Simplifying Algorithms Using Iterator Traits

Angelika Langer (email: langer@roguewave.com)

Abstract

Now that we have iterator traits we can simplify the interfaces of several algorithms. These modifications make the STL easier to understand.

1. Motivation

The find algorithm has two template parameters:

```
template <class InputIterator, class T>
InputIterator find(InputIterator first, InputIterator last,
    const T& value);
```

It is hard to explain why this function template has two parameters, especially as the second one is likely to be the iterator's value type in almost all cases.

2. Proposal

I suggest to simplify the find algorithm and eliminate the second template parameter:

```
template <class InputIterator>
InputIterator find(InputIterator first, InputIterator last,
    const typename iterator_traits<InputIterator>::value_type& value);
```

3. Discussion

3.1 Fewer Template Instances

One advantage is that we enable conversions and thus reduce the number of template instances. This will help to reduce the frequently obeserved code bloat:

```
list<string>.friends;
list<string>::iterator iter1, iter2;
iter1 = find(friends.begin(), friends.end(), "Tom");
string tom("Tom");
iter2 = find(friends.begin(), friends.end(), tom);
```

With the original find algorithm, *two* instances of the find algorithm would be created,

- one for `string`, which is `list<string>`'s value type, and
- one for `char*`.

With the new find algorithm only *one* instance, the one for `string`, would be instantiated. The usual function overloading would then allow to call the find algorithm with a `char*` argument, because a conversion from `char*` to `list<string>::value_type` is defined.

3.2 Better Error Messages

Another benefit is that we make life easier for users who make mistakes like this:

```
map<string,long>.phoneDir;
map<string,long>::iterator tom;
tom = find(phoneDir.begin(), phoneDir.end(), "Tom");[1]
```

Virtually every beginner makes this mistake. Here is a typical error message you receive in such a case:

```
"/home/scratch/langer/stdlib/include/algorithm", line 81: error: no operator
          "!=" matches these operands
            operand types are: std::map<std::string, long,
                      std::less<std::string>, std::allocator>::value_type !=
                      char *const
      while (first != last && *first != value)
                                       ^
          detected during instantiation of
"std::rb_tree<std::map<std::string,
                   long, std::less<std::string>, std::allocator>::key_type,
                   std::map<std::string, long, std::less<std::string>,
                   std::allocator>::value_type,
                   std::select1st<std::map<std::string, long,
                   std::less<std::string>, std::allocator>::value_type,
                   std::map<std::string, long, std::less<std::string>,
                   std::allocator>::key_type>, std::map<std::string, long,
                   std::less<std::string>, std::allocator>::key_compare,
                   std::map<std::string, long, std::less<std::string>,
                   std::allocator>::allocator_type>::iterator
                   std::find(std::rb_tree<std::map<std::string, long,
                   std::less<std::string>, std::allocator>::key_type,
                   std::map<std::string, long, std::less<std::string>,
                   std::allocator>::value_type,
                   std::select1st<std::map<std::string, long,
                   std::less<std::string>, std::allocator>::value_type,
                   std::map<std::string, long, std::less<std::string>,
                   std::allocator>::key_type>, std::map<std::string, long,
                   std::less<std::string>, std::allocator>::key_compare,
                   std::map<std::string, long, std::less<std::string>,
                   std::allocator>::allocator_type>::iterator,
                   std::rb_tree<std::map<std::string, long,
                   std::less<std::string>, std::allocator>::key_type,
                   std::map<std::string, long, std::less<std::string>,
                   std::allocator>::value_type,
                   std::select1st<std::map<std::string, long,
                   std::less<std::string>, std::allocator>::value_type,
                   std::map<std::string, long, std::less<std::string>,
```

[1] The value type of a `map<string,long>` is a `pair<const string, long>` , and never a `string` as was assumed. Here the find member function of class map would do the trick: `tom = phoneDir.find("Tom")`

```
                    std::allocator>::key_type>, std::map<std::string, long,
                    std::less<std::string>, std::allocator>::key_compare,
                    std::map<std::string, long, std::less<std::string>,
                    std::allocator>::allocator_type>::iterator, char *const
&)"
```

