Bring Back the Obvious Definition of count()

Bjarne Stroustrup
AT&T Bell Laboratories
Murray Hill, New Jersey 07974

Alex Stepanov, Matt Austern
Silicon Graphics Inc.

ABSTRACT

The current definition of the count() algorithm is anomalous in the library, and unnecessary hard to teach and use. I propose to replace it with a version based on the original STL definition of count(). The solution to the count() problem introduces the notion of iterator_traits which allows a significant simplification of the internals of STL.

1 The Problem

The count() algorithm is arguably the easiest algorithm to introduce. Even for_each() is more complicated because you have to explain how the function is called for an element.

Relying on memory and pure logic, I wrote:

```c++
void f(list<int>& lc, char cs[], int sz)
{
    int i1 = count(lc.begin(),lc.end(),7);
    int i2 = count(cs,cs+sz,'z');
}
```

Like me, experts will recognize the mistake from when they did it last time. Please realize, though, that this did work in the original STL. We now have to write:

```c++
void f(list<int>& lc, char cs[], int sz)
{
    int i1 = 0; // do remember to initialize, or else!
    int i2 = 0;
    count(lc.begin(),lc.end(),7,i1);
    count(cs,cs+sz,'z',i2);
}
```

Yuck! This eliminates count() as a candidate for introducing algorithms to novices. The calling interface for count() is simply different (and inferior) from all other standard algorithms. Unless we do something about it, we will have to explain this special case forever and apologize for it each time.

The problem is that count() cannot deduce its return type from its iterator type. The reason it cannot do that is that the iterator can be either a class object or a pointer. At the time we generalized the STL to use allocators, this problem was insurmountable. What I propose is to restore the original intuitive interface to count().

2 A Solution

Here is a minimal solution based on partial specialization:
template <class Iterator> struct itrait {
  typedef Iterator::distance_type dist;
};

template <class T> struct itrait<T*> {
  typedef ptrdiff_t dist;
};

template <class In, class T> itrait<In>::dist count(In first, In last, T x) {
  itrait<In>::dist n = 0;
  while (first != last)
    if (*first++ == x)
      ++n;
  return n;
}

Given this definition of count(), my example works as originally written (or it would have, had my compiler supported partial specialization). This definition brings count() into line with every other standard algorithm.

3 Alternatives

I see three alternatives:

[2] Add the outlined version of count() to the library, leaving the current version for compatibility.
[3] Replace the current count() with the version outlined here.

My preference is [3]. The pure addition, [2], is ugly, but manageable; the only purpose for keeping the old version is to keep old code working and old teaching material valid. Because the old versions have 4 arguments and the new 3, there will be no overloading problems.

Status quo, [1], has no defense except that it is status quo. In case of [1], I and probably others will recommend use of the version of count() defined above, but others will prefer the standard version because it is standard and live with the resulting bugs. In addition, people will invent arguments why the standard version is best ("otherwise the standard committee wouldn’t have chosen it") to add to the confusion.

4 Iterator Traits

The solution above is the minimal one that makes count() work as originally intended. However, the notion of an iterator trait is obviously general, and anyone defining one would provide the usual set of type-defs rather than just distance_type:

template <class Iterator> struct iterator_trait {
  typedef Iterator::distance_type distance_type;
  typedef Iterator::value_type value_type;
};

template <class T> struct iterator_trait<T*> {
  typedef ptrdiff_t distance_type;
  typedef T value_type;
};

Should iterator_trait be documented as part of the library or treated as an implementation detail known only to implementors and varying between implementations? I propose iterator_trait as part of the library because something like that is the only way count can be done right. Given that iterator_trait must exist in every implementation and given that it is the basis for treating user-defined and pointer types equivalently in generic programs, it ought to be the same in every implementation.
5 Iterator Simplification using Iterator Traits

The notion of iterator_traits allows a great simplification in the current mechanism for inferring types from iterators. In particular, §24 can be shortened by a few pages!

The current iterator category mechanism gains type information through overload resolution. Given suitable definitions of iterator_traits and iterators this deduction can be done as partial specialization and the distance, value, and category types can be made available as types rather than as values. This is clearly a far more useful form for type information.

