1 迭代器库 [iterators]

1.1 概述 [iterators.general]

1 本章描述C++程序在容器(??章)中、流(??)上或流缓冲区(??)中进行迭代操作时用到的组件。

2 本章后续子章节描述迭代器的需求以及迭代器原语、预定义迭代器和流迭代器的组件，如表1所示。

Table 1 — 迭代器库概览

Subclause		Header(s)
1.2	需求	
1.4	迭代器原语	`<iterator>`
1.5	预定义迭代器	
1.6	流迭代器	

1.2 迭代器需求 [iterator.requirements]

1.2.1 通用 [iterator.requirements.general]

1 迭代器是指针概念的泛化，迭代器使得C++程序以统一的方式使用不同的数据结构（容器）。本库不仅规定了迭代器的接口形式，还规定了迭代器的语义和预期的复杂度，旨在于为各种类型的数据结构设计无误且高效的模板算法。所有输入迭代器`i`都可通过表达式`*i`产生某对象类型`T`的值，`T`称为该迭代器的值类型。所有输出迭代器都支持表达式`*i = o`，其中`o`是迭代器`i`的可写类型集中某个可写类型的对象。使表达式`(*i).m`成立的迭代器`i`也使`i->m`成立，其语义与`(*i).m`相同。凡是定义了等同性的迭代器类型`X`都存在一个称为距离类型的有符号整型与之对应。

2 迭代器是指针的抽象，因此迭代器的语义是C++指针的绝大多数语义的泛化。此举确保所有适用迭代器的函数模板都同样适用常规指针。本标准根据迭代器自身所定义的操作定义了五种迭代器，它们分别是输入迭代器、输出迭代器、前向迭代器、双向迭代器以及随机访问迭代器，如表2所示。

Table 2 — 各种迭代器之间的关系

随机访问	→ **双向**	→ **前向**	→ **输入**
			→ **输出**

3 前向迭代器满足输入迭代器的所有需求，因此可用于所有适用输入迭代器的场合；双向迭代器满足前向迭代器的所有需求，因此可用于所有适用前向迭代器的场合；随机访问迭代器满足双向迭代器的所有需求，因此可用于所有适用双向迭代器的场合。

4 满足输出迭代器需求的迭代器也都称为变值迭代器。非变值迭代器称为常值迭代器。

5 对于整型数值`n`以及可解引用的迭代器值`a`和`(a + n)`，满足`*(a + n)`与`*(addressof(*a) + n)`相等的迭代器也都称为连续迭代器。[注：例如，类型“指向`int`类型的指针”就是连续迭代器，但`reverse_iterator<int *>`则不是。可解引用迭代器`a`的有效迭代范围[a,b)对应的指针表示的范围是[`addressof(*a)`,`addressof(*a) + (b - a)`)，其中`b`可能并不能解引用。 ——结束注]

6 正如可以担保指向数组的常规指针一定存在一个指针值指向数组的接尾部分，所有迭代器类型都存在一个迭代器值指向对应序列的接尾部分，这些值称为接尾值。使表达式`*i`成立的迭代器`i`的值是可解引用的。要注意本库不会做出接尾值可解引用的假设。迭代器可能为异值，说明其不与任何序列相关。[例：虽声明但未初始化的指针`x`（如`int* x;`）应当同时也必须被当作异值对待。 ——结束例]大多数操纵异值的表达式的结果都是未定义的，例外是：销毁异值迭代器；为异值迭代器赋非异值。另外对于满

足DefaultConstructible需求的迭代器，可以将做过值初始化的迭代器拷贝或移动至其中。［注：默认初始化并不作此保证，这种区别对待实际上只影响那些具有传统默认构造函数的类型，如指针或持有指针的聚合体。——结束注］这些情况下异值和非异值一样会被覆盖。可解引用的值一定是非异值。

7 对于迭代器i和j，当且仅当有限次应用表达式++i后可以得到i == j时，称i可达j。i可达j可达喻示着它们指向同一序列的元素。

8 本库大多数操作数据结构的算法模板都存在着使用范围的接口。范围是一对表示计算开始和计算结束的迭代器。范围[i, i)是空范围；通常情况下，范围[i, j)表示某一数据结构中i指向的元素到但不包括j指向的元素之间的所有元素。[i, j)有效，当且仅当i可达j。将本库中的函数应用在无效的范围上，结果是未定义的。

9 All the categories of iterators require only those functions that are realizable for a given category in constant time (amortized). Therefore, requirement tables for the iterators do not have a complexity column.

10 析构迭代器可能使得之前从该迭代器获取的指针或引用失效。

11 无效迭代器是可能为异值的迭代器。[1]

12 后续章节中，a和b表示X类型或const X类型的值；difference_type和reference分别表示类型iterator_traits<X>::difference_type和iterator_traits<X>::reference；n表示difference_type类型的值；u、tmp以及m表示标识符；r表示X&类型的值；t表示T的值类型的值；o表示可写至输出迭代器的类型的值。［注：每个迭代器类型X都必须存在一个iterator_traits<X> (1.4.1)实例。——结束注］

1.2.2 Iterator [iterator.iterators]

1 The Iterator requirements form the basis of the iterator concept taxonomy; every iterator satisfies the Iterator requirements. This set of requirements specifies operations for dereferencing and incrementing an iterator. Most algorithms will require additional operations to read (1.2.3) or write (1.2.4) values, or to provide a richer set of iterator movements (1.2.5, 1.2.6, 1.2.7).)

2 A type X satisfies the Iterator requirements if:

(2.1) — X satisfies the CopyConstructible, CopyAssignable, and Destructible requirements (??) and lvalues of type X are swappable (??), and

(2.2) — the expressions in Table 3 are valid and have the indicated semantics.

Table 3 — Iterator requirements

Expression	Return type	Operational semantics	Assertion/note pre-/post-condition
*r	unspecified		pre: r is dereferenceable.
++r	X&		

1.2.3 Input iterators [input.iterators]

1 A class or pointer type X satisfies the requirements of an input iterator for the value type T if X satisfies the Iterator (1.2.2) and EqualityComparable (Table ??) requirements and the expressions in Table 4 are valid and have the indicated semantics.

2 In Table 4, the term *the domain of* == is used in the ordinary mathematical sense to denote the set of values over which == is (required to be) defined. This set can change over time. Each algorithm places additional requirements on the domain of == for the iterator values it uses. These requirements can be inferred from the uses that algorithm makes of == and !=. ［例： the call find(a,b,x) is defined only if the value of a has

1) 此定义也用于指针，因为指针也是迭代器。解引用无效迭代器会造成未定义的结果。

the property *p* defined as follows: `b` has property *p* and a value `i` has property *p* if `(*i==x)` or if `(*i!=x` and `++i` has property `p`). ——结束例]

Table 4 — Input iterator requirements (in addition to Iterator)

Expression	**Return type**	**Operational semantics**	**Assertion/note pre-/post-condition**
`a != b`	contextually convertible to `bool`	`!(a == b)`	pre: `(a, b)` is in the domain of `==`.
`*a`	`reference`, convertible to `T`		pre: `a` is dereferenceable. The expression `(void)*a, *a` is equivalent to `*a`. If `a == b` and `(a, b)` is in the domain of `==` then `*a` is equivalent to `*b`.
`a->m`		`(*a).m`	pre: `a` is dereferenceable.
`++r`	`X&`		pre: `r` is dereferenceable. post: `r` is dereferenceable or `r` is past-the-end. post: any copies of the previous value of `r` are no longer required either to be dereferenceable or to be in the domain of `==`.
`(void)r++`			equivalent to `(void)++r`
`*r++`	convertible to `T`	`{ T tmp = *r;` `++r;` `return tmp; }`	

3 [注： For input iterators, `a == b` does not imply `++a == ++b`. (Equality does not guarantee the substitution property or referential transparency.) Algorithms on input iterators should never attempt to pass through the same iterator twice. They should be *single pass* algorithms. Value type T is not required to be a `CopyAssignable` type (Table ??). These algorithms can be used with istreams as the source of the input data through the `istream_iterator` class template. ——结束注]

1.2.4 Output iterators [output.iterators]

1 A class or pointer type `X` satisfies the requirements of an output iterator if `X` satisfies the `Iterator` requirements (1.2.2) and the expressions in Table 5 are valid and have the indicated semantics.

Table 5 — Output iterator requirements (in addition to Iterator)

Expression	**Return type**	**Operational semantics**	**Assertion/note pre-/post-condition**
`*r = o`	result is not used		*Remark:* After this operation `r` is not required to be dereferenceable. post: `r` is incrementable.

Table 5 — Output iterator requirements (in addition to Iterator) (continued)

Expression	Return type	Operational semantics	Assertion/note pre-/post-condition
`++r`	`X&`		`&r == &++r`. *Remark:* After this operation `r` is not required to be dereferenceable. post: `r` is incrementable.