I haven't seen any programmer so far who immediately recognized from this message indicating a problem with the innards of a library header file what mistake (s)he had made. With the simplified interface the error message would be more directly related to the source of the error, because the value type would be already determined by the iterator type. I expect the error message to be something like this:

```
"find.C", line 62: error: no instance of function template "find" matches
          the argument list
            argument types are: (std::rb_tree<std::map<std::string, long,
                      std::less<std::string>, std::allocator>::key_type,
                      std::map<std::string, long, std::less<std::string>,
                      std::allocator>::value_type,
                      std::select1st<std::map<std::string, long,
                      std::less<std::string>, std::allocator>::value_type,
                      std::map<std::string, long, std::less<std::string>,
                      std::allocator>::key_type>, std::map<std::string, long,
                      std::less<std::string>, std::allocator>::key_compare,
                      std::map<std::string, long, std::less<std::string>,
                      std::allocator>::allocator_type>::iterator,
                      std::rb_tree<std::map<std::string, long,
                      std::less<std::string>, std::allocator>::key_type,
                      std::map<std::string, long, std::less<std::string>,
                      std::allocator>::value_type,
                      std::select1st<std::map<std::string, long,
                      std::less<std::string>, std::allocator>::value_type,
                      std::map<std::string, long, std::less<std::string>,
                      std::allocator>::key_type>, std::map<std::string, long,
                      std::less<std::string>, std::allocator>::key_compare,
                      std::map<std::string, long, std::less<std::string>,
                      std::allocator>::allocator_type>::iterator, char [4])
    iter1 = find(phoneDir.begin(), phoneDir.end(), "Tom");
```

3.3 Simplicity

The suggested simplification is to remove a template parameter from the find algorithm's interface. This does not make the find algorithm easier to use, because the value type is deduced from the function arguments anyway. However, the simplified find algorithm is easier to explain and understand. There is no compelling reason I know of for having the value type as a template argument. It only confuses people who want to learn the STL.

Some people expressed their concern because this proposal removes some functionality: at present, if you say `find(first, last, value)`, you aren't requiring that `*first` and `value` actually be the same type but only that it is possible to compare `*first` and `value` for equality. Requiring that `*first` and `value` actually be the same type is much stricter. However, I couldn't figure out a real example where the "old" flexibility is really needed.

Consider the following example:

```
long A[100];
find(A, A+100, 7);
```

With the current find algorithm, the compiler deduces from the first two arguments that the iterator type should be a pointer, `long*`, and from the last one that value type should be `int`. It instantiates the according find function. Then the built-in conversion from `int` to `long` will apply.

With the proposed changes this would still be possible:

```
template<class InputIterator>
InputIterator find(InputIterator first, InputIterator last,
   const iterator_traits<InputIterator>::value_type& val);
```

The type of the third function argument is fixed after the iterator type will have been deduced from the first parameters. (What exactly the rules for this kind of type deduction are, is an open issue on John Spicer's list. See section 4 for more details.) Once the type of the third parameter will be determined (type `long` in the example) the usual function overloading will apply, so that the conversion from `int` to `long` will happen, as is expected.

4. Caveat

The proposed changes depend on the availability of partial specialization. This is because I made use of the iterator traits, which require partial specialization. As I had no access to a compiler that supports this language feature, I had no chance to test the proposed changes.

Additionally, the proposed changes depend on a resolution of issue 3.30 in version 15 of John Spicer's template paper in the pre Stockholm mailing. The open issue concerns deduction of one template parameter from another template parameter. Consider for instance:

```
template <class Iterator>
void algo(Iterator first, Iterator last,
          const iterator_traits<Iterator>::value_type& val);
```

- Does the deduction of type `Iterator` determine the related type `iterator_traits <Iterator>:: value_type`?
- Moreover, will function overloading apply to the related type? I.e. if there is a type X that is convertible to type `iterator_traits <Iterator>:: value_type` , can I call `algo(iter,x)` and expect that the conversion from type X to type `iterator_traits <Iterator>:: value_type` will happen?

Both issues need to be resolved in the way suggested above.

For those who are reluctant to consider a modification that depends on a language feature that is not yet available I would like to point out that there is an easy work-around: For the time being, library builders can be retaining the old interfaces.

5. Working Paper Changes

Naturally, not only the find algorithm can be simplified as proposed above. Here are all changes to clause 25 "Algorithms library".

