Listing 12-37 illustrates how to multiply 10 by 2/3 at compile time.

```
#include <ratio>

TEST_CASE("std::ratio") {
  using ten = std::ratio<10, 1>; ❶
  using two_thirds = std::ratio<2, 3>; ❷
  using result = std::ratio_multiply<ten, two_thirds>; ❸
  REQUIRE(result::num == 20); ❹
  REQUIRE(result::den == 3); ❺
}
```

Listing 12-37: Compile time rational arithmetic with `std::ratio`

You declare the `std::ratio` types `ten` ❶ and `two_thirds` ❷ as type aliases. To compute the product of `ten` and `two_thirds`, you again declare another type, `result`, using the `std::ratio_multiply` template ❸. Using the static members `num` and `den`, you can extract the result, 20/3 ❹❺.

Of course, it's always better to do computation at compile time rather than at runtime when you can. Your programs will be more efficient because they'll need to do less computation when they run.

A Partial List of Random Number Distributions

Table 12-16 contains a partial list of the operations provided by stdlib's `<ratio>` library.

Table 12-16: A Partial List of Operations Available in `<ratio>`

Distribution	Notes
`ratio_add<r1, r2>`	Adds `r1` and `r2`
`ratio_subtract<r1, r2>`	Subtracts `r2` from `r1`
`ratio_multiply<r1, r2>`	Multiplies `r1` and `r2`
`ratio_divide<r1, r2>`	Divides `r1` by `r2`
`ratio_equal<r1, r2>`	Tests whether `r1` equals `r2`
`ratio_not_equal<r1, r2>`	Tests whether `r1` is not equal to `r2`
`ratio_less<r1, r2>`	Tests whether `r1` is less than `r2`
`ratio_greater<r1, r2>`	Tests whether `r1` is greater than `r2`
`ratio_less_equal<r1, r2>`	Tests whether `r1` is less than or equal to `r2`
`ratio_greater_equal<r1, r2>`	Tests whether `r1` is greater than or equal to `r2`
`micro`	Literal: `ratio<1, 1000000>`
`milli`	Literal: `ratio<1, 1000>`
`centi`	Literal: `ratio<1, 100>`
`deci`	Literal: `ratio<1, 10>`
`deca`	Literal: `ratio<10, 1>`

Distribution	Notes
hecto	Literal: ratio<100, 1>
kilo	Literal: ratio<1000, 1>
mega	Literal: ratio<1000000, 1>
giga	Literal: ratio<1000000000, 1>

Summary

In this chapter, you examined a potpourri of small, simple, focused utilities that service common programming needs. Data structures, such as `tribool`, `optional`, `pair`, `tuple`, `any`, and `variant` handle many commonplace scenarios in which you need to contain objects within a common structure. In the coming chapters, a few of these data structures will make repeat appearances throughout the stdlib. You also learned about date/time and numerics/math facilities. These libraries implement very specific functionality, but when you have such requirements, these libraries are invaluable.

EXERCISES

12-1. Reimplement the `narrow_cast` in Listing 6-6 to return a `std::optional`. If the cast would result in a narrowing conversion, return an empty optional rather than throwing an exception. Write a unit test that ensures your solution works.

12-2. Implement a program that generates random alphanumeric passwords and writes them to the console. You can store the alphabet of possible characters into a `char[]` and use the discrete uniform distribution with a minimum of zero and a maximum of the last index of your alphabet array. Use a cryptographically secure random number engine.

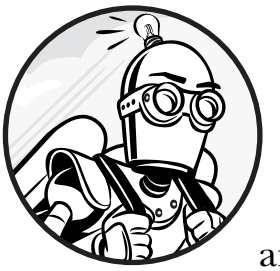
FURTHER READING

- *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)

13

CONTAINERS

Fixing bugs in `std::vector` *is equal parts delight (it is the bestest data structure) and terror (if I mess it up, the world explodes).*
—Stephan T. Lavavej (Principal Developer, Visual C++ Libraries). Tweet dated 3:11 AM on August 22, 2016.



The *standard template library (STL)* is the portion of the stdlib that provides containers and the algorithms to manipulate them, with iterators serving as the interface between the two. In the next three chapters, you'll learn more about each of these components.