`r++`	convertible to `const X&`	`{ X tmp = r;`
		`++r;`
		`return tmp; }`
			Remark: After this operation `r` is not required to be dereferenceable.
			post: `r` is incrementable.

`*r++ = o`	result is not used		*Remark:* After this operation `r` is not required to be dereferenceable. post: `r` is incrementable.

2 [注： The only valid use of an `operator*` is on the left side of the assignment statement. *Assignment through the same value of the iterator happens only once.* Algorithms on output iterators should never attempt to pass through the same iterator twice. They should be *single pass* algorithms. Equality and inequality might not be defined. Algorithms that take output iterators can be used with ostreams as the destination for placing data through the `ostream_iterator` class as well as with insert iterators and insert pointers. ——结束注]

1.2.5 Forward iterators [forward.iterators]

1 A class or pointer type `X` satisfies the requirements of a forward iterator if

(1.1) — `X` satisfies the requirements of an input iterator (1.2.3),

(1.2) — X satisfies the `DefaultConstructible` requirements (??),

(1.3) — if `X` is a mutable iterator, `reference` is a reference to `T`; if `X` is a const iterator, `reference` is a reference to `const T`,

(1.4) — the expressions in Table 6 are valid and have the indicated semantics, and

(1.5) — objects of type `X` offer the multi-pass guarantee, described below.

2 The domain of == for forward iterators is that of iterators over the same underlying sequence. However, value-initialized iterators may be compared and shall compare equal to other value-initialized iterators of the same type. [注： value initialized iterators behave as if they refer past the end of the same empty sequence ——结束注]

3 Two dereferenceable iterators `a` and `b` of type `X` offer the *multi-pass guarantee* if:

(3.1) — `a == b` implies `++a == ++b` and

(3.2) — `X` is a pointer type or the expression `(void)++X(a), *a` is equivalent to the expression `*a`.

4 [注： The requirement that `a == b` implies `++a == ++b` (which is not true for input and output iterators) and the removal of the restrictions on the number of the assignments through a mutable iterator (which applies to output iterators) allows the use of multi-pass one-directional algorithms with forward iterators. ——结束注]

Table 6 — Forward iterator requirements (in addition to input iterator)

Expression	Return type	Operational semantics	Assertion/note pre-/post-condition
`r++`	convertible to `const X&`	`{ X tmp = r;` `++r;` `return tmp; }`	
`*r++`	`reference`		

5 If `a` and `b` are equal, then either `a` and `b` are both dereferenceable or else neither is dereferenceable.

6 If `a` and `b` are both dereferenceable, then `a == b` if and only if `*a` and `*b` are bound to the same object.

1.2.6 Bidirectional iterators [bidirectional.iterators]

1 A class or pointer type `X` satisfies the requirements of a bidirectional iterator if, in addition to satisfying the requirements for forward iterators, the following expressions are valid as shown in Table 7.

Table 7 — Bidirectional iterator requirements (in addition to forward iterator)

Expression	Return type	Operational semantics	Assertion/note pre-/post-condition
`--r`	`X&`		pre: there exists `s` such that `r == ++s`. post: `r` is dereferenceable. `--(++r) == r`. `--r == --s` implies `r == s`. `&r == &--r`.
`r--`	convertible to `const X&`	`{ X tmp = r;` `--r;` `return tmp; }`	
`*r--`	`reference`		

2 [注： Bidirectional iterators allow algorithms to move iterators backward as well as forward. ——结束注]

1.2.7 Random access iterators [random.access.iterators]

1 A class or pointer type `X` satisfies the requirements of a random access iterator if, in addition to satisfying the requirements for bidirectional iterators, the following expressions are valid as shown in Table 8.

Table 8 — Random access iterator requirements (in addition to bidirectional iterator)

Expression	Return type	Operational semantics	Assertion/note pre-/post-condition

`r += n`	`X&`	`{ difference_type m = n;`
		`if (m >= 0)`
		`while (m--)`
		`++r;`
		`else`
		`while (m++)`
		`--r;`
		`return r; }`	

`a + n`
`n + a`
	`X`	`{ X tmp = a;`
		`return tmp += n; }`
			`a + n == n + a`.

`r -= n`	`X&`	`return r += -n;`	

`a - n`	`X`	`{ X tmp = a;`
		`return tmp -= n; }`	

`b - a`	`difference_-` `type`	`return n`	pre: there exists a value `n` of type `difference_type` such that `a + n == b`. `b == a + (b - a)`.
`a[n]`	convertible to `reference`	`*(a + n)`	
`a < b`	contextually convertible to `bool`	`b - a > 0`	`<` is a total ordering relation
`a > b`	contextually convertible to `bool`	`b < a`	`>` is a total ordering relation opposite to `<`.
`a >= b`	contextually convertible to `bool`	`!(a < b)`	
`a <= b`	contextually convertible to `bool`.	`!(a > b)`	

1.3 Header `<iterator>` synopsis [iterator.synopsis]

```
namespace std {
  // 1.4, primitives:
  template<class Iterator> struct iterator_traits;
  template<class T> struct iterator_traits<T*>;

  template<class Category, class T, class Distance = ptrdiff_t,
       class Pointer = T*, class Reference = T&> struct iterator;

  struct input_iterator_tag { };
  struct output_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 { };
```

```
// 1.4.4, iterator operations:
template <class InputIterator, class Distance>
  constexpr void advance(InputIterator& i, Distance n);
template <class InputIterator>
  constexpr typename iterator_traits<InputIterator>::difference_type
  distance(InputIterator first, InputIterator last);
template <class InputIterator>
  constexpr InputIterator next(InputIterator x,
    typename std::iterator_traits<InputIterator>::difference_type n = 1);
template <class BidirectionalIterator>
  constexpr BidirectionalIterator prev(BidirectionalIterator x,
    typename std::iterator_traits<BidirectionalIterator>::difference_type n = 1);

// 1.5, predefined iterators:
template <class Iterator> class reverse_iterator;

template <class Iterator1, class Iterator2>
  constexpr bool operator==(
    const reverse_iterator<Iterator1>& x,
    const reverse_iterator<Iterator2>& y);
template <class Iterator1, class Iterator2>
  constexpr bool operator<(
    const reverse_iterator<Iterator1>& x,
    const reverse_iterator<Iterator2>& y);
template <class Iterator1, class Iterator2>
  constexpr bool operator!=(
    const reverse_iterator<Iterator1>& x,
    const reverse_iterator<Iterator2>& y);
template <class Iterator1, class Iterator2>
  constexpr bool operator>(
    const reverse_iterator<Iterator1>& x,
    const reverse_iterator<Iterator2>& y);
template <class Iterator1, class Iterator2>
  constexpr bool operator>=(
    const reverse_iterator<Iterator1>& x,
    const reverse_iterator<Iterator2>& y);
template <class Iterator1, class Iterator2>
  constexpr bool operator<=(
    const reverse_iterator<Iterator1>& x,
    const reverse_iterator<Iterator2>& y);

template <class Iterator1, class Iterator2>
  constexpr auto operator-(
    const reverse_iterator<Iterator1>& x,
    const reverse_iterator<Iterator2>& y) ->decltype(y.base() - x.base());
template <class Iterator>
  constexpr reverse_iterator<Iterator>
    operator+(
  typename reverse_iterator<Iterator>::difference_type n,
  const reverse_iterator<Iterator>& x);

template <class Iterator>
  constexpr reverse_iterator<Iterator> make_reverse_iterator(Iterator i);
```

```
template <class Container> class back_insert_iterator;
template <class Container>
  back_insert_iterator<Container> back_inserter(Container& x);

template <class Container> class front_insert_iterator;
template <class Container>
  front_insert_iterator<Container> front_inserter(Container& x);

template <class Container> class insert_iterator;
template <class Container>
  insert_iterator<Container> inserter(Container& x, typename Container::iterator i);

template <class Iterator> class move_iterator;
template <class Iterator1, class Iterator2>
  constexpr bool operator==(
    const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);
template <class Iterator1, class Iterator2>
  constexpr bool operator!=(
    const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);
template <class Iterator1, class Iterator2>
  constexpr bool operator<(
    const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);
template <class Iterator1, class Iterator2>
  constexpr bool operator<=(
    const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);
template <class Iterator1, class Iterator2>
  constexpr bool operator>(
    const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);
template <class Iterator1, class Iterator2>
  constexpr bool operator>=(
    const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);

template <class Iterator1, class Iterator2>
  constexpr auto operator-(
  const move_iterator<Iterator1>& x,
  const move_iterator<Iterator2>& y) -> decltype(x.base() - y.base());
template <class Iterator>
  constexpr move_iterator<Iterator> operator+(
    typename move_iterator<Iterator>::difference_type n, const move_iterator<Iterator>& x);
template <class Iterator>
  constexpr move_iterator<Iterator> make_move_iterator(Iterator i);

// 1.6, stream iterators:
template <class T, class charT = char, class traits = char_traits<charT>,
    class Distance = ptrdiff_t>
class istream_iterator;
template <class T, class charT, class traits, class Distance>
  bool operator==(const istream_iterator<T,charT,traits,Distance>& x,
          const istream_iterator<T,charT,traits,Distance>& y);
template <class T, class charT, class traits, class Distance>
  bool operator!=(const istream_iterator<T,charT,traits,Distance>& x,
          const istream_iterator<T,charT,traits,Distance>& y);

template <class T, class charT = char, class traits = char_traits<charT> >
    class ostream_iterator;
  template<class charT, class traits = char_traits<charT> >
    class istreambuf_iterator;
  template <class charT, class traits>
    bool operator==(const istreambuf_iterator<charT,traits>& a,
            const istreambuf_iterator<charT,traits>& b);
  template <class charT, class traits>
    bool operator!=(const istreambuf_iterator<charT,traits>& a,
            const istreambuf_iterator<charT,traits>& b);

  template <class charT, class traits = char_traits<charT> >
    class ostreambuf_iterator;

  // 1.7, range access:
  template <class C> constexpr auto begin(C& c) -> decltype(c.begin());
  template <class C> constexpr auto begin(const C& c) -> decltype(c.begin());
  template <class C> constexpr auto end(C& c) -> decltype(c.end());
  template <class C> constexpr auto end(const C& c) -> decltype(c.end());
  template <class T, size_t N> constexpr T* begin(T (&array)[N]) noexcept;
  template <class T, size_t N> constexpr T* end(T (&array)[N]) noexcept;
  template <class C> constexpr auto cbegin(const C& c) noexcept(noexcept(std::begin(c)))
    -> decltype(std::begin(c));
  template <class C> constexpr auto cend(const C& c) noexcept(noexcept(std::end(c)))
    -> decltype(std::end(c));
  template <class C> constexpr auto rbegin(C& c) -> decltype(c.rbegin());
  template <class C> constexpr auto rbegin(const C& c) -> decltype(c.rbegin());
  template <class C> constexpr auto rend(C& c) -> decltype(c.rend());
  template <class C> constexpr auto rend(const C& c) -> decltype(c.rend());
  template <class T, size_t N> constexpr reverse_iterator<T*> rbegin(T (&array)[N]);
  template <class T, size_t N> constexpr reverse_iterator<T*> rend(T (&array)[N]);
  template <class E> constexpr reverse_iterator<const E*> rbegin(initializer_list<E> il);
  template <class E> constexpr reverse_iterator<const E*> rend(initializer_list<E> il);
  template <class C> constexpr auto crbegin(const C& c) -> decltype(std::rbegin(c));
  template <class C> constexpr auto crend(const C& c) -> decltype(std::rend(c));

  // 1.8, container access:
  template <class C> constexpr auto size(const C& c) -> decltype(c.size());
  template <class T, size_t N> constexpr size_t size(const T (&array)[N]) noexcept;
  template <class C> constexpr auto empty(const C& c) -> decltype(c.empty());
  template <class T, size_t N> constexpr bool empty(const T (&array)[N]) noexcept;
  template <class E> constexpr bool empty(initializer_list<E> il) noexcept;
  template <class C> constexpr auto data(C& c) -> decltype(c.data());
  template <class C> constexpr auto data(const C& c) -> decltype(c.data());
  template <class T, size_t N> constexpr T* data(T (&array)[N]) noexcept;
  template <class E> constexpr const E* data(initializer_list<E> il) noexcept;
}
```

