Next, you reserve 1,024 elements ❸, which doesn't change the vector's empty status but increases its capacity to match ❹. The `vector` now has 1,024 × 1 KiB = 1 MiB of contiguous space reserved. After reserving space, you emplace three arrays and check that `kb_store.size()` increased accordingly ❺.

You've reserved space for 1,024 elements. To release the 1,024 – 3 = 1,021 elements you aren't using back to the allocator, you call `shrink_to_fit`, which reduces the capacity to 3 ❻.

Finally, you invoke `clear` on the `vector` ❼, which destructs all elements and reduces its size to zero. However, the capacity remains unchanged because you haven't made another call to `shrink_to_fit` ❽. This is significant because the vector doesn't want to do extra work if you're going to add elements again.

A Partial List of Supported Operations

Table 13-2 provides a partial list of `vector` operations. In this table, `v`, `v1`, and `v2` are of type `std::vector<T>`, `t` is of type `T`, `alc` is an appropriate allocator, and `itr` is an iterator. An asterisk (*) indicates that this operation invalidates raw pointers and iterators to `v`'s elements in at least some circumstances.

Table 13-2: A Partial List of `std::vector` Operations

Operation	Notes
`vector<T>{ ..., [alc]}`	Performs braced initialization of a newly constructed vector. Uses `alc=std::allocator<T>` by default.
`vector<T>(s,[t], [alc])`	Fills the newly constructed vector with `s` number of copies of `t`. If no `t` is provided, default constructs `T` instances.
`vector<T>(v)`	Deep copy of `v`; allocates new memory.
`vector<T>(move(v))`	Takes ownership of memory, elements in `v`. No allocations.
`~vector`	Destructs all elements contained by the vector and releases dynamic memory.
`v.begin()`	Returns an iterator pointing to the first element.
`v.cbegin()`	Returns a `const` iterator pointing to the first element.
`v.end()`	Returns an iterator pointing to 1 past the last element.
`v.cend()`	Returns a `const` iterator pointing to 1 past the last element.
`v1 = v2`	`v1` destructs its elements; copies each `v2` element. Only allocates if it needs to resize to fit `v2`'s elements.*
`v1 = move(v2)`	`v1` destructs its elements; moves each `v2` element. Only allocates if it needs to resize to fit `v2`'s elements.*
`v.at(0)`	Accesses element `0` of `v`. Throws `std::out_of_range` if out of bounds.
`v[0]`	Accesses element `0` of `v`. Undefined behavior if out of bounds.
`v.front()`	Accesses first element.
`v.back()`	Accesses last element.

Operation	Notes
`v.data()`	Returns a raw pointer to the first element if array is non-empty. For empty arrays, returns a valid but non-dereferencable pointer.
`v.assign({ ... })`	Replaces the contents of `v` with the elements `...`.*
`v.assign(s, t)`	Replaces the contents of `v` with `s` number of copies of `t`.*
`v.empty()`	Returns `true` if vector's size is zero; otherwise `false`.
`v.size()`	Returns the number of elements in the vector.
`v.capacity()`	Returns the maximum number of elements the vector could hold without having to resize.
`v.shrink_to_fit()`	Might reduce the vector's storage so `capacity()` equals `size()`.*
`v.resize(s, [t])`	Resizes `v` to contain `s` elements. If this shrinks `v`, destructs elements at the end. If this grows `v`, inserts default constructed `T`s or copies of `t` if provided.*
`v.reserve(s)`	Increases the vector's storage so it can contain at least `s` elements.*
`v.max_size()`	Returns the maximum possible size the vector can resize to.
`v.clear()`	Removes all elements in `v`, but capacity remains.*
`v.insert(itr, t)`	Inserts a copy of `t` just before the element pointed to by `itr`; `v`'s range must contain `itr`.*
`v.push_back(t)`	Inserts a copy of `t` at the end of `v`.*
`v.emplace(itr, ...)`	Constructs a `T` in place by forwarding the arguments `...` to the appropriate constructor. Element inserted just before the element pointed to by `itr`.*
`v.emplace_back(...)`	Constructs a `T` in place by forwarding the arguments `...` to the appropriate constructor. Element inserted at the end of `v`.*
`v1.swap(v2)` `swap(v1, v2)`	Exchanges each element of `v1` with those of `v2`.*

`v1 == v2`
`v1 != v2`
`v1 > v2`
`v1 >= v2`
`v1 < v2`
`v1 <= v2`
	Equal if all elements are equal.
	Greater than/less than comparisons proceed from first element to last.

Niche Sequential Containers

The `vector` and `array` containers are the clear choice in most situations in which you need a sequential data structure. If you know the number of elements you'll need ahead of time, use an `array`. If you don't, use a `vector`.

You might find yourself in a niche situation where `vector` and `array` don't have the performance characteristics you desire. This section highlights a number of alternative sequential containers that might offer superior performance characteristics in such a situation.

Deque

A *deque* (pronounced "deck") is a sequential container with fast insert and remove operations from the front and back. Deque is a portmanteau of **d**ouble-**e**nded **que**ue. The STL implementation `std::deque` is available from the `<deque>` header.

NOTE *The Boost Container library also contains a `boost::container::deque` in the `<boost/container/deque.hpp>` header.*

A `vector` and a `deque` have very similar interfaces, but internally their storage models are totally different. A `vector` guarantees that all elements are sequential in memory, whereas a `deque`'s memory is usually scattered about, like a hybrid between a `vector` and a `list`. This makes large resizing operations more efficient and enables fast element insertion/deletion at the container's front.

Constructing and accessing members are identical operations for `vectors` and `deques`.

Because the internal structure of `deque` is complex, it doesn't expose a `data` method. In exchange, you gain access to `push_front` and `emplace_front`, which mirror the `push_back` and `emplace_back` that you're familiar with from `vector`. Listing 13-17 illustrates how to use `push_back` and `push_front` to insert values into a `deque` of `chars`.

```
#include <deque>

TEST_CASE("std::deque supports front insertion") {
  std::deque<char> deckard;
  deckard.push_front('a'); ❶ //  a
  deckard.push_back('i'); ❷ //  ai
  deckard.push_front('c');   // cai
  deckard.push_back('n');    // cain
  REQUIRE(deckard[0] == 'c'); ❸
  REQUIRE(deckard[1] == 'a');
  REQUIRE(deckard[2] == 'i');
  REQUIRE(deckard[3] == 'n');
}
```

Listing 13-17: A `deque` supports `push_front` and `push_back`.

After constructing an empty `deque`, you push alternating letters to the front ❶ and back ❷ of the `deque` so it contains the elements `c`, `a`, `i`, and `n` ❸.

NOTE *It would be a very bad idea to attempt to extract a string here, for example, `&deckard[0]`, because `deque` makes no guarantees about internal layout.*

The `vector` methods not implemented by `deque`, along with an explanation for their absence, are as follows:

`capacity`, `reserve` Because the internal structure is complicated, it might not be efficient to compute capacity. Also, `deque` allocations are

relatively fast because a deque doesn't relocate existing elements, so reserving memory ahead of time is unnecessary.

data The elements of deque are not contiguous.

Table 13-3 summarizes the additional operators offered by a deque but not by a vector. In this table, d is of type std::deque<T> and t is of type T. An asterisk (*) indicates that this operation invalidates iterators to v's elements in at least some circumstances. (Pointers to existing elements remain valid.)

Table 13-3: A Partial List of std::deque Operations

Operation	Notes
d.emplace_front(**...**)	Constructs an element in place at the front of the **d** by forwarding all arguments to the appropriate constructor.*
d.push_front(**t**)	Constructs an element in place at the front of the **d** by copying **t**.*
d.pop_front()	Removes the element at the front of **d**.*

List

A *list* is a sequence container with fast insert/remove operations everywhere but with no random element access. The STL implementation std::list is available from the <list> header.

NOTE *The Boost Container library also contains a boost::container::list in the <boost/container/list.hpp> header.*

The list is implemented as a doubly linked list, a data structure composed of *nodes*. Each node contains an element, a forward link ("flink"), and a backward link ("blink"). This is completely different from a vector, which stores elements in contiguous memory. As a result, you cannot use operator[] or at to access arbitrary elements in a list, because such operations would be very inefficient. (These methods are simply not available in list because of their horrible performance characteristics.) The trade-off is that inserting and removing elements in a list is much faster. All you need to update are the flinks and blinks of an element's neighbors rather than shuffling potentially large, contiguous element ranges.

The list container supports the same constructor patterns as vector.

You can perform special operations on lists, such as splicing elements from one list into another using the splice method, removing consecutive duplicate elements using the unique method, and even sorting the elements of a container using the sort method. Consider, for example, the remove_if method. The remove_if method accepts a function object as a parameter, and it traverses the list while invoking the function object on each element. If the result is true, remove_if removes the element. Listing 13-18 illustrates how to use the remove_if method to eliminate all the even numbers of a list with a lambda predicate.

```
#include <list>

TEST_CASE("std::list supports front insertion") {
  std::list<int> odds{ 11, 22, 33, 44, 55 }; ❶
  odds.remove_if([](int x) { return x % 2 == 0; }); ❷
  auto odds_iter = odds.begin(); ❸
  REQUIRE(*odds_iter == 11); ❹
  ++odds_iter; ❺
  REQUIRE(*odds_iter == 33);
  ++odds_iter;
  REQUIRE(*odds_iter == 55);
  ++odds_iter;
  REQUIRE(odds_iter == odds.end()); ❻
}
```

Listing 13-18: A `list` supports `remove_if`.

Here, you use braced initialization to fill a `list` of `int` objects ❶. Next, you use the `remove_if` method to remove all the even numbers ❷. Because only even numbers modulo 2 equal zero, this lambda tests whether a number is even. To establish that `remove_if` has extracted the even elements 22 and 44, you create an iterator pointing at the beginning of the list ❸, check its value ❹, and increment ❺ until you reach the end of the list ❻.

All the `vector` methods not implemented by `list`, along with an explanation for their absence, are as follows:

`capacity`, `reserve`, `shrink_to_fit` Because `list` acquires memory incrementally, it doesn't require periodic resizing.

`operator[]`, `at` Random element access is prohibitively expensive on `list`s.

`data` Unneeded because `list` elements are not contiguous.

Table 13-4 summarizes the additional operators offered by a `list` but not by a `vector`. In this table, `lst`, `lst1`, and `lst2` are of type `std::list<T>`, and `t` is of type `T`. The arguments `itr1`, `itr2a`, and `itr2b` are `list` iterators. An asterisk (*) indicates that the operation invalidates iterators to `v`'s elements in at least some circumstances. (Pointers to existing elements remain valid.)

Table 13-4: A Partial List of `std::list` Operations

Operation	Notes
`lst``.emplace_front(...)`	Constructs an element in place at the front of the **`d`** by forwarding all arguments to the appropriate constructor.
`lst``.push_front(`**`t`**`)`	Constructs an element in place at the front of **`d`** by copying **`t`**.
`lst``.pop_front()`	Removes the element at the front of **`d`**.
`lst``.push_back(`**`t`**`)`	Constructs an element in place at the back of **`d`** by copying **`t`**.
`lst``.pop_back()`	Removes the element at the back of **`d`**.
`lst1``.splice(`**`itr1`**`,`**`lst2`**`, [`**`itr2a`**`], [`**`itr2b`**`])`	Transfers items from **`lst2`** into **`lst1`** at position **`itr1`**. Optionally, only transfer the element at **`itr2a`** or the elements within the half-open range **`itr2a`** to **`itr2b`**.