A *container* is a special data structure that stores objects in an organized way that follows specific access rules. There are three kinds of containers:

- Sequence containers store elements consecutively, as in an array.
- Associative containers store sorted elements.
- Unordered associative containers store hashed objects.

Associative and unordered associative containers offer rapid search for individual elements. All containers are RAII wrappers around their contained objects, so they manage the storage durations and lifetimes of the elements they own. Additionally, each container provides some set of member functions that perform various operations on the object collection.

Modern C++ programs use containers all the time. Which container you choose for a particular application depends on the required operations, the contained objects' characteristics, and efficiencies under particular access patterns. This chapter surveys the vast container landscape covered between the STL and Boost. Because there are so many containers in these libraries, you'll explore the most popular ones.

Sequence Containers

Sequence containers are STL containers that allow sequential member access. That is, you can start from one end of the container and iterate through to the other end. But except for this commonality, sequence containers are a varied and motley crew. Some containers have a fixed length; others can shrink and grow as program needs dictate. Some allow indexing directly into the container, whereas you can only access others sequentially. Additionally, each sequence container has unique performance characteristics that make it desirable in some situations and undesirable in others.

Working with sequence containers should feel intuitive because you've been acquainted with a primitive one since "Arrays" on page 42, where you saw the built-in or "C-style" array `T[]`. You'll begin the survey of sequence containers by looking at the built-in array's more sophisticated, cooler younger brother `std::array`.

Arrays

The STL provides `std::array` in the `<array>` header. An `array` is a sequential container that holds a fixed-size, contiguous series of elements. It combines the sheer performance and efficiency of built-in arrays with the modern conveniences of supporting copy/move construction/assignment, knowing its own size, providing bounds-checked member access, and other advanced features.

You should use `array` instead of built-in arrays in virtually all situations. It supports almost all the same usage patterns as `operator[]` to access elements, so there aren't many situations in which you'll need a built-in array instead.

NOTE *Boost also offers a `boost::array` in Boost Array's `<boost/array.hpp>`. You shouldn't need to use the Boost version unless you have a very old C++ tool chain.*

Constructing

The `array<T, S>` class template takes two template parameters:

- The contained type `T`
- The fixed size of the array `S`

You can construct an array and built-in arrays using the same rules. To summarize these rules from "Arrays" on page 42, the preferred method is to use braced initialization to construct an array. Braced initialization fills the array with the values contained in the braces and fills the remaining elements with zeros. If you omit initialization braces, the array contains uninitialized values depending on its storage duration. Listing 13-1 illustrates braced initialization with several array declarations.

```
#include <array>

std::array<int, 10> static_array; ❶

TEST_CASE("std::array") {
  REQUIRE(static_array[0] == 0); ❷

  SECTION("uninitialized without braced initializers") {
    std::array<int, 10> local_array; ❸
    REQUIRE(local_array[0] != 0); ❹
  }

  SECTION("initialized with braced initializers") {
    std::array<int, 10> local_array{ 1, 1, 2, 3 }; ❺
    REQUIRE(local_array[0] == 1);
    REQUIRE(local_array[1] == 1);
    REQUIRE(local_array[2] == 2);
    REQUIRE(local_array[3] == 3);
    REQUIRE(local_array[4] == 0); ❻
  }
}
```

Listing 13-1: Initializing a `std::array`. You might get compiler warnings from `REQUIRE(local_array[0] != 0);` ❹, since `local_array` has uninitialized elements.

You declare an array of 10 int objects called static_array with static storage duration ❶. You haven't used braced initialization, but its elements initialize to zero anyway ❷, thanks to the initialization rules covered in "Arrays" on page 42.

Next, you try declaring another array of 10 int objects, this time with automatic storage duration ❸. Because you haven't used braced initialization, local_array contains uninitialized elements (that have an extremely low probability of equaling zero ❹).

Finally, you use braced initialization to declare another array and to fill the first four elements ❺. All remaining elements get set to zero ❻.

Element Access

The three main methods by which you can access arbitrary array elements are:

- operator[]
- at
- get

The operator[] and at methods take a single size_t argument corresponding to the index of the desired element. The difference between these two lies in bounds checking: if the index argument is out of bounds, at will throw a std::out_of_range exception, whereas operator[] will cause undefined behavior. The function template get takes a template parameter of the same specification. Because it's a template, the index must be known at compile time.

