The back_insert_iterator transforms iterator writes into calls to the container's push_back, whereas front_insert_iterator calls to push_front. Both of these insert iterators expose a single constructor that accepts a container reference, and their corresponding convenience functions take a single argument. Obviously, the wrapped container must implement the appropriate method. For example, a vector won't work with a front_insert_iterator, and a set won't work with either of them.

The insert_iterator takes two constructor arguments: a container to wrap and an iterator pointing into a position in that container. The insert_iterator then transforms writes into calls to the container's insert method, and it will pass the position you provided on construction as the first argument. For example, you use the insert_iterator to insert into the middle of a sequential container or to add elements into a set with a hint.

NOTE *Internally, all the insert iterators completely ignore* operator++, operator++(int), *and* operator*. *Containers don't need this intermediate step between insertions, but it's generally a requirement for output iterators.*

Listing 14-1 illustrates the basic usages of the three insert iterators by adding elements to a deque.

```
#include <deque>
#include <iterator>

TEST_CASE("Insert iterators convert writes into container insertions.") {
  std::deque<int> dq;
  auto back_instr = std::back_inserter(dq); ❶
  *back_instr = 2; ❷ // 2
  ++back_instr; ❸
  *back_instr = 4; ❹ // 2 4
  ++back_instr;

  auto front_instr = std::front_inserter(dq); ❺
  *front_instr = 1; ❻ // 1 2 4
  ++front_instr;

  auto instr = std::inserter(dq, dq.begin()+2); ❼
  *instr = 3; ❽ // 1 2 3 4
  instr++;

  REQUIRE(dq[0] == 1);
  REQUIRE(dq[1] == 2);
  REQUIRE(dq[2] == 3);
  REQUIRE(dq[3] == 4); ❾
}
```

Listing 14-1: Insert iterators convert writes into container insertions.

First, you build a back_insert_iterator with back_inserter to wrap a deque called dq ❶. When you write into the back_insert_iterator, it translates the write into a push_back, so the deque contains a single element, 2 ❷. Because output iterators require incrementing before you can write again, you follow with an increment ❸. When you write `4` to the `back_insert_iterator`, it again translates the write into a `push_back` so the `deque` contains the elements `2 4` ❹.

Next, you build a `front_insert_iterator` with `front_inserter` to wrap `dq` ❺. Writing `1` into this newly constructed inserter results in a call to `push_front`, so the deque contains the elements `1 2 4` ❻.

Finally, you build an `insert_iterator` with `inserter` by passing `dq` and an iterator pointing to its third element (`4`). When you write 3 into this inserter ❽, it inserts just before the element pointed to by the iterator you passed at construction ❼. This results in `dq` containing the elements `1 2 3 4` ❾.

Table 14-1 summarizes the insert iterators.

Table 14-1: Summary of Insert Iterators

Class	Convenience function	Delegated function	Example containers
`back_insert_iterator`	`back_inserter`	`push_back`	vectors, deques, lists
`front_insert_iterator`	`front_inserter`	`push_front`	deques, lists
`insert_iterator`	`inserter`	`insert`	vectors, deques, lists, sets

List of Supported Output Iterator Operations

Table 14-2 summarizes the output iterator's supported operations.

Table 14-2: Output Iterator's Supported Operations

Operation	Notes
`*itr=t`	Writes into the output iterator. After operation, iterator is incrementable but not necessarily dereferencable.
`++itr` `itr++`	Increments the iterator. After operation, iterator is either dereferencable or exhausted (past the end) but is not necessarily incrementable.
`iterator-type``{ itr }`	Copy-constructs an iterator from `itr`.

Input Iterators

You can use an *input iterator* to read from, increment, and check for equality. It's the foil to the output iterator. You can only iterate through an input iterator once.

The usual pattern when reading from an input iterator is to obtain a half-open range with a *begin* and an *end* iterator. To read through the range, you read the `begin` iterator using `operator*` followed by an increment with `operator++`. Next, you evaluate whether the iterator equals `end`. If it does, you've exhausted the range. If it doesn't, you can continue reading/incrementing.

NOTE *Input iterators are the magic that makes the range expressions discussed in "Range-Based* `for` *Loops" on page 234 work.*

A canonical usage of an input iterator is to wrap a program's standard input (usually the keyboard). Once you've read a value from standard input, it's gone. You cannot go back to the beginning and replay. This behavior matches an input iterator's supported operations really well.

In "A Crash Course in Iterators" on page 412, you learned that every container exposes iterators with begin/cbegin/end/cend methods. All of these methods are *at least* input iterators (and they might support additional functionality). For example, Listing 14-2 illustrates how to extract a range from a forward_list and manipulate the iterators manually for reading.

```
#include <forward_list>

TEST_CASE("std::forward_list begin and end provide input iterators") {
  const std::forward_list<int> easy_as{ 1, 2, 3 }; ❶
  auto itr = easy_as.begin(); ❷
  REQUIRE(*itr == 1); ❸
  itr++; ❹
  REQUIRE(*itr == 2);
  itr++;
  REQUIRE(*itr == 3);
  itr++;
  REQUIRE(itr == easy_as.end()); ❺
}
```

Listing 14-2: Interacting with input iterators from a forward_list

You create a forward_list containing three elements ❶. The container's constness means the elements are immutable, so the iterators support only read operations. You extract an iterator with the begin method of forward_list ❷. Using operator*, you extract the element pointed to by itr ❸ and follow up with the obligatory incrementation ❹. Once you've exhausted the range by reading/incrementing, itr equals the end of the forward_list ❺.

Table 14-3 summarizes the input iterator's supported operations.

Table 14-3: Input Iterator's Supported Operations

Operation	Notes
***itr**	Dereferences the pointed-to member. Might or might not be read-only.
itr->mbr	Dereferences the member **mbr** of the object pointed-to by **itr**.
++itr **itr++**	Increments the iterator. After operation, iterator is either dereferencable or exhausted (past the end).
itr1 == itr2 **itr1 != itr2**	Compares whether the iterators are equal (pointing to the same element).
***iterator-type*{ itr }**	Copy-constructs an iterator from **itr**.

Forward Iterators

A *forward iterator* is an input iterator with additional features: a forward iterator can also traverse multiple times, default construct, and copy assign. You can use a forward iterator in place of an input iterator in all cases.

All STL containers provide forward iterators. Accordingly, the `forward_list` used in Listing 14-2 actually provides a forward iterator (which is also an input iterator).