1.4 Iterator primitives [iterator.primitives]

1 To simplify the task of defining iterators, the library provides several classes and functions:

1.4.1 Iterator traits [iterator.traits]

1 To implement algorithms only in terms of iterators, it is often necessary to determine the value and difference types that correspond to a particular iterator type. Accordingly, it is required that if `Iterator` is the type of an iterator, the types

```
iterator_traits<Iterator>::difference_type
iterator_traits<Iterator>::value_type
iterator_traits<Iterator>::iterator_category
```

be defined as the iterator's difference type, value type and iterator category, respectively. In addition, the types

```
iterator_traits<Iterator>::reference
iterator_traits<Iterator>::pointer
```

shall be defined as the iterator's reference and pointer types, that is, for an iterator object `a`, the same type as the type of `*a` and `a->`, respectively. In the case of an output iterator, the types

```
iterator_traits<Iterator>::difference_type
iterator_traits<Iterator>::value_type
iterator_traits<Iterator>::reference
iterator_traits<Iterator>::pointer
```

may be defined as `void`.

2 If `Iterator` has valid (**??**) member types `difference_type`, `value_type`, `pointer`, `reference`, and `iterator_-category`, `iterator_traits<Iterator>` shall have the following as publicly accessible members and shall have no other members:

```
  typedef typename Iterator::difference_type difference_type;
  typedef typename Iterator::value_type value_type;
  typedef typename Iterator::pointer pointer;
  typedef typename Iterator::reference reference;
  typedef typename Iterator::iterator_category iterator_category;
```

Otherwise, `iterator_traits<Iterator>` shall have no members.

3 It is specialized for pointers as

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

and for pointers to const as

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

4 [例: To implement a generic `reverse` function, a C++ program can do the following:

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

——结束例]

1.4.2 Basic iterator [iterator.basic]

1 The `iterator` template may be used as a base class to ease the definition of required types for new iterators.

```
namespace std {
  template<class Category, class T, class Distance = ptrdiff_t,
    class Pointer = T*, class Reference = T&>
  struct iterator {
    typedef T         value_type;
    typedef Distance  difference_type;
    typedef Pointer   pointer;
    typedef Reference reference;
    typedef Category  iterator_category;
  };
}
```

1.4.3 Standard iterator tags [std.iterator.tags]

1 It is often desirable for a function template specialization 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_traits<Iterator>::iterator_category` shall be defined to be the most specific category tag that describes the iterator's behavior.

```
namespace std {
  struct input_iterator_tag { };
  struct output_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 { };
}
```

2 [例: For a program-defined iterator `BinaryTreeIterator`, it could be included into the bidirectional iterator category by specializing the `iterator_traits` template:

```
template<class T> struct iterator_traits<BinaryTreeIterator<T> > {
  typedef std::ptrdiff_t difference_type;
  typedef T value_type;
  typedef T* pointer;
```

```
    typedef T& reference;
    typedef bidirectional_iterator_tag iterator_category;
  };
```

Typically, however, it would be easier to derive `BinaryTreeIterator<T>` from `iterator<bidirectional_-iterator_tag,T,ptrdiff_t,T*,T&>`. ——结束例]

3 [例: If `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,
      typename iterator_traits<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
  }
```

——结束例]

4 [例: 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, T*, T&> {
    // code implementing ++, etc.
  };
```

5 Then there is no need to specialize the `iterator_traits` template. ——结束例]

1.4.4 Iterator operations [iterator.operations]

1 Since only random access iterators provide `+` and `-` operators, the library provides two function templates `advance` and `distance`. These function templates use `+` and `-` for random access iterators (and are, therefore, constant time for them); for input, forward and bidirectional iterators they use `++` to provide linear time implementations.

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

2 *Requires:* `n` shall be negative only for bidirectional and random access iterators.

3 *Effects:* Increments (or decrements for negative `n`) iterator reference `i` by `n`.

```
template <class InputIterator>
  constexpr typename iterator_traits<InputIterator>::difference_type
    distance(InputIterator first, InputIterator last);
```

4 *Effects:* If `InputIterator` meets the requirements of random access iterator, returns `(last - first)`; otherwise, returns the number of increments needed to get from `first` to `last`.

5 *Requires:* If `InputIterator` meets the requirements of random access iterator, `last` shall be reachable from `first` or `first` shall be reachable from `last`; otherwise, `last` shall be reachable from `first`.

```
template <class InputIterator>
  constexpr InputIterator next(InputIterator x,
    typename std::iterator_traits<InputIterator>::difference_type n = 1);
```

6 *Effects:* Equivalent to `advance(x, n); return x;`

```
template <class BidirectionalIterator>
  constexpr BidirectionalIterator prev(BidirectionalIterator x,
    typename std::iterator_traits<BidirectionalIterator>::difference_type n = 1);
```

7 *Effects:* Equivalent to `advance(x, -n); return x;`

1.5 Iterator adaptors [predef.iterators]

1.5.1 Reverse iterators [reverse.iterators]

1 Class template `reverse_iterator` is an iterator adaptor that iterates from the end of the sequence defined by its underlying iterator to the beginning of that sequence. The fundamental relation between a reverse iterator and its corresponding iterator `i` is established by the identity: `&*(reverse_iterator(i)) == &*(i - 1)`.

