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。凡是定义了相等的迭代器类型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可变迭代器。非可变迭代器也都称为常值迭代器。

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 任何类型的迭代器上的函数都需要实现为（均摊）常数时间。因此，迭代器的需求表都不含复杂度列。

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 Iterator需求是分别迭代器概念的基础，所有迭代器都满足Iterator需求。此组需求规定了迭代器的解引用操作和自增操作。多数算法还需求迭代器能进行读操作(1.2.3)、写操作(1.2.4)或能提供更丰富的迭代器移动操作(1.2.5、1.2.6、1.2.7))。

2 类型X满足Iterator需求，当：

(2.1) — X满足CopyConstructible、CopyAssignable和Destructible(??)需求，并且X类型的左值是可交换的(??)，

(2.2) — 表3中的表达式都成立且语义与表指定的语义相同。

Table 3 — Iterator需求

表达式	返回类型	操作语义	断言/注 前提/后设
*r	未规定		前提：r可解引用。
++r	X&		

1.2.3 输入迭代器 [input.iterators]

1 如果类类型或指针类型X满足Iterator (1.2.2)和EqualityComparable需求(表??)，并能使表4中的表达式都成立且语义与表指定的语义相同，那么X满足值类型T上的输入迭代器的需求。

2 表4中，术语==的域就是在一般的数学层面上表示==（应该）定义在哪些值的集合上，这个集合可能随时不同。特定的算法需要迭代器的==域满足特定的需求，这些需求可能源于该算法用到了依赖==和!=的算法。[例：调用find(a,b,x)仅在a的值具有如下属性p时有定义： b具有属性p且当(*i==x)或(*i!=x且++i具有属性p)时i具有属性p ——结束例]

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。

Table 4 — 输入迭代器需求（在Iterator需求的基础上）

表达式	返回类型	操作 语义	断言/注 前提/后设
`a != b`	可根据上下文转换为`bool`	`!(a == b)`	前提: (`a`，`b`)在`==`的域中。
`*a`	`reference`，可转换为`T`		前提：`a`可解引用。 表达式 `(void)*a`，`*a`等价于`*a`。 若`a == b`且(`a`，`b`)在`==`的域中 则`*a`等价于`*b`。
`a->m`		`(*a).m`	前提：`a`可解引用。
`++r`	`X&`		前提：`r`可解引用。 后设：`r`可解引用或`r`为接尾值。 后设：`r`旧值的任何拷贝都不必再是可解引用的，或不必在`==`的域中。
`(void)r++`			等价于`(void)++r`

`*r++`	可转换为`T`	`{ T tmp = *r;`
		`++r;`
		`return tmp; }`	

3 [注：对于输入迭代器来说，`a == b`并不隐含`++a == ++b`。（Equality does not guarantee the substitution property or referential transparency）输入迭代器上的算法不能通过同一迭代器两次，它们必须是单遍算法。值类型T不必为`CopyAssignable`类型(表??)。可以通过类模板`istream_iterator`将这些算法用于输入数据源是istream的情况。 ——结束注]

1.2.4 输出迭代器 [output.iterators]

1 如果类类型或指针类型`X`满足需求`Iterator`需求(1.2.2)，并能使表4中的表达式都成立且语义与表指定的语义相同，那么`X`满足输出迭代器的需求。

Table 5 — 输出迭代器需求（在Iterator需求的基础上）

表达式	返回类型	操作 语义	断言/注 前提/后设
`*r = o`	值不被使用		备注： 此操作后`r`无需可解引用。 后设：`r`可递增。
`++r`	`X&`		`&r == &++r`。 备注： 此操作后`r`无需可解引用。 后设：`r`可递增。

`r++`	可转换为`const X&`	`{ X tmp = r;`
		`++r;`
		`return tmp; }`
			备注： 此操作后`r`无需可解引用。
			后设：`r`可递增。

`*r++ = o`	值不被使用		备注： 此操作后`r`无需可解引用。 后设：`r`可递增。

2 [注： operator*仅用作赋值语句左边的时候有效。通过同一值的迭代器进行的赋值只能发生一次。输出迭代器上的算法不能通过同一迭代器两次，它们必须是单遍算法。相等和不等未必有定义。可以通过类模板ostream_iterator将适用输出迭代器的算法用于数据输出目标是ostream的情况，适用输出迭代器的算法也可适用插入迭代器和插入指针。 ——结束注]

