

C H A P T E R

9

SEQUENTIAL CONTAINERS

CONTENTS

---

|                 |   |     |
|-----------------|---|-----|
| Section 9.1     | Overview of the Sequential Containers . .           | 326 |
| Section 9.2     | Container Library Overview . . . . .                | 328 |
| Section 9.3     | Sequential Container Operations . . . . .           | 341 |
| Section 9.4     | How a <code>vector</code> Grows . . . . .           | 355 |
| Section 9.5     | Additional <code>string</code> Operations . . . . . | 360 |
| Section 9.6     | Container Adaptors . . . . .                        | 368 |
| Chapter Summary | . . . . .   | 372 |
| Defined Terms   | . . . . .   | 372 |

This chapter expands on the material from Chapter 3 and completes our discussion of the standard-library sequential containers. The order of the elements in a sequential container corresponds to the positions in which the elements are added to the container. The library also defines several associative containers, which hold elements whose position depends on a key associated with each element. We'll cover operations specific to the associative containers in Chapter 11.

The container classes share a common interface, which each of the containers extends in its own way. This common interface makes the library easier to learn; what we learn about one kind of container applies to another. Each kind of container offers a different set of performance and functionality trade-offs.

A *container* holds a collection of objects of a specified type. The **sequential containers** let the programmer control the order in which the elements are stored and accessed. That order does not depend on the values of the elements. Instead, the order corresponds to the position at which elements are put into the container. By contrast, the ordered and unordered associative containers, which we cover in Chapter 11, store their elements based on the value of a key.

The library also provides three container adaptors, each of which adapts a container type by defining a different interface to the container’s operations. We cover the adaptors at the end of this chapter.



This chapter builds on the material covered in § 3.2, § 3.3, and § 3.4. We assume that the reader is familiar with the material covered there.



# 9.1 Overview of the Sequential Containers

The sequential containers, which are listed in Table 9.1, all provide fast sequential access to their elements. However, these containers offer different performance trade-offs relative to

- The costs to add or delete elements to the container
- The costs to perform nonsequential access to elements of the container

Table 9.1: Sequential Container Types

|              |   |
|--------------|---|
| vector       | Flexible-size array. Supports fast random access. Inserting or deleting elements other than at the back may be slow.                    |
| deque        | Double-ended queue. Supports fast random access. Fast insert/delete at front or back.   |
| list         | Doubly linked list. Supports only bidirectional sequential access. Fast insert/delete at any point in the list.                         |
| forward_list | Singly linked list. Supports only sequential access in one direction. Fast insert/delete at any point in the list.                      |
| array        | Fixed-size array. Supports fast random access. Cannot add or remove elements.   |
| string       | A specialized container, similar to <code>vector</code> , that contains characters. Fast random access. Fast insert/delete at the back. |

With the exception of `array`, which is a fixed-size container, the containers provide efficient, flexible memory management. We can add and remove elements, growing and shrinking the size of the container. The strategies that the containers use for storing their elements have inherent, and sometimes significant, impact on the efficiency of these operations. In some cases, these strategies also affect whether a particular container supplies a particular operation.

For example, `string` and `vector` hold their elements in contiguous memory. Because elements are contiguous, it is fast to compute the address of an element

from its index. However, adding or removing elements in the middle of one of these containers takes time: All the elements after the one inserted or removed have to be moved to maintain contiguity. Moreover, adding an element can sometimes require that additional storage be allocated. In that case, every element must be moved into the new storage.

The `list` and `forward_list` containers are designed to make it fast to add or remove an element anywhere in the container. In exchange, these types do not support random access to elements: We can access an element only by iterating through the container. Moreover, the memory overhead for these containers is often substantial, when compared to `vector`, `deque`, and `array`.

A `deque` is a more complicated data structure. Like `string` and `vector`, `deque` supports fast random access. As with `string` and `vector`, adding or removing elements in the middle of a `deque` is a (potentially) expensive operation. However, adding or removing elements at either end of the `deque` is a fast operation, comparable to adding an element to a `list` or `forward_list`.

The `forward_list` and `array` types were added by the new standard. An `array` is a safer, easier-to-use alternative to built-in arrays. Like built-in arrays, library arrays have fixed size. As a result, `array` does not support operations to add and remove elements or to resize the container. A `forward_list` is intended to be comparable to the best handwritten, singly linked list. Consequently, `forward_list` does not have the `size` operation because storing or computing its size would entail overhead compared to a handwritten list. For the other containers, `size` is guaranteed to be a fast, constant-time operation.

**C++  
11**

For reasons we'll explain in § 13.6 (p. 531), the new library containers are dramatically faster than in previous releases. The library containers almost certainly perform as well as (and usually better than) even the most carefully crafted alternatives. Modern C++ programs should use the library containers rather than more primitive structures like arrays.

## Deciding Which Sequential Container to Use



Ordinarily, it is best to use `vector` unless there is a good reason to prefer another container.

There are a few rules of thumb that apply to selecting which container to use:

- Unless you have a reason to use another container, use a `vector`.
- If your program has lots of small elements and space overhead matters, don't use `list` or `forward_list`.
- If the program requires random access to elements, use a `vector` or a `deque`.
- If the program needs to insert or delete elements in the middle of the container, use a `list` or `forward_list`.
- If the program needs to insert or delete elements at the front and the back, but not in the middle, use a `deque`.

- If the program needs to insert elements in the middle of the container only while reading input, and subsequently needs random access to the elements:
  - First, decide whether you actually need to add elements in the middle of a container. It is often easier to append to a `vector` and then call the library `sort` function (which we shall cover in § 10.2.3 (p. 384)) to reorder the container when you're done with input.
  - If you must insert into the middle, consider using a `list` for the input phase. Once the input is complete, copy the `list` into a `vector`.

What if the program needs random access *and* needs to insert and delete elements in the middle of the container? This decision will depend on the relative cost of accessing the elements in a `list` or `forward_list` versus the cost of inserting or deleting elements in a `vector` or `deque`. In general, the predominant operation of the application (whether it does more access or more insertion or deletion) will determine the choice of container type. In such cases, performance testing the application using both containers will probably be necessary.



If you're not sure which container to use, write your code so that it uses only operations common to both `vectors` and `lists`: Use iterators, not subscripts, and avoid random access to elements. That way it will be easy to use either a `vector` or a `list` as necessary.

## EXERCISES SECTION 9.1

**Exercise 9.1:** Which is the most appropriate—a `vector`, a `deque`, or a `list`—for the following program tasks? Explain the rationale for your choice. If there is no reason to prefer one or another container, explain why not.

- Read a fixed number of words, inserting them in the container alphabetically as they are entered. We'll see in the next chapter that associative containers are better suited to this problem.
- Read an unknown number of words. Always insert new words at the back. Remove the next value from the front.
- Read an unknown number of integers from a file. Sort the numbers and then print them to standard output.



## 9.2 Container Library Overview

The operations on the container types form a kind of hierarchy:

- Some operations (Table 9.2 (p. 330)) are provided by all container types.
- Other operations are specific to the sequential (Table 9.3 (p. 335)), the associative (Table 11.7 (p. 438)), or the unordered (Table 11.8 (p. 445)) containers.
- Still others are common to only a smaller subset of the containers.

In this section, we'll cover aspects common to all of the containers. The remainder of this chapter will then focus solely on sequential containers; we'll cover operations specific to the associative containers in Chapter 11.

In general, each container is defined in a header file with the same name as the type. That is, `deque` is in the `deque` header, `list` in the `list` header, and so on. The containers are class templates (§ 3.3, p. 96). As with `vectors`, we must supply additional information to generate a particular container type. For most, but not all, of the containers, the information we must supply is the element type:

```
list<Sales_data>    // list that holds Sales_data objects
deque<double>      // deque that holds doubles
```

## Constraints on Types That a Container Can Hold

Almost any type can be used as the element type of a sequential container. In particular, we can define a container whose element type is itself another container. We define such containers exactly as we do any other container type: We specify the element type (which in this case is a container type) inside angle brackets:

```
vector<vector<string>> lines;    // vector of vectors
```

Here `lines` is a vector whose elements are vectors of strings.

C++  
11



Older compilers may require a space between the angle brackets, for example, `vector<vector<string> >`.

Although we can store almost any type in a container, some container operations impose requirements of their own on the element type. We can define a container for a type that does not support an operation-specific requirement, but we can use an operation only if the element type meets that operation's requirements.

As an example, the sequential container constructor that takes a size argument (§ 3.3.1, p. 98) uses the element type's default constructor. Some classes do not have a default constructor. We can define a container that holds objects of such types, but we cannot construct such containers using only an element count:

```
// assume noDefault is a type without a default constructor
vector<noDefault> v1(10, init); // ok: element initializer supplied
vector<noDefault> v2(10);      // error: must supply an element initializer
```

As we describe the container operations, we'll note the additional constraints, if any, that each container operation places on the element type.

## EXERCISES SECTION 9.2

**Exercise 9.2:** Define a `list` that holds elements that are deques that hold `ints`.

Table 9.2: Container Operations

**Type Aliases**

|                              |   |
|------------------------------|---|
| <code>iterator</code>        | Type of the iterator for this container type  |
| <code>const_iterator</code>  | Iterator type that can read but not change its elements   |
| <code>size_type</code>       | Unsigned integral type big enough to hold the size of the largest possible container of this container type |
| <code>difference_type</code> | Signed integral type big enough to hold the distance between two iterators                                  |
| <code>value_type</code>      | Element type  |
| <code>reference</code>       | Element's lvalue type; synonym for <code>value_type&amp;</code>   |
| <code>const_reference</code> | Element's const lvalue type (i.e., <code>const value_type&amp;</code> )                                     |

**Construction**

|                             |  |
|-----------------------------|--|
| <code>C c;</code>           | Default constructor, empty container (array; see p. 336)   |
| <code>C c1(c2);</code>      | Construct <code>c1</code> as a copy of <code>c2</code>   |
| <code>C c(b, e);</code>     | Copy elements from the range denoted by iterators <code>b</code> and <code>e</code> ; <b>(not valid for array)</b> |
| <code>C c{a,b,c...};</code> | List initialize <code>c</code>   |

**Assignment and swap**

|                              |   |
|------------------------------|---|
| <code>c1 = c2</code>         | Replace elements in <code>c1</code> with those in <code>c2</code>                       |
| <code>c1 = {a,b,c...}</code> | Replace elements in <code>c1</code> with those in the list <b>(not valid for array)</b> |
| <code>a.swap(b)</code>       | Swap elements in <code>a</code> with those in <code>b</code>                            |
| <code>swap(a, b)</code>      | Equivalent to <code>a.swap(b)</code>  |

**Size**

|                           |  |
|---------------------------|--|
| <code>c.size()</code>     | Number of elements in <code>c</code> <b>(not valid for forward_list)</b> |
| <code>c.max_size()</code> | Maximum number of elements <code>c</code> can hold                       |
| <code>c.empty()</code>    | false if <code>c</code> has any elements, true otherwise                 |

**Add/Remove Elements (not valid for array)**

*Note: the interface to these operations varies by container type*

|                               |   |
|-------------------------------|---|
| <code>c.insert(args)</code>   | Copy element(s) as specified by <code>args</code> into <code>c</code> |
| <code>c.emplace(inits)</code> | Use <code>inits</code> to construct an element in <code>c</code>      |
| <code>c.erase(args)</code>    | Remove element(s) specified by <code>args</code>                      |
| <code>c.clear()</code>        | Remove all elements from <code>c</code> ; returns void                |

**Equality and Relational Operators**

|                                       |   |
|---------------------------------------|---|
| <code>==, !=</code>                   | Equality valid for all container types                              |
| <code>&lt;, &lt;=, &gt;, &gt;=</code> | Relationals <b>(not valid for unordered associative containers)</b> |

**Obtain Iterators**

|                                   |   |
|-----------------------------------|---|
| <code>c.begin(), c.end()</code>   | Return iterator to the first, one past the last element in <code>c</code> |
| <code>c.cbegin(), c.cend()</code> | Return <code>const_iterator</code>  |

**Additional Members of Reversible Containers (not valid for forward\_list)**

|                                     |   |
|-------------------------------------|---|
| <code>reverse_iterator</code>       | Iterator that addresses elements in reverse order                         |
| <code>const_reverse_iterator</code> | Reverse iterator that cannot write the elements                           |
| <code>c.rbegin(), c.rend()</code>   | Return iterator to the last, one past the first element in <code>c</code> |
| <code>c.crbegin(), c.crend()</code> | Return <code>const_reverse_iterator</code>                                |

## 9.2.1 Iterators



As with the containers, iterators have a common interface: If an iterator provides an operation, then the operation is supported in the same way for each iterator that supplies that operation. For example, all the iterators on the standard container types let us access an element from a container, and they all do so by providing the dereference operator. Similarly, the iterators for the library containers all define the increment operator to move from one element to the next.

With one exception, the container iterators support all the operations listed in Table 3.6 (p. 107). The exception is that the `forward_list` iterators do not support the decrement (`--`) operator. The iterator arithmetic operations listed in Table 3.7 (p. 111) apply only to iterators for `string`, `vector`, `deque`, and `array`. We cannot use these operations on iterators for any of the other container types.

### Iterator Ranges



The concept of an iterator range is fundamental to the standard library.

An **iterator range** is denoted by a pair of iterators each of which refers to an element, or to *one past the last element*, in the same container. These two iterators, often referred to as `begin` and `end`—or (somewhat misleadingly) as `first` and `last`—mark a range of elements from the container.

The name `last`, although commonly used, is a bit misleading, because the second iterator never refers to the last element of the range. Instead, it refers to a point one past the last element. The elements in the range include the element denoted by `first` and every element from `first` up to but not including `last`.

