Document no: N2820=09-0010
Date: 2009-01-30
Project: Programming Language C++
Reply to: Joaquín Mª López Muñoz
joaquin@tid.es

Adding heterogeneous comparison lookup to associative containers

The current draft of the C++0x standard has extended the applicability of binary search algorithms so that they accept keys and comparison operators of types other than those used for sorting the ranges being searched, provided that some compatibility conditions are met[1]. For instance, this extension allows us to write the following:

```
struct name_entry
{
  std::string family_name;
  std::string given_name;
};

bool operator<(const name_entry& x, const name_entry& y)
{
  // lexicographical order on (family_name, given_name)

  if(x.family_name<y.family_name) return true;
  if(y.family_name<x.family_name) return false;
  return x.given_name<y.given_name;
}

struct comp_family_name
{
  bool operator()(const name_entry& x, const std::string& y) const
  {
    return x.family_name<y;
  }
  bool operator()(const std::string& x, const name_entry& y) const
  {
    return x<y.family_name;
  }
};

int main()
{
  std::vector<name_entry> names;
  ... // populate names;
  std::sort(names.begin(),names.end());
  // look for all Smiths
  std::equal_range(
    names.begin(),names.end(),
    std::string("Smith"),comp_family_name());
}
```

Conceptually, the extension consists in replacing the original formulation based on strict weak orderings with one relying on the notion of *sequence partitioning*, as first proposed by David Abrahams[2]. Unfortunately, this extension process has not been carried out for the lookup operations of associative containers, which are still formulated in terms of strict weak orderings and thus do not allow for heterogeneous

[1] "Binary search requirements overly strict", LWG issue 270, C++ Standard Library Defect Report List, http://www.open-std.org/Jtc1/sc22/wg21/docs/lwg-defects.html#270
[2] David Abrahams, "Binary Search with Heterogeneous Comparison", J16-01/0027 = WG21 N1313, 2001.

comparison. So, if in the example above we had used a set rather than a vector we could not do this:

```
int main()
{
  std::set<name_entry> names;
  ... // populate names;
  // this does not compile
  names.equal_range(std::string("Smith"),comp_family_name());
}
```

and would have to resort to

```
int main()
{
  std::set<name_entry> names;
  ... // populate names;
  // look for all Smiths
  std::equal_range(
    names.begin(),names.end(),
    std::string("Smith"),comp_family_name());
}
```

which, since set iterators are bidirectional, has linear complexity, when set lookup operations are logarithmic. We propose to replace the current `equal_range` operations of associative containers

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

with the following ones based on sequence partitioning concepts:

```
template<class T>
  requires HasLess<T, value_type>
        && HasLess<value_type, T>
  pair<iterator, iterator> equal_range(const T& value);
template<class T, CopyConstructible Compare>
  requires Predicate<Compare, T, value_type>
        && Predicate<Compare, value_type, T>
  pair<iterator, iterator> equal_range(const T& value, Compare comp);
template<class T>
  requires HasLess<T, value_type>
        && HasLess<value_type, T>
  pair<const_iterator, const_iterator> equal_range(const T& value) const;
template<class T, CopyConstructible Compare>
  requires Predicate<Compare, T, value_type>
        && Predicate<Compare, value_type, T>
  pair<const_iterator, const_iterator>
    equal_range(const T& value, Compare comp) const;
```

and similarly for the other lookup operations (`find`, `count`, `lower_bound` and `upper_bound`).

Implementation

At least for realizations of associative containers based on red-black trees, implementing the proposed extension is entirely trivial. For instance, starting from a canonical implementation of `lower_bound`

```
iterator lower_bound(const key_type& x)
{

  Node* top = root();
  Node* y = header();
```

```
    while(top){
      if(!comp(top->value, x)){ // comp is the internal comparison object
        y = top;
        top = top->left;
      }
      else top = top->right;
    }

    return iterator(y);
  }
```

we can easily derive the partitioning-based extension:

```
  template<class T, Predicate<auto, value_type, T> Compare>
    requires CopyConstructible<Compare>
    iterator lower_bound(const T& x, Compare comp)
  {

    Node* top = root();
    Node* y = header();

    while(top){
      if(!comp(top->value, x)){ // comp is provided by the user
        y = top;
        top = top->left;
      }
      else top = top->right;
    }

    return iterator(y);
  }
```

Note that the code remains exactly the same, except that we substitute the user-provided `comp` for the internal comparison object used before. This nice property, for which we provide a formal justification in an annex of this paper, holds for the rest of lookup operations as well.

Existing practice

Some of the components of the Boost MultiIndex library[3] provide lookup facilities with heterogeneous comparison in a manner similar to that described here. The author has received some reports pointing to this functionality as reason alone to use Boost.MultiIndex in place of standard associative containers, leaving aside the more prominent multi-indexing capabilities offered by the library.