Operation	Notes
`lst.remove(t)`	Removes all elements in `lst` equal to `t`.
`lst.remove_if(pred)`	Eliminates elements in `lst` where `pred` returns true; `pred` accepts a single `T` argument.
`lst.unique(pred)`	Eliminates duplicate consecutive elements in `lst` according to the function object `pred`, which accepts two `T` arguments and returns `t1 == t2`.
`lst1.merge(lst2, comp)`	Merges `lst1` and `lst2` according to the function object `comp`, which accepts two `T` arguments and returns `t1 < t2`.
`lst.sort(comp)`	Sorts `lst` according to the function object `comp`.
`lst.reverse()`	Reverses the order of `lst`'s elements (mutates `lst`).

NOTE *The STL also offers a `std::forward_list` in the `<forward_list>` header, which is a singly linked list that only allows iteration in one direction. The `forward_list` is slightly more efficient than `list`, and it's optimized for situations in which you need to store very few (or no) elements.*

Stacks

The STL provides three *container adapters* that encapsulate other STL containers and expose special interfaces for tailored situations. The adapters are the stack, the queue, and the priority queue.

A *stack* is a data structure with two fundamental operations: push and pop. When you *push* an element onto a stack, you insert the element onto the stack's end. When you *pop* an element off a stack, you remove the element from the stack's end. This arrangement is called *last-in, first-out*: the last element to be pushed onto a stack is the first to be popped off.

The STL offers the `std::stack` in the `<stack>` header. The class template `stack` takes two template parameters. The first is the underlying type of the wrapped container, such as `int`, and the second is the type of the wrapped container, such as `deque` or `vector`. This second argument is optional and defaults to `deque`.

To construct a `stack`, you can pass a reference to a `deque`, a `vector`, or a `list` to encapsulate. This way, the `stack` translates its operations, such as `push` and `pop`, into methods that the underlying container understands, like `push_back` and `pop_back`. If you provide no constructor argument, the `stack` uses a `deque` by default. The second template parameter must match this container's type.

To obtain a reference to the element on top of a `stack`, you use the `top` method.

Listing 13-19 illustrates how to use a `stack` to wrap a `vector`.

```
#include <stack>

TEST_CASE("std::stack supports push/pop/top operations") {
  std::vector<int> vec{ 1, 3 }; ❶  // 1 3
  std::stack<int, decltype(vec)> easy_as(vec); ❷
```

```
  REQUIRE(easy_as.top() == 3); ❸
  easy_as.pop(); ❹                // 1
  easy_as.push(2); ❺              // 1 2
  REQUIRE(easy_as.top() == 2); ❻
  easy_as.pop();                 // 1
  REQUIRE(easy_as.top() == 1);
  easy_as.pop();                 //
  REQUIRE(easy_as.empty()); ❼
}
```

Listing 13-19: Using a `stack` to wrap a `vector`

You construct a `vector` of `int`s called `vec` containing the elements 1 and 3 ❶. Next, you pass `vec` into the constructor of a new `stack`, making sure to supply the second template parameter `decltype(vec)` ❷. The top element in `stack` is now 3, because this is the last element in `vec` ❸. After the first `pop` ❹, you push a new element 2 onto the `stack` ❺. Now, the `top` element is 2 ❻. After another `pop`-`top`-`pop` series, the `stack` is empty ❼.

Table 13-5 summarizes the operations of `stack`. In this table, `s`, `s1`, and `s2` are of type `std::stack<T>`; `t` is of type `T`; and `ctr` is a container of type `ctr_type<T>`.

Table 13-5: A Summary of `std::stack` Operations

Operation	Notes
`stack<T, [ctr_type<T>]>([ctr])`	Constructs a stack of `T`s using `ctr` as its internal container reference. If no container is provided, constructs an empty `deque`.
`s.empty()`	Returns `true` if container is empty.
`s.size()`	Returns number of elements in container.
`s.top()`	Returns a reference to the element on top of the stack.
`s.push(t)`	Puts a copy of `t` onto the end of the container.
`s.emplace(...)`	Constructs a `T` in place by forwarding `...` to the appropriate constructor.
`s.pop()`	Removes the element at the end of the container.
`s1.swap(s2)` `swap(s1, s2)`	Exchanges the contents of `s2` with `s1`.

Queues

A *queue* is a data structure that, like a stack, has push and pop as its fundamental operations. Unlike a stack, a queue is *first-in, first-out*. When you push an element into a queue, you insert onto the queue's end. When you pop an element off the queue, you remove from the queue's beginning. This way, the element that has been in the queue the longest is the one to get popped off.

The STL offers the `std::queue` in the `<queue>` header. Like `stack`, `queue` takes two template parameters. The first parameter is the underlying type of the wrapped container, and the optional second parameter is the type of the wrapped container, which also defaults to `deque`.

Among STL containers, you can only use `deque` or `list` as the underlying container for a `queue`, because pushing and popping from the front of a `vector` is inefficient.

You can access the element at the front or back of a queue using the `front` and `back` methods.

Listing 13-20 shows how to use a `queue` to wrap a `deque`.

```
#include <queue>

TEST_CASE("std::queue supports push/pop/front/back") {
  std::deque<int> deq{ 1, 2 }; ❶
  std::queue<int> easy_as(deq); ❷ // 1 2

  REQUIRE(easy_as.front() == 1); ❸
  REQUIRE(easy_as.back() == 2); ❹
  easy_as.pop(); ❺                // 2
  easy_as.push(3); ❻              // 2 3
  REQUIRE(easy_as.front() == 2); ❼
  REQUIRE(easy_as.back() == 3); ❽
  easy_as.pop();                  // 3
  REQUIRE(easy_as.front() == 3);
  easy_as.pop();                  //
  REQUIRE(easy_as.empty()); ❾
}
```

Listing 13-20: Using a queue to wrap a deque

You start with a `deque` containing the elements 1 and 2 ❶, which you pass into a queue called `easy_as` ❷. Using the `front` and `back` methods, you can validate that the queue begins with a 1 ❸ and ends with a 2 ❹. When you `pop` the first element, 1, you're left with a queue containing just the single element 2 ❺. You then `push` 3 ❻, so the method `front` yields 2 ❼ and `back` yields 3 ❽. After two more iterations of `pop`-`front`, you're left with an empty `queue` ❾.

Table 13-6 summarizes the operations of `queue`. In this table, `q`, `q1`, and `q2` are of type `std::queue<T>`; `t` is of type `T`; and `ctr` is a container of type `ctr_type<T>`.

Table 13-6: A Summary of `std::queue` Operations

Operation	Notes
`queue<T, [ctr_type<T>]>([ctr])`	Constructs a `queue` of `T`s using **`ctr`** as its internal container. If no container is provided, constructs an empty `deque`.
`q``.empty()`	Returns `true` if container is empty.
`q``.size()`	Returns number of elements in container.

(continued)

Table 13-6: A Summary of `std::queue` Operations (continued)

Operation	Notes
`q.front()`	Returns a reference to the element in front of the queue.
`q.back()`	Returns a reference to the element in back of the queue.
`q.push(t)`	Puts a copy of `t` onto the end of the container.
`q.emplace(...)`	Constructs a `T` in place by forwarding `...` to the appropriate constructor.
`q.pop()`	Removes the element at the front of the container.
`q1.swap(q2)` `swap(q1, q2)`	Exchanges the contents of `q2` with `q1`.

Priority Queues (Heaps)

A *priority queue* (also called a heap) is a data structure that supports `push` and `pop` operations and keeps elements sorted according to some user-specified *comparator object*. The comparator object is a function object invokable with two parameters, returning `true` if the first argument is less than the second. When you `pop` an element from a priority queue, you remove the element that is greatest, according to the comparator object.

The STL offers the `std::priority_queue` in the `<queue>` header. A `priority_queue` has three template parameters:

- The underlying type of the wrapped container
- The type of the wrapped container
- The type of the comparator object

Only the underlying type is mandatory. The wrapped container type defaults to `vector` (probably because it's the most widely used sequential container), and the comparator object type defaults to `std::less`.

NOTE *The `std::less` class template is available from the `<functional>` header, and it returns `true` if the first argument is less than the second.*

The `priority_queue` has an identical interface to a `stack`. The only difference is that stacks `pop` elements according to the last-in, first-out arrangement, whereas priority queues `pop` elements according to the comparator object criteria.

Listing 13-21 illustrates the basic usage of `priority_queue`.

```
#include <queue>

TEST_CASE("std::priority_queue supports push/pop") {
  std::priority_queue<double> prique; ❶
  prique.push(1.0); // 1.0
  prique.push(2.0); // 2.0 1.0
  prique.push(1.5); // 2.0 1.5 1.0
```

```
  REQUIRE(prique.top() == Approx(2.0)); ❷
  prique.pop();     // 1.5 1.0
  prique.push(1.0); // 1.5 1.0 1.0
  REQUIRE(prique.top() == Approx(1.5)); ❸
  prique.pop();     // 1.0 1.0
  REQUIRE(prique.top() == Approx(1.0)); ❹
  prique.pop();     // 1.0
  REQUIRE(prique.top() == Approx(1.0)); ❺
  prique.pop();     //
  REQUIRE(prique.empty()); ❻
}
```

Listing 13-21: Basic `priority_queue` usage

Here, you default construct a `priority_queue` ❶, which internally initializes an empty `vector` to hold its elements. You push the elements 1.0, 2.0, and 1.5 into the `priority_queue`, which sorts the elements in descending order so the container represents them in the order 2.0 1.5 1.0.

You assert that `top` yields 2.0 ❷, pop this element off the `priority_queue`, and then invoke `push` with the new element 1.0. The container now represents them in the order 1.5 ❸ 1.0 ❹ 1.0 ❺, which you verify with a series of `top`-`pop` operations until the container is empty ❻.

NOTE *A `priority_queue` holds its elements in a tree structure, so if you peered into its underlying container, the memory ordering wouldn't match the orders implied by Listing 13-21.*

Table 13-7 summarizes the operations of `priority_queue`. In this table, `pq`, `pq1`, and `pq2` are of type `std::priority_queue<T>`; `t` is of type `T`; `ctr` is a container of type `ctr_type<T>`; and **`srt`** is a container of type `srt_type<T>`.