This element range is called a **left-inclusive interval**. The standard mathematical notation for such a range is

`[ begin, end )`

indicating that the range begins with `begin` and ends with, but does not include, `end`. The iterators `begin` and `end` must refer to the same container. The iterator `end` may be equal to `begin` but must not refer to an element before the one denoted by `begin`.

#### REQUIREMENTS ON ITERATORS FORMING AN ITERATOR RANGE

Two iterators, `begin` and `end`, form an iterator range, if

- They refer to elements of, or one past the end of, the same container, and
- It is possible to reach `end` by repeatedly incrementing `begin`. In other words, `end` must not precede `begin`.



The compiler cannot enforce these requirements. It is up to us to ensure that our programs follow these conventions.

## Programming Implications of Using Left-Inclusive Ranges

The library uses left-inclusive ranges because such ranges have three convenient properties. Assuming `begin` and `end` denote a valid iterator range, then

- If `begin` equals `end`, the range is empty
- If `begin` is not equal to `end`, there is at least one element in the range, and `begin` refers to the first element in that range
- We can increment `begin` some number of times until `begin == end`

These properties mean that we can safely write loops such as the following to process a range of elements:

```
while (begin != end) {
    *begin = val;    // ok: range isn't empty so begin denotes an element
    ++begin;        // advance the iterator to get the next element
}
```

Given that `begin` and `end` form a valid iterator range, we know that if `begin == end`, then the range is empty. In this case, we exit the loop. If the range is nonempty, we know that `begin` refers to an element in this nonempty range. Therefore, inside the body of the `while`, we know that it is safe to dereference `begin` because `begin` must refer to an element. Finally, because the loop body increments `begin`, we also know the loop will eventually terminate.

### EXERCISES SECTION 9.2.1

**Exercise 9.3:** What are the constraints on the iterators that form iterator ranges?

**Exercise 9.4:** Write a function that takes a pair of iterators to a `vector<int>` and an `int` value. Look for that value in the range and return a `bool` indicating whether it was found.

**Exercise 9.5:** Rewrite the previous program to return an iterator to the requested element. Note that the program must handle the case where the element is not found.

**Exercise 9.6:** What is wrong with the following program? How might you correct it?

```
list<int> lst1;
list<int>::iterator iter1 = lst1.begin(),
                    iter2 = lst1.end();
while (iter1 < iter2) /* ... */
```

## 9.2.2 Container Type Members

Each container defines several types, shown in Table 9.2 (p. 330). We have already used three of these container-defined types: `size_type` (§ 3.2.2, p. 88), `iterator`, and `const_iterator` (§ 3.4.1, p. 108).



In addition to the iterator types we've already used, most containers provide reverse iterators. Briefly, a reverse iterator is an iterator that goes backward through a container and inverts the meaning of the iterator operations. For example, saying ++ on a reverse iterator yields the previous element. We'll have more to say about reverse iterators in § 10.4.3 (p. 407).

The remaining type aliases let us use the type of the elements stored in a container without knowing what that type is. If we need the element type, we refer to the container's `value_type`. If we need a reference to that type, we use `reference` or `const_reference`. These element-related type aliases are most useful in generic programs, which we'll cover in Chapter 16.

To use one of these types, we must name the class of which they are a member:

```
// iter is the iterator type defined by list<string>
list<string>::iterator iter;

// count is the difference_type type defined by vector<int>
vector<int>::difference_type count;
```

These declarations use the scope operator (§ 1.2, p. 8) to say that we want the iterator member of the `list<string>` class and the `difference_type` defined by `vector<int>`, respectively.

## EXERCISES SECTION 9.2.2

**Exercise 9.7:** What type should be used as the index into a vector of ints?

**Exercise 9.8:** What type should be used to read elements in a list of strings? To write them?

## 9.2.3 begin and end Members



The `begin` and `end` operations (§ 3.4.1, p. 106) yield iterators that refer to the first and one past the last element in the container. These iterators are most often used to form an iterator range that encompasses all the elements in the container.

As shown in Table 9.2 (p. 330), there are several versions of `begin` and `end`: The versions with an `r` return reverse iterators (which we cover in § 10.4.3 (p. 407)). Those that start with a `c` return the `const` version of the related iterator:

```
list<string> a = {"Milton", "Shakespeare", "Austen"};
auto it1 = a.begin(); // list<string>::iterator
auto it2 = a.rbegin(); // list<string>::reverse_iterator
auto it3 = a.cbegin(); // list<string>::const_iterator
auto it4 = a.crbegin(); // list<string>::const_reverse_iterator
```

The functions that do not begin with a `c` are overloaded. That is, there are actually two members named `begin`. One is a `const` member (§ 7.1.2, p. 258) that returns the container's `const_iterator` type. The other is `nonconst` and returns the container's `iterator` type. Similarly for `rbegin`, `end`, and `rend`. When we

call one of these members on a nonconst object, we get the version that returns iterator. We get a const version of the iterators *only* when we call these functions on a const object. As with pointers and references to const, we can convert a plain iterator to the corresponding const\_iterator, but not vice versa.

C++  
11

The c versions were introduced by the new standard to support using auto with begin and end functions (§ 2.5.2, p. 68). In the past, we had no choice but to say which type of iterator we want:

```
// type is explicitly specified
list<string>::iterator it5 = a.begin();
list<string>::const_iterator it6 = a.begin();
// iterator or const_iterator depending on a's type of a
auto it7 = a.begin(); // const_iterator only if a is const
auto it8 = a.cbegin(); // it8 is const_iterator
```

When we use auto with begin or end, the iterator type we get depends on the container type. How we intend to use the iterator is irrelevant. The c versions let us get a const\_iterator regardless of the type of the container.

Best  
Practices

When write access is not needed, use cbegin and cend.

## EXERCISES SECTION 9.2.3

**Exercise 9.9:** What is the difference between the begin and cbegin functions?

**Exercise 9.10:** What are the types of the following four objects?

```
vector<int> v1;
const vector<int> v2;
auto it1 = v1.begin(), it2 = v2.begin();
auto it3 = v1.cbegin(), it4 = v2.cbegin();
```



## 9.2.4 Defining and Initializing a Container

Every container type defines a default constructor (§ 7.1.4, p. 263). With the exception of array, the default constructor creates an empty container of the specified type. Again excepting array, the other constructors take arguments that specify the size of the container and initial values for the elements.

### Initializing a Container as a Copy of Another Container

There are two ways to create a new container as a copy of another one: We can directly copy the container, or (excepting array) we can copy a range of elements denoted by a pair of iterators.

To create a container as a copy of another container, the container and element types must match. When we pass iterators, there is no requirement that the container types be identical. Moreover, the element types in the new and original

Table 9.3: Defining and Initializing Containers

|   |  |
|---|--|
| <code>C c;</code>   | Default constructor. If <code>C</code> is array, then the elements in <code>c</code> are default-initialized; otherwise <code>c</code> is empty.   |
| <code>C c1(c2)</code><br><code>C c1 = c2</code>   | <code>c1</code> is a copy of <code>c2</code> . <code>c1</code> and <code>c2</code> must have the same type (i.e., they must be the same container type and hold the same element type; for array must also have the same size).  |
| <code>C c{a, b, c...}</code><br><code>C c = {a, b, c...}</code>                                     | <code>c</code> is a copy of the elements in the initializer list. Type of elements in the list must be compatible with the element type of <code>C</code> . For array, the list must have same number or fewer elements than the size of the array, any missing elements are value-initialized (§ 3.3.1, p. 98). |
| <code>C c(b, e)</code>  | <code>c</code> is a copy of the elements in the range denoted by iterators <code>b</code> and <code>e</code> . Type of the elements must be compatible with the element type of <code>C</code> .<br><b>(Not valid for array.)</b>  |
| <b>Constructors that take a size are valid for sequential containers (not including array) only</b> |  |
| <code>C seq(n)</code>   | <code>seq</code> has <code>n</code> value-initialized elements; this constructor is explicit (§ 7.5.4, p. 296). <b>(Not valid for string.)</b>   |
| <code>C seq(n, t)</code>  | <code>seq</code> has <code>n</code> elements with value <code>t</code> .   |

containers can differ as long as it is possible to convert (§ 4.11, p. 159) the elements we're copying to the element type of the container we are initializing:

```
// each container has three elements, initialized from the given initializers
list<string> authors = {"Milton", "Shakespeare", "Austen"};
vector<const char*> articles = {"a", "an", "the"};
list<string> list2(authors);           // ok: types match
deque<string> authList(authors);      // error: container types don't match
vector<string> words(articles);       // error: element types must match
// ok: converts const char* elements to string
forward_list<string> words(articles.begin(), articles.end());
```



When we initialize a container as a copy of another container, the container type and element type of both containers must be identical.

The constructor that takes two iterators uses them to denote a range of elements that we want to copy. As usual, the iterators mark the first and one past the last element to be copied. The new container has the same size as the number of elements in the range. Each element in the new container is initialized by the value of the corresponding element in the range.

Because the iterators denote a range, we can use this constructor to copy a subsequence of a container. For example, assuming it is an iterator denoting an element in `authors`, we can write

```
// copies up to but not including the element denoted by it
deque<string> authList(authors.begin(), it);
```

## List Initialization



Under the new standard, we can list initialize (§ 3.3.1, p. 98) a container:

```
// each container has three elements, initialized from the given initializers
list<string> authors = {"Milton", "Shakespeare", "Austen"};
vector<const char*> articles = {"a", "an", "the"};
```

When we do so, we explicitly specify values for each element in the container. For types other than array, the initializer list also implicitly specifies the size of the container: The container will have as many elements as there are initializers.

## Sequential Container Size-Related Constructors

In addition to the constructors that sequential containers have in common with associative containers, we can also initialize the sequential containers (other than array) from a size and an (optional) element initializer. If we do not supply an element initializer, the library creates a value-initialized one for us § 3.3.1 (p. 98):

```
vector<int> ivec(10, -1);           // ten int elements, each initialized to -1
list<string> svec(10, "hi!");       // ten strings; each element is "hi!"
forward_list<int> ivec(10);        // ten elements, each initialized to 0
deque<string> svec(10);            // ten elements, each an empty string
```

We can use the constructor that takes a size argument if the element type is a built-in type or a class type that has a default constructor (§ 9.2, p. 329). If the element type does not have a default constructor, then we must specify an explicit element initializer along with the size.



The constructors that take a size are valid *only* for sequential containers; they are not supported for the associative containers.

## Library arrays Have Fixed Size

Just as the size of a built-in array is part of its type, the size of a library array is part of its type. When we define an array, in addition to specifying the element type, we also specify the container size:

```
array<int, 42>           // type is: array that holds 42 ints
array<string, 10>        // type is: array that holds 10 strings
```

To use an array type we must specify both the element type and the size:

```
array<int, 10>::size_type i; // array type includes element type and size
array<int>::size_type j;    // error: array<int> is not a type
```

Because the size is part of the array's type, array does not support the normal container constructors. Those constructors, implicitly or explicitly, determine the size of the container. It would be redundant (at best) and error-prone to allow users to pass a size argument to an array constructor.

The fixed-size nature of arrays also affects the behavior of the constructors that array does define. Unlike the other containers, a default-constructed array

is not empty: It has as many elements as its size. These elements are default initialized (§ 2.2.1, p. 43) just as are elements in a built-in array (§ 3.5.1, p. 114). If we list initialize the array, the number of the initializers must be equal to or less than the size of the array. If there are fewer initializers than the size of the array, the initializers are used for the first elements and any remaining elements are value initialized (§ 3.3.1, p. 98). In both cases, if the element type is a class type, the class must have a default constructor in order to permit value initialization:

```
array<int, 10> ia1;    // ten default-initialized ints
array<int, 10> ia2 = {0,1,2,3,4,5,6,7,8,9}; // list initialization
array<int, 10> ia3 = {42}; // ia3[0] is 42, remaining elements are 0
```

It is worth noting that although we cannot copy or assign objects of built-in array types (§ 3.5.1, p. 114), there is no such restriction on array:

```
int digs[10] = {0,1,2,3,4,5,6,7,8,9};
int cpy[10] = digs; // error: no copy or assignment for built-in arrays
array<int, 10> digits = {0,1,2,3,4,5,6,7,8,9};
array<int, 10> copy = digits; // ok: so long as array types match
```

As with any container, the initializer must have the same type as the container we are creating. For arrays, the element type and the size must be the same, because the size of an array is part of its type.

## EXERCISES SECTION 9.2.4

**Exercise 9.11:** Show an example of each of the six ways to create and initialize a vector. Explain what values each vector contains.

**Exercise 9.12:** Explain the differences between the constructor that takes a container to copy and the constructor that takes two iterators.

**Exercise 9.13:** How would you initialize a `vector<double>` from a `list<int>`? From a `vector<int>`? Write code to check your answers.

## 9.2.5 Assignment and swap

The assignment-related operators, listed in Table 9.4 (overleaf) act on the entire container. The assignment operator replaces the entire range of elements in the left-hand container with copies of the elements from the right-hand operand:

```
c1 = c2;           // replace the contents of c1 with a copy of the elements in c2
c1 = {a,b,c};      // after the assignment c1 has size 3
```


After the first assignment, the left- and right-hand containers are equal. If the containers had been of unequal size, after the assignment both containers would have the size of the right-hand operand. After the second assignment, the size of `c1` is 3, which is the number of values provided in the braced list.

Unlike built-in arrays, the library array type does allow assignment. The left- and right-hand operands must have the same type:

```
array<int, 10> a1 = {0,1,2,3,4,5,6,7,8,9};
array<int, 10> a2 = {0}; // elements all have value 0
a1 = a2; // replaces elements in a1
a2 = {0}; // error: cannot assign to an array from a braced list
```

Because the size of the right-hand operand might differ from the size of the left-hand operand, the array type does not support assign and it does not allow assignment from a braced list of values.

