Pairs do not make good ranges

Revision History

The paper was favourably reviewed in Santa Cruz, but several oversights with the wording were identified that needed clearing up:

- Struck the pair range interface
- Add two-iterator version of make_range
- Fixed up some text formatting

Basic Issue

At the Frankfurt meeting the revised form of the range-based for loop was voted into the working paper. Part of this feature is to provide for-loop support that treats every pair of iterators as a valid range. While often useful, this is not a safe assumption to make in general, and is typically the sort of error we look to the type system to protect us from. Most of the operations in the standard library that return pairs of iterators do not guarantee to be a valid range, for example minmax_element (where order is not guaranteed) or mismatch (where iterators will typically come from entirely separate ranges.)

Note this problem is not entirely new, and a similar paper to this was being prepared for the concept-based form of the same feature.

Which library components are affected?

The following library algorithms return a pair of iterators, or a pair of the same type that may be instantiated with iterators:

- mismatch
- partition_copy
- minmax
- minmax_element
- equal_range

Of these, only equal_range actually guarantees to return a range.

The associative and unordered containers also support an equal_range operation that returns a valid range in a pair of iterators. Conversely, those containers store their data in pairs which may also yield ‘accidental ranges’ e.g. map<vector<T>::iterator, vector<T>::iterator>.

The submatch type in the regular expression library derives from a pair of iterators, and can easily provide its own explicit support for the for loop.

Which user components are affected?

Any code that includes a pair of iterators is at risk, with the common case likely to be a pair of pointers. Likewise, any generic code that returns pairs is at risk in the case the pair is instantiated with iterators.

Proposed solution

Replace the for-loop support for pair<iterator,iterator> with a new range type that wraps a pair of iterators. This type shall be API compatible with pair<iterator,iterator> (either by inheritance or implicit conversion) although will likely break a small set of ABIs that previously relied on the return type of a function being strictly an instance of std::pair. If a user want to call the new for loop with a pair object holding a iterators that they assert form a valid range, this can be done easily be passing calling make_range with the pair. In addition to solving the accidental match problem, the range type is useful in its own right for creating lightweight views on sequences and subranges.

Update the specification for all equal_range operations to return the new range<iterator> type, not a pair<iterator,iterator>.

Finally, add explicit support for free-standing begin/end calls to regex::submatch.

Proposed Wording

Strike subclause 20.3.5 [pair.range]

~~**20.3.6 pair range access [pair.range]**~~

~~template <class InputIterator>~~
~~InputIterator begin(const std::pair<InputIterator, InputIterator>& p);~~
~~1 *Returns:* p.first.~~
~~template <class InputIterator>~~
~~InputIterator end(const std::pair<InputIterator, InputIterator>& p);~~
~~2 *Returns:* p.second.~~

Replace subclause 20.3.5 [pair.range] with

20.3.6 Ranges [utility.range]

The library provides a `range` utility class template that holds a pair of iterators. This class can store a view on any subrange, and can be passed as a Range parameter to the range-based for loop.

```
template<typename Iter>
struct range : pair<Iter, Iter> {
    using pair::pair;
};

template<typename Iter>
Iter begin(const range<Iter> & p );

Returns: p.first

template<typename Iter>
Iter end(const range<Iter> & p );
```

Returns: `p.second`

```
template<typename T>
auto make_range( T & t )-> range<decltype(begin(t))>;
```

Requires: T satisfies the type requirements for a Range suitable for use as the range argument to the range based for-loop.

Returns: `range<decltype(begin(t))>{begin(t), end(t)};`

[Draughting note: T can be deduced to be const, but cannot be a temporary. If T deduces to be const then the iterator type will similarly deduce to const_iterator by the use of decltype. The lack of support for temporaries is intentional, as the most likely effect is storing iterators to stale objects. If a range is used directly in a for loop then the implementation can handle the range directly without requiring this extra library.]

```
template<typename InputIterator>
range<InputIterator> make_range(InputIterator first, InputIterator last);
```

Returns: `range<InputIterator>{first, last};`

Update **23.1.4 [associative.reqmts]** Table 95 — Associative container requirements (in addition to container)

Expression	Return type	Assertion/note pre-/post-condition	Complexity
a.equal_range(k)	~~pair<iterator, iterator>~~range<iterator>; ~~pair<const_iterator, const_iterator>~~ range<const_iterator> for constant a.	equivalent to ~~make_pair~~ make_range(a.lower_bound(k), a.upper_bound(k)).	logarithmic

Update **23.1.5 [unord.req]**Table 97 — Unordered associative container requirements (in addition to container)

Expression	Return type	Assertion/note pre-/post-condition	Complexity
b.equal_range(k)	~~pair<iterator, iterator>~~range<iterator>; ~~pair<const_iterator, const_iterator>~~ range<const_iterator> for constant a.	Returns a range containing all elements with keys equivalent to k. Returns ~~make_pair~~ make_range(b.end(), b.end()) if no such elements exist.	Average case O(b.count(k)). Worst case O(b.size()).