NOTE *Recall from "The size_t Type" on page 41 that a size_t object guarantees that its maximum value is sufficient to represent the maximum size in bytes of all objects. It is for this reason that operator[] and at take a size_t rather than an int, which makes no such guarantee.*

A major bonus of using get is that you get compile-time bounds checking, as illustrated in Listing 13-2.

```
TEST_CASE("std::array access") {
   std::array<int, 4> fib{ 1, 1, 0, 3}; ❶

  SECTION("operator[] can get and set elements") {
    fib[2] = 2; ❷
    REQUIRE(fib[2] == 2); ❸
    // fib[4] = 5; ❹
  }

  SECTION("at() can get and set elements") {
    fib.at(2) = 2; ❺
    REQUIRE(fib.at(2) == 2); ❻
    REQUIRE_THROWS_AS(fib.at(4), std::out_of_range); ❼
  }
  SECTION("get can get and set elements") {
    std::get<2>(fib) = 2; ❽
    REQUIRE(std::get<2>(fib) == 2); ❾
    // std::get<4>(fib); ❿
  }
}
```

Listing 13-2: Accessing elements of an array. Uncommenting // fib[4] = 5; ❹ will cause undefined behavior, whereas uncommenting // std::get<4>(fib); ❿ will cause compilation failure.

You declare an array of length 4 called fib ❶. Using operator[] ❷ you can set elements and retrieve them ❸. The out of bounds write you've commented out would cause undefined behavior; there is no bounds checking with operator[] ❹.

You can use at for the same read ❺ and write ❻ operations, and you can safely perform an out-of-bounds operation thanks to bounds checking ❼.

Finally, you can use std::get to set ❽ and get ❾ elements. The get element also performs bounds checking, so // std::get<4>(fib); ❿ will fail to compile if uncommented.

You've also have a `front` and a `back` method, which return references to the first and last elements of the array. You'll get undefined behavior if you call one of these methods if the array has zero length, as Listing 13-3 illustrates.

```
TEST_CASE("std::array has convenience methods") {
  std::array<int, 4> fib{ 0, 1, 2, 0 };

  SECTION("front") {
    fib.front() = 1; ❶
    REQUIRE(fib.front() == 1); ❷
    REQUIRE(fib.front() == fib[0]); ❸
  }

  SECTION("back") {
    fib.back() = 3; ❹
    REQUIRE(fib.back() == 3); ❺
    REQUIRE(fib.back() == fib[3]); ❻
  }
}
```

Listing 13-3: Using the convenience methods `front` and `back` on a `std::array`

You can use the `front` and `back` methods to set ❶❹ and get ❷❺ the first and last elements of an `array`. Of course, `fib[0]` is identical to `fib.front()` ❸, and `fib[3]` is identical to `fib.back()` ❻. The `front()` and `back()` methods are simply convenience methods. Additionally, if you're writing generic code, some containers will offer `front` and `back` but not `operator[]`, so it's best to use the `front` and `back` methods.

Storage Model

An `array` doesn't make allocations; rather, like a built-in array, it contains all of its elements. This means copies will generally be expensive, because each constituent element needs to be copied. Moves can be expensive, depending on whether the underlying type of the `array` also has move construction and move assignment, which are relatively inexpensive.

Each `array` is just a built-in array underneath. In fact, you can extract a pointer to the first element of an `array` using four distinct methods:

- The go-to method is to use the `data` method. As advertised, this returns a pointer to the first element.
- The other three methods involve using the address-of operator `&` on the first element, which you can obtain using `operator[]`, `at`, and `front`.

You should use `data`. If the `array` is empty, the address-of-based approaches will return undefined behavior.