1.5.1.1 Class template `reverse_iterator` [reverse.iterator]

```
namespace std {
  template <class Iterator>
  class reverse_iterator {
  public:
    typedef Iterator                                              iterator_type;
    typedef typename iterator_traits<Iterator>::iterator_category iterator_category;
    typedef typename iterator_traits<Iterator>::value_type        value_type;
    typedef typename iterator_traits<Iterator>::difference_type   difference_type;
    typedef typename iterator_traits<Iterator>::pointer           pointer;
    typedef typename iterator_traits<Iterator>::reference         reference;

    constexpr reverse_iterator();
    constexpr explicit reverse_iterator(Iterator x);
    template <class U> constexpr reverse_iterator(const reverse_iterator<U>& u);
    template <class U> constexpr reverse_iterator& operator=(const reverse_iterator<U>& u);

    constexpr Iterator base() const;      // explicit
    constexpr reference operator*() const;
    constexpr pointer   operator->() const;

    constexpr reverse_iterator& operator++();
    constexpr reverse_iterator  operator++(int);
    constexpr reverse_iterator& operator--();
    constexpr reverse_iterator  operator--(int);

    constexpr reverse_iterator  operator+ (difference_type n) const;
    constexpr reverse_iterator& operator+=(difference_type n);
    constexpr reverse_iterator  operator- (difference_type n) const;
    constexpr reverse_iterator& operator-=(difference_type n);
```

```
      constexpr unspecified operator[](difference_type n) const;
    protected:
      Iterator current;
    };

    template <class Iterator1, class Iterator2>
      constexpr bool operator==(
        const reverse_iterator<Iterator1>& x,
        const reverse_iterator<Iterator2>& y);
    template <class Iterator1, class Iterator2>
      constexpr bool operator<(
        const reverse_iterator<Iterator1>& x,
        const reverse_iterator<Iterator2>& y);
    template <class Iterator1, class Iterator2>
      constexpr bool operator!=(
        const reverse_iterator<Iterator1>& x,
        const reverse_iterator<Iterator2>& y);
    template <class Iterator1, class Iterator2>
      constexpr bool operator>(
        const reverse_iterator<Iterator1>& x,
        const reverse_iterator<Iterator2>& y);
    template <class Iterator1, class Iterator2>
      constexpr bool operator>=(
        const reverse_iterator<Iterator1>& x,
        const reverse_iterator<Iterator2>& y);
    template <class Iterator1, class Iterator2>
      constexpr bool operator<=(
        const reverse_iterator<Iterator1>& x,
        const reverse_iterator<Iterator2>& y);
    template <class Iterator1, class Iterator2>
      constexpr auto operator-(
        const reverse_iterator<Iterator1>& x,
        const reverse_iterator<Iterator2>& y) -> decltype(y.base() - x.base());
    template <class Iterator>
      constexpr reverse_iterator<Iterator> operator+(
        typename reverse_iterator<Iterator>::difference_type n,
        const reverse_iterator<Iterator>& x);

    template <class Iterator>
      constexpr reverse_iterator<Iterator> make_reverse_iterator(Iterator i);
  }
```

1.5.1.2 `reverse_iterator` requirements [reverse.iter.requirements]

1 The template parameter `Iterator` shall meet all the requirements of a Bidirectional Iterator (1.2.6).

2 Additionally, `Iterator` shall meet the requirements of a Random Access Iterator (1.2.7) if any of the members `operator+` (1.5.1.3.8), `operator-` (1.5.1.3.10), `operator+=` (1.5.1.3.9), `operator-=` (1.5.1.3.11), `operator []` (1.5.1.3.12), or the global operators `operator<` (1.5.1.3.14), `operator>` (1.5.1.3.16), `operator <=` (1.5.1.3.18), `operator>=` (1.5.1.3.17), `operator-` (1.5.1.3.19) or `operator+` (1.5.1.3.20) are referenced in a way that requires instantiation (**??**).

1.5.1.3 `reverse_iterator` operations [reverse.iter.ops]

1.5.1.3.1 `reverse_iterator` constructor [reverse.iter.cons]

```
constexpr reverse_iterator();
```

1 *Effects:* Value initializes `current`. Iterator operations applied to the resulting iterator have defined behavior if and only if the corresponding operations are defined on a value-initialized iterator of type `Iterator`.

```
constexpr explicit reverse_iterator(Iterator x);
```

2 *Effects:* Initializes `current` with `x`.

```
template <class U> constexpr reverse_iterator(const reverse_iterator<U> &u);
```

3 *Effects:* Initializes `current` with `u.current`.

1.5.1.3.2 `reverse_iterator::operator=` [reverse.iter.op=]

```
template <class U>
constexpr reverse_iterator&
  operator=(const reverse_iterator<U>& u);
```

1 *Effects:* Assigns `u.base()` to current.

2 *Returns:* `*this`.

1.5.1.3.3 Conversion [reverse.iter.conv]

```
constexpr Iterator base() const;           // explicit
```

1 *Returns:* `current`.

1.5.1.3.4 `operator*` [reverse.iter.op.star]

```
constexpr reference operator*() const;
```

1 *Effects:*

```
Iterator tmp = current;
return *--tmp;
```

1.5.1.3.5 `operator->` [reverse.iter.opref]

```
constexpr pointer operator->() const;
```

1 *Returns:* `std::addressof(operator*())`.

1.5.1.3.6 `operator++` [reverse.iter.op++]

```
constexpr reverse_iterator& operator++();
```

1 *Effects:* `--current;`

2 *Returns:* `*this`.

```
constexpr reverse_iterator operator++(int);
```

3 *Effects:*

```
reverse_iterator tmp = *this;
--current;
return tmp;
```

1.5.1.3.7 operator-- [reverse.iter.op--]

```
constexpr reverse_iterator& operator--();
```

1 *Effects:* `++current`

2 *Returns:* `*this`.

```
constexpr reverse_iterator operator--(int);
```

3 *Effects:*

```
  reverse_iterator tmp = *this;
  ++current;
  return tmp;
```

1.5.1.3.8 operator+ [reverse.iter.op+]

```
constexpr reverse_iterator
operator+(typename reverse_iterator<Iterator>::difference_type n) const;
```

1 *Returns:* `reverse_iterator(current-n)`.

1.5.1.3.9 operator+= [reverse.iter.op+=]

```
constexpr reverse_iterator&
operator+=(typename reverse_iterator<Iterator>::difference_type n);
```

1 *Effects:* `current -= n;`

2 *Returns:* `*this`.

1.5.1.3.10 operator- [reverse.iter.op-]

```
constexpr reverse_iterator
operator-(typename reverse_iterator<Iterator>::difference_type n) const;
```

1 *Returns:* `reverse_iterator(current+n)`.

1.5.1.3.11 operator-= [reverse.iter.op-=]

```
constexpr reverse_iterator&
operator-=(typename reverse_iterator<Iterator>::difference_type n);
```

1 *Effects:* `current += n;`

2 *Returns:* `*this`.

1.5.1.3.12 operator[] [reverse.iter.opindex]

```
constexpr unspecified operator[](
    typename reverse_iterator<Iterator>::difference_type n) const;
```

1 *Returns:* `current[-n-1]`.

1.5.1.3.13 operator== [reverse.iter.op==]

```
template <class Iterator1, class Iterator2>
  constexpr bool operator==(
    const reverse_iterator<Iterator1>& x,
    const reverse_iterator<Iterator2>& y);
```

1 *Returns:* `x.current == y.current`.

1.5.1.3.14 operator< [reverse.iter.op<]

```
template <class Iterator1, class Iterator2>
  constexpr bool operator<(
    const reverse_iterator<Iterator1>& x,
    const reverse_iterator<Iterator2>& y);
```

1 *Returns:* `x.current > y.current`.

1.5.1.3.15 operator!= [reverse.iter.op!=]

```
template <class Iterator1, class Iterator2>
  constexpr bool operator!=(
    const reverse_iterator<Iterator1>& x,
    const reverse_iterator<Iterator2>& y);
```

1 *Returns:* `x.current != y.current`.

1.5.1.3.16 operator> [reverse.iter.op>]

```
template <class Iterator1, class Iterator2>
  constexpr bool operator>(
    const reverse_iterator<Iterator1>& x,
    const reverse_iterator<Iterator2>& y);
```

1 *Returns:* `x.current < y.current`.

1.5.1.3.17 operator>= [reverse.iter.op>=]

```
template <class Iterator1, class Iterator2>
  constexpr bool operator>=(
    const reverse_iterator<Iterator1>& x,
    const reverse_iterator<Iterator2>& y);
```

1 *Returns:* `x.current <= y.current`.

1.5.1.3.18 operator<= [reverse.iter.op<=]

```
template <class Iterator1, class Iterator2>
  constexpr bool operator<=(
    const reverse_iterator<Iterator1>& x,
    const reverse_iterator<Iterator2>& y);
```

1 *Returns:* `x.current >= y.current`.