Table 9.4: Container Assignment Operations

|   |   |
|---|---|
| c1 = c2   | Replace the elements in c1 with copies of the elements in c2. c1 and c2 must be the same type.  |
| c = {a,b,c...}  | Replace the elements in c1 with copies of the elements in the initializer list. <b>(Not valid for array.)</b>   |
| swap(c1, c2)  | Exchanges elements in c1 with those in c2. c1 and c2 must be the same   |
| c1.swap(c2)   | type. swap is usually <i>much</i> faster than copying elements from c2 to c1.   |
| <b>assign operations not valid for associative containers or array</b>                                |   |
| seq.assign(b,e)   | Replaces elements in seq with those in the range denoted by iterators b and e. The iterators b and e must not refer to elements in seq.   |
| seq.assign(il)  | Replaces the elements in seq with those in the initializer list il.   |
| seq.assign(n,t)   | Replaces the elements in seq with n elements with value t.  |
| <br><b>WARNING</b> | Assignment related operations invalidate iterators, references, and pointers into the left-hand container. Aside from string they remain valid after a swap, and (excepting arrays) the containers to which they refer are swapped. |

Using assign (Sequential Containers Only)

The assignment operator requires that the left-hand and right-hand operands have the same type. It copies all the elements from the right-hand operand into the left-hand operand. The sequential containers (except array) also define a member named assign that lets us assign from a different but compatible type, or assign from a subsequence of a container. The assign operation replaces all the elements in the left-hand container with (copies of) the elements specified by its arguments. For example, we can use assign to assign a range of char\* values from a vector into a list of string:

```
list<string> names;
vector<const char*> oldstyle;
names = oldstyle; // error: container types don't match
// ok: can convert from const char* to string
names.assign(oldstyle.cbegin(), oldstyle.cend());
```

The call to `assign` replaces the elements in `names` with copies of the elements in the range denoted by the iterators. The arguments to `assign` determine how many elements and what values the container will have.



Because the existing elements are replaced, the iterators passed to `assign` must not refer to the container on which `assign` is called.

A second version of `assign` takes an integral value and an element value. It replaces the elements in the container with the specified number of elements, each of which has the specified element value:

```
// equivalent to slist1.clear();
// followed by slist1.insert(slist1.begin(), 10, "Hiya!");
list<string> slist1(1);      // one element, which is the empty string
slist1.assign(10, "Hiya!"); // ten elements; each one is Hiya!
```

## Using swap

The `swap` operation exchanges the contents of two containers of the same type. After the call to `swap`, the elements in the two containers are interchanged:

```
vector<string> svec1(10); // vector with ten elements
vector<string> svec2(24); // vector with 24 elements
swap(svec1, svec2);
```

After the `swap`, `svec1` contains 24 `string` elements and `svec2` contains ten. With the exception of arrays, swapping two containers is guaranteed to be fast—the elements themselves are not swapped; internal data structures are swapped.



Excepting array, `swap` does not copy, delete, or insert any elements and is guaranteed to run in constant time.

The fact that elements are not moved means that, with the exception of `string`, iterators, references, and pointers into the containers are not invalidated. They refer to the same elements as they did before the `swap`. However, after the `swap`, those elements are in a different container. For example, had `iter` denoted the `string` at position `svec1[3]` before the `swap`, it will denote the element at position `svec2[3]` after the `swap`. Differently from the containers, a call to `swap` on a `string` may invalidate iterators, references and pointers.

Unlike how `swap` behaves for the other containers, swapping two arrays does exchange the elements. As a result, swapping two arrays requires time proportional to the number of elements in the array.

After the `swap`, pointers, references, and iterators remain bound to the same element they denoted before the `swap`. Of course, the value of that element has been swapped with the corresponding element in the other array.

In the new library, the containers offer both a member and nonmember version of `swap`. Earlier versions of the library defined only the member version of `swap`. The nonmember `swap` is of most importance in generic programs. As a matter of habit, it is best to use the nonmember version of `swap`.

## EXERCISES SECTION 9.2.5

**Exercise 9.14:** Write a program to assign the elements from a list of `char*` pointers to C-style character strings to a vector of strings.



### 9.2.6 Container Size Operations

With one exception, the container types have three size-related operations. The `size` member (§ 3.2.2, p. 87) returns the number of elements in the container; `empty` returns a `bool` that is `true` if `size` is zero and `false` otherwise; and `max_size` returns a number that is greater than or equal to the number of elements a container of that type can contain. For reasons we'll explain in the next section, `forward_list` provides `max_size` and `empty`, but not `size`.

### 9.2.7 Relational Operators

Every container type supports the equality operators (`==` and `!=`); all the containers except the unordered associative containers also support the relational operators (`>`, `>=`, `<`, `<=`). The right- and left-hand operands must be the same kind of container and must hold elements of the same type. That is, we can compare a `vector<int>` only with another `vector<int>`. We cannot compare a `vector<int>` with a `list<int>` or a `vector<double>`.

Comparing two containers performs a pairwise comparison of the elements. These operators work similarly to the string relationals (§ 3.2.2, p. 88):

- If both containers are the same size and all the elements are equal, then the two containers are equal; otherwise, they are unequal.
- If the containers have different sizes but every element of the smaller one is equal to the corresponding element of the larger one, then the smaller one is less than the other.
- If neither container is an initial subsequence of the other, then the comparison depends on comparing the first unequal elements.

The following examples illustrate how these operators work:

```
vector<int> v1 = { 1, 3, 5, 7, 9, 12 };
vector<int> v2 = { 1, 3, 9 };
vector<int> v3 = { 1, 3, 5, 7 };
vector<int> v4 = { 1, 3, 5, 7, 9, 12 };
v1 < v2 // true; v1 and v2 differ at element [2]: v1[2] is less than v2[2]
v1 < v3 // false; all elements are equal, but v3 has fewer of them;
v1 == v4 // true; each element is equal and v1 and v4 have the same size ()
v1 == v2 // false; v2 has fewer elements than v1
```



## Relational Operators Use Their Element's Relational Operator



We can use a relational operator to compare two containers only if the appropriate comparison operator is defined for the element type.

The container equality operators use the element's `==` operator, and the relational operators use the element's `<` operator. If the element type doesn't support the required operator, then we cannot use the corresponding operations on containers holding that type. For example, the `Sales_data` type that we defined in Chapter 7 does not define either the `==` or the `<` operation. Therefore, we cannot compare two containers that hold `Sales_data` elements:

```
vector<Sales_data> storeA, storeB;  
if (storeA < storeB) // error: Sales_data has no less-than operator
```

### EXERCISES SECTION 9.2.7

**Exercise 9.15:** Write a program to determine whether two `vector<int>`s are equal.

**Exercise 9.16:** Repeat the previous program, but compare elements in a `list<int>` to a `vector<int>`.

**Exercise 9.17:** Assuming `c1` and `c2` are containers, what (if any) constraints does the following usage place on the types of `c1` and `c2`?

```
if (c1 < c2)
```

## 9.3 Sequential Container Operations

The sequential and associative containers differ in how they organize their elements. These differences affect how elements are stored, accessed, added, and removed. The previous section covered operations common to all containers (those listed in Table 9.2 (p. 330)). We'll cover the operations specific to the sequential containers in the remainder of this chapter.

### 9.3.1 Adding Elements to a Sequential Container



Excepting `array`, all of the library containers provide flexible memory management. We can add or remove elements dynamically changing the size of the container at run time. Table 9.5 (p. 343) lists the operations that add elements to a (nonarray) sequential container.

When we use these operations, we must remember that the containers use different strategies for allocating elements and that these strategies affect performance. Adding elements anywhere but at the end of a `vector` or `string`, or anywhere but the beginning or end of a `deque`, requires elements to be moved.

Moreover, adding elements to a vector or a string may cause the entire object to be reallocated. Reallocating an object requires allocating new memory and moving elements from the old space to the new.

## Using `push_back`

In § 3.3.2 (p. 100) we saw that `push_back` appends an element to the back of a vector. Aside from `array` and `forward_list`, every sequential container (including the `string` type) supports `push_back`.

As an example, the following loop reads one string at a time into `word`:

```
// read from standard input, putting each word onto the end of container
string word;
while (cin >> word)
    container.push_back(word);
```

The call to `push_back` creates a new element at the end of `container`, increasing the size of `container` by 1. The value of that element is a copy of `word`. The type of `container` can be any of `list`, `vector`, or `deque`.

Because `string` is just a container of characters, we can use `push_back` to add characters to the end of the string:

```
void pluralize(size_t cnt, string &word)
{
    if (cnt > 1)
        word.push_back('s'); // same as word += 's'
}
```

### KEY CONCEPT: CONTAINER ELEMENTS ARE COPIES

When we use an object to initialize a container, or insert an object into a container, a copy of that object's value is placed in the container, not the object itself. Just as when we pass an object to a nonreference parameter (§ 6.2.1, p. 209), there is no relationship between the element in the container and the object from which that value originated. Subsequent changes to the element in the container have no effect on the original object, and vice versa.

## Using `push_front`

In addition to `push_back`, the `list`, `forward_list`, and `deque` containers support an analogous operation named `push_front`. This operation inserts a new element at the front of the container:

```
list<int> ilist;
// add elements to the start of ilist
for (size_t ix = 0; ix != 4; ++ix)
    ilist.push_front(ix);
```

This loop adds the elements 0, 1, 2, 3 to the beginning of `ilist`. Each element is inserted at the *new beginning* of the list. That is, when we insert 1, it goes in

front of 0, and 2 in front of 1, and so forth. Thus, the elements added in a loop such as this one wind up in reverse order. After executing this loop, `ilist` holds the sequence 3, 2, 1, 0.

Note that `deque`, which like `vector` offers fast random access to its elements, provides the `push_front` member even though `vector` does not. A `deque` guarantees constant-time insert and delete of elements at the beginning and end of the container. As with `vector`, inserting elements other than at the front or back of a `deque` is a potentially expensive operation.

**Table 9.5: Operations That Add Elements to a Sequential Container**

These operations change the size of the container; they are not supported by `array`.  
**`forward_list` has special versions of `insert` and `emplace`; see § 9.3.4 (p. 350).**  
**`push_back` and `emplace_back` not valid for `forward_list`.**  
**`push_front` and `emplace_front` not valid for `vector` or `string`.**

|                                    |   |
|------------------------------------|---|
| <code>c.push_back(t)</code>        | Creates an element with value <code>t</code> or constructed from <code>args</code> at the end of <code>c</code> . Returns <code>void</code> .   |
| <code>c.emplace_back(args)</code>  |   |
| <code>c.push_front(t)</code>       | Creates an element with value <code>t</code> or constructed from <code>args</code> on the front of <code>c</code> . Returns <code>void</code> .   |
| <code>c.emplace_front(args)</code> |   |
| <code>c.insert(p, t)</code>        | Creates an element with value <code>t</code> or constructed from <code>args</code> before the element denoted by iterator <code>p</code> . Returns an iterator referring to the element that was added.   |
| <code>c.emplace(p, args)</code>    |   |
| <code>c.insert(p, n, t)</code>     | Inserts <code>n</code> elements with value <code>t</code> before the element denoted by iterator <code>p</code> . Returns an iterator to the first element inserted; if <code>n</code> is zero, returns <code>p</code> .  |
| <code>c.insert(p, b, e)</code>     | Inserts the elements from the range denoted by iterators <code>b</code> and <code>e</code> before the element denoted by iterator <code>p</code> . <code>b</code> and <code>e</code> may not refer to elements in <code>c</code> . Returns an iterator to the first element inserted; if the range is empty, returns <code>p</code> . |
| <code>c.insert(p, il)</code>       | <code>il</code> is a braced list of element values. Inserts the given values before the element denoted by the iterator <code>p</code> . Returns an iterator to the first inserted element; if the list is empty returns <code>p</code> .   |



**WARNING**

Adding elements to a `vector`, `string`, or `deque` potentially invalidates all existing iterators, references, and pointers into the container.

## Adding Elements at a Specified Point in the Container

The `push_back` and `push_front` operations provide convenient ways to insert a single element at the end or beginning of a sequential container. More generally, the `insert` members let us insert zero or more elements at any point in the container. The `insert` members are supported for `vector`, `deque`, `list`, and `string`. `forward_list` provides specialized versions of these members that we'll cover in § 9.3.4 (p. 350).

Each of the `insert` functions takes an iterator as its first argument. The iterator indicates where in the container to put the element(s). It can refer to any position in the container, including one past the end of the container. Because the iterator

might refer to a nonexistent element off the end of the container, and because it is useful to have a way to insert elements at the beginning of a container, element(s) are inserted *before* the position denoted by the iterator. For example, this statement

```
slist.insert(iter, "Hello!"); // insert "Hello!" just before iter
```

inserts a string with value "Hello" just before the element denoted by `iter`.

Even though some containers do not have a `push_front` operation, there is no similar constraint on `insert`. We can insert elements at the beginning of a container without worrying about whether the container has `push_front`:

```
vector<string> svec;
list<string> slist;

// equivalent to calling slist.push_front("Hello!");
slist.insert(slist.begin(), "Hello!");

// no push_front on vector but we can insert before begin()
// warning: inserting anywhere but at the end of a vector might be slow
svec.insert(svec.begin(), "Hello!");
```



It is legal to insert anywhere in a vector, deque, or string. However, doing so can be an expensive operation.

## Inserting a Range of Elements

The arguments to `insert` that appear after the initial iterator argument are analogous to the container constructors that take the same parameters. The version that takes an element count and a value adds the specified number of identical elements before the given position:

```
svec.insert(svec.end(), 10, "Anna");
```

This code inserts ten elements at the end of `svec` and initializes each of those elements to the string "Anna".