Table 13-7: A Summary of `std::priority_queue` Operations

Operation	Notes
`priority_queue <T, [ctr_type<T>], [cmp_type]>([cmp], [ctr])`	Constructs a `priority_queue` of `T`s using **`ctr`** as its internal container and **`srt`** as its comparator object. If no container is provided, constructs an empty `deque`. Uses `std::less` as default sorter.
`pq``.empty()`	Returns `true` if container is empty.
`pq``.size()`	Returns number of elements in container.
`pq``.top()`	Returns a reference to the greatest element in the container.
`pq``.push(`**`t`**`)`	Puts a copy of **`t`** onto the end of the container.
`pq``.emplace(...)`	Constructs a `T` in place by forwarding **`...`** to the appropriate constructor.
`pq``.pop()`	Removes the element at the end of the container.
`pq1``.swap(`**`pq2`**`)` `swap(`**`pq1, pq2`**`)`	Exchanges the contents of **`s2`** with **`s1`**.

Bitsets

A *bitset* is a data structure that stores a fixed-size bit sequence. You can manipulate each bit.

The STL offers the `std::bitset` in the `<bitset>` header. The class template `bitset` takes a single template parameter corresponding to the desired size. You could achieve similar functionality using a `bool array`, but `bitset` is optimized for space efficiency and provides some special convenience operations.

NOTE *The STL specializes `std::vector<bool>`, so it might benefit from the same space efficiencies as `bitset`. (Recall from "Template Specialization" on page 178 that template specialization is the process of making certain kinds of template instantiations more efficient.) Boost offers `boost::dynamic_bitset`, which provides dynamic sizing at runtime.*

A default constructed `bitset` contains all zero (false) bits. To initialize bitsets with other contents, you can provide an `unsigned long long` value. This integer's bitwise representation sets the value of `bitset`. You can access individual bits in the `bitset` using `operator[]`. Listing 13-22 demonstrates how to initialize a `bitset` with an integer literal and extract its elements.

```
#include <bitset>

TEST_CASE("std::bitset supports integer initialization") {
  std::bitset<4> bs(0b0101); ❶
  REQUIRE_FALSE(bs[0]); ❷
  REQUIRE(bs[1]); ❸
  REQUIRE_FALSE(bs[2]); ❹
  REQUIRE(bs[3]); ❺
}
```

Listing 13-22: Initializing a `bitset` with an integer

You initialize a `bitset` with the 4-bit *nybble* `0101` ❶. So, the first ❷ and third ❹ elements are zero, and the second ❸ and fourth ❺ elements are 1.

You can also provide a string representation of the desired `bitset`, as shown in Listing 13-23.

```
TEST_CASE("std::bitset supports string initialization") {
  std::bitset<4> bs1(0b0110); ❶
  std::bitset<4> bs2("0110"); ❷
  REQUIRE(bs1 == bs2); ❸
}
```

Listing 13-23: Initializing a `bitset` with a string

Here, you construct a `bitset` called `bs1` using the same integer nybble `0b0110` ❶ and another `bitset` called `bs2` using the string literal `0110` ❷. Both of these initialization approaches produce identical `bitset` objects ❸.

Table 13-8 summarizes the operations of `bitset`. In this table, `bs`, `bs 1`, and `bs 2` are of type `std::bitset<N>`, and `i` is a `size_t`.

Table 13-8: A Summary of `std::bitset` Operations

Operation	Notes
`bitset<N>([val])`	Constructs a `bitset` with initial value **`val`**, which can be either a string of **`0`**s and **`1`**s or an `unsigned long long`. Default constructor initializes all bits to zero.
`bs``[i]`	Returns the value of the **`i`**-th bit: 1 returns true; 0 returns false.
`bs``.test(i)`	Returns the value of the **`i`**-th bit: 1 returns true; 0 returns false. Performs bounds checking; throws `std::out_of_range`.
`bs``.set()`	Sets all bits to 1.
`bs``.set(i, val)`	Sets the **`i`**-th bit to **`val`**. Performs bounds checking; throws `std::out_of_range`.
`bs``.reset()`	Sets all bits to 0.
`bs``.reset(i)`	Sets the **`i`**-th bit to zero. Performs bounds checking; throws `std::out_of_range`.
`bs``.flip()`	Flips all the bits: (0 becomes 1; 1 becomes 0).
`bs``.flip(i)`	Flips the **`i`**-th bit to zero. Performs bounds checking; throws `std::out_of_range`.
`bs``.count()`	Returns the number of bits set to 1.
`bs``.size()`	Returns the size **`N`** of the bitset.
`bs``.any()`	Returns `true` if any bits are set to 1.
`bs``.none()`	Returns `true` if all bits are set to 0.
`bs``.all()`	Returns `true` if all bits are set to 1.
`bs``.to_string()`	Returns the `string` representation of the **`bitset`**.
`bs``.to_ulong()`	Returns the `unsigned long` representation of the **`bitset`**.
`bs``.to_ullong()`	Returns the `unsigned long long` representation of the **`bitset`**.

Special Sequential Boost Containers

Boost provides an abundance of special containers, and there simply isn't enough room to explore all their features here. Table 13-9 provides the names, headers, and brief descriptions of a number of them.

NOTE *Refer to the Boost Container documentation for more information.*

Table 13-9: Special Boost Containers

Class/Header	Description
`boost::intrusive::*` `<boost/intrusive/*.hpp>`	Intrusive containers impose requirements on the elements they contain (such as inheriting from a particular base class). In exchange, they offer substantial performance gains.
`boost::container::stable_vector` `<boost/container/stable_vector.hpp>`	A vector without contiguous elements but guarantees that iterators and references to elements remain valid as long as the element isn't erased (as with `list`).
`boost::container::slist` `<boost/container/slist.hpp>`	A `forward_list` with a fast `size` method.
`boost::container::static_vector` `<boost/container/static_vector.hpp>`	A hybrid between array and vector that stores a dynamic number of elements up to a fixed size. Elements are stored within the memory of `stable_vector`, like an `array`.
`boost::container::small_vector` `<boost/container/small_vector.hpp>`	A `vector`-like container optimized for holding a small number of elements. Contains some preallocated space, avoiding dynamic allocation.
`boost::circular_buffer` `<boost/circular_buffer.hpp>`	A fixed-capacity, queue-like container that fills elements in a circular fashion; a new element overwrites the oldest element once capacity is reached.
`boost::multi_array` `<boost/multi_array.hpp>`	An array-like container that accepts multiple dimensions. Rather than having, for example, an array of arrays of arrays, you can specify a three-dimensional `multi_array` **x** that allows element access, such as **x**`[5][1][2]`.
`boost::ptr_vector` `boost::ptr_list` `<boost/ptr_container/*.hpp>`	Having a collection of smart pointers can be suboptimal. Pointer vectors manage a collection of dynamic objects in a more efficient and user-friendly way.

NOTE *Boost Intrusive also contains some specialized containers that provide performance benefits in certain situations. These are primarily useful for library implementers.*

Associative Containers

Associative containers allow for very fast element search. Sequential containers have some natural ordering that allows you to iterate from the beginning of the container to the end in a well-specified order. Associative containers are a bit different. This container family splits along three axes:

- Whether elements contain keys (a set) or key-value pairs (a map)
- Whether elements are ordered
- Whether keys are *unique*

Sets

The `std::set` available in the STL's `<set>` header is an associative container that contains sorted, unique elements called *keys*. Because `set` stores sorted elements, you can insert, remove, and search efficiently. In addition, `set` supports sorted iteration over its elements, and you have complete control over how keys sort using comparator objects.

NOTE *Boost also provides a `boost::container::set` in the `<boost/container/set.hpp>` header.*

Constructing

The class template `set<T, Comparator, Allocator>` takes three template parameters:

- The key type `T`
- The comparator type that defaults to `std::less`
- The allocator type that defaults to `std::allocator<T>`

You have a lot of flexibility when constructing `set`s. Each of the following constructors accepts an optional comparator and allocator (whose types must match their corresponding template parameters):

- A default constructor that initializes an empty `set`
- Move and copy constructors with the usual behavior
- A range constructor that copies the elements from the range into the set
- A braced initializer

Listing 13-24 showcases each of these constructors.

```
#include <set>

TEST_CASE("std::set supports") {
  std::set<int> emp; ❶
  std::set<int> fib{ 1, 1, 2, 3, 5 }; ❷
  SECTION("default construction") {
    REQUIRE(emp.empty()); ❸
  }
  SECTION("braced initialization") {
    REQUIRE(fib.size() == 4); ❹
  }
  SECTION("copy construction") {
    auto fib_copy(fib);
    REQUIRE(fib.size() == 4); ❺
    REQUIRE(fib_copy.size() == 4); ❻
  }
  SECTION("move construction") {
    auto fib_moved(std::move(fib));
    REQUIRE(fib.empty()); ❼
    REQUIRE(fib_moved.size() == 4); ❽
```

```
  }
  SECTION("range construction") {
    std::array<int, 5> fib_array{ 1, 1, 2, 3, 5 };
    std::set<int> fib_set(fib_array.cbegin(), fib_array.cend());
    REQUIRE(fib_set.size() == 4); ❾
  }
}
```

Listing 13-24: The constructors of a set

You default construct ❶ and brace initialize ❷ two different `set`s. The default constructed `set` called `emp` is empty ❸, and the braced initialized `set` called `fib` has four elements ❹. You include five elements in the braced initializer, so why only four elements? Recall that `set` elements are unique, so the 1 enters only once.

Next, you copy construct `fib`, which results in two `set`s with size 4 ❺❻. On the other hand, the move constructor empties the moved-from `set` ❼ and transfers the elements to the new `set` ❽.

Then you can initialize a `set` from a range. You construct an `array` with five elements and then pass it as a range to a `set` constructor using the `cbegin` and `cend` methods. As with the braced initialization earlier in the code, the `set` contains only four elements because duplicates are discarded ❾.

Move and Copy Semantics

In addition to move/copy constructors, move/copy assignment operators are also available. As with other container copy operations, `set` copies are potentially very slow because each element needs to get copied, and move operations are usually fast because elements reside in dynamic memory. A `set` can simply pass ownership without disturbing the elements.

Element Access

You have several options for extracting elements from a `set`. The basic method is `find`, which takes a `const` reference to a key and returns an iterator. If the `set` contains an element-matching key, `find` will return an iterator pointing to the found element. If the `set` does not, it will return an iterator pointing to `end`. The `lower_bound` method returns an iterator to the first element *not less than* the key argument, whereas the `upper_bound` method returns the first element *greater than* the given key.

The `set` class supports two additional lookup methods, mainly for compatibility of non-unique associative containers:

- The `count` method returns the number of elements matching the key. Because `set` elements are unique, `count` returns either 0 or 1.
- The `equal_range` method returns a half-open range containing all the elements matching the given key. The range returns a `std::pair` of iterators with `first` pointing to the matching element and `second` pointing to

the element after `first`. If `equal_range` finds no matching element, `first` and `second` both point to the first element greater than the given key. In other words, the pair returned by `equal_range` is equivalent to a `pair` of `lower_bound` as `first` and `upper_bound` as `second`.