1.5.1.3.19 operator- [reverse.iter.opdiff]

```
template <class Iterator1, class Iterator2>
    constexpr auto operator-(
    const reverse_iterator<Iterator1>& x,
    const reverse_iterator<Iterator2>& y) -> decltype(y.base() - x.base());
```

1 *Returns:* `y.current - x.current`.

1.5.1.3.20 operator+ [reverse.iter.opsum]

```
template <class Iterator>
  constexpr reverse_iterator<Iterator> operator+(
    typename reverse_iterator<Iterator>::difference_type n,
    const reverse_iterator<Iterator>& x);
```

1 *Returns:* `reverse_iterator<Iterator> (x.current - n)`.

1.5.1.3.21 Non-member function `make_reverse_iterator()` [reverse.iter.make]

```
template <class Iterator>
  constexpr reverse_iterator<Iterator> make_reverse_iterator(Iterator i);
```

1 *Returns:* `reverse_iterator<Iterator>(i)`.

1.5.2 Insert iterators [insert.iterators]

1 To make it possible to deal with insertion in the same way as writing into an array, a special kind of iterator adaptors, called *insert iterators*, are provided in the library. With regular iterator classes,

```
  while (first != last) *result++ = *first++;
```

causes a range `[first, last)` to be copied into a range starting with result. The same code with `result` being an insert iterator will insert corresponding elements into the container. This device allows all of the copying algorithms in the library to work in the *insert mode* instead of the *regular overwrite* mode.

2 An insert iterator is constructed from a container and possibly one of its iterators pointing to where insertion takes place if it is neither at the beginning nor at the end of the container. Insert iterators satisfy the requirements of output iterators. `operator*` returns the insert iterator itself. The assignment `operator=(const T& x)` is defined on insert iterators to allow writing into them, it inserts `x` right before where the insert iterator is pointing. In other words, an insert iterator is like a cursor pointing into the container where the insertion takes place. `back_insert_iterator` inserts elements at the end of a container, `front_insert_iterator` inserts elements at the beginning of a container, and `insert_iterator` inserts elements where the iterator points to in a container. `back_inserter`, `front_inserter`, and `inserter` are three functions making the insert iterators out of a container.

1.5.2.1 Class template `back_insert_iterator` [back.insert.iterator]

```
  namespace std {
    template <class Container>
    class back_insert_iterator {
    protected:
      Container* container;

    public:
      typedef output_iterator_tag iterator_category;
      typedef void value_type;
      typedef void difference_type;
      typedef void pointer;
      typedef void reference;
      typedef Container container_type;
      explicit back_insert_iterator(Container& x);
      back_insert_iterator& operator=(const typename Container::value_type& value);
      back_insert_iterator& operator=(typename Container::value_type&& value);

      back_insert_iterator& operator*();
      back_insert_iterator& operator++();
      back_insert_iterator  operator++(int);
    };

    template <class Container>
      back_insert_iterator<Container> back_inserter(Container& x);
  }
```

1.5.2.2 `back_insert_iterator` operations [back.insert.iter.ops]

1.5.2.2.1 `back_insert_iterator` constructor [back.insert.iter.cons]

```
explicit back_insert_iterator(Container& x);
```

1 *Effects:* Initializes `container` with `std::addressof(x)`.

1.5.2.2.2 `back_insert_iterator::operator=` [back.insert.iter.op=]

```
back_insert_iterator& operator=(const typename Container::value_type& value);
```

1 *Effects:* `container->push_back(value);`

2 *Returns:* `*this`.

```
back_insert_iterator& operator=(typename Container::value_type&& value);
```

3 *Effects:* `container->push_back(std::move(value));`

4 *Returns:* `*this`.

1.5.2.2.3 `back_insert_iterator::operator*` [back.insert.iter.op*]

```
back_insert_iterator& operator*();
```

1 *Returns:* `*this`.

1.5.2.2.4 `back_insert_iterator::operator++` [back.insert.iter.op++]

```
back_insert_iterator& operator++();
back_insert_iterator  operator++(int);
```

1 *Returns:* `*this`.

1.5.2.2.5 `back_inserter` [back.inserter]

```
template <class Container>
  back_insert_iterator<Container> back_inserter(Container& x);
```

1 *Returns:* `back_insert_iterator<Container>(x)`.

1.5.2.3 Class template `front_insert_iterator` [front.insert.iterator]

```
  namespace std {
    template <class Container>
    class front_insert_iterator {
    protected:
      Container* container;

    public:
      typedef output_iterator_tag iterator_category;
      typedef void value_type;
      typedef void difference_type;
      typedef void pointer;
      typedef void reference;
      typedef Container container_type;
      explicit front_insert_iterator(Container& x);
      front_insert_iterator& operator=(const typename Container::value_type& value);
      front_insert_iterator& operator=(typename Container::value_type&& value);

      front_insert_iterator& operator*();
      front_insert_iterator& operator++();
      front_insert_iterator  operator++(int);
    };
    template <class Container>
      front_insert_iterator<Container> front_inserter(Container& x);
  }
```

1.5.2.4 `front_insert_iterator` operations [front.insert.iter.ops]

1.5.2.4.1 `front_insert_iterator` constructor [front.insert.iter.cons]

```
explicit front_insert_iterator(Container& x);
```

1 *Effects:* Initializes `container` with `std::addressof(x)`.

1.5.2.4.2 `front_insert_iterator::operator=` [front.insert.iter.op=]

```
front_insert_iterator& operator=(const typename Container::value_type& value);
```

1 *Effects:* `container->push_front(value);`

2 *Returns:* `*this`.

```
front_insert_iterator& operator=(typename Container::value_type&& value);
```

3 *Effects:* `container->push_front(std::move(value));`

4 *Returns:* `*this`.

1.5.2.4.3 `front_insert_iterator::operator*` [front.insert.iter.op*]

```
front_insert_iterator& operator*();
```

1 *Returns:* `*this`.

1.5.2.4.4 `front_insert_iterator::operator++` [front.insert.iter.op++]

```
front_insert_iterator& operator++();
front_insert_iterator  operator++(int);
```

1 *Returns:* `*this`.

1.5.2.4.5 `front_inserter` [front.inserter]

```
template <class Container>
  front_insert_iterator<Container> front_inserter(Container& x);
```

1 *Returns:* `front_insert_iterator<Container>(x)`.

1.5.2.5 Class template `insert_iterator` [insert.iterator]

```
  namespace std {
    template <class Container>
    class insert_iterator {
    protected:
      Container* container;
      typename Container::iterator iter;

    public:
      typedef output_iterator_tag iterator_category;
      typedef void value_type;
      typedef void difference_type;
      typedef void pointer;
      typedef void reference;
      typedef Container container_type;
      insert_iterator(Container& x, typename Container::iterator i);
      insert_iterator& operator=(const typename Container::value_type& value);
      insert_iterator& operator=(typename Container::value_type&& value);

      insert_iterator& operator*();
      insert_iterator& operator++();
      insert_iterator& operator++(int);
    };

    template <class Container>
      insert_iterator<Container> inserter(Container& x, typename Container::iterator i);
  }
```

1.5.2.6 `insert_iterator` operations [insert.iter.ops]

1.5.2.6.1 `insert_iterator` constructor [insert.iter.cons]

```
insert_iterator(Container& x, typename Container::iterator i);
```

1 *Effects:* Initializes `container` with `std::addressof(x)` and `iter` with `i`.

1.5.2.6.2 `insert_iterator::operator=` [insert.iter.op=]

```
insert_iterator& operator=(const typename Container::value_type& value);
```

1 *Effects:*

```
iter = container->insert(iter, value);
++iter;
```

2 *Returns:* `*this`.

```
insert_iterator& operator=(typename Container::value_type&& value);
```

3 *Effects:*

```
iter = container->insert(iter, std::move(value));
++iter;
```

4 *Returns:* `*this`.

1.5.2.6.3 `insert_iterator::operator*` [insert.iter.op*]

```
insert_iterator& operator*();
```

1 *Returns:* `*this`.

1.5.2.6.4 `insert_iterator::operator++` [insert.iter.op++]

```
insert_iterator& operator++();
insert_iterator& operator++(int);
```

1 *Returns:* `*this`.

1.5.2.6.5 `inserter` [inserter]

```
template <class Container>
  insert_iterator<Container> inserter(Container& x, typename Container::iterator i);
```

1 *Returns:* `insert_iterator<Container>(x, i)`.

1.5.3 Move iterators [move.iterators]

1 Class template `move_iterator` is an iterator adaptor with the same behavior as the underlying iterator except that its indirection operator implicitly converts the value returned by the underlying iterator's indirection operator to an rvalue. Some generic algorithms can be called with move iterators to replace copying with moving.