Listing 14-3 updates Listing 14-2 to iterate over the `forward_list` multiple times.

```
TEST_CASE("std::forward_list's begin and end provide forward iterators") {
  const std::forward_list<int> easy_as{ 1, 2, 3 }; ❶
  auto itr1 = easy_as.begin(); ❷
  auto itr2{ itr1 }; ❸
  int double_sum{};
  while (itr1 != easy_as.end()) ❹
    double_sum += *(itr1++);
  while (itr2 != easy_as.end()) ❺
    double_sum += *(itr2++);
  REQUIRE(double_sum == 12); ❻
}
```

Listing 14-3: Traversing a forward iterator twice

Again you create a `forward_list` containing three elements ❶. You extract an iterator called `itr1` with the `begin` method of `forward_list` ❷, then create a copy called `itr2` ❸. You exhaust `itr1` ❹ and `itr2` ❺, iterating over the range twice while summing both times. The resulting `double_sum` equals 12 ❻.

Table 14-4 summarizes the forward iterator's supported operations.

Table 14-4: Forward Iterator's Supported Operations

Operation	Notes
`*itr`	Dereferences the pointed-to member. Might or might not be read-only.
`itr->mbr`	Dereferences the member `mbr` of the object pointed-to by `itr`.
`++itr` `itr++`	Increments the iterator so it points to the next element.
`itr1 == itr2` `itr1 != itr2`	Compares whether the iterators are equal (pointing to the same element).
`iterator-type``{}`	Default constructs an iterator.
`iterator-type``{ itr }`	Copy-constructs an iterator from `itr`.
`itr1 = itr2`	Assigns an iterator `itr1` from `itr2`.

Bidirectional Iterators

A *bidirectional iterator* is a forward iterator that can also iterate backward. You can use a bidirectional iterator in place of a forward or input iterator in all cases.

Bidirectional iterators permit backward iteration with `operator--` and `operator–(int)`. The STL containers that provide bidirectional iterators are `array`, `list`, `deque`, `vector`, and all of the ordered associative containers.

Listing 14-4 illustrates how to iterate in both directions using the bidirectional iterator of `list`.

```
#include <list>

TEST_CASE("std::list begin and end provide bidirectional iterators") {
  const std::list<int> easy_as{ 1, 2, 3 }; ❶
  auto itr = easy_as.begin(); ❷
  REQUIRE(*itr == 1); ❸
  itr++; ❹
  REQUIRE(*itr == 2);
  itr--; ❺
  REQUIRE(*itr == 1); ❻
  REQUIRE(itr == easy_as.cbegin());
}
```

Listing 14-4: The `std::list` methods `begin` and `end` provide bidirectional iterators.

Here, you create a `list` containing three elements ❶. You extract an iterator called `itr` with the `begin` method of `list` ❷. As with the input and forward iterators, you can dereference ❸ and increment ❹ the iterator. Additionally, you can decrement the iterator ❺ so you can go back to elements you've already iterated over ❻.

Table 14-5 summarizes a bidirectional iterator's supported operations.

Table 14-5: Bidirectional Iterator's Supported Operations

Operation	Notes
`*itr`	Dereferences the pointed-to member. Might or might not be read-only.
`itr->mbr`	Dereferences the member `mbr` of the object pointed to by `itr`.
`++itr` `itr++`	Increments the iterator so it points to the next element.
`--itr` `itr--`	Decrements the iterator so it points to the previous element.
`itr1 == itr2` `itr1 != itr2`	Compares whether the iterators are equal (pointing to the same element).
`iterator-type``{}`	Default constructs an iterator.
`iterator-type``{ itr }`	Copy-constructs an iterator from `itr`.
`itr1 = itr2`	Assigns an iterator `itr1` from `itr2`.

Random-Access Iterators

A *random-access iterator* is a bidirectional iterator that supports random element access. You can use a random-access iterator in place of bidirectional, forward, and input iterators in all cases.

Random-access iterators permit random access with `operator[]` and also iterator arithmetic, such as adding or subtracting integer values and subtracting other iterators to find distances. The STL containers that provide random-access iterators are array, vector, and deque. Listing 14-5 illustrates how to access arbitrary elements using a random-access iterator from a vector.

```
#include <vector>

TEST_CASE("std::vector begin and end provide random-access iterators") {
  const std::vector<int> easy_as{ 1, 2, 3 }; ❶
  auto itr = easy_as.begin(); ❷
  REQUIRE(itr[0] == 1); ❸
  itr++; ❹
  REQUIRE(*(easy_as.cbegin() + 2) == 3); ❺
  REQUIRE(easy_as.cend() - itr == 2); ❻
}
```

Listing 14-5: Interacting with a random-access iterator

You create a vector containing three elements ❶. You extract an iterator called itr with the begin method of vector ❷. Because this is a random-access iterator, you can use operator[] to dereference arbitrary elements ❸. Of course, you can still increment the iterator using operator++ ❹. You can also add to or subtract from an iterator to access elements at a given offset ❺❻.

List of Supported Random-Access Iterator Operations

Table 14-6 summarizes the random-access iterator's supported operations.

Table 14-6: Random-Access Iterator's Supported Operations

Operation	Notes
itr[**n**]	Dereferences the element with index **n**.
itr+**n** **itr**-**n**	Returns the iterator at offset **n** from **itr**.
itr2-**itr1**	Computes the distance between **itr1** and **itr2**.
***itr**	Dereferences the pointed-to member. Might or might not be read-only.
itr->**mbr**	Dereferences the member **mbr** of the object pointed to by **itr**.
++**itr** **itr**++	Increments the iterator so it points to the next element.
--**itr** **itr**--	Decrements the iterator so it points to the previous element.
itr1 == **itr2** **itr1** != **itr2**	Compares whether the iterators are equal (pointing to the same element).
iterator-type{}	Default constructs an iterator.
iterator-type{ **itr** }	Copy-constructs an iterator from **itr**.
itr1 < **itr2** **itr1** > **itr2** **itr1** <= **itr2** **itr1** >= **itr2**	Performs the corresponding comparison to the iterators' positions.

Contiguous Iterators

A *contiguous iterator* is a random-access iterator with elements adjacent in memory. For a contiguous iterator `itr`, all elements `itr[n]` and `itr[n+1]` satisfy the following relation for all valid selections of indices `n` and offsets `i`:

```
&itr[n] + i == &itr[n+i]
```

The `vector` and `array` containers provide contiguous iterators, but `list` and `deque` don't.

Mutable Iterators

All forward iterators, bidirectional iterators, random-access iterators, and contiguous iterators can support read-only or read-and-write modes. If an iterator supports read and write, you can assign values to the references returned by dereferencing an iterator. Such iterators are called *mutable iterators*. For example, a bidirectional iterator that supports reading and writing is called a mutable bidirectional iterator.

In each of the examples so far, the containers used to underpin the iterators have been `const`. This produces iterators to `const` objects, which are of course not writable. Listing 14-6 extracts a mutable, random-access iterator from a (non-`const`) `deque`, allowing you to write into arbitrary elements of the container.

```
#include <deque>

TEST_CASE("Mutable random-access iterators support writing.") {
  std::deque<int> easy_as{ 1, 0, 3 }; ❶
  auto itr = easy_as.begin(); ❷
  itr[1] = 2; ❸
  itr++; ❹
  REQUIRE(*itr == 2); ❺
}
```

Listing 14-6: A mutable random-access iterator permits writing.