To make category information available for use in iterator traits, the basic iterators (§24.2.2) should present their category as a typedef. For example:

```cpp
template<class T, class Distance=ptrdiff_t> struct input_iterator {
    typedef T value_type;
    typedef Distance distance_type;
    typedef input_iterator_tag iterator_category;
};

template<class Iterator> struct iterator_traits {
    typedef Iterator::distance_type distance_type;
    typedef Iterator::value_type value_type;
    typedef Iterator::iterator_category iterator_category;
};
```

Given these definitions, iterator_traits provides all of the information currently provided by iterator_category(), value_type(), and distance_type(); these functions are thus redundant, and can be eliminated.

Since iterator_traits<Iterator>::value_type is a typedef, we can define value_type to be the return type of Iterator::operator* rather than a pointer to that type. At present, value_type(Iterator) returns a pointer. Since it is possible to declare variables of type iterator_traits<Iterator>::value_type directly, it is much rarer for auxiliary functions to be necessary. In particular, auxiliary functions are needed only when discriminating between different iterator categories.

A further simplification is now possible: since input_iterator, output_iterator, forward_iterator, bidirectional_iterator, and random_access_iterator are identical except for the iterator_category typedefs, are eliminated in favor of a single template:

```cpp
template<class Category, class T, class Distance=ptrdiff_t> struct iterator {
    typedef T value_type;
    typedef Distance distance_type;
    typedef Category iterator_category;
};
```

Thus, iterator_traits provide a simpler, smaller, and more versatile mechanism for dealing with iterator categories.

6 Working Paper Changes

Replace §25.1.6 [lib.alg.count] by

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

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

Requires: Type T is EqualityComparable (lib.equalitycomparable).

Effects: Returns the number of iterators i in the range [first, last) for which the following corresponding conditions hold: *i==value, pred(*i)==true.

Complexity: Exactly last-first applications of the corresponding predicate.
In §17.3.1.1 [lib.contents], remove bidirectional_iterator, forward_iterator, input_iterator, and random_access_iterator from Table 4, and add iterator and iterator_traits. Remove distance_type, iterator_category, and value_type from Table 6. Remove output_iterator from Table 8. Remove iterator_category from Table 10.

In §20.4 [lib.memory] change the comment “For output_iterator” to read “For iterator”. In §20.4.2 [lib.storage.iterator] change the declaration of raw_storage_iterator so that it is derived from iterator<output_iterator_tag,void,void>.

This proposal involves many changes to §24. However, the vast majority are deletions as opposed to additions. To ease the job of the editor, we supply the complete revised text for §24.1.6 rather than listing the minor changes individually; in this case also, the replacement text is noticeably shorter than the current text.

Replace §24.1.6 with:

To implement algorithms only in terms of iterators, it is often necessary to infer both the value type and the distance type from the iterator. To enable this task it is required that for an iterator of type Iterator the types iterator_traits<Iterator>::distance_type, iterator_traits<Iterator>::value_type, and iterator_traits<Iterator>::iterator_category be defined as the iterator’s distance type, value type, and iterator category. In the case of an output iterator, iterator_traits<Iterator>::distance_type and iterator_traits<Iterator>::value_type are defined as void.

[Example: to implement a generic reverse function, a C++ program can do the following:

```cpp
template<class BidirectionalIterator>
void reverse(BidirectionalIterator first, BidirectionalIterator last) {
    iterator_traits<BidirectionalIterator>::distance_type n =
        distance(first, last);
    --n;
    while(n > 0) {
        iterator_traits<BidirectionalIterator>::value_type tmp = *first;
        *first++ = *--last;
        *last = tmp;
        n -= 2;
    }
}
```

--end example]

The template iterator_traits<Iterator> is defined as

```cpp
template<class Iterator> struct iterator_traits {
    typedef Iterator::distance_type distance_type;
    typedef Iterator::value_type value_type;
    typedef Iterator::iterator_category iterator_category;
};
```

It is specialized for pointers as

```cpp
template<class T> struct iterator_traits<T*> {
    typedef ptrdiff_t distance_type;
    typedef T value_type;
    typedef random_access_iterator_tag iterator_category;
};
```