The versions of `insert` that take a pair of iterators or an initializer list insert the elements from the given range before the given position:

```
vector<string> v = {"quasi", "simba", "frollo", "scar"};
// insert the last two elements of v at the beginning of slist
slist.insert(slist.begin(), v.end() - 2, v.end());
slist.insert(slist.end(), {"these", "words", "will",
                          "go", "at", "the", "end"});

// run-time error: iterators denoting the range to copy from
// must not refer to the same container as the one we are changing
slist.insert(slist.begin(), slist.begin(), slist.end());
```

When we pass a pair of iterators, those iterators may not refer to the same container as the one to which we are adding elements.

Under the new standard, the versions of `insert` that take a count or a range return an iterator to the first element that was inserted. (In prior versions of the library, these operations returned `void`.) If the range is empty, no elements are inserted, and the operation returns its first parameter.

## Using the Return from `insert`

We can use the value returned by `insert` to repeatedly insert elements at a specified position in the container:

```
list<string> lst;
auto iter = lst.begin();
while (cin >> word)
    iter = lst.insert(iter, word); // same as calling push_front
```



It is important to understand how this loop operates—in particular, to understand why the loop is equivalent to calling `push_front`.

Before the loop, we initialize `iter` to `lst.begin()`. The first call to `insert` takes the string we just read and puts it in front of the element denoted by `iter`. The value returned by `insert` is an iterator referring to this new element. We assign that iterator to `iter` and repeat the `while`, reading another word. As long as there are words to insert, each trip through the `while` inserts a new element ahead of `iter` and reassigns to `iter` the location of the newly inserted element. That element is the (new) first element. Thus, each iteration inserts an element ahead of the first element in the list.

## Using the Emplace Operations

The new standard introduced three new members—`emplace_front`, `emplace`, and `emplace_back`—that construct rather than copy elements. These operations correspond to the `push_front`, `insert`, and `push_back` operations in that they let us put an element at the front of the container, in front of a given position, or at the back of the container, respectively.

C++  
11

When we call a `push` or `insert` member, we pass objects of the element type and those objects are copied into the container. When we call an `emplace` member, we pass arguments to a constructor for the element type. The `emplace` members use those arguments to construct an element directly in space managed by the container. For example, assuming `c` holds `Sales_data` (§7.1.4, p. 264) elements:

```
// construct a Sales_data object at the end of c
// uses the three-argument Sales_data constructor
c.emplace_back("978-0590353403", 25, 15.99);

// error: there is no version of push_back that takes three arguments
c.push_back("978-0590353403", 25, 15.99);

// ok: we create a temporary Sales_data object to pass to push_back
c.push_back(Sales_data("978-0590353403", 25, 15.99));
```

The call to `emplace_back` and the second call to `push_back` both create new `Sales_data` objects. In the call to `emplace_back`, that object is created directly in space managed by the container. The call to `push_back` creates a local temporary object that is pushed onto the container.

The arguments to an `emplace` function vary depending on the element type. The arguments must match a constructor for the element type:

```
// iter refers to an element in c, which holds Sales_data elements
c.emplace_back(); // uses the Sales_data default constructor
c.emplace(iter, "999-999999999"); // uses Sales_data(string)
// uses the Sales_data constructor that takes an ISBN, a count, and a price
c.emplace_front("978-0590353403", 25, 15.99);
```



The *emplace* functions construct elements in the container. The arguments to these functions must match a constructor for the element type.

### EXERCISES SECTION 9.3.1

**Exercise 9.18:** Write a program to read a sequence of strings from the standard input into a deque. Use iterators to write a loop to print the elements in the deque.

**Exercise 9.19:** Rewrite the program from the previous exercise to use a list. List the changes you needed to make.

**Exercise 9.20:** Write a program to copy elements from a `list<int>` into two deques. The even-valued elements should go into one deque and the odd ones into the other.

**Exercise 9.21:** Explain how the loop from page 345 that used the return from `insert` to add elements to a list would work if we inserted into a vector instead.

**Exercise 9.22:** Assuming `iv` is a vector of ints, what is wrong with the following program? How might you correct the problem(s)?

```
vector<int>::iterator iter = iv.begin(),
                    mid = iv.begin() + iv.size()/2;
while (iter != mid)
    if (*iter == some_val)
        iv.insert(iter, 2 * some_val);
```



## 9.3.2 Accessing Elements

Table 9.6 lists the operations we can use to access elements in a sequential container. The access operations are undefined if the container has no elements.

Each sequential container, including `array`, has a `front` member, and all except `forward_list` also have a `back` member. These operations return a reference to the first and last element, respectively:

```
// check that there are elements before dereferencing an iterator or calling front or back
if (!c.empty()) {
    // val and val2 are copies of the value of the first element in c
    auto val = *c.begin(), val2 = c.front();
    // val3 and val4 are copies of the value of the last element in c
    auto last = c.end();
    auto val3 = *(--last); // can't decrement forward_list iterators
    auto val4 = c.back(); // not supported by forward_list
}
```

This program obtains references to the first and last elements in `c` in two different ways. The direct approach is to call `front` or `back`. Indirectly, we can obtain a reference to the same element by dereferencing the iterator returned by `begin` or decrementing and then dereferencing the iterator returned by `end`.

Two things are noteworthy in this program: The `end` iterator refers to the (nonexistent) element one past the end of the container. To fetch the last element we must first decrement that iterator. The other important point is that before calling `front` or `back` (or dereferencing the iterators from `begin` or `end`), we check that `c` isn't empty. If the container were empty, the operations inside the `if` would be undefined.

**Table 9.6: Operations to Access Elements in a Sequential Container**

**at** and subscript operator valid only for **string**, **vector**, **deque**, and **array**.  
**back** not valid for **forward\_list**.

|                        |   |
|------------------------|---|
| <code>c.back()</code>  | Returns a reference to the last element in <code>c</code> . Undefined if <code>c</code> is empty.   |
| <code>c.front()</code> | Returns a reference to the first element in <code>c</code> . Undefined if <code>c</code> is empty.  |
| <code>c[n]</code>      | Returns a reference to the element indexed by the unsigned integral value <code>n</code> . Undefined if <code>n &gt;= c.size()</code> .     |
| <code>c.at(n)</code>   | Returns a reference to the element indexed by <code>n</code> . If the index is out of range, throws an <code>out_of_range</code> exception. |



Calling `front` or `back` on an empty container, like using a subscript that is out of range, is a serious programming error.

## The Access Members Return References

The members that access elements in a container (i.e., `front`, `back`, subscript, and `at`) return references. If the container is a `const` object, the return is a reference to `const`. If the container is not `const`, the return is an ordinary reference that we can use to change the value of the fetched element:

```
if (!c.empty()) {
    c.front() = 42;           // assigns 42 to the first element in c
    auto &v = c.back();       // get a reference to the last element
    v = 1024;                 // changes the element in c
    auto v2 = c.back();       // v2 is not a reference; it's a copy of c.back()
    v2 = 0;                   // no change to the element in c
}
```

As usual, if we use `auto` to store the return from one of these functions and we want to use that variable to change the element, we must remember to define our variable as a reference type.

## Subscripting and Safe Random Access

The containers that provide fast random access (`string`, `vector`, `deque`, and `array`) also provide the subscript operator (§ 3.3.3, p. 102). As we've seen, the

subscript operator takes an index and returns a reference to the element at that position in the container. The index must be “in range,” (i.e., greater than or equal to 0 and less than the size of the container). It is up to the program to ensure that the index is valid; the subscript operator does not check whether the index is in range. Using an out-of-range value for an index is a serious programming error, but one that the compiler will not detect.

If we want to ensure that our index is valid, we can use the `at` member instead. The `at` member acts like the subscript operator, but if the index is invalid, `at` throws an `out_of_range` exception (§ 5.6, p. 193):

```
vector<string> svec; // empty vector
cout << svec[0];    // run-time error: there are no elements in svec!
cout << svec.at(0); // throws an out_of_range exception
```

### EXERCISES SECTION 9.3.2

**Exercise 9.23:** In the first program in this section on page 346, what would the values of `val`, `val2`, `val3`, and `val4` be if `c.size()` is 1?

**Exercise 9.24:** Write a program that fetches the first element in a vector using `at`, the subscript operator, `front`, and `begin`. Test your program on an empty vector.



### 9.3.3 Erasing Elements

Just as there are several ways to add elements to a (nonarray) container there are also several ways to remove elements. These members are listed in Table 9.7.



**WARNING**

The members that remove elements do not check their argument(s). The programmer must ensure that element(s) exist before removing them.

### The `pop_front` and `pop_back` Members

The `pop_front` and `pop_back` functions remove the first and last elements, respectively. Just as there is no `push_front` for `vector` and `string`, there is also no `pop_front` for those types. Similarly, `forward_list` does not have `pop_back`. Like the element access members, we may not use a pop operation on an empty container.

These operations return `void`. If you need the value you are about to pop, you must store that value before doing the pop:

```
while (!ilist.empty()) {
    process(ilist.front()); // do something with the current top of ilist
    ilist.pop_front();      // done; remove the first element
}
```



**Table 9.7: erase Operations on Sequential Containers**

These operations change the size of the container and so are not supported by array.  
**forward\_list** has a special version of **erase**; see § 9.3.4 (p. 350).  
**pop\_back** not valid for **forward\_list**; **pop\_front** not valid for **vector** and **string**.

|                            |  |
|----------------------------|--|
| <code>c.pop_back()</code>  | Removes last element in <code>c</code> . Undefined if <code>c</code> is empty. Returns <code>void</code> .   |
| <code>c.pop_front()</code> | Removes first element in <code>c</code> . Undefined if <code>c</code> is empty. Returns <code>void</code> .  |
| <code>c.erase(p)</code>    | Removes the element denoted by the iterator <code>p</code> and returns an iterator to the element after the one deleted or the off-the-end iterator if <code>p</code> denotes the last element. Undefined if <code>p</code> is the off-the-end iterator. |
| <code>c.erase(b, e)</code> | Removes the range of elements denoted by the iterators <code>b</code> and <code>e</code> . Returns an iterator to the element after the last one that was deleted, or an off-the-end iterator if <code>e</code> is itself an off-the-end iterator.       |
| <code>c.clear()</code>     | Removes all the elements in <code>c</code> . Returns <code>void</code> .   |

**WARNING**

Removing elements anywhere but the beginning or end of a deque invalidates all iterators, references, and pointers. Iterators, references, and pointers to elements after the erasure point in a vector or string are invalidated.

## Removing an Element from within the Container

The `erase` members remove element(s) at a specified point in the container. We can delete a single element denoted by an iterator or a range of elements marked by a pair of iterators. Both forms of `erase` return an iterator referring to the location *after* the (last) element that was removed. That is, if `j` is the element following `i`, then `erase(i)` will return an iterator referring to `j`.

As an example, the following loop erases the odd elements in a list:

```
list<int> lst = {0,1,2,3,4,5,6,7,8,9};
auto it = lst.begin();
while (it != lst.end())
    if (*it % 2)                // if the element is odd
        it = lst.erase(it);    // erase this element
    else
        ++it;
```

On each iteration, we check whether the current element is odd. If so, we erase that element, setting `it` to denote the element after the one we erased. If `*it` is even, we increment `it` so we'll look at the next element on the next iteration.

## Removing Multiple Elements

The iterator-pair version of `erase` lets us delete a range of elements:

```
// delete the range of elements between two iterators
// returns an iterator to the element just after the last removed element
elem1 = slist.erase(elem1, elem2); // after the call elem1 == elem2
```

The iterator `elem1` refers to the first element we want to erase, and `elem2` refers to one past the last element we want to remove.

To delete all the elements in a container, we can either call `clear` or pass the iterators from `begin` and `end` to `erase`:

```
slist.clear(); // delete all the elements within the container
slist.erase(slist.begin(), slist.end()); // equivalent
```

EXERCISES SECTION 9.3.3

**Exercise 9.25:** In the program on page 349 that erased a range of elements, what happens if `elem1` and `elem2` are equal? What if `elem2` or both `elem1` and `elem2` are the off-the-end iterator?

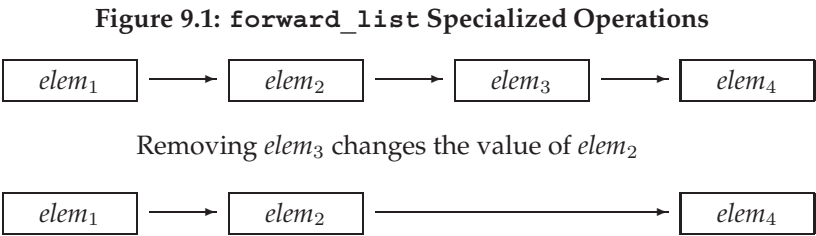
**Exercise 9.26:** Using the following definition of `ia`, copy `ia` into a vector and into a list. Use the single-iterator form of `erase` to remove the elements with odd values from your list and the even values from your vector.

```
int ia[] = { 0, 1, 1, 2, 3, 5, 8, 13, 21, 55, 89 };
```



### 9.3.4 Specialized forward\_list Operations

To understand why `forward_list` has special versions of the operations to add and remove elements, consider what must happen when we remove an element from a singly linked list. As illustrated in Figure 9.1, removing an element changes the links in the sequence. In this case, removing `elem3` changes `elem2`; `elem2` had pointed to `elem3`, but after we remove `elem3`, `elem2` points to `elem4`.



When we add or remove an element, the element before the one we added or removed has a different successor. To add or remove an element, we need access to its predecessor in order to update that element’s links. However, `forward_list` is a singly linked list. In a singly linked list there is no easy way to get to an element’s predecessor. For this reason, the operations to add or remove elements in a `forward_list` operate by changing the element *after* the given element. That way, we always have access to the elements that are affected by the change.