You construct a `deque` containing three elements ❶ and then obtain an iterator pointing to the first element ❷. Next, you write the value 2 to the second element ❸. Then, you increment the iterator so it points to the element you just modified ❹. When you dereference the pointed-to element, you get back the value you wrote in ❺.

Figure 14-1 illustrates the relationship between the input iterator and all its more featureful descendants.

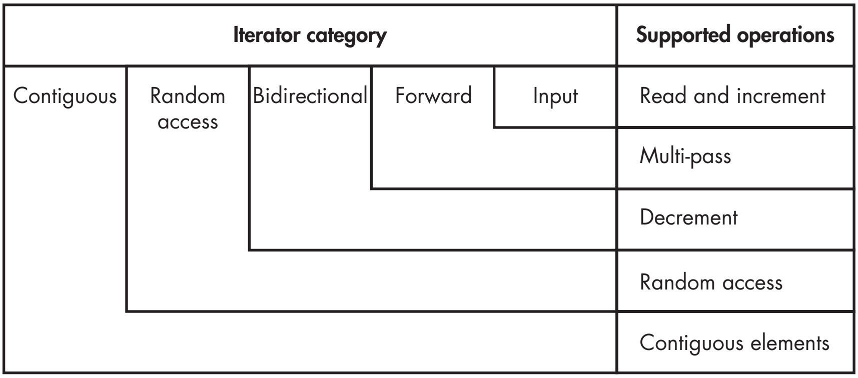


Figure 14-1: Input iterator categories and their nested relationships

To summarize, the input iterator supports only read and increment. Forward iterators are also input iterators, so they also support read and increment but additionally allow you to iterate over their range multiple times ("multi-pass"). Bidirectional iterators are also forward iterators, but they additionally permit decrement operations. Random access iterators are also bidirectional iterators, but you can access arbitrary elements in the sequence directly. Finally, contiguous iterators are random-access iterators that guarantee their elements are contiguous in memory.

Auxiliary Iterator Functions

If you write generic code dealing with iterators, you should use *auxiliary iterator functions* from the `<iterator>` header to manipulate iterators rather than using the supported operations directly. These iterator functions perform common tasks of traversing, swapping, and computing distances between iterators. The major advantage of using the auxiliary functions instead of direct iterator manipulation is that the auxiliary functions will inspect an iterator's type traits and determine the most efficient method for performing the desired operation. Additionally, auxiliary iterator functions make generic code even more generic because it will work with the widest range of iterators.

std::advance

The `std::advance` auxiliary iterator function allows you to increment or decrement by the desired amount. This function template accepts an iterator reference and an integer value corresponding to the distance you want to move the iterator:

```
void std::advance(InputIterator&❶ itr, Distance❷ d);
```

The `InputIterator` template parameter must be at least an input iterator ❶, and the `Distance` template parameter is usually an integer ❷.

The advance function doesn't perform bounds checking, so you must ensure that you've not exceeded the valid range for the iterator's position.

Depending on the iterator's category, advance will perform the most efficient operation that achieves the desired effect:

Input iterator The advance function will invoke itr++ the correct number of times; dist cannot be negative.

Bidirectional iterator The function will invoke itr++ or itr-- the correct number of times.

Random access iterator It will invoke itr+=dist; dist can be negative.

NOTE *Random-access iterators will be more efficient than lesser iterators with* advance*, so you might want to use* operator+= *instead of* advance *if you want to forbid the worst-case (linear-time) performance.*

Listing 14-7 illustrates how to use advance to manipulate a random-access iterator.

```
#include <iterator>

TEST_CASE("advance modifies input iterators") {
  std::vector<unsigned char> mission{ ❶
    0x9e, 0xc4, 0xc1, 0x29,
    0x49, 0xa4, 0xf3, 0x14,
    0x74, 0xf2, 0x99, 0x05,
    0x8c, 0xe2, 0xb2, 0x2a
  };
  auto itr = mission.begin(); ❷
  std::advance(itr, 4); ❸
  REQUIRE(*itr == 0x49);
  std::advance(itr, 4); ❹
  REQUIRE(*itr == 0x74);
  std::advance(itr, -8); ❺
  REQUIRE(*itr == 0x9e);
}
```

Listing 14-7: Using advance *to manipulate a contiguous iterator*

Here, you initialize a vector called mission with 16 unsigned char objects ❶. Next, you extract an iterator called itr using the begin method of mission ❷ and invoke advance on itr to advance four elements so it points at the fourth element (with value 0x49) ❸. You advance again four elements to the eighth element (with value 0x74) ❹. Finally, you invoke advance with –8 to retreat eight values, so the iterator again points to the first element (with value 0x9e) ❺.

std::next and std::prev

The std::next and std::prev auxiliary iterator functions are function templates that compute offsets from a given iterator. They return a new iterator pointing to the desired element without modifying the original iterator, as demonstrated here:

```
ForwardIterator std::next(ForwardIterator& itr❶, Distance d=1❷);
BidirectionalIterator std::prev(BidirectionalIterator& itr❸, Distance d=1❹);
```

The function `next` accepts at least a forward iterator ❶ and optionally a distance ❷, and it returns an iterator pointing to the corresponding offset. This offset can be negative if `itr` is bidirectional. The `prev` function template works like `next` in reverse: it accepts at least a bidirectional iterator ❸ and optionally a distance ❹ (which can be negative).

Neither `next` nor `prev` performs bounds checking. This means you must be absolutely sure that your math is correct and that you're staying within the sequence; otherwise, you'll get undefined behavior.

NOTE *For both* `next` *and* `prev`*,* `itr` *remains unchanged unless it's an rvalue, in which case* `advance` *is used for efficiency.*

Listing 14-8 illustrates how to use `next` to obtain a new iterator pointing to the element at a given offset.

```
#include <iterator>

TEST_CASE("next returns iterators at given offsets") {
  std::vector<unsigned char> mission{
    0x9e, 0xc4, 0xc1, 0x29,
    0x49, 0xa4, 0xf3, 0x14,
    0x74, 0xf2, 0x99, 0x05,
    0x8c, 0xe2, 0xb2, 0x2a
  };
  auto itr1 = mission.begin(); ❶
  std::advance(itr1, 4); ❷
  REQUIRE(*itr1 == 0x49); ❸

  auto itr2 = std::next(itr1); ❹
  REQUIRE(*itr2 == 0xa4); ❺

  auto itr3 = std::next(itr1, 4); ❻
  REQUIRE(*itr3 == 0x74); ❼

  REQUIRE(*itr1 == 0x49); ❽
}
```

Listing 14-8: Using next to obtain offsets from an iterator

As in Listing 14-7, you initialize a vector containing 16 `unsigned chars` and extract an iterator `itr1` pointing to the first element ❶. You use `advance` to increment the iterator four elements ❷ so it points to the element with the value 0x49 ❸. The first use of `next` omits a distance argument, which defaults to 1 ❹. This produces a new iterator, `itr2`, which is one past `itr1` ❺.