[Note: If there is an additional pointer type __far such that the difference of two __far pointers is of type long, an implementation may define
template<class T> struct iterator_trait<T __far*> { 
    typedef long distance_type;
    typedef T value_type;
    typedef random_access_iterator_tag iterator_category;
};

--end note

It is often desirable for a template function to find out what is the most specific category of its iterator argument, so that the function can select the most efficient algorithm at compile time. To facilitate this, the library introduces category tag classes which are used as compile time tags for algorithm selection. They are: input_iterator_tag, output_iterator_tag, forward_iterator_tag, bidirectional_iterator_tag, and random_access_iterator_tag. For every iterator of type Iterator, iterator_trait<Iterator>::iterator_category must be defined to be the most specific category tag that describes the iterator’s behavior.

[Example: For a program-defined iterator BinaryTreeIterator, it could be included into the bidirectional iterator category by specializing the iterator_trait template:

    template<class T> struct iterator_trait<BinaryTreeIterator<T> > { 
        typedef ptrdiff_t distance_type;
        typedef T value_type;
        typedef bidirectional_iterator_tag iterator_category;
    };

    Typically, however, it would be easier to derive BinaryTreeIterator<T> from iterator<bidirectional_iterator_tag,T,ptrdiff_t>.

    --end example]

[Example: If a template function evolve() is well defined for bidirectional iterators, but can be implemented more efficiently for random access iterators, then the implementation is as follows:

    template <class BidirectionalIterator>
    inline void evolve(BidirectionalIterator first, BidirectionalIterator last) 
    { 
        evolve(first, last,  
                iterator_trait<BidirectionalIterator>::iterator_category()); 
    }

    template <class BidirectionalIterator>
    void evolve(BidirectionalIterator first, BidirectionalIterator last,  
                bidirectional_iterator_tag) {  
        // ... more generic, but less efficient algorithm 
    }

    template <class RandomAccessIterator>
    void evolve(RandomAccessIterator first, RandomAccessIterator last,  
                random_access_iterator_tag) { 
        // ... more efficient, but less generic algorithm 
    }

    --end example]

[Example: If a C++ program wants to define a bidirectional iterator for some data structure containing double and such that it works on a large memory model of the implementation, it can do so with:

    class MyIterator : public iterator<bidirectional_iterator_tag, double, long> { 
        // code implementing ++, etc. 
    };

    Then there is no need to specialize the iterator_trait template. --end example]
#include <cstddef> // for ptrdiff_t
#include <iosfwd>   // for istream, ostream
#include <ios>      // for ios_traits
#include <streambuf> // for streambuf

namespace std {

// subclause _lib.library.primitives_, primitives:
struct input_iterator_tag {};
struct output_iterator_tag {};
struct forward_iterator_tag {};
struct bidirectional_iterator_tag {};
struct random_access_iterator_tag {};

template<class Category, class T, class Distance = ptrdiff_t> struct iterator;

template<class Iterator> struct iterator_trait;
template<class T> struct iterator_trait<T*>;

// subclause _lib.iterator.operations_, iterator operations:
template <class InputIterator, class Distance>
void advance(InputIterator& i, Distance n);
template <class InputIterator>
iterator_trait<InputIterator>::iterator_trait
distance(InputIterator first, InputIterator last);

// subclause _lib.predef.iterators_, predefined iterators:
template <class BidirectionalIterator, class T, class Reference = T&, class Pointer = T*, class Distance = ptrdiff_t>
class reverse_bidirectional_iterator :
   public iterator<bidirectional_iterator_tag,T,Distance>;

template <class BidirectionalIterator, class T, class Reference, class Pointer, class Distance>
bool operator==(
   const reverse_bidirectional_iterator
   <BidirectionalIterator,T,Reference,Pointer,Distance>& x,
   const reverse_bidirectional_iterator
   <BidirectionalIterator,T,Reference,Pointer,Distance>& y);

template <class RandomAccessIterator, class T, class Reference = T&, class Pointer = T*, class Distance = ptrdiff_t>
class reverse_iterator : public
iterator<random_access_iterator_tag,T,Distance>;