Listing 13-25 illustrates these two access methods.

```
TEST_CASE("std::set allows access") {
  std::set<int> fib{ 1, 1, 2, 3, 5 }; ❶
  SECTION("with find") { ❷
    REQUIRE(*fib.find(3) == 3);
    REQUIRE(fib.find(100) == fib.end());
  }
  SECTION("with count") { ❸
    REQUIRE(fib.count(3) == 1);
    REQUIRE(fib.count(100) == 0);
  }
  SECTION("with lower_bound") { ❹
    auto itr = fib.lower_bound(3);
    REQUIRE(*itr == 3);
  }
  SECTION("with upper_bound") { ❺
    auto itr = fib.upper_bound(3);
    REQUIRE(*itr == 5);
  }
  SECTION("with equal_range") { ❻
    auto pair_itr = fib.equal_range(3);
    REQUIRE(*pair_itr.first == 3);
    REQUIRE(*pair_itr.second == 5);
  }
}
```

Listing 13-25: A `set` member access

First, you construct a `set` with the four elements 1 2 3 5 ❶. Using `find`, you can extract an iterator to the element 3. You can also determine that 8 isn't in the `set`, because `find` returns an iterator pointing to `end` ❷. You can determine similar information with `count`, which returns 1 when you give the key 3 and 0 when you give the key 8 ❸. When you pass 3 to the `lower_bound` method, it returns an iterator pointing to 3 because this is the first element that's not less than the argument ❹. When you pass this to `upper_bound`, on the other hand, you obtain a pointer to the element 5, because this is the first element greater than the argument ❺. Finally, when you pass 3 to the `equal_range` method, you obtain a `pair` of iterators. The `first` iterator points to 3, and the `second` iterator points to 5, the element just after 3 ❻.

A `set` also exposes iterators through its `begin` and `end` methods, so you can use range-based `for` loops to iterate through the `set` from least element to greatest.

Adding Elements

You have three options when adding elements to a `set`:

- `insert` to copy an existing element into the `set`
- `emplace` to in-place construct a new element into the `set`
- `emplace_hint` to in-place construct a new element, just like `emplace` (because adding an element requires sorting). The difference is the `emplace_hint` method takes an iterator as its first argument. This iterator is the search's starting point (a hint). If the iterator is close to the correct position for the newly inserted element, this can provide a substantial speedup.

  Listing 13-26 illustrates the several ways to insert elements into a `set`.

```
TEST_CASE("std::set allows insertion") {
  std::set<int> fib{ 1, 1, 2, 3, 5 };
  SECTION("with insert") { ❶
    fib.insert(8);
    REQUIRE(fib.find(8) != fib.end());
  }
  SECTION("with emplace") { ❷
    fib.emplace(8);
    REQUIRE(fib.find(8) != fib.end());
  }
  SECTION("with emplace_hint") { ❸
    fib.emplace_hint(fib.end(), 8);
    REQUIRE(fib.find(8) != fib.end());
  }
}
```

Listing 13-26: Inserting into a set

Both `insert` ❶ and `emplace` ❷ add the element 8 into `fib`, so when you invoke `find` with 8, you get an iterator pointing to the new element. You can achieve the same effect a bit more efficiently with `emplace_hint` ❸. Because you know ahead of time that the new element 8 is greater than all the other elements in the `set`, you can use `end` as the hint.

If you attempt to `insert`, `emplace`, or `emplace_hint` a key that's already present in the `set`, the operation has no effect. All three of these methods return a `std::pair<Iterator, bool>` where the `second` element indicates whether the operation resulted in insertion (`true`) or not (`false`). The iterator at `first` points to either the newly inserted element or the existing element that prevented insertion.

Removing Elements

You can remove elements from a set using `erase`, which is overloaded to accept a key, an iterator, or a half-open range, as shown in Listing 13-27.

```
TEST_CASE("std::set allows removal") {
  std::set<int> fib{ 1, 1, 2, 3, 5 };
  SECTION("with erase") { ❶
```

```
    fib.erase(3);
    REQUIRE(fib.find(3) == fib.end());
  }
  SECTION("with clear") { ❷
    fib.clear();
    REQUIRE(fib.empty());
  }
}
```

Listing 13-27: Removing from a set

In the first test, you call `erase` with the key 3, which removes the corresponding element from the `set`. When you invoke `find` on 3, you get an iterator pointing to the `end`, indicating that no matching element was found ❶. In the second test, you invoke `clear`, which eliminates all the elements from the `set` ❷.

Storage Model

Set operations are fast because sets are typically implemented as *red-black trees*. These structures treat each element as a node. Each node has one parent and up to two children, its left and right legs. Each node's children are sorted so all children to the left are less than the children to the right. This way, you can perform searches much quicker than with linear iteration, as long as a tree's branches are roughly balanced (equal in length). Red-black trees have additional facilities for rebalancing branches after insertions and deletions.

NOTE *For details on red-black trees, refer to* Data Structures and Algorithms in C++ *by Adam Drozdek.*

A Partial List of Supported Operations

Table 13-10 summarizes the operations of `set`. Operations `s`, `s1`, and `s2` are of type `std::set<T,[cmp_type<T>]>`. `T` is the contained element/key type, and `itr`, `beg`, and `end` are `set` iterators. The variable `t` is a `T`. A dagger (†) denotes a method that returns a `std::pair<Iterator, bool>`, where the iterator points to the resulting element and the `bool` equals `true` if the method inserted an element and `false` if the element already existed.

Table 13-10: A Summary of `std::set`

Operation	Notes
`set<T>{ ..., [cmp], [alc] }`	Performs braced initialization of a newly constructed set. Uses **`cmp`**`=std::less<T>` and **`alc`**`=std::allocator<T>` by default.
`set<T>{ beg, end, [cmp], [alc] }`	Range constructor that copies elements from the half-open range **`beg`** to **`end`**. Uses **`cmp`**`=std::less<T>` and **`alc`**`=std::allocator<T>` by default.
`set<T>(s)`	Deep copy of **`s`**; allocates new memory.

(continued)

Table 13-10: A Summary of `std::set` *(continued)*

Operation	Notes
`set<T>(move(s))`	Takes ownership of memory; elements in `s`. No allocations.
`~set`	Destructs all elements contained by the set and releases dynamic memory.
`s1 = s2`	`s1` destructs its elements; copies each `s2` element. Only allocates if it needs to resize to fit `s2`'s elements.
`s1 = move(s2)`	`s1` destructs its elements; moves each `s2` element. Only allocates if it needs to resize to fit `s2`'s elements.
`s.begin()`	Returns an iterator pointing to the first element.
`s.cbegin()`	Returns a `const` iterator pointing to the first element.
`s.end()`	Returns an iterator pointing to 1 past the last element.
`s.cend()`	Returns a `const` iterator pointing to 1 past the last element.
`s.find(t)`	Returns an iterator pointing to the element matching `t` or `s.end()` if no such element exists.
`s.count(t)`	Returns 1 if set contains `t`; otherwise 0.
`s.equal_range(t)`	Returns a `pair` of iterators corresponding to the half-open range of elements matching `t`.
`s.lower_bound(t)`	Returns an iterator pointing to the first element not less than `t` or `s.end()` if no such element exists.
`s.upper_bound(t)`	Returns an iterator pointing to the first element greater than `t` or `s.end()` if no such element exists.
`s.clear()`	Removes all elements from the set.
`s.erase(t)`	Removes the element equal to `t`.
`s.erase(itr)`	Removes the element pointed to by `itr`.
`s.erase(beg, end)`	Removes all elements on the half-open range from `beg` to `end`.
`s.insert(t)`	Inserts a copy of `t` into the set.†
`s.emplace(...)`	Constructs a `T` in place by forwarding the arguments `...`.†
`s.emplace_hint(itr, ...)`	Constructs a `T` in place by forwarding the arguments `...`. Uses `itr` as a hint for where to insert the new element.†
`s.empty()`	Returns `true` if set's size is zero; otherwise `false`.
`s.size()`	Returns the number of elements in the set.
`s.max_size()`	Returns the maximum number of elements in the set.
`s.extract(t)` `s.extract(itr)`	Obtains a node handle that owns the element matching `t` or pointed to by `itr`. (This is the only way to remove a move-only element.)
`s1.merge(s2)` `s1.merge(move(s2))`	Splices each element of `s2` into `s1`. If argument is an rvalue, will move the elements into `s1`.
`s1.swap(s2)` `swap(s1, s2)`	Exchanges each element of `s1` with those of `s2`.

Multisets

The `std::multiset` available in the STL's `<set>` header is an associative container that contains sorted, *non-unique* keys. A `multiset` supports the same operations as a `set`, but it will store redundant elements. This has important ramifications for two methods:

- The method `count` can return values other than 0 or 1. The `count` method of `multiset` will tell you how many elements matched the given key.
- The method `equal_range` can return half-open ranges containing more than one element. The `equal_range` method of `multiset` will return a range containing all the elements matching the given key.

You might want to use a `multiset` rather than a `set` if it's important that you store multiple elements with the same key. For example, you could store all of an address's occupants by treating the address as a key and each member of the house as an element. If you used a `set`, you'd be stuck having only a single occupant.

Listing 13-28 illustrates using a `multiset`.

```
TEST_CASE("std::multiset handles non-unique elements") {
  std::multiset<int> fib{ 1, 1, 2, 3, 5 };
  SECTION("as reflected by size") {
    REQUIRE(fib.size() == 5); ❶
  }
  SECTION("and count returns values greater than 1") {
    REQUIRE(fib.count(1) == 2); ❷
  }
  SECTION("and equal_range returns non-trivial ranges") {
    auto [begin, end] = fib.equal_range(1); ❸
    REQUIRE(*begin == 1); ❹
    ++begin;
    REQUIRE(*begin == 1); ❺
    ++begin;
    REQUIRE(begin == end); ❻
  }
}
```

Listing 13-28: Accessing `multiset` elements

Unlike `set` in Listing 13-24, `multiset` permits multiple 1s, so size returns 5, the number of elements you provided in the braced initializers ❶. When you count the number of 1s, you get 2 ❷. You can use `equal_range` to iterate over these elements. Using structured binding syntax, you obtain a `begin` and `end` iterator ❸. You iterate over the two 1s ❹❺ and arrive at the end of the half-open range ❻.

Every operation in Table 13-10 works for `multiset`.

NOTE *Boost also provides a `boost::container::multiset` in the `<boost/container/set.hpp>` header.*

Unordered Sets

The `std::unordered_set` available in the STL's `<unordered_set>` header is an associative container that contains *unsorted*, unique keys. The `unordered_set` supports most of the same operations as `set` and `multiset`, but its internal storage model is completely different.

NOTE *Boost also provides a `boost::unordered_set` in the `<boost/unordered_set.hpp>` header.*

Rather than using a comparator to sort elements into a red-black tree, an `unordered_set` is usually implemented as a hash table. You might want to use an `unordered_set` in a situation in which there is no natural ordering among the keys and you don't need to iterate through the collection in such an order. You might find that in many situations, you could use either a `set` or an `unordered_set`. Although they appear quite similar, their internal representations are fundamentally different, so they'll have different performance characteristics. If performance is an issue, measure how both perform and use the one that's more appropriate.

Storage Model: Hash Tables

A hash function, or a *hasher*, is a function that accepts a key and returns a unique `size_t` value called a hash code. The `unordered_set` organizes its elements into a hash table, which associates a hash code with a collection of one or more elements called a *bucket*. To find an element, an `unordered_set` computes its hash code and then searches through the corresponding bucket in the hash table.