You invoke next a second time with a distance argument of 4 ❻. This produces another new iterator, itr3, which points to four past the element of itr1 ❼. Neither of these invocations affects the original iterator itr1 ❽.

std::distance

The std::distance auxiliary iterator function enables you to compute the distance between two input iterators itr1 and itr2:

```
Distance std::distance(InputIterator itr1, InputIterator itr2);
```

If the iterators are not random access, itr2 must refer to an element after itr1. It's a good idea to ensure that itr2 comes after itr1, because you'll get undefined behavior if you accidentally violate this requirement and the iterators are not random access.

Listing 14-9 illustrates how to compute the distance between two random access iterators.

```
#include <iterator>

TEST_CASE("distance returns the number of elements between iterators") {
  std::vector<unsigned char> mission{ ❶
    0x9e, 0xc4, 0xc1, 0x29,
    0x49, 0xa4, 0xf3, 0x14,
    0x74, 0xf2, 0x99, 0x05,
    0x8c, 0xe2, 0xb2, 0x2a
  };
  auto eighth = std::next(mission.begin(), 8); ❷
  auto fifth = std::prev(eighth, 3); ❸
  REQUIRE(std::distance(fifth, eighth) == 3); ❹
}
```

Listing 14-9: Using distance to obtain the distance between iterators

After initializing your vector ❶, you create an iterator pointing to the eighth element using std::next ❷. You use std::prev on eighth to obtain an iterator to the fifth element by passing 3 as the second argument ❸. When you pass fifth and eighth as the arguments to distance, you get 3 ❹.

std::iter_swap

The std::iter_swap auxiliary iterator function allows you to swap the values pointed to by two forward iterators itr1 and itr2:

```
Distance std::iter_swap(ForwardIterator itr1, ForwardIterator itr2);
```

The iterators don't need to have the same type, as long as their pointed-to types are assignable to one another. Listing 14-10 illustrates how to use iter_swap to exchange two vector elements.

```
#include <iterator>

TEST_CASE("iter_swap swaps pointed-to elements") {
  std::vector<long> easy_as{ 3, 2, 1 }; ❶
  std::iter_swap(easy_as.begin()❷, std::next(easy_as.begin(), 2)❸);
  REQUIRE(easy_as[0] == 1); ❹
  REQUIRE(easy_as[1] == 2);
  REQUIRE(easy_as[2] == 3);
}
```

Listing 14-10: Using `iter_swap` to exchange pointed-to elements

After you construct a `vector` with the elements `3 2 1` ❶, you invoke `iter_swap` on the first element ❷ and the last element ❸. After the exchange, the `vector` contains the elements `1 2 3` ❹.

Additional Iterator Adapters

In addition to insert iterators, the STL provides move iterator adapters and reverse iterator adapters to modify iterator behavior.

NOTE *The STL also provides stream iterator adapters, which you'll learn about in Chapter 16 alongside streams.*

Move Iterator Adapters

A *move iterator adapter* is a class template that converts all iterator accesses into move operations. The convenience function template `std::make_move_iterator` in the `<iterator>` header accepts a single iterator argument and returns a move iterator adapter.

The canonical use of a move iterator adapter is to move a range of objects into a new container. Consider the toy class `Movable` in Listing 14-11, which stores an `int` value called `id`.

```
struct Movable{
  Movable(int id) : id{ id } { } ❶
  Movable(Movable&& m) {
    id = m.id; ❷
    m.id = -1; ❸
  }
  int id;
};
```

Listing 14-11: The `Movable` class stores an `int`.

The `Movable` constructor takes an `int` and stores it into its `id` field ❶. Movable is also move constructible; it will steal the `id` from its move-constructor argument ❷, replacing it with –1 ❸.

Listing 14-12 constructs a `vector` of `Movable` objects called `donor` and moves them into a `vector` called `recipient`.

```
#include <iterator>

TEST_CASE("move iterators convert accesses into move operations") {
  std::vector<Movable> donor; ❶
  donor.emplace_back(1); ❷
  donor.emplace_back(2);
  donor.emplace_back(3);
  std::vector<Movable> recipient{
    std::make_move_iterator(donor.begin()), ❸
    std::make_move_iterator(donor.end()),
  };
  REQUIRE(donor[0].id == -1); ❹
  REQUIRE(donor[1].id == -1);
  REQUIRE(donor[2].id == -1);
  REQUIRE(recipient[0].id == 1); ❺
  REQUIRE(recipient[1].id == 2);
  REQUIRE(recipient[2].id == 3);
}
```

Listing 14-12: Using the move iterator adapter to convert iterator operations into move operations

Here, you default construct a vector called donor ❶, which you use to emplace_back three Movable objects with id fields 1, 2, and 3 ❷. You then use the range constructor of vector with the begin and end iterators of donor, which you pass to make_move_iterator ❸. This converts all iterator operations into move operations, so the move constructor of Movable gets called. As a result, all the elements of donor are in a moved-from state ❹, and all the elements of recipient match the previous elements of donor ❺.

Reverse Iterator Adapters

A *reverse iterator adapter* is a class template that swaps an iterator's increment and decrement operators. The net effect is that you can reverse the input to an algorithm by applying a reverse iterator adapter. One common scenario where you might want to use a reverse iterator is when searching backward from the end of a container. For example, perhaps you've been pushing logs onto the end of a deque and want to find the latest entry that meets some criterion.

Almost all containers in Chapter 13 expose reverse iterators with rbegin/rend/crbegin/crend methods. For example, you can create a container with the reverse sequence of another container, as shown in Listing 14-13.

```
TEST_CASE("reverse iterators can initialize containers") {
  std::list<int> original{ 3, 2, 1 }; ❶
  std::vector<int> easy_as{ original.crbegin(), original.crend() }; ❷
  REQUIRE(easy_as[0] == 1); ❸
  REQUIRE(easy_as[1] == 2);
  REQUIRE(easy_as[2] == 3);
}
```

Listing 14-13: Creating a container with the reverse of another container's elements

Here, you create a `list` containing the elements `3 2 1` ❶. Next, you construct a `vector` with the reverse of the sequence by using the `crbegin` and `crend` methods ❷. The `vector` contains `1 2 3`, the reverse of the list elements ❸.