2 [例：

```
list<string> s;
// populate the list s
vector<string> v1(s.begin(), s.end());          // copies strings into v1
vector<string> v2(make_move_iterator(s.begin()),
                  make_move_iterator(s.end())); // moves strings into v2
```

——结束例]

1.5.3.1 Class template `move_iterator` [move.iterator]

```
namespace std {
  template <class Iterator>
  class move_iterator {
  public:
    typedef Iterator                                              iterator_type;
    typedef typename iterator_traits<Iterator>::difference_type   difference_type;
    typedef Iterator                                              pointer;
    typedef typename iterator_traits<Iterator>::value_type        value_type;
    typedef typename iterator_traits<Iterator>::iterator_category iterator_category;
    typedef see below                                             reference;

    constexpr move_iterator();
    constexpr explicit move_iterator(Iterator i);
    template <class U> constexpr move_iterator(const move_iterator<U>& u);
    template <class U> constexpr move_iterator& operator=(const move_iterator<U>& u);

    constexpr iterator_type base() const;
    constexpr reference operator*() const;
    constexpr pointer operator->() const;

    constexpr move_iterator& operator++();
    constexpr move_iterator operator++(int);
    constexpr move_iterator& operator--();
    constexpr move_iterator operator--(int);

    constexpr move_iterator operator+(difference_type n) const;
    constexpr move_iterator& operator+=(difference_type n);
    constexpr move_iterator operator-(difference_type n) const;
    constexpr move_iterator& operator-=(difference_type n);
    constexpr unspecified operator[](difference_type n) const;

  private:
    Iterator current;   // exposition only
  };

  template <class Iterator1, class Iterator2>
    constexpr bool operator==(
      const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);
  template <class Iterator1, class Iterator2>
    constexpr bool operator!=(
      const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);
  template <class Iterator1, class Iterator2>
    constexpr bool operator<(
      const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);
  template <class Iterator1, class Iterator2>
    constexpr bool operator<=(
      const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);
  template <class Iterator1, class Iterator2>
    constexpr bool operator>(
      const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);
  template <class Iterator1, class Iterator2>
    constexpr bool operator>=(
      const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);

  template <class Iterator1, class Iterator2>
    constexpr auto operator-(
      const move_iterator<Iterator1>& x,
      const move_iterator<Iterator2>& y) -> decltype(x.base() - y.base());
  template <class Iterator>
    constexpr move_iterator<Iterator> operator+(
      typename move_iterator<Iterator>::difference_type n, const move_iterator<Iterator>& x);
  template <class Iterator>
    constexpr move_iterator<Iterator> make_move_iterator(Iterator i);
}
```

1 Let *R* be `iterator_traits<Iterator>::reference`. If `is_reference<`*R*`>::value` is `true`, the template specialization `move_iterator<Iterator>` shall define the nested type named `reference` as a synonym for `remove_reference<`*R*`>::type&&`, otherwise as a synonym for *R*.

1.5.3.2 `move_iterator` requirements [move.iter.requirements]

1 The template parameter `Iterator` shall meet the requirements for an Input Iterator (1.2.3). Additionally, if any of the bidirectional or random access traversal functions are instantiated, the template parameter shall meet the requirements for a Bidirectional Iterator (1.2.6) or a Random Access Iterator (1.2.7), respectively.

1.5.3.3 `move_iterator` operations [move.iter.ops]

1.5.3.3.1 `move_iterator` constructors [move.iter.op.const]

```
constexpr move_iterator();
```

1 *Effects:* Constructs a `move_iterator`, value initializing `current`. Iterator operations applied to the resulting iterator have defined behavior if and only if the corresponding operations are defined on a value-initialized iterator of type `Iterator`.

```
constexpr explicit move_iterator(Iterator i);
```

2 *Effects:* Constructs a `move_iterator`, initializing `current` with `i`.

```
template <class U> constexpr move_iterator(const move_iterator<U>& u);
```

3 *Effects:* Constructs a `move_iterator`, initializing `current` with `u.base()`.

4 *Requires:* `U` shall be convertible to `Iterator`.

1.5.3.3.2 `move_iterator::operator=` [move.iter.op=]

```
template <class U> constexpr move_iterator& operator=(const move_iterator<U>& u);
```

1 *Effects:* Assigns `u.base()` to `current`.

2 *Requires:* `U` shall be convertible to `Iterator`.

1.5.3.3.3 `move_iterator` conversion [move.iter.op.conv]

```
constexpr Iterator base() const;
```

1 *Returns:* `current`.

1.5.3.3.4 `move_iterator::operator*` [move.iter.op.star]

```
constexpr reference operator*() const;
```

1 *Returns:* `static_cast<reference>(*current)`.

1.5.3.3.5 `move_iterator::operator->` [move.iter.op.ref]

```
constexpr pointer operator->() const;
```

1 *Returns:* `current`.

1.5.3.3.6 `move_iterator::operator++` [move.iter.op.incr]

```
constexpr move_iterator& operator++();
```

1 *Effects:* `++current`.

2 *Returns:* `*this`.

```
constexpr move_iterator operator++(int);
```

3 *Effects:*

```
  move_iterator tmp = *this;
  ++current;
  return tmp;
```

1.5.3.3.7 `move_iterator::operator--` [move.iter.op.decr]

```
constexpr move_iterator& operator--();
```

1 *Effects:* `--current`.

2 *Returns:* `*this`.

```
constexpr move_iterator operator--(int);
```

3 *Effects:*

```
  move_iterator tmp = *this;
  --current;
  return tmp;
```

1.5.3.3.8 `move_iterator::operator+` [move.iter.op.+]

```
constexpr move_iterator operator+(difference_type n) const;
```

1 *Returns:* `move_iterator(current + n)`.

1.5.3.3.9 `move_iterator::operator+=` [move.iter.op.+=]

```
constexpr move_iterator& operator+=(difference_type n);
```

1 *Effects:* `current += n`.

2 *Returns:* `*this`.

1.5.3.3.10 `move_iterator::operator-` [move.iter.op.-]

```
constexpr move_iterator operator-(difference_type n) const;
```

1 *Returns:* `move_iterator(current - n)`.

1.5.3.3.11 `move_iterator::operator-=` [move.iter.op.-=]

```
constexpr move_iterator& operator-=(difference_type n);
```

1 *Effects:* `current -= n`.

2 *Returns:* `*this`.

1.5.3.3.12 `move_iterator::operator[]` [move.iter.op.index]

```
constexpr unspecified operator[](difference_type n) const;
```

1 *Returns:* `std::move`(current[n]).

1.5.3.3.13 `move_iterator` comparisons [move.iter.op.comp]

```
template <class Iterator1, class Iterator2>
constexpr bool operator==(const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);
```

1 *Returns:* `x.base() == y.base()`.

```
template <class Iterator1, class Iterator2>
constexpr bool operator!=(const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);
```

2 *Returns:* `!(x == y)`.

```
template <class Iterator1, class Iterator2>
constexpr bool operator<(const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);
```

3 *Returns:* `x.base() < y.base()`.

```
template <class Iterator1, class Iterator2>
constexpr bool operator<=(const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);
```

4 *Returns:* `!(y < x)`.

```
template <class Iterator1, class Iterator2>
constexpr bool operator>(const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);
```

5 *Returns:* `y < x`.

```
template <class Iterator1, class Iterator2>
constexpr bool operator>=(const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);
```

6 *Returns:* `!(x < y)`.

1.5.3.3.14 `move_iterator` non-member functions [move.iter.nonmember]

```
template <class Iterator1, class Iterator2>
    constexpr auto operator-(
    const move_iterator<Iterator1>& x,
    const move_iterator<Iterator2>& y) -> decltype(x.base() - y.base());
```

1 *Returns:* `x.base() - y.base()`.

```
template <class Iterator>
  constexpr move_iterator<Iterator> operator+(
    typename move_iterator<Iterator>::difference_type n, const move_iterator<Iterator>& x);
```

2 *Returns:* `x + n`.

```
template <class Iterator>
constexpr move_iterator<Iterator> make_move_iterator(Iterator i);
```

3 *Returns:* `move_iterator<Iterator>(i)`.

1.6 Stream iterators [stream.iterators]

1 To make it possible for algorithmic templates to work directly with input/output streams, appropriate iterator-like class templates are provided.

[例：

```
  partial_sum(istream_iterator<double, char>(cin),
    istream_iterator<double, char>(),
    ostream_iterator<double, char>(cout, "\n"));
```

reads a file containing floating point numbers from `cin`, and prints the partial sums onto `cout`. ——结束例]

1.6.1 Class template istream_iterator [istream.iterator]

1 The class template `istream_iterator` is an input iterator (1.2.3) that reads (using `operator>>`) successive elements from the input stream for which it was constructed. After it is constructed, and every time `++` is used, the iterator reads and stores a value of `T`. If the iterator fails to read and store a value of `T` (`fail()` on the stream returns `true`), the iterator becomes equal to the *end-of-stream* iterator value. The constructor with no arguments `istream_iterator()` always constructs an end-of-stream input iterator object, which is the only legitimate iterator to be used for the end condition. The result of `operator*` on an end-of-stream iterator is not defined. For any other iterator value a `const T&` is returned. The result of `operator->` on an end-of-stream iterator is not defined. For any other iterator value a `const T*` is returned. The behavior of a program that applies `operator++()` to an end-of-stream iterator is undefined. It is impossible to store things into istream iterators.