If you've never seen a hash table before, this information might be a lot to take in, so let's look at an example. Imagine you had a large group of people that you needed to sort into some kind of sensible groups to find an individual easily. You could group people by birthday, which would give you 365 groups (well, 366 if you count February 29 for leap years). The birthday is like a hash function that returns one of 365 values for each person. Each value forms a bucket, and all people in the same bucket have the same birthday. In this example, to find a person, you first determine their birthday, which gives you the correct bucket. Then you can search through the bucket to find the person you're looking for.

As long as the hash function is quick and there aren't too many elements per bucket, `unordered_sets` have even more impressive performance than their ordered counterparts: the contained element count doesn't increase insertion, search, and deletion times. When two different keys have the same hash code, it's called a *hash collision*. When you have a hash collision, it means that the two keys will reside in the same bucket. In the preceding birthday example, many people will have the same birthday, so there will be a lot of hash collisions. The more hash collisions there are, the larger the buckets will be, and the more time you'll spend searching through a bucket for the correct element.

A hash function has several requirements:

- It accepts a `Key` and returns a `size_t` hash code.
- It doesn't throw exceptions.
- Equal keys yield equal hash codes.
- Unequal keys yield unequal hash codes with high probability. (There is a low probability of a hash collision.)

The STL provides the hasher class template `std::hash<T>` in the `<functional>` header, which contains specializations for fundamental types, enumeration types, pointer types, `optional`, `variant`, smart pointers, and more. As an example, Listing 13-29 illustrates how `std::hash<long>` meets the equivalence criteria.

```
#include <functional>
TEST_CASE("std::hash<long> returns") {
  std::hash<long> hasher; ❶
  auto hash_code_42 = hasher(42); ❷
  SECTION("equal hash codes for equal keys") {
    REQUIRE(hash_code_42 == hasher(42)); ❸
  }
  SECTION("unequal hash codes for unequal keys") {
    REQUIRE(hash_code_42 != hasher(43)); ❹
  }
}
```

Listing 13-29: The `std::hash<long>` returns equal hash codes for equal keys and unequal hash codes for unequal keys.

You construct a hasher of type `std::hash<long>` ❶ and use it to compute the hash code of 42, storing the result into `size_t hash_code_42` ❷. When you invoke `hasher` with 42 again, you obtain the same value ❸. When you invoke hasher with `43` instead, you obtain a different value ❹.

Once an `unordered_set` hashes a key, it can obtain a bucket. Because the bucket is a list of possible matching elements, you need a function object that determines equality between a key and a bucket element. The STL provides the class template `std::equal_to<T>` in the `<functional>` header, which simply invokes `operator==` on its arguments, as Listing 13-30 illustrates.

```
#include <functional>
TEST_CASE("std::equal_to<long> returns") {
  std::equal_to<long> long_equal_to; ❶
  SECTION("true when arguments equal") {
    REQUIRE(long_equal_to(42, 42)); ❷
  }
  SECTION("false when arguments unequal") {
    REQUIRE_FALSE(long_equal_to(42, 43)); ❸
  }
}
```

Listing 13-30: The `std::equal_to<long>` calls `operator==` on its arguments to determine equality.

Here, you've initialized an `equal_to<long>` called `long_equal_to` ❶. When you invoke `long_equal_to` with equal arguments, it returns `true` ❷. When you invoke it with unequal arguments, it returns `false` ❸.

NOTE *For brevity, this chapter won't cover implementing your own hashing and equivalence functions, which you'll need if you want to construct unordered containers given user-defined key types. See Chapter 7 of* The C++ Standard Library, *2nd Edition, by Nicolai Josuttis.*

Constructing

The class template `std::unordered_set<T, Hash, KeyEqual, Allocator>` takes four template parameters:

- Key type `T`
- The `Hash` hash function type, which defaults to `std::hash<T>`
- The `KeyEqual` equality function type, which defaults to `std::equal_to<T>`
- The `Allocator` allocator type, which defaults to `std::allocator<T>`

An `unordered_set` supports equivalent constructors to `set` with adjustments for the different template parameters (`set` needs a `Comparator`, whereas `unordered_set` needs a `Hash` and a `KeyEqual`). For example, you can use `unordered_set` as a drop-in replacement for `set` in Listing 13-24, because `unordered_set` has range constructors and copy/move constructors and supports braced initialization.

Supported set Operations

An `unordered_set` supports all `set` operations in Table 13-10 except for `lower_bound` and `upper_bound`, because `unordered_set` doesn't sort its elements.

Bucket Management

Generally, the reason you reach for an `unordered_set` is its high performance. Unfortunately, this performance comes at a cost: `unordered_set` objects have a somewhat complicated interior structure. You have various knobs and dials you can use to inspect and modify this internal structure at runtime.

The first control measure you have is to customize the bucket count of the `unordered_set` (that is, the number of buckets, not the number of elements in a particular bucket). Each `unordered_set` constructor takes a `size_t bucket_count` as its first argument, which defaults to some implementation-defined value. Table 13-11 lists the main `unordered_set` constructors.

Table 13-11: The `unordered_set` Constructors

Operation	Notes
`unordered_set<T>( [bck], [hsh], [keq], [alc])`	Bucket size **bck** has an implementation-defined default value. Uses **hsh**=`std::hash<T>`, **keq**=`std::equal_to<T>`, and **alc**=`std::allocator<T>` by default.

Operation	Notes
`unordered_set<T>(..., [bck], [hsh], [keq], [alc])`	Performs braced initialization of a newly constructed unordered set.
`unordered_set<T>(beg, end [bck], [hsh], [keq], [alc])`	Constructs an unordered set with the elements on the half-open range from **beg** to **end**.
`unordered_set<T>(s)`	Deep copy of **s**; allocates new memory.
`unordered_set<T>(move(s))`	Takes ownership of memory; elements in **s**. No allocations.

You can inspect the number of buckets in an `unordered_set` using the `bucket_count` method. You can also obtain the maximum bucket count using the `max_bucket_count` method.

An important concept in the runtime performance of `unordered_set` is its *load factor*, the average number of elements per bucket. You can obtain the load factor of an `unordered_set` using the `load_factor` method, which is equivalent to `size()` divided by `bucket_count()`. Each `unordered_set` has a maximum load factor, which triggers an increase in the bucket count and a potentially expensive rehashing of all the contained elements. A *rehashing* is an operation where elements get reorganized into new buckets. This requires that you generate new hashes for each element, which can be a relatively computationally expensive operation.

You can obtain the maximum load factor using the `max_load_factor`, which is overloaded, so you can set a new maximum load factor (it defaults to 1.0).

To avoid expensive rehashing at inopportune times, you can manually trigger a rehashing using the `rehash` method, which accepts a `size_t` argument for the desired bucket count. You can also use the `reserve` method, which instead accepts a `size_t` argument for the desired *element* count.

Listing 13-31 illustrates some of these basic bucket management operations.

```
#include <unordered_set>
TEST_CASE("std::unordered_set") {
  std::unordered_set<unsigned long> sheep(100); ❶
  SECTION("allows bucket count specification on construction") {
    REQUIRE(sheep.bucket_count() >= 100); ❷
    REQUIRE(sheep.bucket_count() <= sheep.max_bucket_count()); ❸
    REQUIRE(sheep.max_load_factor() == Approx(1.0)); ❹
  }
  SECTION("allows us to reserve space for elements") {
    sheep.reserve(100'000); ❺
    sheep.insert(0);
    REQUIRE(sheep.load_factor() <= 0.00001); ❻

    while(sheep.size() < 100'000)
      sheep.insert(sheep.size()); ❼
    REQUIRE(sheep.load_factor() <= 1.0); ❽
  }
}
```

Listing 13-31: The `unordered_set` bucket management

You construct an `unordered_set` and specify a bucket count of 100 ❶. This results in a `bucket_count` of at least 100 ❷, which must be less than or equal to the `max_bucket_count` ❸. By default, the `max_load_factor` is 1.0 ❹.

In the next test, you invoke `reserve` with enough space for a hundred thousand elements ❺. After inserting an element, the `load_factor` should be less than or equal to one one-hundred-thousandth (0.00001) ❻ because you've reserved enough space for a hundred thousand elements. As long as you stay below this threshold, you won't need a rehashing. After inserting a hundred thousand elements ❼, the `load_factor` should still be less than or equal to 1 ❽. This demonstrates that you needed no rehashing, thanks to `reserve`.

Unordered Multisets

The `std::unordered_multiset` available in the STL's `<unordered_set>` header is an associative container that contains unsorted, *non-unique* keys. An `unordered_multiset` supports all the same constructors and operations as an `unordered_set`, but it will store redundant elements. This relationship is analogous to `unordered_sets` and `sets`: both `equal_range` and `count` have slightly different behavior to account for the non-uniqueness of keys.

NOTE *Boost also provides a `boost::unordered_multiset` in the `<boost/unordered_set.hpp>` header.*

Maps

The `std::map` available in the STL's `<map>` header is an associative container that contains key-value pairs. The keys of a `map` are sorted and unique, and `map` supports all the same operations as `set`. In fact, you can think of a `set` as a special kind of `map` containing keys and empty values. Accordingly, `map` supports efficient insertion, removal, and search, and you have control over sorting with comparator objects.

The major advantage of working with a map instead of a set of pairs is that `map` works as an *associative array*. An associative array takes a key rather than an integer-valued index. Think of how you use the `at` and `operator[]` methods to access indices in sequential containers. Because sequential containers have a natural ordering of elements, you use an integer to refer to them. The associative array allows you to use types other than integers to refer to elements. For example, you could use a string or a `float` as a key.

To enable associative array operations, `map` supports a number of useful operations; for example, allowing you to insert, modify, and retrieve values by their associated keys.

Constructing

The class template `map<Key, Value, Comparator, Allocator>` takes four template parameters. The first is the key type `Key`. The second is the value type `Value`. The third is the comparator type, which defaults to `std::less`. The fourth parameter is the allocator type, which defaults to `std::allocator<T>`.

The map constructors are direct analogues to the constructors of `set`: a default constructor that initializes an empty `map`; move and copy constructors with the usual behavior; a range constructor that copies the elements from the range into the map; and a braced initializer. The main difference is in the braced initializer, because you need to initialize key-value pairs instead of just keys. To achieve this nested initialization, you use nested initializer lists, as Listing 13-32 illustrates.

```
#include <map>

auto colour_of_magic = "Colour of Magic";
auto the_light_fantastic = "The Light Fantastic";
auto equal_rites = "Equal Rites";
auto mort = "Mort";

TEST_CASE("std::map supports") {
  SECTION("default construction") {
    std::map<const char*, int> emp; ❶
    REQUIRE(emp.empty()); ❷
  }
  SECTION("braced initialization") {
    std::map<const char*, int> pub_year { ❸
      { colour_of_magic, 1983 }, ❹
      { the_light_fantastic, 1986 },
      { equal_rites, 1987 },
      { mort, 1987 },
    };
    REQUIRE(pub_year.size() == 4); ❺
  }
}
```

Listing 13-32: A `std::map` supports default construction and braced initialization.