Although containers usually expose reverse iterators directly, you can also convert a normal iterator into a reverse iterator manually. The convenience function template `std::make_reverse_iterator` in the `<iterator>` header accepts a single iterator argument and returns a reverse iterator adapter.

Reverse iterators are designed to work with half-open ranges that are exactly opposite of normal half-open ranges. Internally, a *reverse half-open range* has an `rbegin` iterator that refers to 1 past a half-open range's `end` and an `rend` iterator that refers to the half-open range's `begin`, as shown in Figure 14-2.

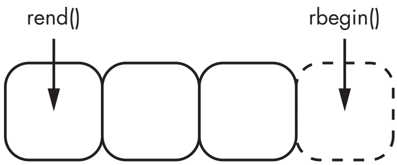


Figure 14-2: A reverse half-open range

However, these implementation details are all transparent to the user. The iterators dereference as you would expect. As long as the range isn't empty, you can dereference the reverse-begin iterator, and it will return the first element. But you *cannot* dereference the reverse-end iterator.

Why introduce this representational complication? With this design, you can easily swap the begin and end iterators of a half-open range to produce a reverse half-open range. For example, Listing 14-14 uses `std::make_reverse_iterator` to convert normal iterators to reverse iterators, accomplishing the same task as Listing 14-13.

```
TEST_CASE("make_reverse_iterator converts a normal iterator") {
  std::list<int> original{ 3, 2, 1 };
  auto begin = std::make_reverse_iterator(original.cend()); ❶
  auto end = std::make_reverse_iterator(original.cbegin()); ❷
  std::vector<int> easy_as{ begin, end }; ❸
  REQUIRE(easy_as[0] == 1);
  REQUIRE(easy_as[1] == 2);
  REQUIRE(easy_as[2] == 3);
}
```

Listing 14-14: The `make_reverse_iterator` function converts a normal iterator to a reverse iterator

Pay special attention to the iterators you're extracting from `original`. To create the `begin` iterator, you extract an `end` iterator from `original` and pass it to `make_reverse_iterator` ❶. The reverse iterator adapter will swap increment and decrement operators, but it needs to start in the right place. Likewise, you need to terminate at the original's beginning, so you pass the result of `cbegin` to `make_reverse_iterator` to produce the correct end ❷. Passing these to the range constructor of `easy_as` ❸ produces identical results to Listing 14-13.

NOTE *All reverse iterators expose a* `base` *method, which will convert the reverse iterator back into a normal iterator.*

Summary

In this short chapter, you learned all the iterator categories: output, input, forward, bidirectional, random-access, and contiguous. Knowing the basic properties of each category provides you with a framework for understanding how containers connect with algorithms. The chapter also surveyed iterator adapters, which enable you to customize iterator behavior, and the auxiliary iterator functions, which help you write generic code with iterators.

EXERCISES

14-1. Create a corollary to Listing 14-8 using `std::prev` rather than `std::next`.

14-2. Write a function template called `sum` that accepts a half-open range of `int` objects and returns the sum of the sequence.

14-3. Write a program that uses the `Stopwatch` class in Listing 12-25 to determine the runtime performance of `std::advance` when given a forward iterator from a large `std::forward_list` and a large `std::vector`. How does the runtime change with the number of elements in the container? (Try hundreds of thousands or millions of elements.)

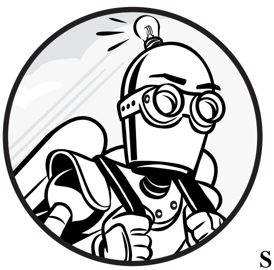
FURTHER READING

- *The C++ Standard Library: A Tutorial and Reference*, 2nd Edition, by Nicolai M. Josuttis (Addison-Wesley Professional, 2012)
- *C++ Templates: The Complete Guide*, 2nd Edition, by David Vandevoorde et al. (Addison-Wesley, 2017)

15

STRINGS

If you talk to a man in a language he understands, that goes to his head. If you talk to him in his language, that goes to his heart.
—Nelson Mandela



The STL provides a special *string container* for human-language data, such as words, sentences, and markup languages. Available in the `<string>` header, the `std::basic_string` is a class template that you can specialize on a string's underlying character type. As a sequential container, `basic_string` is essentially similar to a `vector` but with some special facilities for manipulating language.

STL `basic_string` provides major safety and feature improvements over C-style or null-terminated strings, and because human-language data inundates most modern programs, you'll probably find `basic_string` indispensable.

std::string

The STL provides four `basic_string` specializations in the `<string>` header. Each specialization implements a string using one of the fundamental character types that you learned about in Chapter 2:

- `std::string` for `char` is used for character sets like ASCII.
- `std::wstring` for `wchar_t` is large enough to contain the largest character of the implementation's locale.
- `std::u16string` for `char16_t` is used for character sets like UTF-16.
- `std::u32string` for `char32_t` is used for character sets like UTF-32.

You'll use the specialization with the appropriate underlying type. Because these specializations have the same interface, all the examples in this chapter will use `std::string`.

Constructing

The `basic_string` container takes three template parameters:

- The underlying character type, `T`
- The underlying type's traits, `Traits`
- An allocator, `Alloc`

Of these, only `T` is required. The STL's `std::char_traits` template class in the `<string>` header abstracts character and string operations from the underlying character type. Also, unless you plan on supporting a custom character type, you won't need to implement your own type traits, because `char_traits` has specializations available for `char`, `wchar_t`, `char16_t`, and `char32_t`. When the stdlib provides specializations for a type, you won't need to provide it yourself unless you require some kind of exotic behavior.

Together, a `basic_string` specialization looks like this, where `T` is a character type:

```
std::basic_string<T, Traits=std::char_traits<T>, Alloc=std::allocator<T>>
```

NOTE *In most cases, you'll be dealing with one of the predefined specializations, especially* `string` *or* `wstring`*. However, if you need a custom allocator, you'll need to specialize* `basic_string` *appropriately.*

The `basic_string<T>` container supports the same constructors as `vector<T>`, plus additional convenience constructors for converting a C-style string. In other words, a `string` supports the constructors of `vector<char>`, a `wstring` supports the constructors of `vector<wchar_t>`, and so on. As with `vector`, use parentheses for all `basic_string` constructors except when you actually want an initializer list.

You can default construct an empty string, or if you want to fill a `string` with a repeating character, you can use the fill constructor by passing a `size_t` and a `char`, as Listing 15-1 illustrates.

```
#include <string>
TEST_CASE("std::string supports constructing") {
  SECTION("empty strings") {
    std::string cheese; ❶
    REQUIRE(cheese.empty()); ❷
  }
  SECTION("repeated characters") {
    std::string roadside_assistance(3, 'A'); ❸
    REQUIRE(roadside_assistance == "AAA"); ❹
  }
}
```

Listing 15-1: The default and fill constructors of `string`

After you default construct a `string` ❶, it contains no elements ❷. If you want to fill the `string` with repeating characters, you can use the fill constructor by passing in the number of elements you want to fill and their value ❸. The example fills a string with three `A` characters ❹.