Listing 13-4 illustrates how to obtain a pointer using these four methods.

```
TEST_CASE("We can obtain a pointer to the first element using") {
  std::array<char, 9> color{ 'o',  'c', 't', 'a', 'r', 'i', 'n', 'e' };
  const auto* color_ptr = color.data(); ❶

  SECTION("data") {
    REQUIRE(*color_ptr == 'o'); ❷
  }
  SECTION("address-of front") {
    REQUIRE(&color.front() == color_ptr); ❸
  }
  SECTION("address-of at(0)") {
    REQUIRE(&color.at(0) == color_ptr); ❹
  }
  SECTION("address-of [0]") {
    REQUIRE(&color[0] == color_ptr); ❺
  }
}
```

Listing 13-4: Obtaining a pointer to the first element of a `std::array`

After initializing the `array color`, you obtain a pointer to the first element, the letter `o`, using the `data` method ❶. When you dereference the resulting `color_ptr`, you obtain the letter `o` as expected ❷. This pointer is identical to the pointer obtained from the `address-of-plus-front` ❸, `-at` ❹, and `-operator[]` ❺ approaches.

To conclude arrays, you can query the size of an `array` using either the `size` or `max_size` methods. (These are identical for an `array`.) Because an `array` has a fixed size, these method's values are static and known at compile time.

A Crash Course in Iterators

The interface between containers and algorithms is the iterator. An iterator is a type that knows the internal structure of a container and exposes simple, pointer-like operations to a container's elements. Chapter 14 is dedicated entirely to iterators, but you need to know the very basics here so you can explore how to use iterators to manipulate containers and how containers expose iterators to users.

Iterators come in various flavors, but they all support at least the following operations:

1. Get the current element (`operator*`)
2. Go to the next element (`operator++`)
3. Assign an iterator equal to another iterator (`operator=`)

You can extract iterators from all STL containers (including `array`) using their `begin` and `end` methods. The `begin` method returns an iterator pointing to the first element, and the `end` method returns a pointer to one element past the last element. Figure 13-1 illustrates where the `begin` and `end` iterators point in an array of three elements.

Figure 13-1: A half-open range over an array of three elements

The arrangement in Figure 13-1, where `end()` points after the last element, is called a *half-open range*. It might seem counterintuitive at first—why not have a closed range where `end()` points to the last element—but a half-open range has some advantages. For example, if a container is empty, `begin()` will return the same value as `end()`. This allows you to know that, regardless of whether the container is empty, if the iterator equals `end()`, you've traversed the container.

Listing 13-5 illustrates what happens with half-open range iterators and empty containers.

```
TEST_CASE("std::array begin/end form a half-open range") {
  std::array<int, 0> e{}; ❶
  REQUIRE(e.begin()❷ == e.end()❸);
}
```

Listing 13-5: With an empty `array`, the `begin` iterator equals the `end` iterator.

Here, you construct an empty array `e` ❶, and the `begin` ❷ and `end` ❸ iterators are equal.

Listing 13-6 examines how to use iterators to perform pointer-like operations over a non-empty `array`.

```
TEST_CASE("std::array iterators are pointer-like") {
  std::array<int, 3> easy_as{ 1, 2, 3 }; ❶
  auto iter = easy_as.begin(); ❷
  REQUIRE(*iter == 1); ❸
  ++iter; ❹
  REQUIRE(*iter == 2);
  ++iter;
  REQUIRE(*iter == 3); ❺
  ++iter; ❻
  REQUIRE(iter == easy_as.end()); ❼
}
```

Listing 13-6: Basic `array` iterator operations

The `array` `easy_as` contains the elements 1, 2, and 3 ❶. You invoke `begin` on `easy_as` to obtain an iterator `iter` pointing to the first element ❷. The dereference operator yields the first element 1, because this is the first element in the `array` ❸. Next, you increment `iter` so it points to the next element ❹. You continue in this fashion until you reach the last element ❺. Incrementing the pointer one last time puts you 1 past the last element ❻, so `iter` equals `easy_as.end()`, indicating that you've traversed the `array` ❼.

Recall from "Range Expressions" on page 235 that you can build your own types for use in range expressions by exposing a `begin` and an `end` method, as implemented in the `FibonacciIterator` in Listing 8-29. Well, containers already do all this work for you, meaning you can use any STL container as a range expression. Listing 13-7 illustrates by iterating over an `array`.

```
TEST_CASE("std::array can be used as a range expression") {
  std::array<int, 5> fib{ 1, 1, 2, 3, 5 }; ❶
  int sum{}; ❷
  for (const auto element : fib) ❸
    sum += element; ❹
  REQUIRE(sum == 12);
}
```

Listing 13-7: Range-based `for` loops and arrays

You initialize an `array` ❶ and a `sum` variable ❷. Because `array` is a valid range, you can use it in a ranged-based `for` loop ❸. This enables you to accumulate the `sum` of each `element` ❹.