Here, you default construct a `map` with keys of type `const char*` and values of type `int` ❶. This results in an empty `map` ❷. In the second test, you again have a `map` with keys of type `const char*` and values of type `int` ❸, but this time you use braced initialization ❹ to pack four elements into the `map` ❺.

Move and Copy Semantics

The move and copy semantics of `map` are identical to those of `set`.

Storage Model

Both `map` and `set` use the same red-black tree internal structure.

Element Access

The major advantage to using a `map` instead of a `set` of `pair` objects is that `map` offers two associative array operations: `operator[]` and `at`. Unlike the sequential containers supporting these operations, like `vector` and `array`, which take a `size_t` index argument, `map` takes a `Key` argument and returns

a reference to the corresponding value. As with sequential containers, `at` will throw a `std::out_of_range` exception if the given `key` doesn't exist in the `map`. Unlike with sequential containers, `operator[]` won't cause undefined behavior if the `key` doesn't exist; instead, it will (silently) default construct a `Value` and insert the corresponding key-value pair into the map, even if you only intended to perform a read, as Listing 13-33 illustrates.

```
TEST_CASE("std::map is an associative array with") {
  std::map<const char*, int> pub_year { ❶
    { colour_of_magic, 1983 },
    { the_light_fantastic, 1986 },
  };
  SECTION("operator[]") {
    REQUIRE(pub_year[colour_of_magic] == 1983); ❷

    pub_year[equal_rites] = 1987; ❸
    REQUIRE(pub_year[equal_rites] == 1987); ❹

    REQUIRE(pub_year[mort] == 0); ❺
  }
  SECTION("an at method") {
    REQUIRE(pub_year.at(colour_of_magic) == 1983); ❻

    REQUIRE_THROWS_AS(pub_year.at(equal_rites), std::out_of_range); ❼
  }
}
```

Listing 13-33: A `std::map` is an associative array with several access methods.

You construct a `map` called `pub_year` containing two elements ❶. Next, you use `operator[]` to extract the value corresponding to the key `colour_of_magic` ❷. You also use `operator[]` to insert the new key-value pair `equal_rites`, `1987` ❸ and then retrieve it ❹. Notice that when you attempt to retrieve an element with the key `mort` (which doesn't exist), the map has silently default-initialized an `int` for you ❺.

Using `at`, you can still set and retrieve ❻ elements, but if you attempt to access a key that doesn't exist, you get a `std::out_of_range` exception ❼.

A `map` supports all the `set`-like, element-retrieval operations. For example, `map` supports `find`, which accepts a `key` argument and returns an iterator pointing to the key-value `pair` or, if no matching key is found, to the end of `map`. Also similarly supported are `count`, `equal_range`, `lower_bound`, and `upper_bound`.

Adding Elements

In addition to the element access methods `operator[]` and `at`, you also have all the `insert` and `emplace` methods available from `set`. You simply need to treat each key-value pair as a `std::pair<Key, Value>`. As with `set`, `insert` returns a pair containing an iterator and a `bool`. The iterator points to the inserted element, and the `bool` answers whether `insert` added a new element (`true`) or not (`false`), as Listing 13-34 illustrates.

```
TEST_CASE("std::map supports insert") {
  std::map<const char*, int> pub_year; ❶
  pub_year.insert({ colour_of_magic, 1983 }); ❷
  REQUIRE(pub_year.size() == 1); ❸

  std::pair<const char*, int> tlfp{ the_light_fantastic, 1986 }; ❹
  pub_year.insert(tlfp); ❺
  REQUIRE(pub_year.size() == 2); ❻

  auto [itr, is_new] = pub_year.insert({ the_light_fantastic, 9999 }); ❼
  REQUIRE(itr->first == the_light_fantastic);
  REQUIRE(itr->second == 1986); ❽
  REQUIRE_FALSE(is_new); ❾
  REQUIRE(pub_year.size() == 2); ❿
}
```

Listing 13-34: A `std::map` supports `insert` to add new elements.

You default construct a `map` ❶ and use the `insert` method with a braced initializer for a `pair` ❷. This construction is roughly equivalent to the following:

```
pub_year.insert(std::pair<const char*, int>{ colour_of_magic, 1983 });
```

After insertion, the `map` now contains one element ❸. Next, you create a stand-alone `pair` ❹ and then pass it as an argument to `insert` ❺. This inserts a copy into the `map`, so it now contains two elements ❻.

When you attempt to invoke `insert` with a new element with the same `the_light_fantastic` key ❼, you get an iterator pointing to the element you already inserted ❺. The key (`first`) and the value (`second`) match ❽. The return value `is_new` indicates that no new element was inserted ❾, and you still have two elements ❿. This behavior mirrors the `insert` behavior of `set`.

A `map` also offers an `insert_or_assign` method, which, unlike `insert`, will overwrite an existing value. Also unlike `insert`, `insert_or_assign` accepts separate key and value arguments, as Listing 13-35 illustrates.

```
TEST_CASE("std::map supports insert_or_assign") {
  std::map<const char*, int> pub_year{ ❶
    { the_light_fantastic, 9999 }
  };
  auto [itr, is_new] = pub_year.insert_or_assign(the_light_fantastic, 1986); ❷
  REQUIRE(itr->second == 1986); ❸
  REQUIRE_FALSE(is_new); ❹
}
```

Listing 13-35: A `std::map` supports `insert_or_assign` to overwrite existing elements.

You construct a `map` with a single element ❶ and then call `insert_or_assign` to reassign the value associated with the key `the_light_fantastic` to 1986 ❷. The iterator points to the existing element, and when you query the corresponding value with `second`, you see the value updated to 1986 ❸. The `is_new` return value also indicates that you've updated an existing element rather than inserting a new one ❹.

Removing Elements

Like set, map supports erase and clear to remove elements, as shown in Listing 13-36.

```
TEST_CASE("We can remove std::map elements using") {
    std::map<const char*, int> pub_year {
      { colour_of_magic, 1983 },
      { mort, 1987 },
    }; ❶
  SECTION("erase") {
    pub_year.erase(mort); ❷
    REQUIRE(pub_year.find(mort) == pub_year.end()); ❸
  }
  SECTION("clear") {
    pub_year.clear(); ❹
    REQUIRE(pub_year.empty()); ❺
  }
}
```

Listing 13-36: A `std::map` supports element removal.

You construct a map with two elements ❶. In the first test, you invoke erase on the element with key mort ❷, so when you try to find it, you get back end ❸. In the second test, you clear map ❹, which causes empty to return true ❺.

List of Supported Operations

Table 13-12 summarizes the supported operations of map. A key k has type K. A value v has type V. P is the type pair<K, V>, and p is of type P. The map m is map<K, V>. A dagger (†) denotes a method that returns a std::pair<Iterator, bool>, where the iterator points to the resulting element and the bool equals true if the method inserted an element and false if the element already existed.

Table 13-12: A Partial List of Supported map Operations

Operation	Notes
map<T>{ ..., [**cmp**], [**alc**] }	Performs braced initialization of a newly constructed map. Uses **cmp**=std::less<**T**> and **alc**=std::allocator<**T**> by default.
map<T>{ **beg**, **end**, [**cmp**], [**alc**] }	Range constructor that copies elements from the half-open range **beg** to **end**. Uses **cmp**=std::less<**T**> and **alc**=std::allocator<**T**> by default.
map<T>(**m**)	Deep copy of **m**; allocates new memory.
map<T>(move(**m**))	Takes ownership of memory; elements in **m**. No allocations.
~map	Destructs all elements contained by the map and releases dynamic memory.
m1 = **m2**	**m1** destructs its elements; copies each **m2** element. Only allocates if it needs to resize to fit **m2**'s elements.

Operation	Notes
`m1 = move(m2)`	`m1` destructs its elements; moves each `m2` element. Only allocates if it needs to resize to fit `m2`'s elements.
`m.at(k)`	Accesses the value corresponding to the key `k`. Throws `std::out_of_bounds` if key not found.
`m[k]`	Accesses the value corresponding to the key `k`. If the key is not found, inserts a new key-value pair using `k` and a default initialized value.
`m.begin()`	Returns an iterator pointing to the first element.
`m.cbegin()`	Returns a `const` iterator pointing to the first element.
`m.end()`	Returns an iterator pointing to 1 past the last element.
`m.cend()`	Returns a `const` iterator pointing to 1 past the last element.
`m.find(k)`	Returns an iterator pointing to the element matching `k`, or `m.end()` if no such element exists.
`m.count(k)`	Returns 1 if the map contains `k`; otherwise 0.
`m.equal_range(k)`	Returns a `pair` of iterators corresponding to the half-open range of elements matching `k`.
`m.lower_bound(k)`	Returns an iterator pointing to the first element not less than `k`, or `t.end()` if no such element exists.
`m.upper_bound(k)`	Returns an iterator pointing to the first element greater than `k`, or `t.end()` if no such element exists.
`m.clear()`	Removes all elements from the map.
`m.erase(k)`	Removes the element with key `k`.
`m.erase(itr)`	Removes the element pointed to by `itr`.
`m.erase(beg, end)`	Removes all elements on the half-open range from `beg` to `end`.
`m.insert(p)`	Inserts a copy of the pair `p` into the map.†
`m.insert_or_assign(k, v)`	If `k` exists, overwrites the corresponding value with `v`. If `k` doesn't exist, inserts the pair `k`, `v` into the map.†
`m.emplace(...)`	Constructs a `P` in place by forwarding the arguments `...`.†
`m.emplace_hint(k, ...)`	Constructs a `P` in place by forwarding the arguments `...`. Uses `itr` as a hint for where to insert the new element.†
`m.try_emplace(itr, ...)`	If key `k` exists, does nothing. If `k` doesn't exist, constructs a `V` in place by forwarding the arguments `...`.
`m.empty()`	Returns `true` if map's size is zero; otherwise `false`.
`m.size()`	Returns the number of elements in the map.
`m.max_size()`	Returns the maximum number of elements in the map.

(continued)

Table 13-12: A Partial List of Supported `map` Operation (continued)

Operation	Notes
`m.extract(k)` `m.extract(itr)`	Obtains a node handle that owns the element matching `k` or pointed to by `itr`. (This is the only way to remove a move-only element.)
`m1.merge(m2)` `m1.merge(move(m2))`	Splices each element of `m2` into `m1`. If argument is an rvalue, will move the elements into `m1`.
`m1.swap(m2)` `swap(m1, m2)`	Exchanges each element of `m1` with those of `m2`.

Multimaps

The `std::multimap` available in the STL's `<map>` header is an associative container that contains key-value pairs with *non-unique* keys. Because the keys are not unique, `multimap` doesn't support the associative array features that `map` does. Namely, `operator[]` and `at` aren't supported. As with `multiset`, `multimap` offers element access primarily through the `equal_range` method, as Listing 13-37 illustrates.

```
TEST_CASE("std::multimap supports non-unique keys") {
  std::array<char, 64> far_out {
    "Far out in the uncharted backwaters of the unfashionable end..."
  }; ❶
  std::multimap<char, size_t> indices; ❷
  for(size_t index{}; index<far_out.size(); index++)
    indices.emplace(far_out[index], index); ❸

  REQUIRE(indices.count('a') == 6); ❹

  auto [itr, end] = indices.equal_range('d'); ❺
  REQUIRE(itr->second == 23); ❻
  itr++;
  REQUIRE(itr->second == 59); ❼
  itr++;
  REQUIRE(itr == end);
}
```

Listing 13-37: A `std::multimap` supports non-unique keys.