template <class RandomAccessIterator, class T, class Reference, class Pointer, class Distance>
bool operator==(
   const reverse_iterator
   <RandomAccessIterator,T,Reference,Pointer,Distance>& x,
   const reverse_iterator
   <RandomAccessIterator,T,Reference,Pointer,Distance>& y);
template <class RandomAccessIterator, class T, class Reference, class Pointer, class Distance>
bool operator<(const reverse_iterator<RandomAccessIterator,T,Reference,Pointer,Distance>& x, const reverse_iterator<RandomAccessIterator,T,Reference,Pointer,Distance>& y);

template <class RandomAccessIterator, class T, class Reference, class Pointer, class Distance>
Distance operator-(const reverse_iterator<RandomAccessIterator,T,Reference,Pointer,Distance>& x, const reverse_iterator<RandomAccessIterator,T,Reference,Pointer,Distance>& y);

template <class RandomAccessIterator, class T, class Reference, class Pointer, class Distance>
reverse_iterator<RandomAccessIterator,T,Reference,Pointer,Distance> operator+(Distance n, const reverse_iterator<RandomAccessIterator,T,Reference,Pointer,Distance>& x);

// subclauses _lib.stream.iterators_, stream iterators:
template <class T, class Distance = ptrdiff_t> class istream_iterator;
template <class T, class Distance>
bool operator==(const istream_iterator<T,Distance>& x, const istream_iterator<T,Distance>& y);

template <class T> class ostream_iterator;
template <class charT, class traits = ios_traits<charT>, class Distance = ptrdiff_t>
class istreambuf_iterator;
template <class charT, class traits = ios_traits<charT> >
bool operator==(istreambuf_iterator<charT,traits>& a, istreambuf_iterator<charT,traits>& b);

template <class charT, class traits = ios_traits<charT> >
bool operator!=(istreambuf_iterator<charT,traits>& a, istreambuf_iterator<charT,traits>& b);

}
template<class Category, class T, class Distance> struct iterator {
    typedef T value_type;
    typedef Distance distance_type;
    typedef Category iterator_category;
};

Delete §24.2.3, §24.2.4, and §24.2.5.

Replace the definition distance() in section §24.2.6 with

    template <class InputIterator>
    iterator_trait<InputIterator>::distance_type
    distance(InputIterator first, InputIterator last);

    Effects: Returns the number of times it takes to get from first to last.

and delete the footnote in that section.

In §24.3.1.1, add iterator_trait<BidirectionalIterator>::value_type as a default for T, and eliminate the footnote that says that this is impossible! Change the declaration of reverse_bidirectional_iterator so that it is derived from iterator<bidirectional_iterator_tag,T,Distance>. Make the same changes in §24.3.1.3: add iterator_trait<RandomAccessIterator>::value_type as a default for T, and change reverse_iterator so that it is derived from iterator<random_access_iterator_tag,T,Distance>.

In §24.3.2.1 [lib.back.insert.iterator], derive back_insert_iterator from iterator<output_iterator_tag, void, void> instead of from output_iterator.

In §24.3.2.3 [lib.front.insert.iterator], make the same change: derive front_insert_iterator from iterator<output_iterator_tag, void, void>.

In §24.3.2.5 [lib.insert.iterator], derive insert_iterator from iterator<output_iterator_tag, void, void>.

In §24.4.1 [lib.istream.iterator], and §24.4.2 [lib.ostream.iterator], derive istream_iterator and ostream_iterator from, respectively, iterator<input_iterator_tag,T,Distance> and iterator<output_iterator_tag, void, void>.

In §24.4.3 [lib.istreambuf.iterator] change the declaration of istreambuf_iterator to (this also fixes a bug in the current draft not related to the new definition of iterator):

    namespace std {
        template<class charT, class traits = ios_traits<charT>,
             class Distance = ptrdiff_t>
        : public iterator<input_iterator_tag, charT, Distance>
        class istreambuf_iterator {
            // no change from current WP
        };
    }

In §24.4.4 [lib.ostreambuf.iterator], derive ostreambuf_iterator from iterator<output_iterator_tag, void, void>.

Delete the definition of the iterator_category() function in §24.4.3.6, §24.4.4, and §24.4.4.3.

7 Caveat

Note that although we believe this code to be correct, we have not tested it: we do not have access to a compiler that supports partial specialization.