A Partial List of Supported Operations

Table 13-1 provides a partial list of `array` operations. In this table, `a`, `a1`, and `a2` are of type `std::array<T, S>`, `t` is of type `T`, `S` is the fixed length of the array, and `i` is of type `size_t`.

Table 13-1: A Partial List of `std::array` Operations

Operation	Notes
`array<T, S>{ ... }`	Performs braced initialization of a newly constructed array.
`~array`	Destructs all elements contained by the array.
`a1` `=` **`a2`**	Copy-assigns all the members of **`a1`** with the members of **`a2`**.
`a``.at(`**`i`**`)`	Returns a reference to element **`i`** of **`a`**. Throws `std::out_of_range` if out of bounds.
`a``[`**`i`**`]`	Returns a reference to element **`i`** of **`a`**. Undefined behavior if out of bounds.
`get<i>(`**`a`**`)`	Returns a reference to element **`i`** of **`a`**. Fails to compile if out of bounds.
`a``.front()`	Returns a reference to first element.
`a``.back()`	Returns a reference to last element.
`a``.data()`	Returns a raw pointer to the first element if the array is non-empty. For empty arrays, returns a valid but non-dereferencable pointer.
`a``.empty()`	Returns `true` if the array's size is zero; otherwise `false`.
`a``.size()`	Returns the size of the array.
`a``.max_size()`	Identical to **`a`**`.size()`.
`a``.fill(`**`t`**`)`	Copy-assigns **`t`** to every element of **`a`**.

Operation	Notes
`a1.swap(a2)` `swap(a1, a2)`	Exchanges each element of `a1` with those of `a2`.
`a.begin()`	Returns an iterator pointing to the first element.
`a.cbegin()`	Returns a `const` iterator pointing to the first element.
`a.end()`	Returns an iterator pointing to 1 past the last element.
`a.cend()`	Returns a `const` iterator pointing to 1 past the last element.

`a1 == a2`
`a1 != a2`
`a1 > a2`
`a1 >= a2`
`a1 < a2`
`a1 <= a2`
	Equal if all elements are equal.
	Greater than/less than comparisons proceed from first element to last.

NOTE *The partial operations in Table 13-1 function as quick, reasonably comprehensive references. For gritty details, refer to the freely available online references* https://cppreference.com/ *and* http://cplusplus.com/*, as well as Chapter 31 of* The C++ Programming Language*, 4th Edition, by Bjarne Stroustrup and Chapters 7, 8, and 12 of* The C++ Standard Library*, 2nd Edition, by Nicolai M. Josuttis.*

Vectors

The `std::vector` available in the STL's `<vector>` header is a sequential container that holds a dynamically sized, contiguous series of elements. A `vector` manages its storage dynamically, requiring no outside help from the programmer.

The `vector` is the workhorse of the sequential-data-structure stable. For a very modest overhead, you gain substantial flexibility over the `array`. Plus, `vector` supports almost all of the same operations as an `array` and adds a slew of others. If you have a fixed number of elements on hand, you should strongly consider an `array` because you'll get some small reductions in overhead versus a `vector`. In all other situations, your go-to sequential container is the `vector`.

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

Constructing

The class template `std::vector<T, Allocator>` takes two template parameters. The first is the contained type `T`, and the second is the allocator type `Allocator`, which is optional and defaults to `std::allocator<T>`.

You have much more flexibility in constructing `vector`s than you do with arrays. A `vector` supports user-defined allocators because vectors need to allocate dynamic memory. You can default construct a `vector` so it contains no elements. You might want to construct an empty vector so you can fill it with a variable number of elements depending on what happens during

runtime. Listing 13-8 illustrates default constructing a `vector` and checking that it contains no elements.

```
#include <vector>
TEST_CASE("std::vector supports default construction") {
  std::vector<const char*❶> vec; ❷
  REQUIRE(vec.empty()); ❸
}
```

Listing 13-8: A `vector` supports default construction.