NOTE *You'll learn about `std::string` comparisons with `operator==` later in the chapter. Because you generally handle C-style strings with raw pointers or raw arrays, `operator==` returns true only when given the same object. However, for `std::string`, `operator==` returns true if the contents are equivalent. As you can see in Listing 15-1, the comparison works even when one of the operands is a C-style string literal.*

The `string` constructor also offers two `const char*`-based constructors. If the argument points to a null-terminated `string`, the `string` constructor can determine the input's length on its own. If the pointer does *not* point to a null-terminated string or if you only want to use the first part of a `string`, you can pass a length argument that informs the `string` constructor of how many elements to copy, as Listing 15-2 illustrates.

```
TEST_CASE("std::string supports constructing substrings ") {
  auto word = "gobbledygook"; ❶
  REQUIRE(std::string(word) == "gobbledygook"); ❷
  REQUIRE(std::string(word, 6) == "gobble"); ❸
}
```

Listing 15-2: Constructing a `string` from C-style strings

You create a `const char*` called `word` pointing to the C-style string literal `gobbledygook` ❶. Next, you construct a `string` by passing `word`. As expected, the resulting `string` contains `gobbledygook` ❷. In the next test, you pass the number `6` as a second argument. This causes `string` to only take the first six characters of `word`, resulting in the `string` containing `gobble` ❸.

Additionally, you can construct strings from other strings. As an STL container, string fully supports copy and move semantics. You can also construct a string from a *substring*—a contiguous subset of another string. Listing 15-3 illustrates these three constructors.

```
TEST_CASE("std::string supports") {
  std::string word("catawampus"); ❶
  SECTION("copy constructing") {
    REQUIRE(std::string(word) == "catawampus"); ❷
  }
  SECTION("move constructing") {
    REQUIRE(std::string(move(word)) == "catawampus"); ❸
  }
  SECTION("constructing from substrings") {
    REQUIRE(std::string(word, 0, 3) == "cat"); ❹
    REQUIRE(std::string(word, 4) == "wampus"); ❺
  }
}
```

Listing 15-3: Copy, move, and substring construction of `string` *objects*

NOTE *In Listing 15-3,* `word` *is in a moved-from state, which, you'll recall from "Move Semantics" on page 122, means it can only be reassigned or destructed.*

Here, you construct a `string` called `word` containing the characters `catawampus` ❶. Copy construction yields another `string` containing a copy of the characters of `word` ❷. Move construction steals the characters of `word`, resulting in a new `string` containing `catawampus` ❸. Finally, you can construct a new `string` based on substrings. By passing `word`, a starting position of 0, and a length of 3, you construct a new `string` containing the characters `cat` ❹. If you instead pass `word` and a starting position of 4 (without a length), you get all the characters from the fourth to the end of the original string, resulting in `wampus` ❺.

The `string` class also supports literal construction with `std::string_literals::operator""s`. The major benefit is notational convenience, but you can also use `operator""s` to embed null characters within a `string` easily, as Listing 15-4 illustrates.

```
TEST_CASE("constructing a string with") {
  SECTION("std::string(char*) stops at embedded nulls") {
    std::string str("idioglossia\0ellohay!"); ❶
    REQUIRE(str.length() == 11); ❷
  }
  SECTION("operator\"\"s incorporates embedded nulls") {
    using namespace std::string_literals; ❸
    auto str_lit = "idioglossia\0ellohay!"s; ❹
    REQUIRE(str_lit.length() == 20); ❺
  }
}
```

Listing 15-4: Constructing a `string`

In the first test, you construct a `string` using the literal `idioglossia\0ellohay!` ❶, which results in a `string` containing `idioglossia` ❷. The remainder of the literal didn't get copied into the `string` due to embedded nulls. In the second test, you bring in the `std::string_literals` namespace ❸ so you can use `operator""s` to construct a `string` from a literal directly ❹. Unlike the `std::string` constructor ❶, `operator""s` yields a string containing the entire literal—embedded null bytes and all ❺.

Table 15-1 summarizes the options for constructing a `string`. In this table, `c` is a `char`, `n` and `pos` are `size_t`, `str` is a `string` or a C-style string, `c_str` is a C-style string, and `beg` and `end` are input iterators.

Table 15-1: Supported `std::string` Constructors

Constructor	Produces a `string` containing
`string()`	No characters.
`string(n, c)`	`c` repeated `n` times.
`string(str, pos, [n])`	The half-open range `pos` to `pos+n` of `str`. Substring extends from `pos` to `str`'s end if `n` is omitted.
`string(c_str, [n])`	A copy of `c_str`, which has length `n`. If `c_str` is null terminated, `n` defaults to the null-terminated string's length.
`string(beg, end)`	A copy of the elements in the half-open range from `beg` to `end`.
`string(str)`	A copy of `str`.
`string(move(str))`	The contents of `str`, which is in a moved-from state after construction.
`string{ c1, c2, c3 }`	The characters `c1`, `c2`, and `c3`.
`"my string literal"s`	A string containing the characters `my string literal`.

String Storage and Small String Optimizations

Exactly like `vector`, `string` uses dynamic storage to store its constituent elements contiguously. Accordingly, `vector` and `string` have very similar copy/move-construction/assignment semantics. For example, copy operations are potentially more expensive than move operations because the contained elements reside in dynamic memory.

The most popular STL implementations have *small string optimizations (SSO)*. The SSO places the contents of a `string` within the object's storage (rather than dynamic storage) if the contents are small enough. As a general rule, a `string` with fewer than 24 bytes is an SSO candidate. Implementers make this optimization because in many modern programs, most `string`s are short. (A `vector` doesn't have any small optimizations.)

NOTE *Practically, SSO affects moves in two ways. First, any references to the elements of a* `string` *will invalidate if the* `string` *moves. Second, moves are potentially slower for* `string`*s than* `vector`*s because* `string`*s need to check for SSO.*

A string has a *size* (or *length*) and a *capacity*. The size is the number of characters contained in the string, and the capacity is the number of characters that the string can hold before needing to resize.

Table 15-2 contains methods for reading and manipulating the size and capacity of a string. In this table, n is a size_t. An asterisk (*) indicates that this operation invalidates raw pointers and iterators to the elements of s in at least some circumstances.

Table 15-2: Supported std::string Storage and Length Methods

Method	Returns
s.empty()	true if s contains no characters; otherwise false.
s.size()	The number of characters in s.
s.length()	Identical to s.size()
s.max_size()	The maximum possible size of s (due to system/runtime limitations).
s.capacity()	The number of characters s can hold before needing to resize.
s.shrink_to_fit()	void; issues a non-binding request to reduce s.capacity() to s.size().*
s.reserve([n])	void; if n > s.capacity(), resizes so s can hold at least n elements; otherwise, issues a non-binding request* to reduce s.capacity() to n or s.size(), whichever is greater.