2 Two end-of-stream iterators are always equal. An end-of-stream iterator is not equal to a non-end-of-stream iterator. Two non-end-of-stream iterators are equal when they are constructed from the same stream.

```
  namespace std {
    template <class T, class charT = char, class traits = char_traits<charT>,
        class Distance = ptrdiff_t>
    class istream_iterator {
    public:
      typedef input_iterator_tag iterator_category;
      typedef T value_type;
      typedef Distance difference_type;
      typedef const T* pointer;
      typedef const T& reference;
      typedef charT char_type;
      typedef traits traits_type;
      typedef basic_istream<charT,traits> istream_type;
      see below istream_iterator();
      istream_iterator(istream_type& s);
      istream_iterator(const istream_iterator& x) = default;
     ~istream_iterator() = default;

      const T& operator*() const;
      const T* operator->() const;
      istream_iterator& operator++();
      istream_iterator  operator++(int);
    private:
      basic_istream<charT,traits>* in_stream; // exposition only
      T value;                                // exposition only
    };

    template <class T, class charT, class traits, class Distance>
      bool operator==(const istream_iterator<T,charT,traits,Distance>& x,
              const istream_iterator<T,charT,traits,Distance>& y);
    template <class T, class charT, class traits, class Distance>
      bool operator!=(const istream_iterator<T,charT,traits,Distance>& x,
              const istream_iterator<T,charT,traits,Distance>& y);
  }
```

1.6.1.1 `istream_iterator` constructors and destructor [istream.iterator.cons]

`see below istream_iterator();`

1 *Effects:* Constructs the end-of-stream iterator. If `T` is a literal type, then this constructor shall be a `constexpr` constructor.

2 *Postcondition:* `in_stream == 0`.

`istream_iterator(istream_type& s);`

3 *Effects:* Initializes *in_stream* with `addressof(s)`. *value* may be initialized during construction or the first time it is referenced.

4 *Postcondition:* `in_stream == addressof(s)`.

`istream_iterator(const istream_iterator& x) = default;`

5 *Effects:* Constructs a copy of `x`. If `T` is a literal type, then this constructor shall be a trivial copy constructor.

6 *Postcondition:* `in_stream == x.in_stream`.

`~istream_iterator() = default;`

7 *Effects:* The iterator is destroyed. If `T` is a literal type, then this destructor shall be a trivial destructor.

1.6.1.2 `istream_iterator` operations [istream.iterator.ops]

`const T& operator*() const;`

1 *Returns: value.*

`const T* operator->() const;`

2 *Returns:* `addressof(operator*())`.

`istream_iterator& operator++();`

3 *Requires:* `in_stream != 0`.

4 *Effects:* `*in_stream >>value`.

5 *Returns:* `*this`.

`istream_iterator operator++(int);`

6 *Requires:* `in_stream != 0`.

7 *Effects:*

```
      istream_iterator tmp = *this;
      *in_stream >> value;
      return (tmp);
```

```
template <class T, class charT, class traits, class Distance>
  bool operator==(const istream_iterator<T,charT,traits,Distance> &x,
                  const istream_iterator<T,charT,traits,Distance> &y);
```

8 *Returns:* `x.in_stream == y.in_stream`.

```
template <class T, class charT, class traits, class Distance>
  bool operator!=(const istream_iterator<T,charT,traits,Distance> &x,
                  const istream_iterator<T,charT,traits,Distance> &y);
```

9 *Returns:* `!(x == y)`

1.6.2 Class template `ostream_iterator` [ostream.iterator]

1 `ostream_iterator` writes (using `operator<<`) successive elements onto the output stream from which it was constructed. If it was constructed with `charT*` as a constructor argument, this string, called a *delimiter string*, is written to the stream after every `T` is written. It is not possible to get a value out of the output iterator. Its only use is as an output iterator in situations like

```
while (first != last)
  *result++ = *first++;
```

2 `ostream_iterator` is defined as:

```
namespace std {
  template <class T, class charT = char, class traits = char_traits<charT> >
  class ostream_iterator {
  public:
    typedef output_iterator_tag iterator_category;
    typedef void value_type;
    typedef void difference_type;
    typedef void pointer;
    typedef void reference;
    typedef charT char_type;
    typedef traits traits_type;
    typedef basic_ostream<charT,traits> ostream_type;
    ostream_iterator(ostream_type& s);
    ostream_iterator(ostream_type& s, const charT* delimiter);
    ostream_iterator(const ostream_iterator& x);
   ~ostream_iterator();
    ostream_iterator& operator=(const T& value);

    ostream_iterator& operator*();
    ostream_iterator& operator++();
    ostream_iterator& operator++(int);
  private:
    basic_ostream<charT,traits>* out_stream;  // exposition only
    const charT* delim;                       // exposition only
  };
}
```

1.6.2.1 `ostream_iterator` constructors and destructor [ostream.iterator.cons.des]

```
ostream_iterator(ostream_type& s);
```

1 *Effects:* Initializes *out_stream* with `addressof(s)` and *delim* with null.

```
ostream_iterator(ostream_type& s, const charT* delimiter);
```

2 *Effects:* Initializes *out_stream* with `addressof(s)` and *delim* with `delimiter`.

```
ostream_iterator(const ostream_iterator& x);
```

3 *Effects:* Constructs a copy of `x`.

```
~ostream_iterator();
```

4 *Effects:* The iterator is destroyed.

1.6.2.2 `ostream_iterator` operations [ostream.iterator.ops]

```
ostream_iterator& operator=(const T& value);
```

1 *Effects:*

```
*out_stream << value;
if (delim != 0)
  *out_stream << delim;
return *this;
```

```
ostream_iterator& operator*();
```

2 *Returns:* `*this`.

```
ostream_iterator& operator++();
ostream_iterator& operator++(int);
```

3 *Returns:* `*this`.

1.6.3 Class template `istreambuf_iterator` [istreambuf.iterator]

1 The class template `istreambuf_iterator` defines an input iterator (1.2.3) that reads successive *characters* from the streambuf for which it was constructed. `operator*` provides access to the current input character, if any. [注： `operator->` may return a proxy. ——结束注] Each time `operator++` is evaluated, the iterator advances to the next input character. If the end of stream is reached (`streambuf_type::sgetc()` returns `traits::eof()`), the iterator becomes equal to the *end-of-stream* iterator value. The default constructor `istreambuf_iterator()` and the constructor `istreambuf_iterator(0)` both construct an end-of-stream iterator object suitable for use as an end-of-range. All specializations of `istreambuf_iterator` shall have a trivial copy constructor, a `constexpr` default constructor, and a trivial destructor.

2 The result of `operator*()` on an end-of-stream iterator is undefined. For any other iterator value a `char_type` value is returned. It is impossible to assign a character via an input iterator.

```
namespace std {
  template<class charT, class traits = char_traits<charT> >
  class istreambuf_iterator {
  public:
    typedef input_iterator_tag            iterator_category;
    typedef charT                         value_type;
    typedef typename traits::off_type     difference_type;
    typedef unspecified                   pointer;
    typedef charT                       reference;
    typedef charT                       char_type;
    typedef traits                      traits_type;
    typedef typename traits::int_type   int_type;
    typedef basic_streambuf<charT,traits> streambuf_type;
    typedef basic_istream<charT,traits>   istream_type;

    class proxy;                        // exposition only

    constexpr istreambuf_iterator() noexcept;
    istreambuf_iterator(const istreambuf_iterator&) noexcept = default;
    ~istreambuf_iterator() = default;
    istreambuf_iterator(istream_type& s) noexcept;
    istreambuf_iterator(streambuf_type* s) noexcept;
    istreambuf_iterator(const proxy& p) noexcept;
    charT operator*() const;
    pointer operator->() const;
    istreambuf_iterator& operator++();
    proxy operator++(int);
    bool equal(const istreambuf_iterator& b) const;
  private:
    streambuf_type* sbuf_;              // exposition only
  };

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

1.6.3.1 Class template `istreambuf_iterator::proxy` [istreambuf.iterator::proxy]

```
namespace std {
  template <class charT, class traits = char_traits<charT> >
  class istreambuf_iterator<charT, traits>::proxy { // exposition only
    charT keep_;
    basic_streambuf<charT,traits>* sbuf_;
    proxy(charT c, basic_streambuf<charT,traits>* sbuf)
      : keep_(c), sbuf_(sbuf) { }
  public:
    charT operator*() { return keep_; }
  };
}
```

1 Class `istreambuf_iterator<charT,traits>::proxy` is for exposition only. An implementation is permitted to provide equivalent functionality without providing a class with this name. Class `istreambuf_iterator<charT, traits>::proxy` provides a temporary placeholder as the return value of the post-increment operator (`operator++`). It keeps the character pointed to by the previous value of the iterator for some possible future access to get the character.

1.6.3.2 `istreambuf_iterator` constructors [istreambuf.iterator.cons]

`constexpr istreambuf_iterator() noexcept;`

1 *Effects:* Constructs the end-of-stream iterator.