You declare a `vector` containing elements of type `const char*` ❶ called `vec`. Because it's been default constructed ❷, the `vector` contains no elements, and the `empty` method returns `true` ❸.

You can use braced initialization with a `vector`. Similar to how you brace initialize an array, this fills the vector with the specified elements, as Listing 13-9 illustrates.

```
TEST_CASE("std::vector supports braced initialization ") {
    std::vector<int> fib{ 1, 1, 2, 3, 5 }; ❶
    REQUIRE(fib[4] == 5); ❷
}
```

Listing 13-9: A `vector` supports braced initializers.

Here, you construct a `vector` called `fib` and use braced initializers ❶. After initialization, the `vector` contains the five elements 1, 1, 2, 3, and 5 ❷.

If you want to populate a `vector` with many identical values, you can use one of the *fill constructors*. To fill construct a `vector`, you first pass a `size_t` corresponding to the number of elements you want to fill. Optionally, you can pass a `const` reference to an object to copy. Sometimes you want to initialize all your elements to the same value, for example, to keep track of counts related to particular indices. You might also have a `vector` of some user-defined type that keeps track of program state, and you might need to keep track of such state by index.

Unfortunately, the general rule to use braced initialization to construct objects breaks down here. With `vector`, you must use parentheses to invoke these constructors. To the compiler, `std::vector<int>{ 99, 100 }` specifies an initialization list with the elements 99 and 100, which will construct a vector with the two elements 99 and 100. What if you want a vector with 99 copies of the number 100?

In general, the compiler will try very hard to treat the initializer list as elements to fill the vector with. You can try to memorize the rules (refer to Item 7 of *Effective Modern C++* by Scott Meyers) or just commit to using parentheses for stdlib container constructors.

Listing 13-10 highlights the initializer list/braced initialization general rule for STL containers.

```
TEST_CASE("std::vector supports") {
  SECTION("braced initialization") {
    std::vector<int> five_nine{ 5, 9 }; ❶
```

```
    REQUIRE(five_nine[0] == 5); ❷
    REQUIRE(five_nine[1] == 9); ❸
  }
  SECTION("fill constructor") {
    std::vector<int> five_nines(5, 9); ❹
    REQUIRE(five_nines[0] == 9); ❺
    REQUIRE(five_nines[4] == 9); ❻
  }
}
```

Listing 13-10: A `vector` supports braced initializers and fill constructors.

The first example uses braced initialization to construct a vector with two elements ❶: 5 at index 0 ❷ and 9 at index 1 ❸. The second example uses parentheses to invoke the fill constructor ❹, which fills the vector with five copies of the number 9, so the first ❺ and last ❻ elements are both 9.

NOTE *This notational clash is unfortunate and isn't the result of some well-thought-out trade-off. The reasons are purely historical and related to backward compatibility.*

You can also construct `vector`s from a half-open range by passing in the `begin` and `end` iterators of the range you want to copy. In various programming contexts, you might want to splice out a subset of some range and copy it into a `vector` for further processing. For example, you could construct a `vector` that copies all the elements contained by an `array`, as in Listing 13-11.

```
TEST_CASE("std::vector supports construction from iterators") {
  std::array<int, 5> fib_arr{ 1, 1, 2, 3, 5 }; ❶
  std::vector<int> fib_vec(fib_arr.begin(), fib_arr.end()); ❷
  REQUIRE(fib_vec[4] == 5); ❸
  REQUIRE(fib_vec.size() == fib_arr.size()); ❹
}
```

Listing 13-11: Constructing a `vector` from a range

You construct the array `fib_arr` with five elements ❶. To construct the vector `fib_vec` with the elements contained in `fib_arr`, you invoke the `begin` and `end` methods on `fib_arr` ❷. The resulting `vector` has copies of the `array`'s elements ❸ and has the same `size` ❹.

At a high level, you can think of this constructor as taking pointers to the beginning and the end of some target sequence. It will then copy that target sequence.

Move and Copy Semantics

With `vector`s, you have full copy/move construction/assignment support. Any `vector` copy operation is potentially very expensive, because these are element-wise or deep copies. Move operations, on the other hand, are usually very fast, because the contained elements reside in dynamic memory and the moved-from `vector` can simply pass ownership to the moved-into `vector`; there's no need to move the contained elements.