Because these operations behave differently from the operations on the other containers, `forward_list` does not define `insert`, `emplace`, or `erase`. Instead it defines members (listed in Table 9.8) named `insert_after`, `emplace_after`,

and `erase_after`. For example, in our illustration, to remove `elem3`, we'd call `erase_after` on an iterator that denoted `elem2`. To support these operations, `forward_list` also defines `before_begin`, which returns an **off-the-beginning** iterator. This iterator lets us add or remove elements “after” the nonexistent element before the first one in the list.

**Table 9.8: Operations to Insert or Remove Elements in a `forward_list`**

|  |   |
|--|---|
| <code>lst.before_begin()</code>        | Iterator denoting the nonexistent element just before the beginning of the list. This iterator may not be dereferenced.   |
| <code>lst.cbefore_begin()</code>       | <code>cbefore_begin()</code> returns a <code>const_iterator</code> .  |
| <code>lst.insert_after(p, t)</code>    | Inserts element(s) <i>after</i> the one denoted by iterator <code>p</code> . <code>t</code> is an object, <code>n</code> is a count, <code>b</code> and <code>e</code> are iterators denoting a range ( <code>b</code> and <code>e</code> must not refer to <code>lst</code> ), and <code>il</code> is a braced list. Returns an iterator to the <i>last</i> inserted element. If the range is empty, returns <code>p</code> . Undefined if <code>p</code> is the off-the-end iterator. |
| <code>lst.insert_after(p, n, t)</code> |   |
| <code>lst.insert_after(p, b, e)</code> |   |
| <code>lst.insert_after(p, il)</code>   |   |
| <code>emplace_after(p, args)</code>    | Uses <code>args</code> to construct an element after the one denoted by iterator <code>p</code> . Returns an iterator to the new element. Undefined if <code>p</code> is the off-the-end iterator.  |
| <code>lst.erase_after(p)</code>        | Removes the element <i>after</i> the one denoted by iterator <code>p</code> or the range of elements from the one <i>after</i> the iterator <code>b</code> up to but not including the one denoted by <code>e</code> . Returns an iterator to the element after the one deleted, or the off-the-end iterator if there is no such element. Undefined if <code>p</code> denotes the last element in <code>lst</code> or is the off-the-end iterator.                                      |
| <code>lst.erase_after(b, e)</code>     |   |

When we add or remove elements in a `forward_list`, we have to keep track of two iterators—one to the element we’re checking and one to that element’s predecessor. As an example, we’ll rewrite the loop from page 349 that removed the odd-valued elements from a list to use a `forward_list`:

```
forward_list<int> flst = {0,1,2,3,4,5,6,7,8,9};
auto prev = flst.before_begin(); // denotes element "off the start" of flst
auto curr = flst.begin();        // denotes the first element in flst
while (curr != flst.end()) {      // while there are still elements to process
    if (*curr % 2)                 // if the element is odd
        curr = flst.erase_after(prev); // erase it and move curr
    else {
        prev = curr;              // move the iterators to denote the next
        ++curr;                  // element and one before the next element
    }
}
```

Here, `curr` denotes the element we’re checking, and `prev` denotes the element before `curr`. We call `begin` to initialize `curr`, so that the first iteration checks whether the first element is even or odd. We initialize `prev` from `before_begin`, which returns an iterator to the nonexistent element just before `curr`.

When we find an odd element, we pass `prev` to `erase_after`. This call erases the element after the one denoted by `prev`; that is, it erases the element denoted

by `curr`. We reset `curr` to the return from `erase_after`, which makes `curr` denote the next element in the sequence and we leave `prev` unchanged; `prev` still denotes the element before the (new) value of `curr`. If the element denoted by `curr` is not odd, then we have to move both iterators, which we do in the `else`.

EXERCISES SECTION 9.3.4

**Exercise 9.27:** Write a program to find and remove the odd-valued elements in a `forward_list<int>`.


**Exercise 9.28:** Write a function that takes a `forward_list<string>` and two additional `string` arguments. The function should find the first `string` and insert the second immediately following the first. If the first `string` is not found, then insert the second `string` at the end of the list.

### 9.3.5 Resizing a Container

With the usual exception of arrays, we can use `resize`, described in Table 9.9, to make a container larger or smaller. If the current size is greater than the requested size, elements are deleted from the back of the container; if the current size is less than the new size, elements are added to the back of the container:

```
list<int> ilist(10, 42); // ten ints: each has value 42
ilist.resize(15);       // adds five elements of value 0 to the back of ilist
ilist.resize(25, -1);   // adds ten elements of value -1 to the back of ilist
ilist.resize(5);        // erases 20 elements from the back of ilist
```

The `resize` operation takes an optional element-value argument that it uses to initialize any elements that are added to the container. If this argument is absent, added elements are value initialized (§ 3.3.1, p. 98). If the container holds elements of a class type and `resize` adds elements, we must supply an initializer or the element type must have a default constructor.

| Table 9.9: Sequential Container Size Operations  |   |
|--|---|
| resize not valid for array.  |   |
| <code>c.resize(n)</code>   | Resize <code>c</code> so that it has <code>n</code> elements. If <code>n &lt; c.size()</code> , the excess elements are discarded. If new elements must be added, they are value initialized.   |
| <code>c.resize(n,t)</code>   | Resize <code>c</code> to have <code>n</code> elements. Any elements added have value <code>t</code> .   |
| <div><br/>WARNING</div> | If <code>resize</code> shrinks the container, then iterators, references, and pointers to the deleted elements are invalidated; <code>resize</code> on a <code>vector</code> , <code>string</code> , or <code>deque</code> potentially invalidates all iterators, pointers, and references. |

**EXERCISES SECTION 9.3.5**

**Exercise 9.29:** Given that `vec` holds 25 elements, what does `vec.resize(100)` do? What if we next wrote `vec.resize(10)`?

**Exercise 9.30:** What, if any, restrictions does using the version of `resize` that takes a single argument place on the element type?

### 9.3.6 Container Operations May Invalidate Iterators



Operations that add or remove elements from a container can invalidate pointers, references, or iterators to container elements. An invalidated pointer, reference, or iterator is one that no longer denotes an element. Using an invalidated pointer, reference, or iterator is a serious programming error that is likely to lead to the same kinds of problems as using an uninitialized pointer (§ 2.3.2, p. 54).

After an operation that adds elements to a container

- Iterators, pointers, and references to a `vector` or `string` are invalid if the container was reallocated. If no reallocation happens, indirect references to elements before the insertion remain valid; those to elements after the insertion are invalid.
- Iterators, pointers, and references to a `deque` are invalid if we add elements anywhere but at the front or back. If we add at the front or back, iterators are invalidated, but references and pointers to existing elements are not.
- Iterators, pointers, and references (including the off-the-end and the before-the-beginning iterators) to a `list` or `forward_list` remain valid,

It should not be surprising that when we remove elements from a container, iterators, pointers, and references to the removed elements are invalidated. After all, those elements have been destroyed. After we remove an element,

- All other iterators, references, or pointers (including the off-the-end and the before-the-beginning iterators) to a `list` or `forward_list` remain valid.
- All other iterators, references, or pointers to a `deque` are invalidated if the removed elements are anywhere but the front or back. If we remove elements at the back of the `deque`, the off-the-end iterator is invalidated but other iterators, references, and pointers are unaffected; they are also unaffected if we remove from the front.
- All other iterators, references, or pointers to a `vector` or `string` remain valid for elements before the removal point. Note: The off-the-end iterator is always invalidated when we remove elements.

**WARNING**

It is a serious run-time error to use an iterator, pointer, or reference that has been invalidated.

**ADVICE: MANAGING ITERATORS**

When you use an iterator (or a reference or pointer to a container element), it is a good idea to minimize the part of the program during which an iterator must stay valid.

Because code that adds or removes elements to a container can invalidate iterators, you need to ensure that the iterator is repositioned, as appropriate, after each operation that changes the container. This advice is especially important for `vector`, `string`, and `deque`.



## Writing Loops That Change a Container

Loops that add or remove elements of a `vector`, `string`, or `deque` must cater to the fact that iterators, references, or pointers might be invalidated. The program must ensure that the iterator, reference, or pointer is refreshed on each trip through the loop. Refreshing an iterator is easy if the loop calls `insert` or `erase`. Those operations return iterators, which we can use to reset the iterator:

```
// silly loop to remove even-valued elements and insert a duplicate of odd-valued elements
vector<int> vi = {0,1,2,3,4,5,6,7,8,9};
auto iter = vi.begin(); // call begin, not cbegin because we're changing vi
while (iter != vi.end()) {
    if (*iter % 2) {
        iter = vi.insert(iter, *iter); // duplicate the current element
        iter += 2; // advance past this element and the one inserted before it
    } else
        iter = vi.erase(iter); // remove even elements
    // don't advance the iterator; iter denotes the element after the one we erased
}
```

This program removes the even-valued elements and duplicates each odd-valued one. We refresh the iterator after both the `insert` and the `erase` because either operation can invalidate the iterator.

After the call to `erase`, there is no need to increment the iterator, because the iterator returned from `erase` denotes the next element in the sequence. After the call to `insert`, we increment the iterator twice. Remember, `insert` inserts *before* the position it is given and returns an iterator to the inserted element. Thus, after calling `insert`, `iter` denotes the (newly added) element in front of the one we are processing. We add two to skip over the element we added and the one we just processed. Doing so positions the iterator on the next, unprocessed element.



## Avoid Storing the Iterator Returned from `end`

When we add or remove elements in a `vector` or `string`, or add elements or remove any but the first element in a `deque`, the iterator returned by `end` is *always* invalidated. Thus, loops that add or remove elements should always call `end` rather than use a stored copy. Partly for this reason, C++ standard libraries are usually implemented so that calling `end()` is a very fast operation.

As an example, consider a loop that processes each element and adds a new element following the original. We want the loop to ignore the added elements,

and to process only the original elements. After each insertion, we'll position the iterator to denote the next original element. If we attempt to “optimize” the loop, by storing the iterator returned by `end()`, we'll have a disaster:

```
// disaster: the behavior of this loop is undefined
auto begin = v.begin(),
    end = v.end(); // bad idea, saving the value of the end iterator
while (begin != end) {
    // do some processing
    // insert the new value and reassign begin, which otherwise would be invalid
    ++begin; // advance begin because we want to insert after this element
    begin = v.insert(begin, 42); // insert the new value
    ++begin; // advance begin past the element we just added
}
```

The behavior of this code is undefined. On many implementations, we'll get an infinite loop. The problem is that we stored the value returned by the `end` operation in a local variable named `end`. In the body of the loop, we added an element. Adding an element invalidates the iterator stored in `end`. That iterator neither refers to an element in `v` nor any longer refers to one past the last element in `v`.



Don't cache the iterator returned from `end()` in loops that insert or delete elements in a deque, string, or vector.

Rather than storing the `end()` iterator, we must recompute it after each insertion:

```
// safer: recalculate end on each trip whenever the loop adds/erases elements
while (begin != v.end()) {
    // do some processing
    ++begin; // advance begin because we want to insert after this element
    begin = v.insert(begin, 42); // insert the new value
    ++begin; // advance begin past the element we just added
}
```

## 9.4 How a vector Grows



To support fast random access, vector elements are stored contiguously—each element is adjacent to the previous element. Ordinarily, we should not care about how a library type is implemented; all we should care about is how to use it. However, in the case of vectors and strings, part of the implementation leaks into its interface.

Given that elements are contiguous, and that the size of the container is flexible, consider what must happen when we add an element to a vector or a string: If there is no room for the new element, the container can't just add an element somewhere else in memory—the elements must be contiguous. Instead, the container must allocate new memory to hold the existing elements plus the new one, move the elements from the old location into the new space, add the new element, and deallocate the old memory. If vector did this memory allocation and deallocation each time we added an element, performance would be unacceptably slow.

### EXERCISES SECTION 9.3.6

**Exercise 9.31:** The program on page 354 to remove even-valued elements and duplicate odd ones will not work on a `list` or `forward_list`. Why? Revise the program so that it works on these types as well.

**Exercise 9.32:** In the program on page 354 would it be legal to write the call to `insert` as follows? If not, why not?

```
iter = vi.insert(iter, *iter++);
```

**Exercise 9.33:** In the final example in this section what would happen if we did not assign the result of `insert` to `begin`? Write a program that omits this assignment to see if your expectation was correct.

**Exercise 9.34:** Assuming `vi` is a container of ints that includes even and odd values, predict the behavior of the following loop. After you've analyzed this loop, write a program to test whether your expectations were correct.

```
iter = vi.begin();
while (iter != vi.end())
    if (*iter % 2)
        iter = vi.insert(iter, *iter);
        ++iter;
```

To avoid these costs, library implementors use allocation strategies that reduce the number of times the container is reallocated. When they have to get new memory, `vector` and `string` implementations typically allocate capacity beyond what is immediately needed. The container holds this storage in reserve and uses it to allocate new elements as they are added. Thus, there is no need to reallocate the container for each new element.

This allocation strategy is dramatically more efficient than reallocating the container each time an element is added. In fact, its performance is good enough that in practice a `vector` usually grows more efficiently than a `list` or a `deque`, even though the `vector` has to move all of its elements each time it reallocates memory.

## Members to Manage Capacity

The `vector` and `string` types provide members, described in Table 9.10, that let us interact with the memory-allocation part of the implementation. The `capacity` operation tells us how many elements the container can hold before it must allocate more space. The `reserve` operation lets us tell the container how many elements it should be prepared to hold.



`reserve` does not change the number of elements in the container; it affects only how much memory the `vector` preallocates.

A call to `reserve` changes the capacity of the `vector` only if the requested space exceeds the current capacity. If the requested size is greater than the current



capacity, `reserve` allocates at least as much as (and may allocate more than) the requested amount.