In clause 25 [lib.algorithms] make the following changes:

```
namespace std {
  // subclause 25.1, non-modifying sequence operations:

template<class InputIterator>
    InputIterator find(InputIterator first, InputIterator last,
      const typename iterator_traits<InputIterator>::value_type& value);

template<class InputIterator>
    typename iterator_traits<InputIterator>::distance_type
      count(InputIterator first, InputIterator last,
      const typename iterator_traits<InputIterator>::value_type& value);

template<class InputIterator, class Predicate>
    typename iterator_traits<InputIterator>::distance_type
      count_if(InputIterator first, InputIterator last, Predicate pred);

template<class ForwardIterator >
    ForwardIterator  search_n(ForwardIterator first, ForwardIterator last,
      typename iterator_traits<ForwardIterator>::distance_type count,
      const typename iterator_traits<ForwardIterator>::value_type& value);

template<class ForwardIterator, class BinaryPredicate>
    ForwardIterator1 search_n(ForwardIterator first, ForwardIterator last,
      typename iterator_traits<ForwardIterator>::distance_type count,
      const typename iterator_traits<ForwardIterator>::value_type& value,
      BinaryPredicate pred);


  // subclause 25.2, modifying sequence operations:
template<class ForwardIterator>
    void replace(ForwardIterator first, ForwardIterator last,
       const typename iterator_traits<ForwardIterator>::value_type& old_value,
       const typename iterator_traits<ForwardIterator>::value_type& new_value);

template<class ForwardIterator, class Predicate>
    void replace_if(ForwardIterator first, ForwardIterator last,
      Predicate pred,
      const typename iterator_traits<Forwardterator>::value_type& new_value);

template<class InputIterator, class OutputIterator>
    OutputIterator replace_copy(InputIterator first, InputIterator last,
      OutputIterator result,
      const typename iterator_traits<InputIterator>::value_type& old_value,
      const typename iterator_traits<InputIterator>::value_type& new_value);

template<class Iterator, class OutputIterator, class Predicate>
    OutputIterator replace_copy_if(Iterator first, Iterator last,
      OutputIterator result, Predicate pred,
      const typename iterator_traits<InputIterator>::value_type& new_value);
```

```
template<class ForwardIterator, class T>
    void fill(ForwardIterator first, ForwardIterator last,
      const typename iterator_traits<ForwardIterator>::value_type& value);

template<class OutputIterator, class Generator>[2]
    void generate_n(OutputIterator first,
      typename iterator_traits<OutputIterator>::distance_type n,
      Generator gen);

template<class OutputIterator>[3]
    void fill_n(OutputIterator first,
      typename iterator_traits<OutputIterator>::distance_type n,
      const typename iterator_traits<OutputIterator>::value_type& value);

template<class ForwardIterator>
    ForwardIterator remove(ForwardIterator first, ForwardIterator last,
      const typename iterator_traits<ForwardIterator>::value_type& value);

template<class InputIterator, class OutputIterator, class T>
    OutputIterator remove_copy(InputIterator first, InputIterator last,
      OutputIterator result,
      const typename iterator_traits<InputIterator>::value_type& value);


  // 25.3.3, binary search:
template<class ForwardIterator>
    ForwardIterator lower_bound(ForwardIterator first, ForwardIterator last,
      const typename iterator_traits<ForwardIterator>::value_type& value);

template<class ForwardIterator, class Compare>
    ForwardIterator lower_bound(ForwardIterator first, ForwardIterator last,
      const typename iterator_traits<ForwardIterator>::value_type& value,
      Compare comp);

template<class ForwardIterator>
    ForwardIterator upper_bound(ForwardIterator first, ForwardIterator last,
      const typename iterator_traits<ForwardIterator>::value_type& value);

template<class ForwardIterator, class Compare>
    ForwardIterator upper_bound(ForwardIterator first, ForwardIterator last,
      const typename iterator_traits<ForwardIterator>::value_type& value,
      Compare comp);

template<class ForwardIterator>
    pair<ForwardIterator, ForwardIterator>
      equal_range(ForwardIterator first, ForwardIterator last,
      const typename iterator_traits<ForwardIterator>::value_type& value);

template<class ForwardIterator, class Compare>
    pair<ForwardIterator, ForwardIterator>
      equal_range(ForwardIterator first, ForwardIterator last,
      const typename iterator_traits<ForwardIterator>::value_type& value,
      Compare comp);

template<class ForwardIterator>
    bool binary_search(ForwardIterator first, ForwardIterator last,
      const typename iterator_traits<ForwardIterator>::value_type& value);

template<class ForwardIterator, class Compare>
    bool binary_search(ForwardIterator first, ForwardIterator last,
      const typename iterator_traits<ForwardIterator>::value_type& value,
      Compare comp);
```

---

[2] This modification depends on the fact that output iterators have a meaningful distance type, which is proposed in X3J16/ 96-xxxx WG21/N0xxx.