Element Access

A vector supports most of the same element access operations as array: at, operator[], front, back, and data.

As with an array, you can query the number of contained elements in a vector using the size method. This method's return value can vary at runtime. You can also determine whether a vector contains any elements with the empty method, which returns true if the vector contains no elements; otherwise, it returns false.

Adding Elements

You can use various methods to insert elements into a vector. If you want to replace all the elements in a vector, you can use the assign method, which takes an initialization list and replaces all the existing elements. If needed, the vector will resize to accommodate a larger list of elements, as Listing 13-12 illustrates.

```
TEST_CASE("std::vector assign replaces existing elements") {
  std::vector<int> message{ 13, 80, 110, 114, 102, 110, 101 }; ❶
  REQUIRE(message.size() == 7); ❷
  message.assign({ 67, 97, 101, 115, 97, 114 }); ❸
  REQUIRE(message[5] == 114); ❹
  REQUIRE(message.size() == 6); ❺
}
```

Listing 13-12: The assign method of a vector

Here, you construct a vector ❶ with seven elements ❷. When you assign a new, smaller initializer list ❸, all the elements get replaced ❹, and the vector's size updates to reflect the new contents ❺.

If you want to insert a single new element into a vector, you can use the insert method, which expects two arguments: an iterator and an element to insert. It will insert a copy of the given element just before the existing element pointed to by the iterator, as shown in Listing 13-13.

```
TEST_CASE("std::vector insert places new elements") {
  std::vector<int> zeros(3, 0); ❶
  auto third_element = zeros.begin() + 2; ❷
  zeros.insert(third_element, 10); ❸
  REQUIRE(zeros[2] == 10); ❹
  REQUIRE(zeros.size() == 4); ❺
}
```

Listing 13-13: The insert method of a vector

You initialize a vector with three zeros ❶ and generate an iterator pointing to the third element of zeros ❷. Next, you insert the value 10 immediately before the third element by passing the iterator and the value 10 ❸. The third element of zeros is now 10 ❹. The zeros vector now contains four elements ❺.

Any time you use insert, existing iterators become invalid. For example, in Listing 13-13 you must not reuse third_element: the vector could have resized and relocated in memory, leaving the old iterator dangling in garbage memory.

To insert an element to the end of a vector, you use the push_back method. Unlike insert, push_back doesn't require an iterator argument. You simply provide the element to copy into the vector, as shown in Listing 13-14.

```
TEST_CASE("std::vector push_back places new elements") {
  std::vector<int> zeros(3, 0); ❶
  zeros.push_back(10); ❷
  REQUIRE(zeros[3] == 10); ❸
}
```

Listing 13-14: The push_back method of a vector

Again, you initialize a vector with three zeros ❶, but this time you insert the element 10 to the back of the vector using the push_back method ❷. The vector now contains four elements, the last of which equals 10 ❸.

You can construct new elements in place using the emplace and emplace_back methods. The emplace method is a variadic template that, like insert, accepts an iterator as its first argument. The remaining arguments get forwarded to the appropriate constructor. The emplace_back method is also a variadic template, but like push_back, it doesn't require an iterator. It accepts any number of arguments and forwards those to the appropriate constructor. Listing 13-15 illustrates these two methods by emplacing a few pairs into a vector.

```
#include <utility>

TEST_CASE("std::vector emplace methods forward arguments") {
  std::vector<std::pair<int, int>> factors; ❶
  factors.emplace_back(2, 30); ❷
  factors.emplace_back(3, 20); ❸
  factors.emplace_back(4, 15); ❹
  factors.emplace(factors.begin()❺, 1, 60);
  REQUIRE(factors[0].first == 1); ❻
  REQUIRE(factors[0].second == 60); ❼
}
```

Listing 13-15: The emplace_back and emplace methods of a vector

Here, you default construct a vector containing pairs of ints ❶. Using the emplace_back method, you push three pairs onto the vector: 2, 30 ❷; 3, 20 ❸; and 4, 15 ❹. These values get forwarded directly to the constructor of pair, which gets constructed in place. Next, you use emplace to insert a new pair at the beginning of the vector by passing the result of factors.begin() as the first argument ❺. This causes all the elements in the vector to shift down to make room for the new pair (1 ❻, 60 ❼).