You construct an `array` containing a message ❶. You also default construct a `multimap<char, size_t>` called `indices` that you'll use to store the index of every character in the message ❷. By looping through the array, you can store each character in the message along with its index as a new element in `multimap` ❸. Because you're allowed to have non-unique keys, you can use the `count` method to reveal how many indices you insert with the key `a` ❹. You can also use the `equal_range` method to obtain the half-open range of indices with the key `d` ❺. Using the resulting `begin` and `end` iterators, you can see that the message has the letter `d` at indices 23 ❻ and 59 ❼.

Aside from `operator[]` and `at`, every operation in Table 13-12 works for `multimap` as well. (Note that the `count` method can take on values other than 0 and 1.)

Unordered Maps and Unordered Multimaps

Unordered maps and unordered multimaps are completely analogous to unordered sets and unordered multisets. The `std::unordered_map` and `std::unordered_multimap` are available in the STL's `<unordered_map>` header. These associative containers typically use a red-black tree like their `set` counterparts. They also require a hash function and an equivalence function, and they support the bucket interface.

NOTE *Boost offers the `boost::unordered_map` and `boost::unordered_multimap` in the `<boost/unordered_map.hpp>` header.*

Niche Associative Containers

Use `set`, `map`, and their associated non-unique and unordered counterparts as the default choices when you need associative data structures. When special needs arise, Boost libraries offer a number of specialized associative containers, as highlighted in Table 13-13.

Table 13-13: Special Boost Containers

Class/Header	Description
`boost::container::flat_map` `<boost/container/flat_map.hpp>`	Similar to an STL `map`, but it's implemented like an ordered vector. This means fast random element access.
`boost::container::flat_set` `<boost/container/flat_set.hpp>`	Similar to an STL `set`, but it's implemented like an ordered vector. This means fast random element access.
`boost::intrusive::*` `<boost/intrusive/*.hpp>`	Intrusive containers impose requirements on the elements they contain (such as inheriting from a particular base class). In exchange, they offer substantial performance gains.
`boost::multi_index_container` `<boost/multi_index_container.hpp>`	Permits you to create associative arrays taking multiple indices rather than just one (like a map).
`boost::ptr_map` `boost::ptr_set` `boost::ptr_unordered_map` `boost::ptr_unordered_set` `<boost/ptr_container/*.hpp>`	Having a collection of smart pointers can be suboptimal. Pointer vectors manage a collection of dynamic objects in a more efficient and user-friendly way.
`boost::bimap` `< boost/bimap.hpp>`	A bimap is an associative container that allows both types to be used as a key.
`boost::heap::binomial_heap` `boost::heap::d_ary_heap` `boost::heap::fibonacci_heap` `boost::heap::pairing_heap` `boost::heap::priority_queue` `boost::heap::skew_heap` `<boost/heap/*.hpp>`	The Boost Heap containers implement more advanced, featureful versions of `priority_queue`.

Graphs and Property Trees

This section discusses two specialized Boost libraries that serve niche but valuable purposes: modeling graphs and property trees. A *graph* is a set of objects in which some have a pairwise relation. The objects are called *vertices,* and their relations are called *edges*. Figure 13-3 illustrates a graph containing four vertices and five edges.

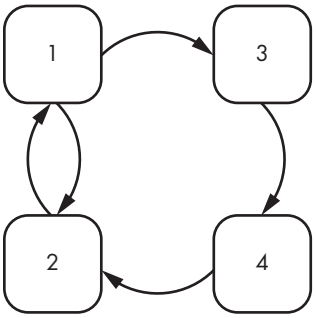


Figure 13-3: A graph containing four vertices and five edges

Each square represents a vertex, and each arrow represents an edge.

A *property tree* is a tree structure storing nested key-value pairs. The hierarchical nature of a property tree's key-value pairs makes it a hybrid between a map and a graph; each key-value pair has a relation to other key-value pairs. Figure 13-4 illustrates an example property tree containing nested key-value pairs.

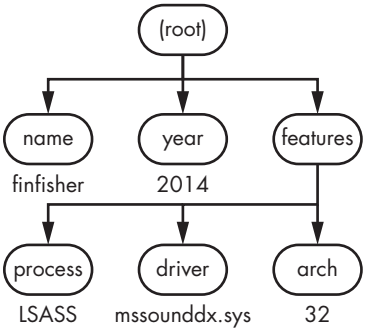


Figure 13-4: An example property tree

The root element has three children: name, year, and features. In Figure 13-4, name has a value finfisher, year has a value 2014, and features has three children: process with value LSASS, driver with value mssounddx .sys, and arch with value 32.

The Boost Graph Library

The *Boost Graph Library* (BGL) is a set of collections and algorithms for storing and manipulating graphs. The BGL offers three containers that represent graphs:

- The `boost::adjacency_list` in the `<boost/graph/adjacency_list.hpp>` header
- The `boost::adjacency_matrix` in the `<boost/graph/adjacency_matrix.hpp>` header
- The `boost::edge_list` in the `<boost/graph/ edge_list.hpp>` header

You use two non-member functions to build graphs: `boost::add_vertex` and `boost::add_edge`. To add a vertex to one of the BGL graph containers, you pass the graph object to `add_vertex`, which will return reference to the new vertex object. To add an edge, we pass the source vertex, the destination vertex, then the graph to `add_edge`.

BGL contains a number of graph-specific algorithms. You can count the number of vertices in a graph object by passing it to the non-member function `boost::num_vertices` and the number of edges using `boost::num_edges`. You can also query a graph for adjacent vertices. Two vertices are *adjacent* if they share an edge. To get the vertices adjacent to a particular vertex, you can pass it and the graph object to the non-member function `boost::adjacent_vertices`. This returns a half-open range as a `std::pair` of iterators.

Listing 13-38 illustrates how you can build the graph represented in Figure 13-3, count its vertices and edges, and compute adjacent vertices.

```
#include <set>
#include <boost/graph/adjacency_list.hpp>

TEST_CASE("boost::adjacency_list stores graph data") {
  boost::adjacency_list<> graph{}; ❶
  auto vertex_1 = boost::add_vertex(graph);
  auto vertex_2 = boost::add_vertex(graph);
  auto vertex_3 = boost::add_vertex(graph);
  auto vertex_4 = boost::add_vertex(graph); ❷
  auto edge_12 = boost::add_edge(vertex_1, vertex_2, graph);
  auto edge_13 = boost::add_edge(vertex_1, vertex_3, graph);
  auto edge_21 = boost::add_edge(vertex_2, vertex_1, graph);
  auto edge_24 = boost::add_edge(vertex_2, vertex_4, graph);
  auto edge_43 = boost::add_edge(vertex_4, vertex_3, graph); ❸

  REQUIRE(boost::num_vertices(graph) == 4); ❹
  REQUIRE(boost::num_edges(graph) == 5); ❺

  auto [begin, end] = boost::adjacent_vertices(vertex_1, graph); ❻
  std::set<decltype(vertex_1)> neighboors_1 { begin, end }; ❼
  REQUIRE(neighboors_1.count(vertex_2) == 1); ❽
  REQUIRE(neighboors_1.count(vertex_3) == 1); ❾
  REQUIRE(neighboors_1.count(vertex_4) == 0); ❿
}
```

Listing 13-38: The `boost::adjacency_list` stores graph data.

Here, you've constructed an `adjacency_list` called `graph` ❶, then added four vertices using `add_vertex` ❷. Next, you add all the edges represented in Figure 13-3 using `add_edge` ❸. Then `num_vertices` shows you that you've added four vertices ❹, and `num_edges` tells you that you've added five edges ❺.

Finally, you've determined the `adjacent_vertices` to `vertex_1`, which you unpack into the iterators `begin` and `end` ❻. You use these iterators to construct a `std::set` ❼, which you use to show that `vertex_2` ❽ and `vertex_3` ❾ are adjacent, but `vertex_4` is not ❿.

Boost Property Trees

Boost offers the `boost::property_tree::ptree` in the `<boost/property_tree/ptree.hpp>` header. This is a property tree that permits us to build and query property trees, as well as some limited serialization into various formats.

The tree `ptree` is default constructible. Default constructing will build an empty `ptree`.

You can insert elements into a `ptree` using its `put` method, which takes a path and a value argument. A *path* is a sequence of one or more nested keys separated by a period (`.`), and a *value* is an arbitrarily typed object.

You can remove subtrees from a `ptree` using the `get_child` method, which takes the path of the desired subtree. If the subtree does not have any children (a so-called *leaf node*), you can also use the method template `get_value` to extract the corresponding value from the key-value pair; `get_value` takes a single template parameter corresponding to the desired output type.

Finally, `ptree` supports serialization and deserialization to several formats including Javascript object notation (JSON), the Windows initialization file (INI) format, the extensible markup language (XML), and a custom, `ptree`-specific format called INFO. For example, to write a `ptree` into a file in JSON format, you could use the `boost::property_tree::write_json` function from the `<boost/property_tree/json_parser.hpp>` header. The function `write_json` accepts two arguments: the path to the desired output file and a `ptree` reference.

Listing 13-39 highlights these basic `ptree` functions by building a `ptree` representing the property tree in Figure 13-4, writing the `ptree` to file as JSON, and reading it back.

```
#include <boost/property_tree/ptree.hpp>
#include <boost/property_tree/json_parser.hpp>

TEST_CASE("boost::property_tree::ptree stores tree data") {
  using namespace boost::property_tree;
  ptree p; ❶
  p.put("name", "finfisher");
  p.put("year", 2014);
  p.put("features.process", "LSASS");
  p.put("features.driver", "mssounddx.sys");
  p.put("features.arch", 32); ❷
```

```
  REQUIRE(p.get_child("year").get_value<int>() == 2014); ❸

  const auto file_name = "rootkit.json";
  write_json(file_name, p); ❹

  ptree p_copy;
  read_json(file_name, p_copy); ❺
  REQUIRE(p_copy == p); ❻
}
--------------------------------------------------------------------------------
{
    "name": "finfisher",
    "year": "2014",
    "features": {
        "process": "LSASS",
        "driver": "mssounddx.sys",
        "arch": "32"
    }
} ❹
```

Listing 13-39: The `boost::property_tree::ptree` method stores tree data. Output shows the contents of `rootkit.json`.

Here, you've default constructed a `ptree` ❶, which you populate with the key values shown in Figure 13-4. Keys with parents, such as `arch` ❷, use periods to show the appropriate path. Using `get_child`, you've extracted the subtree for key `year`. Because it's a leaf node (having no children), you also invoke `get_value`, specifying the output type as `int` ❸.