1.2.5 前向迭代器 [forward.iterators]

1 类类型或指针类型X满足前向迭代器的需求，当

(1.1) — X满足输入迭代器需求(1.2.3)；

(1.2) — X满足DefaultConstructible需求(??),

(1.3) — 如果X是可变迭代器，那么reference是到T的引用； 如果X是常值迭代器，那么reference是到const T的引用；

(1.4) — 能使表6中的表达式都成立且语义与表指定的语义相同；并且

(1.5) — X类型的对象提供下述的多遍担保。

2 前向迭代器的==域就是同一底层序列上的迭代器的==域。另外，做过值初始化的迭代器应当能同相同类型的、并且同样做过值初始化的迭代器相比较并且比较的结果为相等。[注：做过值初始化的迭代器的行为如同它们指向相同的空序列的接尾部分。——结束注]

3 两个可解引用的X类型的迭代器a和b提供多遍担保，当

(3.1) — a == b隐含++a == ++b并且

(3.2) — X是指针类型，或表达式(void)++X(a), *a等价于*a。

4 [注：需求a == b隐含++a == ++b（对于输入和输出迭代器并不成立）并移除可变迭代器上可进行赋值的次数的限制（输出迭代器便有此限制）是为了允许通过前向迭代器使用多遍单向算法。 ——结束注]

Table 6 — 前向迭代器需求（在输入迭代器需求的基础上）

表达式	返回类型	操作 语义	断言/注 前提/后设
r++	可转换为const X&	{ X tmp = r; ++r; return tmp; }	
*r++	reference		

5 如果a和b相等，那么a和b要么都可解引用，要么都不可解引用。

6 如果a和b都可解引用，那么当且仅当*a和*b绑定到同一对象时a == b成立。

1.2.6 双向迭代器 [bidirectional.iterators]

1 如果类类型或指针类型X在满足前向迭代器需求的基础上，能使表7中的表达式都成立，那么X满足双向迭代器的需求。

Table 7 — 双向迭代器需求（在前向迭代器需求的基础上）

表达式	返回类型	操作 语义	断言/注 前提/后设
`--r`	`X&`		前提：存在`s`使得`r == ++s`。 后设：`r`可解引用。 `--(++r) == r`。 `--r == --s`隐含`r == s`。 `&r == &--r`。

`r--`	可转换为`const X&`	`{ X tmp = r;`
		`--r;`
		`return tmp; }`	

`*r--`	`reference`		

2 [注：双向迭代器允许算法同时进行迭代器的前向后向移动。 ——结束注]

1.2.7 随机访问迭代器 [random.access.iterators]

1 如果类类型或指针类型X在满足双向迭代器需求的基础上，能使表8中的表达式都成立，那么X满足随机访问迭代器的需求。

Table 8 — 随机访问迭代器需求（在双向迭代器需求的基础上）

表达式	返回类型	操作 语义	断言/注 前提/后设

`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`	前提：存在`difference_type`类型的值`n`使得`a + n == b`。 `b == a + (b - a)`.
`a[n]`	可转换为`reference`	`*(a + n)`	
`a < b`	可根据上下文转换为`bool`	`b - a > 0`	`<`是全序关系
`a > b`	可根据上下文转换为`bool`	`b < a`	`>`是与`<`相反的全序关系。
`a >= b`	可根据上下文转换为`bool`	`!(a < b)`	

Table 8 — 随机访问迭代器需求（在双向迭代器需求的基础上）(continued)

表达式	返回类型	操作 语义	断言/注 前提/后设
`a <= b`	可根据上下文转换为`bool`.	`!(a > b)`	

1.3 头文件<iterator>概要 [iterator.synopsis]

```
namespace std {
  // 1.4, 原语：
  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, 迭代器操作：
  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, 预定义迭代器：
  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，流迭代器：
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，范围访问：
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, 容器访问:
    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 迭代器原语 [iterator.primitives]

1 为使定义迭代器的工作简化，本库提供了一些类和函数：

1.4.1 迭代器特性 [iterator.traits]

1 为实现仅依赖迭代器的算法，经常需要确定某个特定迭代器类型对应的值类型和距离类型。因此，如果Iterator是迭代器类型，那么需要将类型

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

分别定义为该迭代器的距离类型、值类型以及迭代器类别。此外，类型

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

应该定义为该迭代器的引用类型和指针类型，对于迭代器对象a来说，其引用类型和指针类型分别指*a的类型和a->的类型。 如果是输出迭代器，类型

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

可以定义为void。

2 如果Iterator存在有效的(??)类型成员difference_type、value_type、pointer、reference以及iterator_-category，则iterator_traits<Iterator>应存在并仅存在下列公开可访问成员：