If the requested size is less than or equal to the existing capacity, `reserve` does nothing. In particular, calling `reserve` with a size smaller than `capacity` does not cause the container to give back memory. Thus, after calling `reserve`, the capacity will be greater than or equal to the argument passed to `reserve`.

As a result, a call to `reserve` will never reduce the amount of space that the container uses. Similarly, the `resize` members (§ 9.3.5, p. 352) change only the number of elements in the container, not its capacity. We cannot use `resize` to reduce the memory a container holds in `reserve`.

Under the new library, we can call `shrink_to_fit` to ask a deque, vector, or string to return unneeded memory. This function indicates that we no longer need any excess capacity. However, the implementation is free to ignore this request. There is no guarantee that a call to `shrink_to_fit` will return memory.

C++  
11

| Table 9.10: Container Size Management  |  |
|--|--|
| <b><code>shrink_to_fit</code> valid only for vector, string, and deque.<br/>capacity and reserve valid only for vector and string.</b> |  |
| <code>c.shrink_to_fit()</code>   | Request to reduce <code>capacity()</code> to equal <code>size()</code> .     |
| <code>c.capacity()</code>  | Number of elements <code>c</code> can have before reallocation is necessary. |
| <code>c.reserve(n)</code>  | Allocate space for at least <code>n</code> elements.                         |

capacity and size

It is important to understand the difference between capacity and size. The size of a container is the number of elements it already holds; its capacity is how many elements it can hold before more space must be allocated.

The following code illustrates the interaction between size and capacity:

```
vector<int> ivec;
// size should be zero; capacity is implementation defined
cout << "ivec: size: " << ivec.size()
    << " capacity: " << ivec.capacity() << endl;
// give ivec 24 elements
for (vector<int>::size_type ix = 0; ix != 24; ++ix)
    ivec.push_back(ix);

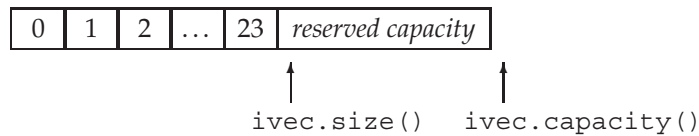
// size should be 24; capacity will be >= 24 and is implementation defined
cout << "ivec: size: " << ivec.size()
    << " capacity: " << ivec.capacity() << endl;
```

When run on our system, this code produces the following output:

```
ivec: size: 0 capacity: 0
ivec: size: 24 capacity: 32
```

We know that the size of an empty vector is zero, and evidently our library also sets the capacity of an empty vector to zero. When we add elements to the vector, we know that the size is the same as the number of elements we've added. The capacity must be at least as large as size but can be larger. The details of how much excess capacity is allocated vary by implementations of the library. Under this implementation, adding 24 elements one at a time results in a capacity of 32.

Visually we can think of the current state of `ivec` as



We can now reserve some additional space:

```
ivec.reserve(50); // sets capacity to at least 50; might be more
// size should be 24; capacity will be >= 50 and is implementation defined
cout << "ivec: size: " << ivec.size()
    << " capacity: " << ivec.capacity() << endl;
```

Here, the output indicates that the call to `reserve` allocated exactly as much space as we requested:

```
ivec: size: 24 capacity: 50
```

We might next use up that reserved capacity as follows:

```
// add elements to use up the excess capacity
while (ivec.size() != ivec.capacity())
    ivec.push_back(0);
// capacity should be unchanged and size and capacity are now equal
cout << "ivec: size: " << ivec.size()
    << " capacity: " << ivec.capacity() << endl;
```

The output indicates that at this point we've used up the reserved capacity, and size and capacity are equal:

```
ivec: size: 50 capacity: 50
```

Because we used only reserved capacity, there is no need for the vector to do any allocation. In fact, as long as no operation exceeds the vector's capacity, the vector must not reallocate its elements.

If we now add another element, the vector will have to reallocate itself:

```
ivec.push_back(42); // add one more element
// size should be 51; capacity will be >= 51 and is implementation defined
cout << "ivec: size: " << ivec.size()
    << " capacity: " << ivec.capacity() << endl;
```

The output from this portion of the program

```
ivec: size: 51 capacity: 100
```

indicates that this vector implementation appears to follow a strategy of doubling the current capacity each time it has to allocate new storage.

We can call `shrink_to_fit` to ask that memory beyond what is needed for the current size be returned to the system:

```
ivec.shrink_to_fit(); // ask for the memory to be returned
// size should be unchanged; capacity is implementation defined
cout << "ivec: size: " << ivec.size()
     << " capacity: " << ivec.capacity() << endl;
```

Calling `shrink_to_fit` is only a request; there is no guarantee that the library will return the memory.



Each vector implementation can choose its own allocation strategy. However, it must not allocate new memory until it is forced to do so.

A vector may be reallocated only when the user performs an insert operation when the size equals capacity or by a call to `resize` or `reserve` with a value that exceeds the current capacity. How much memory is allocated beyond the specified amount is up to the implementation.

Every implementation is required to follow a strategy that ensures that it is efficient to use `push_back` to add elements to a vector. Technically speaking, the execution time of creating an  $n$ -element vector by calling `push_back`  $n$  times on an initially empty vector must never be more than a constant multiple of  $n$ .

## EXERCISES SECTION 9.4

**Exercise 9.35:** Explain the difference between a vector's capacity and its size.

**Exercise 9.36:** Can a container have a capacity less than its size?

**Exercise 9.37:** Why don't list or array have a capacity member?

**Exercise 9.38:** Write a program to explore how vectors grow in the library you use.

**Exercise 9.39:** Explain what the following program fragment does:

```
vector<string> svec;
svec.reserve(1024);
string word;
while (cin >> word)
    svec.push_back(word);
svec.resize(svec.size()+svec.size()/2);
```

**Exercise 9.40:** If the program in the previous exercise reads 256 words, what is its likely capacity after it is resized? What if it reads 512? 1,000? 1,048?

## 9.5 Additional string Operations

The `string` type provides a number of additional operations beyond those common to the sequential containers. For the most part, these additional operations either support the close interaction between the `string` class and C-style character arrays, or they add versions that let us use indices in place of iterators.

The `string` library defines a great number of functions. Fortunately, these functions use repeated patterns. Given the number of functions supported, this section can be mind-numbing on first reading; so readers might want to skim it. Once you know what kinds of operations are available, you can return for the details when you need to use a particular operation.



### 9.5.1 Other Ways to Construct strings

In addition to the constructors we covered in § 3.2.1 (p. 84) and to the constructors that `string` shares with the other sequential containers (Tables 9.3 (p. 335)) the `string` type supports three more constructors that are described in Table 9.11.

| Table 9.11: Additional Ways to Construct strings |  |
|--|--|
| n, len2 and pos2 are all unsigned values         |  |
| <code>string s(cp, n);</code>                    | s is a copy of the first n characters in the array to which cp points. That array must have at least n characters.   |
| <code>string s(s2, pos2);</code>                 | s is a copy of the characters in the string s2 starting at the index pos2. Undefined if pos2 > s2.size().  |
| <code>string s(s2, pos2, len2);</code>           | s is a copy of len2 characters from s2 starting at the index pos2. Undefined if pos2 > s2.size(). Regardless of the value of len2, copies at most s2.size() - pos2 characters. |

The constructors that take a `string` or a `const char*` take additional (optional) arguments that let us specify how many characters to copy. When we pass a `string`, we can also specify the index of where to start the copy:

```
const char *cp = "Hello World!!!"; // null-terminated array
char noNull[] = {'H', 'i'};        // not null terminated
string s1(cp); // copy up to the null in cp; s1 == "Hello World!!!"
string s2(noNull, 2); // copy two characters from no_null; s2 == "Hi"
string s3(noNull); // undefined: noNull not null terminated
string s4(cp + 6, 5); // copy 5 characters starting at cp[6]; s4 == "World"
string s5(s1, 6, 5); // copy 5 characters starting at s1[6]; s5 == "World"
string s6(s1, 6); // copy from s1[6] to end of s1; s6 == "World!!!"
string s7(s1, 6, 20); // ok, copies only to end of s1; s7 == "World!!!"
string s8(s1, 16); // throws an out_of_range exception
```

Ordinarily when we create a `string` from a `const char*`, the array to which the pointer points must be null terminated; characters are copied up to the null. If

we also pass a count, the array does not have to be null terminated. If we do not pass a count and there is no null, or if the given count is greater than the size of the array, the operation is undefined.

When we copy from a `string`, we can supply an optional starting position and a count. The starting position must be less than or equal to the size of the given `string`. If the position is greater than the size, then the constructor throws an `out_of_range` exception (§ 5.6, p. 193). When we pass a count, that many characters are copied, starting from the given position. Regardless of how many characters we ask for, the library copies up to the size of the `string`, but not more.

The `substr` Operation

The `substr` operation (described in Table 9.12) returns a `string` that is a copy of part or all of the original `string`. We can pass `substr` an optional starting position and count:

```
string s("hello world");
string s2 = s.substr(0, 5); // s2 = hello
string s3 = s.substr(6);   // s3 = world
string s4 = s.substr(6, 11); // s3 = world
string s5 = s.substr(12);  // throws an out_of_range exception
```

The `substr` function throws an `out_of_range` exception (§ 5.6, p. 193) if the position exceeds the size of the `string`. If the position plus the count is greater than the size, the count is adjusted to copy only up to the end of the `string`.

| Table 9.12: Substring Operation |  |
|---------------------------------|--|
| <code>s.substr(pos, n)</code>   | Return a <code>string</code> containing <code>n</code> characters from <code>s</code> starting at <code>pos</code> .<br><code>pos</code> defaults to 0. <code>n</code> defaults to a value that causes the library to copy all the characters in <code>s</code> starting from <code>pos</code> . |

EXERCISES SECTION 9.5.1

**Exercise 9.41:** Write a program that initializes a `string` from a `vector<char>`.

**Exercise 9.42:** Given that you want to read a character at a time into a `string`, and you know that you need to read at least 100 characters, how might you improve the performance of your program?

9.5.2 Other Ways to Change a `string`



The `string` type supports the sequential container assignment operators and the `assign`, `insert`, and `erase` operations (§ 9.2.5, p. 337, § 9.3.1, p. 342, and § 9.3.3, p. 348). It also defines additional versions of `insert` and `erase`.

In addition to the versions of `insert` and `erase` that take iterators, `string` provides versions that take an index. The index indicates the starting element to erase or the position before which to insert the given values:

```
s.insert(s.size(), 5, '!'); // insert five exclamation points at the end of s
s.erase(s.size() - 5, 5);   // erase the last five characters from s
```

The `string` library also provides versions of `insert` and `assign` that take C-style character arrays. For example, we can use a null-terminated character array as the value to insert or assign into a `string`:

```
const char *cp = "Stately, plump Buck";
s.assign(cp, 7);           // s == "Stately"
s.insert(s.size(), cp + 7); // s == "Stately, plump Buck"
```

Here we first replace the contents of `s` by calling `assign`. The characters we assign into `s` are the seven characters starting with the one pointed to by `cp`. The number of characters we request must be less than or equal to the number of characters (excluding the null terminator) in the array to which `cp` points.

When we call `insert` on `s`, we say that we want to insert the characters before the (nonexistent) element at `s[size()]`. In this case, we copy characters starting seven characters past `cp` up to the terminating null.

We can also specify the characters to insert or assign as coming from another `string` or substring thereof:

```
string s = "some string", s2 = "some other string";
s.insert(0, s2); // insert a copy of s2 before position 0 in s
// insert s2.size() characters from s2 starting at s2[0] before s[0]
s.insert(0, s2, 0, s2.size());
```

## The append and replace Functions

The `string` class defines two additional members, `append` and `replace`, that can change the contents of a `string`. Table 9.13 summarizes these functions. The `append` operation is a shorthand way of inserting at the end:

```
string s("C++ Primer"), s2 = s; // initialize s and s2 to "C++ Primer"
s.insert(s.size(), " 4th Ed."); // s == "C++ Primer 4th Ed."
s2.append(" 4th Ed."); // equivalent: appends " 4th Ed." to s2; s2 == s2
```

The `replace` operations are a shorthand way of calling `erase` and `insert`:

```
// equivalent way to replace "4th" by "5th"
s.erase(11, 3); // s == "C++ Primer Ed."
s.insert(11, "5th"); // s == "C++ Primer 5th Ed."
// starting at position 11, erase three characters and then insert "5th"
s2.replace(11, 3, "5th"); // equivalent: s2 == s2
```

In the call to `replace`, the text we inserted happens to be the same size as the text we removed. We can insert a larger or smaller `string`:

```
s.replace(11, 3, "Fifth"); // s == "C++ Primer Fifth Ed."
```

In this call we remove three characters but insert five in their place.

