skip elements on increment (filtering, striding)
  - Apply function on dereference (projection)
  - etc.

# SG 9

- Established late 2012 / early 2013
- Headed by Marshall Clow
- “The goal of this study group (SG9) is to research the idea of adding ranges to a future version of the C++ standard library, and to create a proposal for the committee to consider.”

# Goals

- Convenience
- Efficiency
- Safety
- Reuse of old code

# Reuse of Old Code

- Use standard iterators with new algorithms

# Reuse of Old Code

- Use standard iterators with new algorithms
- Use new ranges with old algorithms

# Reuse of Old Code

- Use standard iterators with new algorithms
- Use new ranges with old algorithms
- Massively inhibits freedom of design

# Reuse of Old Code

- Use standard iterators with new algorithms
- Use new ranges with old algorithms
- Massively inhibits freedom of design
- Probably a hard requirements for SG 9 result

# Convenience

- Primary motivation for ranges

```
sort(v.begin(), v.end());
// vs.
sort(v);
```

# Convenience

- Primary motivation for ranges

```
sort(v.begin(), v.end());
// vs.
sort(v);

copy(equal_range(v, 2), insert<int>(cout));
// vs.
auto r = equal_range(v.begin(), v.end(), 2);
copy(r.begin(), r.end(),
     ostream_iterator<int>(cout));
```

# Convenience

- Iterators are hard to write
  - Comparison is often unintuitive

# Convenience

- Iterators are hard to write
  - Comparison is often unintuitive

```
bool equal(iterator that) const
{
  return str_
    ? (that.str_ == str_ ||
        (!that.str_ && !*str_))
    : (!that.str_ || !*that.str_);
}
```

# Convenience

- Iterators are hard to write
  - Comparison is often unintuitive
- Adapters are hard to use
  - Every iterator needs to be wrapped

# Convenience

- Iterators are hard to write
  - Comparison is often unintuitive
- Adapters are hard to use
  - Every iterator needs to be wrapped

```
copy(make_move_iterator(
       make_reverse_iterator(first)),
     make_move_iterator(
       make_reverse_iterator(last)),
     target);
copy(make_move_range(make_reverse_range(r)),
     target);
```

# Convenience

- Iterators are hard to write
  - Comparison is often unintuitive
- Adapters are hard to use
  - Every iterator needs to be wrapped
- Adapters are hard to write
  - Avoiding undefined behavior

# Efficiency

- Ranges must be as efficient as iterators
- Minimal overhead over hand-written algorithm

# Efficiency

- Iterators duplicate information
  - Projection iterators: function in every iterator
  - Filter iterators: must know end iterator

# Efficiency

- Iterators duplicate information
  - Projection iterators: function in every iterator
  - Filter iterators: must know end iterator
- Ranges require good calling convention
  - Iterators may fit into register, ranges don't
  - Needs compiler to split struct across registers

# Safety

- Detect out-of-bounds access
- Avoid invalidation traps

# Sample Libraries

- Boost.Range
- Eric Niebler's Range v3 (Iterables)
- Phobos std.range (D standard library)
- libaccent

# The Big Divide

- Boost.Range — Use Iterators
- Eric Niebler's Range v3 (Iterables)

- Phobos std.range (D standard library)
- libaccent — No Iterators

# Boost.Range

- Range is anything with begin/end
- Iterator is still main primitive

# Boost.Range

- Range is anything with begin/end
- Iterator is still main primitive
- Perfect compatibility

# Boost.Range

- Range is anything with begin/end
- Iterator is still main primitive
- Perfect compatibility
- Inherits all efficiency and safety downsides
- Inherits some convenience downsides

# Boost.Range

- Range is anything with begin/end

```
vector<int> v = { 4, 2, 5, 2, 5, 2, 1, 3 };
rg::sort(v);
```

- And it doesn't have anything else

# Boost.Range

- Iterator is still main primitive

```
template <typename Range>
void sort(Range& r) {
  std::sort(boost::begin(r),
            boost::end(r));
}
```

# Boost.Range

- Inherits all efficiency and safety downsides
  - No protection beyond what iterators have
  - Adapted at iterator level: space explosion
  - Passing ranges by value would mean big objects
  - Ranges are not meant to be passed by value

# Boost.Range

- Inherits some convenience downsides
  - Range adapters just produce iterator adapters
  - Still need to write iterator adapters

# Eric Niebler's Range v3

- Iterator is still main primitive
- Iterator pairs need not be same type

# Eric Niebler's Range v3

- Iterator is still main primitive
- Iterator pairs need not be same type
- Iterable is anything with begin/end
- Range is a homogenous Iterable

# Eric Niebler's Range v3

- Iterator pairs need not be same type
  - istream_iterator knows when it's done
  - An “end” istream_iterator is ugly to implement
  - Instead, have a sentinel “end” iterator
  - Comparison with sentinel is real iterator's is_done

# Eric Niebler's Range v3

- Iterator pairs need not be same type
  - Can represent arbitrary end predicate
  - Only works up to forward ranges
  - Bidirectional ranges must be homogenous
  - Can this represent counted ranges effectively?

# Lifetime issues

- Is this code valid?

```
auto foo(const vector<int>& v) {
  auto r = reverse(
             transform(SomeFunction(),
               v));
  auto it = find(r, 99);
  if (it == r.end()) return it;
  return next(it);
}
```

In Boost.Range? In Range v3?

# Lifetime issues

- Boost.Range
  - Don't know. Doesn't document it.
- Range v3
  - No. Iterators depend on ranges.

# The Big Divide

- Iterator pairs can be very awkward
- Relaxing requirements solves some problems
- Andrei Alexandrescu: Iterators Must Go!