```
  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;
```

否则，iterator_traits<Iterator> 不应该存在成员。

3 这里，指针特化为

```
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;
  };
}
```

指向const的指针特化为

```
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： C++程序实现通用的`reverse`函数的途径之一是：

```
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 基本迭代器 [iterator.basic]

1 `iterator`模板可以作为新建迭代器的基类减少其定义必要类型的工作量。

```
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 标准迭代器标签 [std.iterator.tags]

1 函数模板在特化过程中经常希望获取其迭代器参数的most specific类别以便于在编译时为该函数选择最高效的算法。为方便该功能，本库引入类别标签类用于编译时算法选择。 它们是：`input_iterator_-`

tag、output_iterator_tag、forward_iterator_tag、bidirectional_iterator_tag以及random_access_-iterator_tag。iterator_traits<Iterator>::iterator_category应该为所有Iterator类型的迭代器定义描述该迭代器行为的most specific类别标签。

```
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：程序定义的迭代器BinaryTreeIterator可通过特化iterator_traits模板将该迭代器包含在双向迭代器类别中：

```
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;
};
```

不过典型地，更简单的方法是从iterator<bidirectional_iterator_tag,T,ptrdiff_t,T*,T&>继承BinaryTreeIterator<
——结束例]

3 [例：如果双向迭代器定义了evolve()，但随机访问迭代器上的evolve()可以实现得更加高效，那么可以如下实现：

```
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) {
  // 更通用，但不是最高效的
}

template <class RandomAccessIterator>
void evolve(RandomAccessIterator first, RandomAccessIterator last,
  random_access_iterator_tag) {
  // 更高效，但不是最通用的
}
```

——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 这里就无需特化iterator_traits模板。 ——结束例]

1.4.4 迭代器操作 [iterator.operations]

1 由于只有随机访问迭代器提供+和-操作符， 本库提供了两个函数模板advance和distance。这些函数模板对随机访问迭代器使用+和-（并且因此对于随机访问迭代器来说是常量时间）；对输入、前向和双向迭代器使用++提供线性时间的实现。

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

2 需要： n仅在双向迭代器和随机访问迭代器的情况下能为负值。

3 效果：将迭代器引用i递增(对于n是负值来说则是递减)n次。

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

4 效果：如果InputIterator符合随机访问迭代器的需求，返回(last - first)；否则返回first递增到last需要的次数。

5 需要：如果InputIterator符合随机访问迭代器的需求，则first应可达last或last可达first；否则first应可达last。

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

6 效果：等价于advance(x, n); return x;

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

7 效果：等价于advance(x, -n); return x;

1.5 迭代器适配器 [predef.iterators]

1.5.1 反向迭代器 [reverse.iterators]

1 类模板reverse_iterator是一种迭代器适配器，它的遍历顺序是从它的底层迭代器定义的序列尾遍历到序列头。反向迭代器和它对应的迭代器i之间的基本关系建立在标志：&*(reverse_iterator(i)) == &*(i - 1)上。

1.5.1.1 reverse_iterator类模板 [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需求 [reverse.iter.requirements]

1 模板参数Iterator应符合双向迭代器(1.2.6)的所有需求。

2 另外，当Iterator的以下成员： 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) 中的任何一个或者全局操作符 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)或 operator+ (1.5.1.3.20) 中的任何一个被引用而需要实例化(??)时， Iterator应符合随机访问迭代器(1.2.7)的需求。

1.5.1.3 reverse_iterator操作 [reverse.iter.ops]

1.5.1.3.1 reverse_iterator构造函数 [reverse.iter.cons]

```
constexpr reverse_iterator();
```

1 效果：将current做值初始化。作用在结果迭代器上的迭代器操作行为有定义，当且仅当做过值初始化的Iterator类型的迭代器在对应的操作上有定义。

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

2 效果：将current初始化为x。

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

3 效果：将current初始化为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 效果：将u.base()赋值给current。

2 返回： *this.

1.5.1.3.3 转换 [reverse.iter.conv]

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

1 返回： current.

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

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

1 效果：

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

1.5.1.3.5 operator-> [reverse.iter.opref]

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

1 返回： std::addressof(operator*()).

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

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

1 效果： --current;

2 返回： *this.

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

3 效果：

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

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

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

1 效果： `++current`

2 返回： `*this`.

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

3 效果：

```
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 返回： `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 效果： `current -= n;`

2 返回： `*this`.