[3] This modification depends on the fact that output iterators have a meaningful distance type, which is proposed in X3J16/ 96-xxxx WG21/N0xxx.

Change clause 25.1.2 [lib.alg.find]:

```
  template<class InputIterator>
    InputIterator find(InputIterator first, InputIterator last,
      const typename iterator_traits<InputIterator>::value_type& value);
```

Requires: Type iterator_traits<InputIterator>::value_type is EqualityComparable (_lib.equalitycomparable_).

Change clause 5.1.6 [lib.alg.count]:

```
template<class InputIterator>
      typename iterator_traits<InputIterator>::distance_type
         count(InputIterator first, InputIterator last,
         const typename iterator_traits<InputIterator>::value_type& value);

template<class InputIterator, class Predicate>
      typename iterator_traits<InputIterator>::distance_type
        count_if(InputIterator first, InputIterator last, Predicate pred);
```

Requires: Type iterator_trait<InputIterator>::value_type is EqualityComparable (_lib.equalitycomparable_) .

Change clause 25.1. [lib.alg.search]:

```
  template<class ForwardIterator>
    ForwardIterator
      search_n(ForwardIterator first, ForwardIterator last,
      typename iterator_traits<ForwardIterator>::distance_type count,
      const typename iterator_traits<ForwardIterator>::value_type& value);

  template<class ForwardIterator, class BinaryPredicate>
    ForwardIterator
      search_n(ForwardIterator first, ForwardIterator last,
      typename iterator_traits<ForwardIterator>::distance_type count,
      const typename iterator_traits<ForwardIterator>::value_type& value,
      BinaryPredicate pred);
```

Change clause 25.2.4 [lib.alg.replace]:

```
template<class ForwardIterator>
    void replace(ForwardIterator first, ForwardIterator last,
      const typename iterator_traits<ForwardIterator>::value_type& old_value,
      const typename iterator_traits<ForwardIterator>::value_type& new_value);

  template<class ForwardIterator, class Predicate>
    void replace_if(ForwardIterator first, ForwardIterator last,
      Predicate pred,
      const typename iterator_traits<ForwardIterator>::value_type& new_value);
```

Requires: Type iterator_traits<ForwardIterator>::value_type is Assignable (23.1) (and, for replace(), EqualityComparable (_lib.equalitycomparable_).

```
  template<class InputIterator, class OutputIterator>
    OutputIterator
```

```
    replace_copy(InputIterator first, InputIterator last,
    OutputIterator result,
    const typename iterator_traits<InputIterator>::value_type& old_value,
    const typename iterator_traits<InputIterator>::value_type& new_value);

template<class Iterator, class OutputIterator, class Predicate>
  OutputIterator
    replace_copy_if(Iterator first, Iterator last,
    OutputIterator result,
    Predicate pred,
    const typename iterator_traits<InputIterator>::value_type& new_value);
```

Requires: Type iterator_traits<InputIterator>::value_type is Assignable (23.1) (and, for replace_copy(), EqualityComparable (_lib.equalitycomparable_). The ranges [first, last) and [result, result+(last-first)) shall not overlap.

Change clause 25.2. [lib.alg.fill]:

```
template<class ForwardIterator>
  void fill(ForwardIterator first, ForwardIterator last,
  const typename iterator_traits<ForwardIterator>::value_type& value);
```

Requires: Type iterator_traits<ForwardIterator>::value_type is Assignable (23.1), Size is convertible to an integral type (_conv.integral_, _class.conv_).

```
template<class OutputIterator>[4]
    void fill_n(OutputIterator first,
    typename iterator_traits<OutputIterator>::distance_type n,
    const typename iterator_traits<OutputIterator>::value_type& value);
```

Change clause 25.2.6 [lib.alg.generate]:

```
template<class OutputIterator, class Generator>[5]
  void generate_n(OutputIterator first,
    typename iterator_traits<OutputIterator>::distance_type n,
    Generator gen);
```