NOTE *At press time, the draft C++20 standard changes the behavior of the* reserve *method when its argument is less than the size of the* string. *This will match the behavior of* vector, *where there is no effect rather than being equivalent to invoking* shrink_to_fit.

Note that the size and capacity methods of string match those of vector very closely. This is a direct result of the closeness of their storage models.

Element and Iterator Access

Because string offers random-access iterators to contiguous elements, it accordingly exposes similar element- and iterator-access methods to vector.

For interoperation with C-style APIs, string also exposes a c_str method, which returns a non-modifiable, null-terminated version of the string as a const char*, as Listing 15-5 illustrates.

```
TEST_CASE("string's c_str method makes null-terminated strings") {
  std::string word("horripilation"); ❶
  auto as_cstr = word.c_str(); ❷
  REQUIRE(as_cstr[0] ==  'h'); ❸
  REQUIRE(as_cstr[1] ==  'o');
  REQUIRE(as_cstr[11] == 'o');
  REQUIRE(as_cstr[12] == 'n');
  REQUIRE(as_cstr[13] == '\0'); ❹
}
```

Listing 15-5: Extracting a null-terminated string from a string

You construct a string called word containing the characters horripilation ❶ and use its c_str method to extract a null-terminated string called as_cstr ❷. Because as_cstr is a const char*, you can use operator[] to illustrate that it contains the same characters as word ❸ and that it is null terminated ❹.

NOTE *The std::string class also supports operator[], which has the same behavior as with a C-style string.*

Generally, c_str and data produce identical results except that references returned by data can be non-const. Whenever you manipulate a string, implementations usually ensure that the contiguous memory backing the string ends with a null terminator. The program in Listing 15-6 illustrates this behavior by printing the results of calling data and c_str alongside their addresses.

```
#include <string>
#include <cstdio>

int main() {
  std::string word("pulchritudinous");
  printf("c_str: %s at 0x%p\n", word.c_str(), word.c_str()); ❶
  printf("data:  %s at 0x%p\n", word.data(), word.data()); ❷
}
--------------------------------------------------------------------------
c_str: pulchritudinous at 0x0000002FAE6FF8D0 ❶
data:  pulchritudinous at 0x0000002FAE6FF8D0 ❷
```

Listing 15-6: Illustrating that c_str and data return equivalent addresses

Both c_str and data produce identical results because they point to the same addresses ❶❷. Because the address is the beginning of a null-terminated string, printf yields identical output for both invocations.

Table 15-3 lists the access methods of string. Note that n is a size_t in the table.

Table 15-3: Supported std::string Element and Iterator Access Methods

Method	Returns
s.begin()	An iterator pointing to the first element.
s.cbegin()	A const iterator pointing to the first element.
s.end()	An iterator pointing to one past the last element.
s.cend()	A const iterator pointing to one past the last element.
s.at(n)	A reference to element n of s. Throws std::out_of_range if out of bounds.
s[n]	A reference to element n of s. Undefined behavior if n > s.size(). Also s[s.size()] must be 0, so writing a non-zero value into this character is undefined behavior.
s.front()	A reference to first element.
s.back()	A reference to last element.

(continued)

Table 15-3: Supported `std::string` Element and Iterator Access Methods (continued)

Method	Returns
`s.data()`	A raw pointer to the first element if string is non-empty. For an empty string, returns a pointer to a null character.
`s.c_str()`	Returns a non-modifiable, null-terminated version of the contents of `s`.

String Comparisons

Note that `string` supports comparisons with other strings and with raw C-style strings using the usual comparison operators. For example, the equality `operator==` returns `true` if the size and contents of the left and right size are equal, whereas the inequality `operator!=` returns the opposite. The remaining comparison operators perform *lexicographical comparison*, meaning they sort alphabetically where $A < Z < a < z$ and where, if all else is equal, shorter words are less than longer words (for example, *pal* < *palindrome*). Listing 15-7 illustrates comparisons.

NOTE *Technically, lexicographical comparison depends on the encoding of the `string`. It's theoretically possible that a system could use a default encoding where the alphabet is in some completely jumbled order (such as the nearly obsolete EBCDIC encoding, which put lowercase letters before uppercase letters), which would affect `string` comparison. For ASCII-compatible encodings, you don't need to worry since they imply the expected lexicographical behavior.*

```
TEST_CASE("std::string supports comparison with") {
  using namespace std::literals::string_literals; ❶
  std::string word("allusion"); ❷
  SECTION("operator== and !=") {
    REQUIRE(word == "allusion"); ❸
    REQUIRE(word == "allusion"s); ❹
    REQUIRE(word != "Allusion"s); ❺
    REQUIRE(word != "illusion"s); ❻
    REQUIRE_FALSE(word == "illusion"s); ❼
  }
  SECTION("operator<") {
    REQUIRE(word < "illusion"); ❽
    REQUIRE(word < "illusion"s); ❾
    REQUIRE(word > "Illusion"s); ❿
  }
}
```

Listing 15-7: The `string` class supports comparison

Here, you bring in the `std::literals::string_literals` namespace so you can easily construct a `string` with `operator""s` ❶. You also construct a `string` called `word` containing the characters `allusion` ❷. In the first set of tests, you examine `operator==` and `operator!=`.

You can see that word equals (==) allusion as both a C-style string ❸ and a string ❹, but it doesn't equal (!=) strings containing Allusion ❺ or illusion ❻. As usual, operator== and operator!= always return opposite results ❼.

The next set of tests uses operator< to show that allusion is less than illusion ❽, because *a* is lexicographically less than *i*. Comparisons work with C-style strings and strings ❾. Listing 15-7 also shows that Allusion is less than allusion ❿ because *A* is lexicographically less than *a*.

Table 15-4 lists the comparison methods of string. Note that other is a string or char* C-style string in the table.

Table 15-4: Supported std::string Comparison Operators

Method	Returns
s == **other**	true if **s** and **other** have identical characters and lengths; otherwise false
s != **other**	The opposite of operator==
s.compare(**other**)	Returns 0 if **s == other**, a negative number if **s < other**, and a positive number if **s > other**
s < **other** s > **other** s <= **other** s >= **other**	The result of the corresponding comparison operation, according to lexicographical sort

Manipulating Elements

For manipulating elements, string has *a lot* of methods. It supports all the methods of vector<char> plus many others useful to manipulating human-language data.