NOTE *There's absolutely nothing special about a* `std::vector<std::pair<int, int>>`*. It's just like any other* `vector`*. The individual elements in this sequential container just happen to be a* `pair`*. Because* `pair` *has a constructor that accepts two arguments, one for* `first` *and one for* `second`*,* `emplace_back` *can add a new element by simply passing the two values it should write into the newly created* `pair`*.*

Because the emplacement methods can construct elements in place, it seems they should be more efficient than the insertion methods. This intuition is often correct, but for complicated and unsatisfying reasons it's not always faster. As a general rule, use the emplacement methods. If you determine a performance bottleneck, also try the insertion methods. See Item 42 of *Effective Modern C++* by Scott Meyers for a treatise.

Storage Model

Although `vector` elements are contiguous in memory, like an `array`, the similarities stop there. A `vector` has dynamic size, so it must be able to resize. The allocator of a `vector` manages the dynamic memory underpinning the `vector`.

Because allocations are expensive, a `vector` will request more memory than it needs to contain the current number of elements. Once it can no longer add any more elements, it will request additional memory. The memory for a `vector` is always contiguous, so if there isn't enough space at the end of the existing vector, it will allocate a whole new region of memory and move all the elements of the `vector` into the new region. The number of elements a `vector` holds is called its *size*, and the number of elements it could theoretically hold before having to resize is called its *capacity*. Figure 13-2 illustrates a `vector` containing three elements with additional capacity for three more.

Figure 13-2: The `vector` *storage model*

As Figure 13-2 shows, the `vector` continues past the last element. The capacity determines how many elements the `vector` could hold in this space. In this figure, the size is three and the capacity is six. You can think of the memory in a `vector` as an auditorium: it might have a capacity of 500 but a crowd size of only 250.

The upshot of this design is that inserting at the end of a `vector` is extremely fast (unless the `vector` needs to resize). Inserting anywhere else incurs additional cost, because the `vector` needs to move elements around to make room.

You can obtain the vector's current capacity via the `capacity` method, and you can obtain the absolute maximum capacity that a `vector` could resize to with the `max_size` method.

If you know ahead of time that you'll need a certain capacity, you can use the `reserve` method, which takes a single `size_t` argument corresponding to the number of elements you want capacity for. On the other hand, if you've just removed several elements and want to return memory to the allocator, you can use the `shrink_to_fit` method, which declares that you have excess capacity. The allocator can decide to reduce capacity or not (it's a non-binding call).

Additionally, you can delete all the elements in a vector and set its size to zero using the `clear` method.

Listing 13-16 demonstrates all these storage-related methods in a cohesive story: you create an empty vector, reserve a bunch of space, add some elements, release excess capacity, and finally empty the vector.

```
#include <cstdint>
#include <array>

TEST_CASE("std::vector exposes size management methods") {
  std::vector<std::array<uint8_t, 1024>> kb_store; ❶
  REQUIRE(kb_store.max_size() > 0);
  REQUIRE(kb_store.empty()); ❷

  size_t elements{ 1024 };
  kb_store.reserve(elements); ❸
  REQUIRE(kb_store.empty());
  REQUIRE(kb_store.capacity() == elements); ❹

  kb_store.emplace_back();
  kb_store.emplace_back();
  kb_store.emplace_back();
  REQUIRE(kb_store.size() == 3); ❺

  kb_store.shrink_to_fit();
  REQUIRE(kb_store.capacity() >= 3); ❻

  kb_store.clear(); ❼
  REQUIRE(kb_store.empty());
  REQUIRE(kb_store.capacity() >= 3); ❽
}
```

Listing 13-16: The storage management functions of a `vector`. (Strictly speaking, `kb_store.capacity() >= 3` ❻❽ is not guaranteed because the call is non-binding.)

You construct a `vector` of `array` objects called `kb_store`, which stores 1 KiB chunks ❶. Unless you're using a peculiar platform with no dynamic memory, `kb_store.max_size()` will be greater than zero; because you default initialize the `vector`, it's empty ❷.