# Phobos std.range

- Implements ideas of Alexei Alexandrescu's “Iterators Must Go” keynote of 2009
- Range is the main primitive

# Phobos std.range

- Implements ideas of Alexei Alexandrescu's “Iterators Must Go” keynote of 2009
- Range is the main primitive
- Strictly less powerful than bidirectional iterator
  - Ranges can never grow

# Phobos std.range

- Implements ideas of Alexei Alexandrescu's “Iterators Must Go” keynote of 2009
- Range is the main primitive
- Strictly less powerful than bidirectional iterator
  - Ranges can never grow
- Sufficient for all purposes, but can be unintuiti
  - What does find() return?

# Range Operations

- Forward traversal
  - empty?
  - access first element
  - drop first element

# Range Operations

- Forward traversal
  - empty?
  - access first element
  - drop first element
- Bidirectional traversal
  - access last element
  - drop last element

# Range Operations

- Forward traversal
  - empty?
  - access first element
  - drop first element
- Bidirectional traversal
  - access last element
  - drop last element
- Random access traversal
  - Drop arbitrary number of elements on either side

# Phobos std.range

- What does find() return?

# Phobos std.range

- What does find() return?
  - find() returns range from found element

# Phobos std.range

- What does find() return?
  - find() returns range from found element
  - findSkip() returns range after found element

# Phobos std.range

- What does find() return?
  - find() returns range from found element
  - findSkip() returns range after found element
  - findSplit() returns before match, match, after matc

# Phobos std.range

- What does find() return?
  - find() returns range from found element
  - findSkip() returns range after found element
  - findSplit() returns before match, match, after matc
  - until() returns until before or after match (flag)

# Phobos std.range

- What does find() return?
  - find() returns range from found element
  - findSkip() returns range after found element
  - findSplit() returns before match, match, after matc
  - until() returns until before or after match (flag)
- Suddenly four (seven?) algorithms

# libaccent

- Phobos-style Range
- Additional primitive to represent position

# libaccent

- Phobos-style Range
- Additional primitive to represent position
  - Knows whether it refers to something

# libaccent

- Phobos-style Range
- Additional primitive to represent position
  - Knows whether it refers to something
  - Can be dereferenced

# libaccent

- Phobos-style Range
- Additional primitive to represent position
  - Knows whether it refers to something
  - Can be dereferenced
  - Can be used to cut ranges short

# libaccent

- Phobos-style Range
- Additional primitive to represent position
  - Knows whether it refers to something
  - Can be dereferenced
  - Can be used to cut ranges short
  - Cannot be incremented

# libaccent

- Phobos-style Range
- Additional primitive to represent position
  - Knows whether it refers to something
  - Can be dereferenced
  - Can be used to cut ranges short
  - Cannot be incremented

```
if (auto p = find(rng, is(42))) {
  std::cout << *p << '\n';
  auto before = until(rng, p);
  auto after = after(rng, p);
}
```

# Lifetime Issues

- Ranges always used by value
  - Containers are not ranges

# Lifetime Issues

- Ranges always used by value
  - Containers are not ranges
- Positions do not depend on ranges

# Lifetime Issues

- Ranges always used by value
  - Containers are not ranges
- Positions do not depend on ranges
- Invalidation only happens if underlying sequence goes away
  - Might be garbage-collected class in D

# Iterators vs No Iterators

- Ranges more flexible in implementation
  - Delimited, counted, infinite ...

# Iterators vs No Iterators

- Ranges more flexible in implementation
  - Delimited, counted, infinite ...
- Iterators more flexible for algorithms
  - Two iterators == one range

# Iterators vs No Iterators

- Ranges more flexible in implementation
  - Delimited, counted, infinite ...
- Iterators more flexible for algorithms
  - Two iterators == one range
  - Three iterators == three ranges
    - range + find() result
  - Four iterators == six ranges
    - range + equal_range() result

# Iterators vs No Iterators

- Ranges are safer
  - Know when they are empty
  - Can never grow

# Iterators vs No Iterators

- Ranges are safer
  - Know when they are empty
  - Can never grow
- Ranges work more easily with adapters
  - Range adapters easier than iterator adapters
  - No iterators with dependent lifetimes

# Iterators vs No Iterators

- Iterators are harder to invalidate
  - List splice can invalidate ranges
  - List modification invalidates ranges that hold coun

# Iterators vs No Iterators

- Iterators are harder to invalidate
  - List splice can invalidate ranges
  - List modification invalidates ranges that hold coun
- any_range far simpler than any_iterator

# Iterators vs No Iterators

- Iterators are harder to invalidate
  - List splice can invalidate ranges
  - List modification invalidates ranges that hold coun
- any_range far simpler than any_iterator
- Iterators are used by existing code

# Iterator-Based Range Libraries

| Library | Adapters | Invalidation | Copy Semantics |
|---|---|---|---|
| **Boost.Range** | Ranges very thin wrapper | Undocumented Iterators held by value | Reference semantics |
| **Range v3 (Eric)** | Ranges hold adapter logic | Iterators depend on ranges | Reference semantics |
| **Ranges (Chandler)** | Ranges own elements and hold logic | Iterators depend on ranges | Value semantics |
| **Ranges (Arno Schödl)** | ??? | Iterators independent of ranges | Non-copyable |

# Discussion

- Boost.Range:
  http://www.boost.org/doc/libs/1_55_0/libs/range/doc/html/index.htm
- Range v3:
  http://ericniebler.com/2014/02/16/delimited-ranges/
- Chandler's Ranges: ???
- Phobos:
  http://dlang.org/phobos/std_range.html
- libaccent (use “rewrite” repository):
  https://code.google.com/p/libaccent/