Change clause 25.2.7 [lib.alg.remove]:

```
template<class ForwardIterator>
  ForwardIterator remove(ForwardIterator first, ForwardIterator last,
    const typename iterator_traits<ForwardIterator>::value_type& value);
```

Requires: Type iterator_traits<ForwardIterator>::value_type is EqualityComparable (_lib.equalitycomparable_).

```
template<class InputIterator, class OutputIterator>
  OutputIterator
```

[4] This modification depends on the fact that output iterators have a meaningful distance type, which is proposed in X3J16/ 96-xxxx WG21/N0xxx.

[5] This modification depends on the fact that output iterators have a meaningful distance type, which is proposed in X3J16/ 96-xxxx WG21/N0xxx.

```
    remove_copy(InputIterator first, InputIterator last,
    OutputIterator result,
    const typename iterator_traits<InputIterator>::value_type& value);
```

Requires: Type iterator_traits<InputIterator>::value_type is EqualityComparable (_lib.equalitycomparable_). The ranges [first, last) and [result, result+(last-first)) shall not overlap.

Change clause 25.3.3. [lib.lower.bound]:

```
template<class ForwardIterator>
  ForwardIterator
    lower_bound(ForwardIterator first, ForwardIterator last,
    const typename iterator_traits<ForwardIterator>::value_type& value);

template<class ForwardIterator, class T, class Compare>
  ForwardIterator
    lower_bound(ForwardIterator first, ForwardIterator last,
    const typename iterator_traits<ForwardIterator>::value_type& value,
    Compare comp);
```

Requires: Type iterator_traits<ForwardIterator>::value_type is LessThanComparable (_lib.lessthancomparable_).

Change clause 25.3.3. [lib.upper.bound]:

```
template<class ForwardIterator>
  ForwardIterator
    upper_bound(ForwardIterator first, ForwardIterator last,
    const typename iterator_traits<ForwardIterator>::value_type & value);

template<class ForwardIterator, class Compare>
  ForwardIterator
    upper_bound(ForwardIterator first, ForwardIterator last,
    const typename iterator_traits<ForwardIterator>::value_type& value,
    Compare comp);
```

Requires: Type iterator_traits<ForwardIterator>::value_type is LessThanComparable (_lib.lessthancomparable_).

Change clause 25.3.3.3 [lib.equal.range]:

```
template<class ForwardIterator>
  pair<ForwardIterator, ForwardIterator>
    equal_range(ForwardIterator first, ForwardIterator last,
    const typename iterator_traits<ForwardIterator>::value_type& value);

template<class ForwardIterator, class Compare>
  pair<ForwardIterator, ForwardIterator>
    equal_range(ForwardIterator first, ForwardIterator last,
    const typename iterator_traits<ForwardIterator>::value_type& value,
    Compare comp);
```

Requires: Type iterator_traits<ForwardIterator>::value_type is LessThanComparable (_lib.lessthancomparable_).

Change clause 25.3.3.4 [lib.binary.search]:

```
template<class ForwardIterator>
  bool binary_search(ForwardIterator first, ForwardIterator last,
    const typename iterator_traits<ForwardIterator>::value_type& value);

template<class ForwardIterator, class Compare>
  bool binary_search(ForwardIterator first, ForwardIterator last,
    const typename iterator_traits<ForwardIterator>::value_type& value,
    Compare comp);
```

Requires: Type iterator_traits<ForwardIterator>::value_type is LessThanComparable (_lib.lessthancomparable_).

In clause 24 [lib.iterators] make changes to `advance`:

Change the description of `advance` in clause 24.1.6 [lib.iterator.tags]:

```
template <class InputIterator>
    void advance(InputIterator& i,
                 typename iterator_traits<InputIterator>::distance_type n);
```
[6]

Change the description of `advance` in clause 24.2.6 [lib.iterator.operations]:

```
template <class InputIterator>
  void advance(InputIterator& i,
               typename iterator_traits<InputIterator>::distance_type n);
```

[6] I eliminated the second template parameter, the distance type, and chose to deduce it from the iterator type. Should this be thought too restrictive, one can retain the second template parameter. Here's the original interface:

```
template <class InputIterator, class Distance>
  void advance(InputIterator& i, Distance n);
```