Proposed resolution

1. Change 23.1.4 [associative.reqmts] paragraph 7 from:

   > […] `k` denotes a value of `X::key_type` and `c` denotes a value of type `X::key_compare`.[…]

   to:

   > […] `k` denotes a value of `X::key_type` and `c` denotes a value of type `X::key_compare`; `kl` is a value such that `a` is partitioned (25.3) with respect to `c(r, kl)`, with `r` the key value of `e` and `e` in `a`; `kcl` is a value

[3] Joaquín Mª López Muñoz, Boost Multi-index Containers Library, http://www.boost.org/libs/multi_index

and `cl` a copy constructible value such that that a is partitioned with respect to `cl(r, kcl)`; `ku` is a value such that a is partitioned with respect to `!c(ku, r)`; `kcu` is a value and `cu` a copy constructible value such that that a is partitioned with respect to `!cu(kcu, r)`; `ke` is a value such that a is partitioned with respect to `c(r, ke)` and `!c(ke, r)`, with `c(r, ke)` implying `!c(ke, r)`; `kce` is a value and `ce` a copy constructible value such that that a is partitioned with respect to `ce(r, kce)` and `!ce(kce, r)`, with `ce(r, kce)` implying `!ce(kce, r)`.[…]

2. Replace the following entries from Table 96 of section 23.1.4 [associative.reqmts]:

Expression	**Return type**	**Assertion/note pre- / post-condition**	**Complexity**
`a.find(k)`	`iterator`; `const_-iterator` for constant `a`.	returns an iterator pointing to an element with the key equivalent to `k`, or `a.end()` if such an element is not found	logarithmic
`a.count(k)`	`size_type`	returns the number of elements with key equivalent to `k`	log(size()) + count(*k*)
`a.lower_-bound(k)`	`iterator`; `const_-iterator` for constant `a`.	returns an iterator pointing to the first element with key not less than `k`, or `a.end()` if such an element is not found.	logarithmic
`a.upper_-bound(k)`	`iterator`; `const_-iterator` for constant `a`.	returns an iterator pointing to the first element with key greater than `k`, or `a.end()` if such an element is not found.	logarithmic
`a.equal_-range(k)`	pair<iterator, iterator>; pair<const_-iterator, const_-iterator> for constant a.	equivalent to `make_-pair(a.lower_bound(k), a.upper_bound(k))`.	logarithmic

with:

Expression	**Return type**	**Assertion/note pre- / post-condition**	**Complexity**
`a.find(ke)`	`iterator`; `const_-iterator` for constant `a`.	returns an iterator pointing to an element with key `r` such that `!c(r, ke) && !c(ke, r)`, or `a.end()` if such an element is not found	logarithmic
`a.find(kce, ce)`	`Iterator`; `const_-iterator` for constant `a`.	returns an iterator pointing to an element with key `r` such that `!ce(r, kce) && !ce(kce, r)`, or `a.end()` if such an element is not found	logarithmic
`a.count(ke)`	`size_type`	returns the number of elements with key `r` such that `!c(r, ke) && !c(ke, r)`	log(size()) + count(*ke*)
`a.count(kce, ce)`	`size_type`	returns the number of elements with key `r` such that `!ce(r, kce) && !ce(kce, r)`	log(size()) + count(*kce*, *ce*)
`a.lower_-bound(kl)`	`iterator`; `const_-`	returns an iterator pointing to	logarithmic

	`iterator` for constant `a`.	the first element with key `r` such that `!c(r, kl)`, or `a.end()` if such an element is not found.	
`a.lower_bound(kcl, cl)`	`iterator`; `const_iterator` for constant `a`.	returns an iterator pointing to the first element with key `r` such that `!cl(r, kcl)`, or `a.end()` if such an element is not found.	logarithmic
`a.upper_bound(ku)`	`iterator`; `const_iterator` for constant `a`.	returns an iterator pointing to the first element with key `r` such that `c(ku, r)`, or `a.end()` if such an element is not found.	logarithmic
`a.upper_bound(kcu, cu)`	`iterator`; `const_iterator` for constant `a`.	returns an iterator pointing to the first element with key `r` such that `cu(kcu, r)`, or `a.end()` if such an element is not found.	logarithmic
`a.equal_range(ke)`	`pair<iterator, iterator>`; `pair<const_iterator, const_iterator>` for constant `a`.	equivalent to `make_pair( a.lower_bound(ke), a.upper_bound(ke))`.	logarithmic
`a.equal_range(kce, ce)`	`pair<iterator, iterator>`; `pair<const_iterator, const_iterator>` for constant `a`.	equivalent to `make_pair( a.lower_bound(kce, ce), a.upper_bound(kce, ce))`.	logarithmic

3. In 23.3.1 [map], 23.3.2 [multimap], 23.3.3 [set] and 23.3.4 [multiset], replace:

```
iterator       find(const key_type& x);
const_iterator find(const key_type& x) const;

size_type count(const key_type& x) const;

iterator       lower_bound(const key_type& x);
const_iterator lower_bound(const key_type& x) const;

iterator       upper_bound(const key_type& x);
const_iterator upper_bound(const key_type& x) const;

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

with:

```
template<class T>
  requires HasLess<T, value_type>
        && HasLess<value_type, T>
  iterator find(const T& value);
template<class T, CopyConstructible Compare>
  requires Predicate<Compare, T, value_type>
        && Predicate<Compare, value_type, T>
  iterator find(const T& value, Compare comp);
template<class T>
  requires HasLess<T, value_type>
        && HasLess<value_type, T>
  const_iterator find(const T& value) const;
template<class T, CopyConstructible Compare>
  requires Predicate<Compare, T, value_type>
        && Predicate<Compare, value_type, T>
  const_iterator find(const T& value, Compare comp) const;
```

```
template<class T>
  requires HasLess<T, value_type>
        && HasLess<value_type, T>
  size_type count(const T& value) const;
template<class T, CopyConstructible Compare>
  requires Predicate<Compare, T, value_type>
        && Predicate<Compare, value_type, T>
  size_type count(const T& value, Compare comp) const;

template<class T>
  requires HasLess<value_type, T>
  iterator lower_bound(const T& value);
template<class T, Predicate<auto, value_type, T> Compare>
  requires CopyConstructible<Compare>
  iterator lower_bound(const T& value, Compare comp);
template<class T>
  requires HasLess<value_type, T>
  const_iterator lower_bound(const T& value) const;
template<class T, Predicate<auto, value_type, T> Compare>
  requires CopyConstructible<Compare>
  const_iterator lower_bound(const T& value, Compare comp) const;

template<class T>
  requires HasLess<T, value_type>
  iterator upper_bound(const T& value);
template<class T, Predicate<auto, T, value_type> Compare>
  requires CopyConstructible<Compare>
  iterator upper_bound(const T& value, Compare comp);
template<class T>
  requires HasLess<T, value_type>
  const_iterator upper_bound(const T& value) const;
template<class T, Predicate<auto, T, value_type> Compare>
  requires CopyConstructible<Compare>
  const_iterator upper_bound(const T& value, Compare comp) const;

template<class T>
  requires HasLess<T, value_type>
        && HasLess<value_type, T>
  pair<iterator, iterator> equal_range(const T& value);
template<class T, CopyConstructible Compare>
  requires Predicate<Compare, T, value_type>
        && Predicate<Compare, value_type, T>
  pair<iterator, iterator> equal_range(const T& value, Compare comp);
template<class T>
  requires HasLess<T, value_type>
        && HasLess<value_type, T>
  pair<const_iterator, const_iterator> equal_range(const T& value) const;
template<class T, CopyConstructible Compare>
  requires Predicate<Compare, T, value_type>
        && Predicate<Compare, value_type, T>
  pair<const_iterator, const_iterator>
    equal_range(const T& value, Compare comp) const;
```

Acknowledgements

Kevin Sopp has kindly reviewed the paper before its submission to the committee.

Annex

As we have seen before, usual lookup algorithms on a sorted range are formally equivalent to the extended algorithms needed to accommodate partitioning-based semantics: it only takes to utilize the user-provided heterogeneous comparison object in place of the internal comparison predicate used to sort the range. To prove this fact we need the following

Proposition. Let T be an arbitrary set with an associated strict weak order $<_T$ and L, U subsets of T such that

$$
\begin{aligned}
&L \cap U = \emptyset, \\
&\forall a,b \in T \quad a \in L,\ b <_T a \rightarrow b \in L, \\
&\forall a,b \in T \quad a \in U,\ a <_T b \rightarrow b \in U.
\end{aligned}
$$

We create the set Q by augmenting T with an additional element x, and define the binary relationship $<_Q$ on Q as follows:

$$
\begin{aligned}
&a <_Q b := a <_T b, \\
&a <_Q x := a \in L, \\
&x <_Q a := a \in U, \\
&x <_Q x := \text{false},
\end{aligned}
$$

for all $a,b \in T$. Under these conditions, $<_Q$ is a strict weak order on Q. (Proof trivial.)

Returning to our original scenario, let `[first,last)` be a range of values of a type `T` sorted by some strict weak ordering, `x` a value of a type other than `T` and `comp` a heterogeneous comparison object such that `[first,last)` is partitioned (in the C++0x sense) both with respect to `comp(·,x)` and `!comp(x,·)`, with `comp(e,x)` implying `!comp(x,e)`. Now, we can regard `[first,last)` as a range of elements of the set `Q` = {`e` in `[first,last)`} ∪ {`x`} which is sorted with respect to an extended strict weak order defined on `Q` in the manner shown in the proposition above (with L = {`e` in `[first,last)` : `comp(e,x)`}, U = {`e` in `[first,last)` : `comp(x,e)`}). So, any lookup algorithm operating on `[first,last)` with input values of type `T` is formally a valid algorithm when input values are taken from `Q` and the proper strict weak order, with which `comp` is compatible, is observed.