
9.5. Iterator Traits

The various iterator categories (see Section 9.2, page 433) represent special iterator abilities. It might be useful or even necessary to be able to overload behavior for different iterator categories. By using iterator tags and iterator traits (both provided in `<iterator>`), such an overloading can be performed.

For each iterator category, the C++ standard library provides an *iterator tag* that can be used as a "label" for iterators:

```
namespace std {
    struct output_iterator_tag {
    };
    struct input_iterator_tag {
    };
    struct forward_iterator_tag
     : public input_iterator_tag {
    };
    struct bidirectional_iterator_tag
     : public forward_iterator_tag {
    };
    struct random_access_iterator_tag
     : public bidirectional_iterator_tag {
    };
}
```

Note that inheritance is used. So, for example, any forward iterator *is a* kind of input iterator. However, note that the tag for forward iterators is derived only from the tag for input iterators, not from the tag for output iterators. Thus, any forward iterator *is not a* kind of output iterator. Only a *mutable* forward iterator also fulfills the requirements of output iterators (see Section 9.2, page 433), but no specific category exists for this.

If you write generic code, you might not be interested only in the iterator category. For example, you may need the type of the elements to which the iterator refers. Therefore, the C++ standard library provides a special template structure to define the *iterator traits*. This structure contains all relevant information about an iterator and is used as a common interface for all the type definitions an iterator should have (the category, the type of the elements, and so on):

Click here to view code image

```
namespace std {
    template <typename T>
    struct iterator_traits {
        typedef typename T::iterator_category iterator_category;
        typedef typename T::value_type        value_type;
        typedef typename T::difference_type   difference_type;
        typedef typename T::pointer           pointer;
        typedef typename T::reference         reference;
    };
}
```

In this template, `T` stands for the type of the iterator. Thus, you can write code that, for any iterator, uses its category, the type of its elements, and so on. For example, the following expression yields the value type of iterator type `T`:

```
typename std::iterator_traits<T>::value_type
```

This structure has two advantages:

1. It ensures that an iterator provides all type definitions.
2. It can be (partially) specialized for (sets of) special iterators.

The latter is done for ordinary pointers that also can be used as iterators:

Click here to view code image

```
namespace std {
    template <typename T>
    struct iterator_traits<T*> {
        typedef T                         value_type;
        typedef ptrdiff_t                 difference_type;
        typedef random_access_iterator_tag iterator_category;
        typedef T*                        pointer;
        typedef T&                        reference;
    };
}
```

Thus, for any type "pointer to `T`," it is defined as having the random-access iterator category. A corresponding partial specialization exists for constant pointers (`const T*`).

Note that output iterators can be used only to write something. Thus, in the case of an output iterator, `value_type`, `difference_type`, `pointer`, and `reference` may be defined as `void`.

9.5.1. Writing Generic Functions for Iterators

Using iterator traits, you can write generic functions that derive type definitions or use different implementation code depending on the iterator category.

Using Iterator Types

A simple example of the use of iterator traits is an algorithm that needs a temporary variable for the elements. Such a temporary value is declared simply like this:

```
typename std::iterator_traits<T>::value_type tmp;
```

where `T` is the type of the iterator.

Another example is an algorithm that shifts elements cyclically:

Click here to view code image

```
template <typename ForwardIterator>
void shift_left (ForwardIterator beg, ForwardIterator end)
{
    // temporary variable for first element
    typedef typename
     std::iterator_traits<ForwardIterator>::value_type value_type;

    if (beg != end) {
        // save value of first element
        value_type tmp(*beg);

        // shift following values
        ...
    }
}
```

Using Iterator Categories

To use different implementations for different iterator categories you must follow these two steps:

1. Let your function template call another function with the iterator category as an additional argument. For example:

Click here to view code image

```
template <typename Iterator>
inline void foo (Iterator beg, Iterator end)
{
    foo (beg, end,
         std::iterator_traits<Iterator>::iterator_category());
}
```

2. Implement that other function for any iterator category that provides a special implementation that is not derived from another iterator category. For example:

```
// foo() for bidirectional iterators
template <typename BiIterator>
void foo (BiIterator beg, BiIterator end,
          std::bidirectional_iterator_tag)
{
    ...
}

// foo() for random-access iterators
template <typename RaIterator>
void foo (RaIterator beg, RaIterator end,
          std::random_access_iterator_tag)
{
    ...
}
```

The version for random-access iterators could, for example, use random-access operations, whereas the version for bidirectional iterators would not. Due to the hierarchy of iterator tags (see Section 9.5, page 466), you could provide one implementation for more than one iterator category.

Implementation of `distance()`

An example of following those two steps is the implementation of the auxiliary `distance()` iterator function, which returns the distance between two iterator positions and their elements (see Section 9.3.3, page 445). The implementation for random-access iterators uses only the operator `-`. For all other iterator categories, the number of increments to reach the end of the range is returned:

Click here to view code image

```
// general distance()
template <typename Iterator>
typename std::iterator_traits<Iterator>::difference_type
distance (Iterator pos1, Iterator pos2)
{
    return distance (pos1, pos2,
                     std::iterator_traits<Iterator>
                        ::iterator_category());
}

// distance() for random-access iterators
template <typename RaIterator>
typename std::iterator_traits<RaIterator>::difference_type
distance (RaIterator pos1, RaIterator pos2,
          std::random_access_iterator_tag)
{
     return pos2 - pos1;
}

// distance() for input, forward, and bidirectional iterators
template <typename InIterator>
typename std::iterator_traits<InIterator>::difference_type
distance (InIterator pos1, InIterator pos2,
          std::input_iterator_tag)
{
     typename std::iterator_traits<InIterator>::difference_type d;
     for (d=0; pos1 != pos2; ++pos1, ++d) {
         ;
     }
     return d;
}
```

The difference type of the iterator is used as the return type. Note that the second version uses the tag for input iterators, so this implementation is also used by forward and bidirectional iterators because their tags are derived from `input_iterator_tag`.