1.5.1.3.10 operator- [reverse.iter.op-]

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

1 返回： `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 效果： `current += n;`

2 返回： `*this`.

1.5.1.3.12 operator[] [reverse.iter.opindex]

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

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

1.5.1.3.21 非成员函数make_reverse_iterator() [reverse.iter.make]

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

1 返回： `reverse_iterator<Iterator>(i)`.

1.5.2 插入迭代器 [insert.iterators]

1 为了令插入操作和写入数组操作写法相仿，本库提供了一种特殊的迭代器适配器，称为插入迭代器。对于普通的迭代器类来说，

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

会导致范围`[first，last)`拷贝至result开始的范围。 同一份代码，如果`result`是插入迭代器，则会将对应的元素插入到容器。 这种手段允许本库所有拷贝算法以插入模式而不是常规覆盖模式工作。

2 构造插入迭代器需要指定一个容器，如果要插入的地方既不是容器的头部又不是容器的尾部，可以在构造时指定指向容器要插入的地方的迭代器。插入迭代器满足输出迭代器的需求。`operator*`返回插入迭代器自身。插入迭代器定义了赋值操作`operator=(const T& x)`以用来向它们写入，该赋值操作将x插入在插入迭代器指向的地方前。换句话说，插入迭代器就像光标一样指向容器将要插入的位置。`back_insert_iterator`将元素插入容器的尾部，`front_insert_iterator`将元素插入容器的头部，`insert_iterator`将元素插入其指向的容器位置。`back_inserter`、`front_inserter`和`inserter`是三个脱离容器构造插入迭代器的函数。

1.5.2.1 类模板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操作 [back.insert.iter.ops]

1.5.2.2.1 back_insert_iterator构造函数 [back.insert.iter.cons]

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

1 效果：以std::addressof(x)初始化container。

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

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

1 效果： container->push_back(value);

2 返回： *this。

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

3 效果： container->push_back(std::move(value));

4 返回： *this。

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

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

1 返回： *this。

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

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

1 返回： *this。

1.5.2.2.5 back_inserter [back.inserter]

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

1 返回： back_insert_iterator<Container>(x)。

1.5.2.3 类模板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操作 [front.insert.iter.ops]

1.5.2.4.1 front_insert_iterator构造函数 [front.insert.iter.cons]

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

1 效果：以std::addressof(x)初始化container。

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

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

1 效果： container->push_front(value);

2 返回： *this。

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

3 效果： container->push_front(std::move(value));

4 返回： *this。

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

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

1 返回： *this。

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

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

1 返回： *this。

1.5.2.4.5 front_inserter [front.inserter]

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

1 返回： front_insert_iterator<Container>(x)。

1.5.2.5 类模板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操作 [insert.iter.ops]

1.5.2.6.1 insert_iterator构造函数 [insert.iter.cons]

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

1 效果：以std::addressof(x)初始化container，以i初始化iter。

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

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

1 效果：

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

2 返回：*this。

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

3 效果：

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

4 返回：*this。

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

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

1 返回：*this。

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

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

1 返回：*this。

1.5.2.6.5 inserter [inserter]

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

1 返回： insert_iterator<Container>(x, i)。

1.5.3 移动迭代器 [move.iterators]

1 类模板move_iterator是一种迭代器适配器，它和其底层迭代器的行为相同但其间接操作符将其底层迭代器的间接操作符返回的值隐式转换为右值。为某些通用算法调用传递移动迭代器可以将拷贝操作改为移动操作。

2 [例：

```
list<string> s;
// 稳化链表s
vector<string> v1(s.begin(), s.end());          // 将字符串拷贝至v1
vector<string> v2(make_move_iterator(s.begin()),
                  make_move_iterator(s.end())); // 将字符串移动至v2
```

——结束例]

1.5.3.1 类模板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 令R为`iterator_traits<Iterator>::reference`。如果`is_reference<`R`>::value`为`true`，则模板特化`move_iterator<Iterator>`应该定义名为`reference`的内嵌类型作为`remove_reference<`R`>::type&&`的同义词，否则应该作为R的同义词。