```
istreambuf_iterator(basic_istream<charT,traits>& s) noexcept;
istreambuf_iterator(basic_streambuf<charT,traits>* s) noexcept;
```

2 *Effects:* Constructs an `istreambuf_iterator<>` that uses the `basic_streambuf<>` object `*(s.rdbuf())`, or `*s`, respectively. Constructs an end-of-stream iterator if `s.rdbuf()` is null.

```
istreambuf_iterator(const proxy& p) noexcept;
```

3 *Effects:* Constructs a `istreambuf_iterator<>` that uses the `basic_streambuf<>` object pointed to by the `proxy` object's constructor argument `p`.

1.6.3.3 `istreambuf_iterator::operator*` [istreambuf.iterator::op*]

```
charT operator*() const
```

1 *Returns:* The character obtained via the `streambuf` member `sbuf_->sgetc()`.

1.6.3.4 `istreambuf_iterator::operator++` [istreambuf.iterator::op++]

```
istreambuf_iterator& operator++();
```

1 *Effects:* `sbuf_->sbumpc()`.

2 *Returns:* `*this`.

```
proxy operator++(int);
```

3 *Returns:* `proxy(sbuf_->sbumpc(), sbuf_)`.

1.6.3.5 `istreambuf_iterator::equal` [istreambuf.iterator::equal]

```
bool equal(const istreambuf_iterator& b) const;
```

1 *Returns:* `true` if and only if both iterators are at end-of-stream, or neither is at end-of-stream, regardless of what `streambuf` object they use.

1.6.3.6 `operator==` [istreambuf.iterator::op==]

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

1 *Returns:* `a.equal(b)`.

1.6.3.7 `operator!=` [istreambuf.iterator::op!=]

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

1 *Returns:* `!a.equal(b)`.

1.6.4 Class template `ostreambuf_iterator` [ostreambuf.iterator]

```
namespace std {
  template <class charT, class traits = char_traits<charT> >
  class ostreambuf_iterator {
  public:
    typedef output_iterator_tag          iterator_category;
    typedef void                         value_type;
    typedef void                         difference_type;
    typedef void                         pointer;
```

```
      typedef void                          reference;
      typedef charT                         char_type;
      typedef traits                        traits_type;
      typedef basic_streambuf<charT,traits> streambuf_type;
      typedef basic_ostream<charT,traits>   ostream_type;

      ostreambuf_iterator(ostream_type& s) noexcept;
      ostreambuf_iterator(streambuf_type* s) noexcept;
      ostreambuf_iterator& operator=(charT c);

      ostreambuf_iterator& operator*();
      ostreambuf_iterator& operator++();
      ostreambuf_iterator& operator++(int);
      bool failed() const noexcept;

    private:
      streambuf_type* sbuf_;                // exposition only
    };
  }
```

1 The class template `ostreambuf_iterator` writes successive *characters* onto the output stream from which it was constructed. It is not possible to get a character value out of the output iterator.

1.6.4.1 `ostreambuf_iterator` constructors [ostreambuf.iter.cons]

```
ostreambuf_iterator(ostream_type& s) noexcept;
```

1 *Requires:* `s.rdbuf()` shall not be a null pointer.

2 *Effects:* Initializes `sbuf_` with `s.rdbuf()`.

```
ostreambuf_iterator(streambuf_type* s) noexcept;
```

3 *Requires:* `s` shall not be a null pointer.

4 *Effects:* Initializes `sbuf_` with `s`.

1.6.4.2 `ostreambuf_iterator` operations [ostreambuf.iter.ops]

```
ostreambuf_iterator& operator=(charT c);
```

1 *Effects:* If `failed()` yields `false`, calls `sbuf_->sputc(c)`; otherwise has no effect.

2 *Returns:* `*this`.

```
ostreambuf_iterator& operator*();
```

3 *Returns:* `*this`.

```
ostreambuf_iterator& operator++();
ostreambuf_iterator& operator++(int);
```

4 *Returns:* `*this`.

```
bool failed() const noexcept;
```

5 *Returns:* `true` if in any prior use of member `operator=`, the call to `sbuf_->sputc()` returned `traits::eof()`; or `false` otherwise.

1.7 Range access [iterator.range]

1 In addition to being available via inclusion of the `<iterator>` header, the function templates in 1.7 are available when any of the following headers are included: `<array>`, `<deque>`, `<forward_list>`, `<list>`, `<map>`, `<regex>`, `<set>`, `<string>`, `<unordered_map>`, `<unordered_set>`, and `<vector>`.

```
template <class C> constexpr auto begin(C& c) -> decltype(c.begin());
template <class C> constexpr auto begin(const C& c) -> decltype(c.begin());
```

2 *Returns:* `c.begin()`.

```
template <class C> constexpr auto end(C& c) -> decltype(c.end());
template <class C> constexpr auto end(const C& c) -> decltype(c.end());
```

3 *Returns:* `c.end()`.

```
template <class T, size_t N> constexpr T* begin(T (&array)[N]) noexcept;
```

4 *Returns:* `array`.

```
template <class T, size_t N> constexpr T* end(T (&array)[N]) noexcept;
```

5 *Returns:* `array + N`.

```
template <class C> constexpr auto cbegin(const C& c) noexcept(noexcept(std::begin(c)))
  -> decltype(std::begin(c));
```

6 *Returns:* `std::begin(c)`.

```
template <class C> constexpr auto cend(const C& c) noexcept(noexcept(std::end(c)))
  -> decltype(std::end(c));
```

7 *Returns:* `std::end(c)`.

```
template <class C> constexpr auto rbegin(C& c) -> decltype(c.rbegin());
template <class C> constexpr auto rbegin(const C& c) -> decltype(c.rbegin());
```

8 *Returns:* `c.rbegin()`.

```
template <class C> constexpr auto rend(C& c) -> decltype(c.rend());
template <class C> constexpr auto rend(const C& c) -> decltype(c.rend());
```

9 *Returns:* `c.rend()`.

```
template <class T, size_t N> constexpr reverse_iterator<T*> rbegin(T (&array)[N]);
```

10 *Returns:* `reverse_iterator<T*>(array + N)`.

```
template <class T, size_t N> constexpr reverse_iterator<T*> rend(T (&array)[N]);
```

11 *Returns:* `reverse_iterator<T*>(array)`.

```
template <class E> constexpr reverse_iterator<const E*> rbegin(initializer_list<E> il);
```

12 *Returns:* `reverse_iterator<const E*>(il.end())`.

```
template <class E> constexpr reverse_iterator<const E*> rend(initializer_list<E> il);
```

13 *Returns:* `reverse_iterator<const E*>(il.begin())`.

```
template <class C> constexpr auto crbegin(const C& c) -> decltype(std::rbegin(c));
```

14 *Returns:* `std::rbegin(c)`.

```
template <class C> constexpr auto crend(const C& c) -> decltype(std::rend(c));
```

15 *Returns:* `std::rend(c)`.

1.8 Container access [iterator.container]

1 In addition to being available via inclusion of the `<iterator>` header, the function templates in 1.8 are available when any of the following headers are included: `<array>`, `<deque>`, `<forward_list>`, `<list>`, `<map>`, `<regex>`, `<set>`, `<string>`, `<unordered_map>`, `<unordered_set>`, and `<vector>`.

```
template <class C> constexpr auto size(const C& c) -> decltype(c.size());
```

2 *Returns:* `c.size()`.

```
template <class T, size_t N> constexpr size_t size(const T (&array)[N]) noexcept;
```

3 *Returns:* `N`.

```
template <class C> constexpr auto empty(const C& c) -> decltype(c.empty());
```

4 *Returns:* `c.empty()`.

```
template <class T, size_t N> constexpr bool empty(const T (&array)[N]) noexcept;
```

5 *Returns:* `false`.

```
template <class E> constexpr bool empty(initializer_list<E> il) noexcept;
```

6 *Returns:* `il.size() == 0`.

```
template <class C> constexpr auto data(C& c) -> decltype(c.data());
template <class C> constexpr auto data(const C& c) -> decltype(c.data());
```

7 *Returns:* `c.data()`.

```
template <class T, size_t N> constexpr T* data(T (&array)[N]) noexcept;
```

8 *Returns:* `array`.

```
template <class E> constexpr const E* data(initializer_list<E> il) noexcept;
```

9 *Returns:* `il.begin()`.