Next, you write the `ptree`'s JSON representation to the file `rootkit.json` ❹. To ensure that you get the same property tree back, you default construct another `ptree` called `p_copy` and pass it into `read_json` ❺. This copy is equivalent to the original ❻, illustrating that the serialization-deserialization operation is successful.

Initializer Lists

You can accept initializer lists in your user-defined types by incorporating the `std::initializer_list` container available in the STL's `<initializer_list>` header. The `initializer_list` is a class template that takes a single template parameter corresponding to the underlying type contained in the initializer list. This template serves as a simple proxy for accessing the elements of an initializer list.

The `initializer_list` is immutable and supports three operations:

- The `size` method returns the number of elements in the `initializer_list`.
- The `begin` and `end` methods return the usual half-open-range iterators.

Generally, you should design functions to accept an `initializer_list` by value.

Listing 13-40 implements a SquareMatrix class that stores a matrix with equal numbers of rows and columns. Internally, the class will hold elements in a vector of vectors.

```
#include <cmath>
#include <stdexcept>
#include <initializer_list>
#include <vector>

size_t square_root(size_t x) { ❶
  const auto result = static_cast<size_t>(sqrt(x));
  if (result * result != x) throw std::logic_error{ "Not a perfect square." };
  return result;
}

template <typename T>
struct SquareMatrix {
  SquareMatrix(std::initializer_list<T> val) ❷
    : dim{ square_root(val.size()) }, ❸
      data(dim, std::vector<T>{}) { ❹
    auto itr = val.begin(); ❺
    for(size_t row{}; row<dim; row++){
      data[row].assign(itr, itr+dim); ❻
      itr += dim; ❼
    }
  }
  T& at(size_t row, size_t col) {
    if (row >= dim || col >= dim)
      throw std::out_of_range{ "Index invalid." }; ❽
    return data[row][col]; ❾
  }
  const size_t dim;
private:
  std::vector<std::vector<T>> data;
};
```

Listing 13-40: An implementation of a SquareMatrix

Here, you declare a convenience square_root function that finds the square root of a size_t, throwing an exception if the argument isn't a perfect square ❶. The SquareMatrix class template defines a single constructor that accepts a std::initializer called val ❷. This permits braced initialization.

First, you need to determine the dimensions of SquareMatrix. Use the square_root function to compute the square root of val.size() ❸ and store this into the dim field, which represents the number of rows and columns of the SquareMatrix instance. You can then use dim to initialize the vector of vectors data using its fill constructor ❹. Each of these vectors will correspond to a row in SquareMatrix. Next, you extract an iterator pointing to the first element in initializer_list ❺. You iterate over each row in SquareMatrix, assigning the corresponding vector to the appropriate half-open range ❻. You increment the iterator on each iteration to point to the next row ❼.

Finally, you implement an `at` method to permit element access. You perform bounds checking ❽ and then return a reference to the desired element by extracting the appropriate `vector` and element ❾.

Listing 13-41 illustrates how to use braced initialization to generate a `SquareMatrix` object.

```
TEST_CASE("SquareMatrix and std::initializer_list") {
  SquareMatrix<int> mat { ❶
     1,  2,  3,  4,
     5,  0,  7,  8,
     9, 10, 11, 12,
    13, 14, 15, 16
  };
  REQUIRE(mat.dim == 4); ❷
  mat.at(1, 1) = 6; ❸
  REQUIRE(mat.at(1, 1) == 6); ❹
  REQUIRE(mat.at(0, 2) ==  3); ❺
}
```

Listing 13-41: Using braced initializers with a `SquareMatrix`

You use braced initializers to set up `SquareMatrix` ❶. Because the initializer list contains 16 elements, you end up with a `dim` of 4 ❷. You can use `at` to obtain a reference to any element, meaning you can set ❸ and get ❹❺ elements.

Summary

This chapter began with a discussion of the two go-to sequence containers, `array` and `vector`, which offer you a great balance between performance and features in a wide range of applications. Next, you learned about several sequence containers—`deque`, `list`, `stack`, `queue`, `priority_queue`, and `bitset`—that fill in when `vector` doesn't meet the demands of a particular application. Then you explored the major associative containers, `set` and `map`, and their unordered/multipermutations. You also learned about two niche Boost containers, `graph` and `ptree`. The chapter wrapped up with a brief discussion of incorporating `initializer_lists` into user-defined types.

EXERCISES

13-1. Write a program that default constructs a `std::vector` of unsigned longs. Print the `capacity` of `vector` and then `reserve` 10 elements. Next, append the first 20 elements of the Fibonacci series to the vector. Print `capacity` again. Does `capacity` match the number of elements in the vector? Why or why not? Print the elements of `vector` using a range-based `for` loop.

(continued)

13-2. Rewrite Listings 2-9, 2-10, and 2-11 in Chapter 2 using `std::array`.

13-3. Write a program that accepts any number of command line arguments and prints them in alphanumerically sorted order. Use a `std::set<const char*>` to store the elements, then iterate over the `set` to obtain the sorted result. You'll need to implement a custom comparator that compares two C-style strings.

13-4. Write a program that default constructs a `std::vector` of unsigned longs. Print the `capacity` of `vector` and then `reserve` 10 elements. Next, append the first 20 elements of the Fibonacci series to the vector. Print `capacity` again. Does `capacity` match the number of elements in the vector? Why or why not? Print the elements of `vector` using a range-based `for` loop.

13-5. Consider the following program that profiles the performance of a function summing a Fibonacci series:

```
#include <chrono>
#include <cstdio>
#include <random>

long fib_sum(size_t n) { ❶
  // TODO: Adapt code from Exercise 12.1
  return 0;
}

long random() { ❷
  static std::mt19937_64 mt_engine{ 102787 };
  static std::uniform_int_distribution<long> int_d{ 1000, 2000 };
  return int_d(mt_engine);
}

struct Stopwatch { ❸
  Stopwatch(std::chrono::nanoseconds& result)
    : result{ result },
    start{ std::chrono::system_clock::now() } { }
  ~Stopwatch() {
    result = std::chrono::system_clock::now() - start;
  }
private:
  std::chrono::nanoseconds& result;
  const std::chrono::time_point<std::chrono::system_clock> start;
};

long cached_fib_sum(const size_t& n) { ❹
  static std::map<long, long> cache;
  // TODO: Implement me
  return 0;
}

int main() {

  size_t samples{ 1'000'000 };
  std::chrono::nanoseconds elapsed;
```

```
  {
    Stopwatch stopwatch{elapsed};
    volatile double answer;
    while(samples--) {
      answer = fib_sum(random()); ❺
      //answer = cached_fib_sum(random()); ❻
    }
  }
  printf("Elapsed: %g s.\n", elapsed.count() / 1'000'000'000.); ❼
}
```

This program contains a computationally intensive function `fib_sum` ❶ that computes the sum of a Fibonacci series with a given length. Adapt your code from Exercise 13-1 by (a) generating the appropriate vector and (b) summing over the result with a range-based `for` loop. The `random` function ❷ returns a random number between 1,000 and 2,000, and the `Stopwatch` class ❸ adopted from Listing 12-25 in Chapter 12 helps you determine elapsed time. In the program's `main`, you perform a million evaluations of the `fib_sum` function using random input ❺. You time how long this takes and print the result before exiting the program ❼. Compile the program and run it a few times to get an idea of how long your program takes to run. (This is called a *baseline*.)

13-6. Next, comment out ❺ and uncomment ❻. Implement the function `cached_fib_sum` ❹ so you first check whether you've computed `fib_sum` for the given length yet. (Treat the length `n` as a key into the cache.) If the key is present in the cache, simply return the result. If the key isn't present, compute the correct answer with `fib_sum`, store the new key-value entry into `cache`, and return the result. Run the program again. Is it faster? Try `unordered_map` instead of `map`. Could you use a `vector` instead? How fast can you get the program to run?

13-7. Implement a `Matrix` class like `SquareMatrix` in Listing 13-38. Your `Matrix` should allow unequal numbers of rows and columns. Accept as your constructor's first argument the number of rows in `Matrix`.

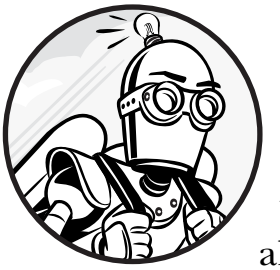
FURTHER READING

- *ISO International Standard ISO/IEC (2017) — Programming Language C++* (International Organization for Standardization; Geneva, Switzerland; *https://isocpp.org/std/the-standard/*)
- *The Boost C++ Libraries*, 2nd Edition, by Boris Schäling (XML Press, 2014)
- *The C++ Standard Library: A Tutorial and Reference*, 2nd Edition, by Nicolai M. Josuttis (Addison-Wesley Professional, 2012)

14

ITERATORS

Say "friend" and enter.
—J.R.R. Tolkein, The Lord of the Rings



Iterators are the STL component that provides the interface between containers and algorithms to manipulate them. An iterator is an interface to a type that knows how to traverse a particular sequence and exposes simple, pointer-like operations to elements.

Every iterator supports at least the following operations:

- Access the current element (`operator*`) for reading and/or writing
- Go to the next element (`operator++`)
- Copy construct

Iterators are categorized based on which additional operations they support. These categories determine which algorithms are available and what you can do with an iterator in your generic code. In this chapter, you'll learn about these iterator categories, convenience functions, and adapters.

Iterator Categories

An iterator's category determines the operations it supports. These operations include reading and writing elements, iterating forward and backward, reading multiple times, and accessing random elements.

Because code that accepts an iterator is usually generic, the iterator's type is typically a template parameter that you can encode with concepts, which you learned about in "Concepts" on page 163. Although you probably won't have to interact with iterators directly (unless you're writing a library), you'll still need to know the iterator categories so you don't try to apply an algorithm to inappropriate iterators. If you do, you're likely to get cryptic compiler errors. Recall from "Type Checking in Templates" on page 161 that because of how templates instantiate, error messages generated from inappropriate type arguments are usually inscrutable.

Output Iterators

You can use an *output iterator* to write into and increment but nothing else. Think of an output iterator as a bottomless pit that you throw data into.

When using an output iterator, you write, then increment, then write, then increment, ad nauseam. Once you've written to an output iterator, you cannot write again until you've incremented at least once. Likewise, once you've incremented an output iterator, you cannot increment again before writing.

To write to an output iterator, dereference the iterator using the dereference operator (*) and assign a value to the resulting reference. To increment an output iterator, use `operator++` or `operator++(int)`.

Again, unless you're writing a C++ library, it's unlikely that you'll have to *implement* your own output iterator types; however, you'll *use* them quite a lot.

One prominent usage is writing into containers as if they were output iterators. For this, you use insert iterators.

Insert Iterators

An *insert iterator* (or *inserter*) is an output iterator that wraps a container and transforms writes (assignments) into insertions. Three insert iterators exist in the STL's `<iterator>` header as class templates:

- `std::back_insert_iterator`
- `std::front_insert_iterator`
- `std::insert_iterator`

The STL also offers three convenience functions for building these iterators:

- `std::back_inserter`
- `std::front_inserter`
- `std::inserter`