1.5.3.2 move_iterator需求 [move.iter.requirements]

1 模板参数`Iterator`应该符合输入迭代器的需求(1.2.3)。此外，如果实例化了任何双向遍历或随机访问遍历函数，模板参数`Iterator`还应分别符合双向迭代器(1.2.6)或随机访问器(1.2.7)的需求。

1.5.3.3 move_iterator操作 [move.iter.ops]

1.5.3.3.1 move_iterator构造函数 [move.iter.op.const]

```
constexpr move_iterator();
```

1 效果：构造`move_iterator`，将`current`做值初始化。作用在结果迭代器上的迭代器操作行为有定义，当且仅当做过值初始化的`Iterator`类型的迭代器在对应的操作上有定义。

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

2 效果：构造`move_iterator`，以`i`初始化`current`。

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

3 效果：构造`move_iterator`，以`u.base()`初始化`current`。

4 需要：`U`应能转换为`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 效果：将u.base()赋值给current.

2 需要： U shall be convertible to Iterator.

1.5.3.3.3 move_iterator转换 [move.iter.op.conv]

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

1 返回： current.

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

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

1 返回： static_cast<reference>(*current).

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

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

1 返回： current.

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

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

1 效果： ++current.

2 返回： *this.

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

3 效果：

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

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

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

1 效果： --current.

2 返回： *this.

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

3 效果：

```
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 返回： move_iterator(current + n).

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

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

1 效果： `current += n`.

2 返回： `*this`.

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

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

1 返回： `move_iterator(current - n)`.

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

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

1 效果： `current -= n`.

2 返回： `*this`.

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

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

1 返回： `std::move`(current[n]).

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

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

1 返回： `x.base() == y.base()`.

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

2 返回： `!(x == y)`.

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

3 返回： `x.base() < y.base()`.

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

4 返回： `!(y < x)`.

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

5 返回： `y < x`.

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

6 返回： `!(x < y)`.

1.5.3.3.14 move_iterator非成员函数 [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 返回： `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 返回： `x + n`.

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

3 返回： `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 效果： 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 效果： 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 效果： 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 效果： 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 返回： *value*.

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

2 返回： `addressof(operator*())`.

```
istream_iterator& operator++();
```

3 需要： `in_stream != 0`.

4 效果： `*in_stream >>value`.

5 返回： `*this`.

```
istream_iterator operator++(int);
```

6 需要： `in_stream != 0`.

7 效果：

```
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 返回： `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 返回： `!(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 效果： Initializes *out_stream* with `addressof(s)` and *delim* with null.

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

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

`ostream_iterator(const ostream_iterator& x);`

3 效果： Constructs a copy of `x`.

`~ostream_iterator();`

4 效果： The iterator is destroyed.

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

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

1 效果：

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

`ostream_iterator& operator*();`

2 返回： `*this`.

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

3 返回： `*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 效果： Constructs the end-of-stream iterator.

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

2 效果： 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 效果： 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 返回： The character obtained via the streambuf member sbuf_->sgetc().

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

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

1 效果： sbuf_->sbumpc().

2 返回： *this.

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

3 返回： proxy(sbuf_->sbumpc(), sbuf_).

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

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

1 返回： 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 返回： 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 返回： !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 需要： `s.rdbuf()` shall not be a null pointer.

2 效果： Initializes `sbuf_` with `s.rdbuf()`.

`ostreambuf_iterator(streambuf_type* s) noexcept;`

3 需要： `s` shall not be a null pointer.

4 效果： Initializes `sbuf_` with `s`.

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

`ostreambuf_iterator& operator=(charT c);`

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

2 返回： `*this`.

`ostreambuf_iterator& operator*();`

3 返回： `*this`.

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

4 返回： *this.

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

5 返回： 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 返回： 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 返回： c.end().

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

4 返回： array.

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

5 返回： array + N.

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

6 返回： std::begin(c).

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

7 返回： 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 返回： 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 返回： c.rend().

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

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

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

11 返回： reverse_iterator<T*>(array).

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

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

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

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

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

14 返回： `std::rbegin(c)`.

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

15 返回： `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 返回： `c.size()`.

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

3 返回： `N`.

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

4 返回： `c.empty()`.

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

5 返回： `false`.

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

6 返回： `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 返回： `c.data()`.

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

8 返回： `array`.

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

9 返回： `il.begin()`.