**Table 9.13: Operations to Modify strings**

|   |  |   |  |                                      |
|---|--|---|--|--------------------------------------|
| <code>s.insert(pos, args)</code>  | Insert characters specified by <i>args</i> before <i>pos</i> . <i>pos</i> can be an index or an iterator. Versions taking an index return a reference to <i>s</i> ; those taking an iterator return an iterator denoting the first inserted character. |   |  |                                      |
| <code>s.erase(pos, len)</code>  | Remove <i>len</i> characters starting at position <i>pos</i> . If <i>len</i> is omitted, removes characters from <i>pos</i> to the end of the <i>s</i> . Returns a reference to <i>s</i> .   |   |  |                                      |
| <code>s.assign(args)</code>   | Replace characters in <i>s</i> according to <i>args</i> . Returns a reference to <i>s</i> .  |   |  |                                      |
| <code>s.append(args)</code>   | Append <i>args</i> to <i>s</i> . Returns a reference to <i>s</i> .   |   |  |                                      |
| <code>s.replace(range, args)</code>   | Remove <i>range</i> of characters from <i>s</i> and replace them with the characters formed by <i>args</i> . <i>range</i> is either an index and a length or a pair of iterators into <i>s</i> . Returns a reference to <i>s</i> .                     |   |  |                                      |
| <b><i>args</i> can be one of the following; <i>append</i> and <i>assign</i> can use all forms<br/><i>str</i> must be distinct from <i>s</i> and the iterators <i>b</i> and <i>e</i> may not refer to <i>s</i></b> |  |   |  |                                      |
| <i>str</i>  | The string <i>str</i> .  |   |  |                                      |
| <i>str</i> , <i>pos</i> , <i>len</i>  | Up to <i>len</i> characters from <i>str</i> starting at <i>pos</i> .   |   |  |                                      |
| <i>cp</i> , <i>len</i>  | Up to <i>len</i> characters from the character array pointed to by <i>cp</i> .   |   |  |                                      |
| <i>cp</i>   | Null-terminated array pointed to by pointer <i>cp</i> .  |   |  |                                      |
| <i>n</i> , <i>c</i>   | <i>n</i> copies of character <i>c</i> .  |   |  |                                      |
| <i>b</i> , <i>e</i>   | Characters in the range formed by iterators <i>b</i> and <i>e</i> .  |   |  |                                      |
| <i>initializer list</i>   | Comma-separated list of characters enclosed in braces.   |   |  |                                      |
| <b><i>args</i> for <i>replace</i> and <i>insert</i> depend on how <i>range</i> or <i>pos</i> is specified.</b>  |  |   |  |                                      |
| <i>replace</i><br>( <i>pos</i> , <i>len</i> , <i>args</i> )   | <i>replace</i><br>( <i>b</i> , <i>e</i> , <i>args</i> )  | <i>insert</i><br>( <i>pos</i> , <i>args</i> ) | <i>insert</i><br>( <i>iter</i> , <i>args</i> ) | <i>args</i> can be                   |
| yes   | yes  | yes   | no   | <i>str</i>                           |
| yes   | no   | yes   | no   | <i>str</i> , <i>pos</i> , <i>len</i> |
| yes   | yes  | yes   | no   | <i>cp</i> , <i>len</i>               |
| yes   | yes  | no  | no   | <i>cp</i>                            |
| yes   | yes  | yes   | yes  | <i>n</i> , <i>c</i>                  |
| no  | yes  | no  | yes  | <i>b2</i> , <i>e2</i>                |
| no  | yes  | no  | yes  | <i>initializer list</i>              |

### The Many Overloaded Ways to Change a string

The `append`, `assign`, `insert`, and `replace` functions listed Table 9.13 have several overloaded versions. The arguments to these functions vary as to how we specify what characters to add and what part of the string to change. Fortunately, these functions share a common interface.

The `assign` and `append` functions have no need to specify what part of the string is changed: `assign` always replaces the entire contents of the string and `append` always adds to the end of the string.

The `replace` functions provide two ways to specify the range of characters to remove. We can specify that range by a position and a length, or with an iterator

range. The `insert` functions give us two ways to specify the insertion point: with either an index or an iterator. In each case, the new element(s) are inserted in front of the given index or iterator.

There are several ways to specify the characters to add to the `string`. The new characters can be taken from another `string`, from a character pointer, from a brace-enclosed list of characters, or as a character and a count. When the characters come from a `string` or a character pointer, we can pass additional arguments to control whether we copy some or all of the characters from the argument.

Not every function supports every version of these arguments. For example, there is no version of `insert` that takes an index and an initializer list. Similarly, if we want to specify the insertion point using an iterator, then we cannot pass a character pointer as the source for the new characters.

## EXERCISES SECTION 9.5.2

**Exercise 9.43:** Write a function that takes three `strings`, `s`, `oldVal`, and `newVal`. Using iterators, and the `insert` and `erase` functions replace all instances of `oldVal` that appear in `s` by `newVal`. Test your function by using it to replace common abbreviations, such as “tho” by “though” and “thru” by “through”.

**Exercise 9.44:** Rewrite the previous function using an index and `replace`.

**Exercise 9.45:** Write a function that takes a `string` representing a name and two other `strings` representing a prefix, such as “Mr.” or “Ms.” and a suffix, such as “Jr.” or “III”. Using iterators and the `insert` and `append` functions, generate and return a new `string` with the suffix and prefix added to the given name.

**Exercise 9.46:** Rewrite the previous exercise using a position and length to manage the `strings`. This time use only the `insert` function.



## 9.5.3 `string` Search Operations

The `string` class provides six different search functions, each of which has four overloaded versions. Table 9.14 describes the search members and their arguments. Each of these search operations returns a `string::size_type` value that is the index of where the match occurred. If there is no match, the function returns a static member (§ 7.6, p. 300) named `string::npos`. The library defines `npos` as a `const string::size_type` initialized with the value `-1`. Because `npos` is an unsigned type, this initializer means `npos` is equal to the largest possible size any `string` could have (§ 2.1.2, p. 35).



**WARNING**

The `string` search functions return `string::size_type`, which is an unsigned type. As a result, it is a bad idea to use an `int`, or other signed type, to hold the return from these functions (§ 2.1.2, p. 36).

The `find` function does the simplest search. It looks for its argument and returns the index of the first match that is found, or `npos` if there is no match:



```
string name("AnnaBelle");
auto pos1 = name.find("Anna"); // pos1 == 0
```

returns 0, the index at which the substring "Anna" is found in "AnnaBelle".

Searching (and other string operations) are case sensitive. When we look for a value in the string, case matters:

```
string lowercase("annabelle");
pos1 = lowercase.find("Anna"); // pos1 == npos
```

This code will set `pos1` to `npos` because Anna does not match anna.

A slightly more complicated problem requires finding a match to any character in the search string. For example, the following locates the first digit within `name`:

```
string numbers("0123456789"), name("r2d2");
// returns 1, i.e., the index of the first digit in name
auto pos = name.find_first_of(numbers);
```

Instead of looking for a match, we might call `find_first_not_of` to find the first position that is *not* in the search argument. For example, to find the first non-numeric character of a string, we can write

```
string dept("03714p3");
// returns 5, which is the index to the character 'p'
auto pos = dept.find_first_not_of(numbers);
```

**Table 9.14: string Search Operations**

**Search operations return the index of the desired character or `npos` if not found**

|  |   |
|--|---|
| <code>s.find(args)</code>              | Find the first occurrence of <i>args</i> in <i>s</i> .                    |
| <code>s.rfind(args)</code>             | Find the last occurrence of <i>args</i> in <i>s</i> .                     |
| <code>s.find_first_of(args)</code>     | Find the first occurrence of any character from <i>args</i> in <i>s</i> . |
| <code>s.find_last_of(args)</code>      | Find the last occurrence of any character from <i>args</i> in <i>s</i> .  |
| <code>s.find_first_not_of(args)</code> | Find the first character in <i>s</i> that is not in <i>args</i> .         |
| <code>s.find_last_not_of(args)</code>  | Find the last character in <i>s</i> that is not in <i>args</i> .          |

*args must be one of*

|                         |  |
|-------------------------|--|
| <code>c, pos</code>     | Look for the character <i>c</i> starting at position <i>pos</i> in <i>s</i> . <i>pos</i> defaults to 0.  |
| <code>s2, pos</code>    | Look for the string <i>s2</i> starting at position <i>pos</i> in <i>s</i> . <i>pos</i> defaults to 0.  |
| <code>cp, pos</code>    | Look for the C-style null-terminated string pointed to by the pointer <i>cp</i> . Start looking at position <i>pos</i> in <i>s</i> . <i>pos</i> defaults to 0.                       |
| <code>cp, pos, n</code> | Look for the first <i>n</i> characters in the array pointed to by the pointer <i>cp</i> . Start looking at position <i>pos</i> in <i>s</i> . No default for <i>pos</i> or <i>n</i> . |

## Specifying Where to Start the Search

We can pass an optional starting position to the `find` operations. This optional argument indicates the position from which to start the search. By default, that position is set to zero. One common programming pattern uses this optional argument to loop through a string finding all occurrences:

```

string::size_type pos = 0;
// each iteration finds the next number in name
while ((pos = name.find_first_of(numbers, pos))
       != string::npos) {
    cout << "found number at index: " << pos
          << " element is " << name[pos] << endl;
    ++pos; // move to the next character
}

```

The condition in the `while` resets `pos` to the index of the first number encountered, starting from the current value of `pos`. So long as `find_first_of` returns a valid index, we print the current result and increment `pos`.

Had we neglected to increment `pos`, the loop would never terminate. To see why, consider what would happen if we didn't do the increment. On the second trip through the loop we start looking at the character indexed by `pos`. That character would be a number, so `find_first_of` would (repeatedly) return `pos`!

## Searching Backward

The `find` operations we've used so far execute left to right. The library provides analogous operations that search from right to left. The `rfind` member searches for the last—that is, right-most—occurrence of the indicated substring:

```

string river("Mississippi");
auto first_pos = river.find("is"); // returns 1
auto last_pos = river.rfind("is"); // returns 4

```

`find` returns an index of 1, indicating the start of the first "is", while `rfind` returns an index of 4, indicating the start of the last occurrence of "is".

Similarly, the `find_last` functions behave like the `find_first` functions, except that they return the *last* match rather than the first:

- `find_last_of` searches for the last character that matches any element of the search string.
- `find_last_not_of` searches for the last character that does not match any element of the search string.

Each of these operations takes an optional second argument indicating the position within the string to begin searching.



## 9.5.4 The compare Functions

In addition to the relational operators (§ 3.2.2, p. 88), the `string` library provides a set of compare functions that are similar to the C library `strcmp` function (§ 3.5.4, p. 122). Like `strcmp`, `s.compare` returns zero or a positive or negative value depending on whether `s` is equal to, greater than, or less than the string formed from the given arguments.

EXERCISES SECTION 9.5.3

**Exercise 9.47:** Write a program that finds each numeric character and then each alphabetic character in the string "ab2c3d7R4E6". Write two versions of the program. The first should use `find_first_of`, and the second `find_first_not_of`.

**Exercise 9.48:** Given the definitions of `name` and `numbers` on page 365, what does `numbers.find(name)` return?

**Exercise 9.49:** A letter has an ascender if, as with `d` or `f`, part of the letter extends above the middle of the line. A letter has a descender if, as with `p` or `g`, part of the letter extends below the line. Write a program that reads a file containing words and reports the longest word that contains neither ascenders nor descenders.

As shown in Table 9.15, there are six versions of `compare`. The arguments vary based on whether we are comparing two strings or a string and a character array. In both cases, we might compare the entire string or a portion thereof.

Table 9.15: Possible Arguments to `s.compare`

|                                     |  |
|-------------------------------------|--|
| <code>s2</code>                     | Compare <code>s</code> to <code>s2</code> .  |
| <code>pos1, n1, s2</code>           | Compares <code>n1</code> characters starting at <code>pos1</code> from <code>s</code> to <code>s2</code> .   |
| <code>pos1, n1, s2, pos2, n2</code> | Compares <code>n1</code> characters starting at <code>pos1</code> from <code>s</code> to the <code>n2</code> characters starting at <code>pos2</code> in <code>s2</code> . |
| <code>cp</code>                     | Compares <code>s</code> to the null-terminated array pointed to by <code>cp</code> .   |
| <code>pos1, n1, cp</code>           | Compares <code>n1</code> characters starting at <code>pos1</code> from <code>s</code> to <code>cp</code> .   |
| <code>pos1, n1, cp, n2</code>       | Compares <code>n1</code> characters starting at <code>pos1</code> from <code>s</code> to <code>n2</code> characters starting from the pointer <code>cp</code> .            |

9.5.5 Numeric Conversions



Strings often contain characters that represent numbers. For example, we represent the numeric value 15 as a string with two characters, the character '1' followed by the character '5'. In general, the character representation of a number differs from its numeric value. The numeric value 15 stored in a 16-bit short has the bit pattern 0000000000001111, whereas the character string "15" represented as two Latin-1 chars has the bit pattern 0011000100110101. The first byte represents the character '1' which has the octal value 061, and the second byte represents '5', which in Latin-1 is octal 065.

The new standard introduced several functions that convert between numeric data and library strings:



```
int i = 42;
string s = to_string(i); // converts the int i to its character representation
double d = stod(s);      // converts the string s to floating-point
```

Table 9.16: Conversions between strings and Numbers

|                              |  |
|------------------------------|--|
| <code>to_string(val)</code>  | Overloaded functions returning the string representation of <code>val</code> . <code>val</code> can be any arithmetic type (§ 2.1.1, p. 32). There are versions of <code>to_string</code> for each floating-point type and integral type that is <code>int</code> or larger. Small integral types are promoted (§ 4.11.1, p. 160) as usual.  |
| <code>stoi(s, p, b)</code>   | Return the initial substring of <code>s</code> that has numeric content as an <code>int</code> , <code>long</code> , <code>unsigned long</code> , <code>long long</code> , <code>unsigned long long</code> , respectively. <code>b</code> indicates the numeric base to use for the conversion; <code>b</code> defaults to 10. <code>p</code> is a pointer to a <code>size_t</code> in which to put the index of the first nonnumeric character in <code>s</code> ; <code>p</code> defaults to 0, in which case the function does not store the index. |
| <code>stol(s, p, b)</code>   |  |
| <code>stoul(s, p, b)</code>  |  |
| <code>stoll(s, p, b)</code>  |  |
| <code>stoull(s, p, b)</code> |  |
| <code>stof(s, p)</code>      | Return the initial numeric substring in <code>s</code> as a <code>float</code> , <code>double</code> , or <code>long double</code> , respectively. <code>p</code> has the same behavior as described for the integer conversions.  |
| <code>stod(s, p)</code>      |  |
| <code>stold(s, p)</code>     |  |


Here we call `to_string` to convert 42 to its corresponding string representation and then call `stod` to convert that string to floating-point.