Adding Elements

To add elements to a string, you can use push_back, which inserts a single character at the end. When you want to insert more than one character to the end of a string, you can use operator+= to append a character, a null-terminated char* string, or a string. You can also use the append method, which has three overloads. First, you can pass a string or a null-terminated char* string, an optional offset into that string, and an optional number of characters to append. Second, you can pass a length and a char, which will append that number of chars to the string. Third, you can append a half-open range. Listing 15-8 illustrates all of these operations.

```
TEST_CASE("std::string supports appending with") {
  std::string word("butt"); ❶
  SECTION("push_back") {
    word.push_back('e'); ❷
    REQUIRE(word == "butte");
  }
  SECTION("operator+=") {
    word += "erfinger"; ❸
```

```
    REQUIRE(word == "butterfinger");
  }
  SECTION("append char") {
    word.append(1, 's'); ❹
    REQUIRE(word == "butts");
  }
  SECTION("append char*") {
    word.append("stockings", 5); ❺
    REQUIRE(word == "buttstock");
  }
  SECTION("append (half-open range)") {
    std::string other("onomatopoeia"); ❻
    word.append(other.begin(), other.begin()+2); ❼
    REQUIRE(word == "button");
  }
}
```

Listing 15-8: Appending to a `string`

To begin, you initialize a `string` called `word` containing the characters `butt` ❶. In the first test, you invoke `push_back` with the letter `e` ❷, which yields `butte`. Next, you add `erfinger` to `word` using `operator+=` ❸, yielding `butterfinger`. In the first invocation of `append`, you append a single `s` ❹ to yield `butts`. (This setup works just like `push_back`.) A second overload of `append` allows you to provide a `char*` and a length. By providing `stockings` and length `5`, you add `stock` to `word` to yield `buttstock` ❺. Because `append` works with half-open ranges, you can also construct a `string` called `other` containing the characters `onomatopoeia` ❻ and append the first two characters via a half-open range to yield `button` ❼.

NOTE *Recall from "Test Cases and Sections" on page 308 that each `SECTION` of a Catch unit test runs independently, so modifications to `word` are independent of each other: the setup code resets `word` for each test.*

Removing Elements

To remove elements from a `string`, you have several options. The simplest method is to use `pop_back`, which follows `vector` in removing the last character from a `string`. If you want to instead remove all the characters (to yield an empty `string`), use the `clear` method. When you need more precision in removing elements, use the `erase` method, which provides several overloads. You can provide an index and a length, which removes the corresponding characters. You can also provide an iterator to remove a single element or a half-open range to remove many. Listing 15-9 illustrates removing elements from a `string`.

```
TEST_CASE("std::string supports removal with") {
  std::string word("therein"); ❶
  SECTION("pop_back") {
    word.pop_back();
    word.pop_back(); ❷
    REQUIRE(word == "there");
  }
```

```
  SECTION("clear") {
    word.clear(); ❸
    REQUIRE(word.empty());
  }
  SECTION("erase using half-open range") {
    word.erase(word.begin(), word.begin()+3); ❹
    REQUIRE(word == "rein");
  }
  SECTION("erase using an index and length") {
    word.erase(5, 2);
    REQUIRE(word == "there"); ❺
  }
}
```

Listing 15-9: Removing elements from a `string`

You construct a `string` called `word` containing the characters `therein` ❶. In the first test, you call `pop_back` twice to first remove the letter `n` followed by the letter `i` so `word` contains the characters `there` ❷. Next, you invoke `clear`, which removes all the characters from `word` so it's `empty` ❸. The last two tests use `erase` to remove some subset of the characters in `word`. In the first usage, you remove the first three characters with a half-open range so `word` contains `rein` ❹. In the second, you remove the characters starting at index 5 (`i` in `therein`) and extending two characters ❺. Like the first test, this yields the characters `there`.

Replacing Elements

To insert and remove elements simultaneously, use `string` to expose the `replace` method, which has many overloads.

First, you can provide a half-open range and a null-terminated `char*` or a `string`, and `replace` will perform a simultaneous `erase` of all the elements within the half-open range and an `insert` of the provided `string` where the range used to be. Second, you can provide two half-open ranges, and `replace` will insert the second range instead of a `string`.

Instead of replacing a range, you can use either an index or a single iterator and a length. You can supply a new half-open range, a character and a size, or a `string`, and `replace` will substitute new elements over the implied range. Listing 15-10 demonstrates some of these possibilities.

```
TEST_CASE("std::string replace works with") {
  std::string word("substitution"); ❶
  SECTION("a range and a char*") {
    word.replace(word.begin()+9, word.end(), "e"); ❷
    REQUIRE(word == "substitute");
  }
  SECTION("two ranges") {
    std::string other("innuendo");
    word.replace(word.begin(), word.begin()+3,
                 other.begin(), other.begin()+2); ❸
    REQUIRE(word == "institution");
  }
```

```
  SECTION("an index/length and a string") {
    std::string other("vers");
    word.replace(3, 6, other); ❹
    REQUIRE(word == "subversion");
  }
}
```

Listing 15-10: Replacing elements of a `string`

Here, you construct a `string` called `word` containing `substitution` ❶. In the first test, you replace all the characters from index 9 to the end with the letter `e`, resulting in the word `substitute` ❷. Next, you replace the first three letters of `word` with the first two letters of a `string` containing `innuendo` ❸, resulting in `institution`. Finally, you use an alternate way of specifying the target sequence with an index and a length to replace the characters `stitut` with the characters `vers`, yielding `subversion` ❹.

The `string` class offers a `resize` method to manually set the length of `string`. The `resize` method takes two arguments: a new length and an optional `char`. If the new length of `string` is smaller, `resize` ignores the `char`. If the new length of `string` is larger, `resize` appends the `char` the implied number of times to achieve the desired length. Listing 15-11 illustrates the `resize` method.

```
TEST_CASE("std::string resize") {
  std::string word("shamp"); ❶
  SECTION("can remove elements") {
    word.resize(4); ❷
    REQUIRE(word == "sham");
  }
  SECTION("can add elements") {
    word.resize(7, 'o'); ❸
    REQUIRE(word == "shampoo");
  }
}
```

Listing 15-11: Resizing a `string`

You construct a `string` called `word` containing the characters `shamp` ❶. In the first test, you resize `word` to length `4` so it contains `sham` ❷. In the second, you `resize` to a length of 7 and provide the optional character `o` as the value to extend `word` with ❸. This results in `word` containing `shampoo`.

The "Constructing" section on page 482 explained a substring constructor that can extract contiguous sequences of characters to create a new `string`. You can also generate substrings using the `substr` method, which takes two optional arguments: a position argument and a length. The position defaults to 0 (the beginning of the `string`), and the length defaults to the remainder of the `string`. Listing 15-12 illustrates how to use `substr`.

```
TEST_CASE("std::string substr with") {
  std::string word("hobbits"); ❶
  SECTION("no arguments copies the string") {
```