23.3.1p2 [map]

```
namespace std {
  template <class Key, class T, class Compare = less<Key>,
```

```
                    class Allocator = allocator<pair<const Key, T> > >
  class map {
  public:
    ...
    pair<iterator,iterator>
    range<iterator>
      equal_range(const key_type& x);
    pair<const_iterator,const_iterator>
    range<const_iterator>
      equal_range(const key_type& x) const;
  };
```

23.3.1.4 [map.ops]

```
  pair<iterator,iterator>
  range<iterator>
    equal_range(const key_type& x);
  pair<const_iterator,const_iterator>
  range<const_iterator>
    equal_range(const key_type& x) const;
```

23.3.2 [multimap]

```
namespace std {
  template <class Key, class T, class Compare = less<Key>,
                    class Allocator = allocator<pair<const Key, T> > >
  class multimap {
  public:
    ...
    pair<iterator,iterator>
    range<iterator>
      equal_range(const key_type& x);
    pair<const_iterator,const_iterator>
    range<const_iterator>
      equal_range(const key_type& x) const;
  };
```

23.3.2.3 [multimap.ops]

```
    pair<iterator,iterator>
    range<iterator>
      equal_range(const key_type& x);
    pair<const_iterator,const_iterator>
    range<const_iterator>
      equal_range(const key_type& x) const;
```

23.3.3 Class template set [set]

```
namespace std {
  template <class Key, class Compare = less<Key>,
                    class Allocator = allocator<Key> >
  class set {
  public:
    ...
    pair<iterator,iterator>range<iterator> equal_range(const key_type& x);
    pair<const_iterator,const_iterator>range<const_iterator> equal_range(const
key_type& x) const;
  };
```

23.3.4 [multiset]

```
namespace std {
  template <class Key, class Compare = less<Key>,
                    class Allocator = allocator<Key> >
  class multiset {
```

```
  public:
    ...
    pair<iterator,iterator>range<iterator> equal_range(const key_type& x);
    pair<const_iterator,const_iterator>range<const_iterator> equal_range(const
key_type& x) const;
  };
```

23.4.1 [unord.map]

```
template <class Key,
                 class T,
                 class Hash = hash<Key>,
                 class Pred = std::equal_to<Key>,
                 class Alloc = std::allocator<std::pair<const Key, T> > >
class unordered_map
{
public:
    ...
    pair<iterator,iterator>range<iterator> equal_range(const key_type& x);
    pair<const_iterator,const_iterator>range<const_iterator> equal_range(const
key_type& x) const;
  };
```

23.4.2 [unord.multimap]

```
template <class Key,
                 class T,
                 class Hash = hash<Key>,
                 class Pred = std::equal_to<Key>,
                 class Alloc = std::allocator<std::pair<const Key, T> > >
class unordered_multimap
{
public:
    ...
    pair<iterator,iterator>range<iterator> equal_range(const key_type& x);
    pair<const_iterator,const_iterator>range<const_iterator> equal_range(const
key_type& x) const;
  };
```

23.4.3 [unord.set]

```
template <class Value,
                 class Hash = hash<Value>,
                 class Pred = std::equal_to<Value>,
                 class Alloc = std::allocator<Value> >
class unordered_set
{
public:
    ...
    pair<iterator,iterator>range<iterator> equal_range(const key_type& x);
    pair<const_iterator,const_iterator>range<const_iterator> equal_range(const
key_type& x) const;
  };
```

23.4.4 [unord.multiset]

```
template <class Value,
                 class Hash = hash<Value>,
                 class Pred = std::equal_to<Value>,
                 class Alloc = std::allocator<Value> >
class unordered_multiset
{
public:
    ...
    pair<iterator,iterator>range<iterator> equal_range(const key_type& x);
```

```
    pair<const_iterator,const_iterator>range<const_iterator> equal_range(const
key_type& x) const;
  };
```

25p2 [algorithms]

```
template<class ForwardIterator, class T>
  pair<ForwardIterator, ForwardIterator>
  range<ForwardIterator>
    equal_range(ForwardIterator first, ForwardIterator last, const T& value);
template<class ForwardIterator, class T, class Compare>
  pair<ForwardIterator, ForwardIterator>
  range<ForwardIterator>
    equal_range(ForwardIterator first, ForwardIterator last, const T& value,
Compare comp);
```

25.3.3.3 equal_range [equal.range]

```
template<class ForwardIterator, class T>
  pair<ForwardIterator, ForwardIterator>
  range<ForwardIterator>
    equal_range(ForwardIterator first, ForwardIterator last, const T& value);
template<class ForwardIterator, class T, class Compare>
  pair<ForwardIterator, ForwardIterator>
  range<ForwardIterator>
    equal_range(ForwardIterator first, ForwardIterator last, const T& value,
Compare comp);
```

8 *Requires:* The elements e of `[first,last)` shall be partitioned with respect to the expressions e `< value` and `!(value < e)` or `comp(e, value)` and `!comp(value, e)`. Also, for all elements e of `[first, last)`, `e < value` shall imply `!(value < e)` or `comp(e, value)` shall imply `!comp(value, e)`.

9 *Returns:*

```
      make_pairmake_range(lower_bound(first, last, value),
                          upper_bound(first, last, value))
```

or

```
      make_pairmake_range(lower_bound(first, last, value, comp),
                         upper_bound(first, last, value, comp))
```

10 *Complexity:* At most $2 * \log_2(last - first) + O(1)$ comparisons.

28.9 [re.submatch]

1 Class template sub_match denotes the sequence of characters matched by a particular marked sub-expression.

```
namespace std {
  template <class BidirectionalIterator>
  class sub_match : public std::pairrange<BidirectionalIterator,
BidirectionalIterator> {
  ...
  };
}
```