The first non-whitespace character in the string we convert to numeric value must be a character that can appear in a number:

```
string s2 = "pi = 3.14";  
// convert the first substring in s that starts with a digit, d = 3.14  
d = stod(s2.substr(s2.find_first_of("+- .0123456789")));
```

In this call to `stod`, we call `find_first_of` (§ 9.5.3, p. 364) to get the position of the first character in `s` that could be part of a number. We pass the substring of `s` starting at that position to `stod`. The `stod` function reads the string it is given until it finds a character that cannot be part of a number. It then converts the character representation of the number it found into the corresponding double-precision floating-point value.

The first non-whitespace character in the string must be a sign (+ or -) or a digit. The string can begin with `0x` or `0X` to indicate hexadecimal. For the functions that convert to floating-point the string may also start with a decimal point (.) and may contain an `e` or `E` to designate the exponent. For the functions that convert to integral type, depending on the base, the string can contain alphabetic characters corresponding to numbers beyond the digit 9.



If the string can't be converted to a number, These functions throw an `invalid_argument` exception (§ 5.6, p. 193). If the conversion generates a value that can't be represented, they throw `out_of_range`.



## 9.6 Container Adaptors

In addition to the sequential containers, the library defines three sequential container adaptors: `stack`, `queue`, and `priority_queue`. An **adaptor** is a general

EXERCISES SECTION 9.5.5

**Exercise 9.50:** Write a program to process a `vector<string>s` whose elements represent integral values. Produce the sum of all the elements in that vector. Change the program so that it sums of strings that represent floating-point values.

**Exercise 9.51:** Write a class that has three unsigned members representing year, month, and day. Write a constructor that takes a string representing a date. Your constructor should handle a variety of date formats, such as January 1, 1900, 1/1/1900, Jan 1, 1900, and so on.

concept in the library. There are container, iterator, and function adaptors. Essentially, an adaptor is a mechanism for making one thing act like another. A container adaptor takes an existing container type and makes it act like a different type. For example, the `stack` adaptor takes a sequential container (other than `array` or `forward_list`) and makes it operate as if it were a `stack`. Table 9.17 lists the operations and types that are common to all the container adaptors.

Table 9.17: Operations and Types Common to the Container Adaptors

|                             |   |
|-----------------------------|---|
| <code>size_type</code>      | Type large enough to hold the size of the largest object of this type.  |
| <code>value_type</code>     | Element type.   |
| <code>container_type</code> | Type of the underlying container on which the adaptor is implemented.   |
| <code>A a;</code>           | Create a new empty adaptor named <code>a</code> .   |
| <code>A a(c);</code>        | Create a new adaptor named <code>a</code> with a copy of the container <code>c</code> .   |
| <i>relational operators</i> | Each adaptor supports all the relational operators: <code>==</code> , <code>!=</code> , <code>&lt;</code> , <code>&lt;=</code> , <code>&gt;</code> , <code>&gt;=</code> . These operators return the result of comparing the underlying containers. |
| <code>a.empty()</code>      | <code>false</code> if <code>a</code> has any elements, <code>true</code> otherwise.   |
| <code>a.size()</code>       | Number of elements in <code>a</code> .  |
| <code>swap(a, b)</code>     | Swaps the contents of <code>a</code> and <code>b</code> ; <code>a</code> and <code>b</code> must have the same type, including the type of the container on which they are implemented.   |
| <code>a.swap(b)</code>      |   |

Defining an Adaptor

Each adaptor defines two constructors: the default constructor that creates an empty object, and a constructor that takes a container and initializes the adaptor by copying the given container. For example, assuming that `deq` is a `deque<int>`, we can use `deq` to initialize a new `stack` as follows:

```
stack<int> stk(deq); // copies elements from deq into stk
```

By default both `stack` and `queue` are implemented in terms of `deque`, and a `priority_queue` is implemented on a `vector`. We can override the default container type by naming a sequential container as a second type argument when we create the adaptor:

```
// empty stack implemented on top of vector
stack<string, vector<string>> str_stk;

// str_stk2 is implemented on top of vector and initially holds a copy of svec
stack<string, vector<string>> str_stk2(svec);
```

There are constraints on which containers can be used for a given adaptor. All of the adaptors require the ability to add and remove elements. As a result, they cannot be built on an array. Similarly, we cannot use `forward_list`, because all of the adaptors require operations that add, remove, or access the last element in the container. A stack requires only `push_back`, `pop_back`, and back operations, so we can use any of the remaining container types for a stack. The queue adaptor requires `back`, `push_back`, `front`, and `push_front`, so it can be built on a list or deque but not on a vector. A `priority_queue` requires random access in addition to the `front`, `push_back`, and `pop_back` operations; it can be built on a vector or a deque but not on a list.

Stack Adaptor

The stack type is defined in the stack header. The operations provided by a stack are listed in Table 9.18. The following program illustrates the use of stack:

```
stack<int> intStack; // empty stack
// fill up the stack
for (size_t ix = 0; ix != 10; ++ix)
    intStack.push(ix); // intStack holds 0...9 inclusive
while (!intStack.empty()) { // while there are still values in intStack
    int value = intStack.top();
    // code that uses value
    intStack.pop(); // pop the top element, and repeat
}
```

The declaration

```
stack<int> intStack; // empty stack
```

defines `intStack` to be an empty stack that holds integer elements. The `for` loop adds ten elements, initializing each to the next integer in sequence starting from zero. The `while` loop iterates through the entire stack, examining the top value and popping it from the stack until the stack is empty.

| Table 9.18: Stack Operations in Addition to Those in Table 9.17        |  |
|--|--|
| Uses deque by default; can be implemented on a list or vector as well. |  |
| s.pop()  | Removes, but does not return, the top element from the stack.        |
| s.push(item)   | Creates a new top element on the stack by copying or moving item, or |
| s.emplace(args)  | by constructing the element from args.                               |
| s.top()  | Returns, but does not remove, the top element on the stack.          |

Each container adaptor defines its own operations in terms of operations provided by the underlying container type. We can use only the adaptor operations and cannot use the operations of the underlying container type. For example,

```
intStack.push(ix);    // intStack holds 0...9 inclusive
```

calls `push_back` on the `deque` object on which `intStack` is based. Although `stack` is implemented by using a `deque`, we have no direct access to the `deque` operations. We cannot call `push_back` on a `stack`; instead, we must use the `stack` operation named `push`.

The Queue Adaptors

The `queue` and `priority_queue` adaptors are defined in the `queue` header. Table 9.19 lists the operations supported by these types.

| Table 9.19: <code>queue</code> , <code>priority_queue</code> Operations in Addition to Table 9.17  |   |
|--|---|
| By default <code>queue</code> uses <code>deque</code> and <code>priority_queue</code> uses <code>vector</code> ; <code>queue</code> can use a <code>list</code> or <code>vector</code> as well, <code>priority_queue</code> can use a <code>deque</code> . |   |
| <code>q.pop()</code>   | Removes, but does not return, the front element or highest-priority element from the <code>queue</code> or <code>priority_queue</code> , respectively.                                    |
| <code>q.front()</code>   | Returns, but does not remove, the front or back element of <code>q</code> .   |
| <code>q.back()</code>  | <b>Valid only for <code>queue</code></b>  |
| <code>q.top()</code>   | Returns, but does not remove, the highest-priority element.<br><b>Valid only for <code>priority_queue</code>.</b>   |
| <code>q.push(item)</code>  | Create an element with value <code>item</code> or constructed from <code>args</code> at the end of the <code>queue</code> or in its appropriate position in <code>priority_queue</code> . |
| <code>q.emplace(args)</code>   |   |

The library `queue` uses a first-in, first-out (FIFO) storage and retrieval policy. Objects entering the `queue` are placed in the back and objects leaving the `queue` are removed from the front. A restaurant that seats people in the order in which they arrive is an example of a FIFO `queue`.

A `priority_queue` lets us establish a priority among the elements held in the `queue`. Newly added elements are placed ahead of all the elements with a lower priority. A restaurant that seats people according to their reservation time, regardless of when they arrive, is an example of a priority `queue`. By default, the library uses the `<` operator on the element type to determine relative priorities. We'll learn how to override this default in § 11.2.2 (p. 425).

EXERCISES SECTION 9.6

**Exercise 9.52:** Use a `stack` to process parenthesized expressions. When you see an open parenthesis, note that it was seen. When you see a close parenthesis after an open parenthesis, `pop` elements down to and including the open parenthesis off the `stack`. `push` a value onto the `stack` to indicate that a parenthesized expression was replaced.

## CHAPTER SUMMARY

---

The library containers are template types that holds objects of a given type. In a sequential container, elements are ordered and accessed by position. The sequential containers share a common, standardized interface: If two sequential containers offer a particular operation, then the operation has the same interface and meaning for both containers.

All the containers (except `array`) provide efficient dynamic memory management. We may add elements to the container without worrying about where to store the elements. The container itself manages its storage. Both `vector` and `string` provide more detailed control over memory management through their `reserve` and `capacity` members.

For the most part, the containers define surprisingly few operations. Containers define constructors, operations to add or remove elements, operations to determine the size of the container, and operations to return iterators to particular elements. Other useful operations, such as sorting or searching, are defined not by the container types but by the standard algorithms, which we shall cover in Chapter 10.

When we use container operations that add or remove elements, it is essential to remember that these operations can invalidate iterators, pointers, or references to elements in the container. Many operations that invalidate an iterator, such as `insert` or `erase`, return a new iterator that allows the programmer to maintain a position within the container. Loops that use container operations that change the size of a container should be particularly careful in their use of iterators, pointers, and references.

## DEFINED TERMS

---

**adaptor** Library type, function, or iterator that, given a type, function, or iterator, makes it act like another. There are three sequential container adaptors: `stack`, `queue`, and `priority_queue`. Each adaptor defines a new interface on top of an underlying sequential container type.

**array** Fixed-size sequential container. To define an array, we must give the size in addition to specifying the element type. Elements in an array can be accessed by their positional index. Supports fast random access to elements.

**begin** Container operation that returns an iterator referring to the first element in the container, if there is one, or the off-the-end iterator if the container is empty. Whether the returned iterator is `const` depends on the type of the container.

**cbegin** Container operation that returns a `const_iterator` referring to the first element in the container, if there is one, or the off-the-end iterator if the container is empty.

**cend** Container operation that returns a `const_iterator` referring to the (nonexistent) element one past the end of the container.

**container** Type that holds a collection of objects of a given type. Each library container type is a template type. To define a container, we must specify the type of the elements stored in the container. With the exception of `array`, the library containers are variable-size.

**deque** Sequential container. Elements in a deque can be accessed by their positional



index. Supports fast random access to elements. Like a `vector` in all respects except that it supports fast insertion and deletion at the front of the container as well as at the back and does not relocate elements as a result of insertions or deletions at either end.

**end** Container operation that returns an iterator referring to the (nonexistent) element one past the end of the container. Whether the returned iterator is `const` depends on the type of the container.

**forward\_list** Sequential container that represents a singly linked list. Elements in a `forward_list` may be accessed only sequentially; starting from a given element, we can get to another element only by traversing each element between them. Iterators on `forward_list` do not support decrement (`--`). Supports fast insertion (or deletion) anywhere in the `forward_list`. Unlike other containers, insertions and deletions occur *after* a given iterator position. As a consequence, `forward_list` has a “before-the-beginning” iterator to go along with the usual off-the-end iterator. Iterators remain valid when new elements are added. When an element is removed, only the iterators to that element are invalidated.

**iterator range** Range of elements denoted by a pair of iterators. The first iterator denotes the first element in the sequence, and the second iterator denotes one past the last element. If the range is empty, then the iterators are equal (and vice versa—if the iterators are unequal, they denote a nonempty range). If the range is not empty, then it must be possible to reach the second iterator by repeatedly incrementing the first iterator. By incrementing the iterator, each element in the sequence can be processed.

**left-inclusive interval** A range of values that includes its first element but not its last. Typically denoted as `[i, j)`, meaning the sequence starting at and including `i` up to but excluding `j`.

**list** Sequential container representing a doubly linked list. Elements in a `list` may

be accessed only sequentially; starting from a given element, we can get to another element only by traversing each element between them. Iterators on `list` support both increment (`++`) and decrement (`--`). Supports fast insertion (or deletion) anywhere in the `list`. Iterators remain valid when new elements are added. When an element is removed, only the iterators to that element are invalidated.

**off-the-beginning iterator** Iterator denoting the (nonexistent) element just before the beginning of a `forward_list`. Returned from the `forward_list` member `before_begin`. Like the `end()` iterator, it may not be dereferenced.

**off-the-end iterator** Iterator that denotes one past the last element in the range. Commonly referred to as the “end iterator”.

**priority\_queue** Adaptor for the sequential containers that yields a queue in which elements are inserted, not at the end but according to a specified priority level. By default, priority is determined by using the less-than operator for the element type.

**queue** Adaptor for the sequential containers that yields a type that lets us add elements to the back and remove elements from the front.

**sequential container** Type that holds an ordered collection of objects of a single type. Elements in a sequential container are accessed by position.

**stack** Adaptor for the sequential containers that yields a type that lets us add and remove elements from one end only.

**vector** Sequential container. Elements in a `vector` can be accessed by their positional index. Supports fast random access to elements. We can efficiently add or remove `vector` elements only at the back. Adding elements to a `vector` might cause it to be reallocated, invalidating all iterators into the `vector`. Adding (or removing) an element in the middle of a `vector` invalidates all iterators to elements after the insertion (or deletion) point.