Chapter Summary

Among the most important library types are `vector` and `string`. A `string` is a variable-length sequence of characters, and a `vector` is a container of objects of a single type.

Iterators allow indirect access to objects stored in a container. Iterators are used to access and navigate between the elements in `string`s and `vector`s.

Arrays and pointers to array elements provide low-level analogs to the `vector` and `string` libraries. In general, the library classes should be used in preference to low-level array and pointer alternatives built into the language.

Defined Terms

begin Member of `string` and `vector` that returns an iterator to the first element. Also, free-standing library function that takes an array and returns a pointer to the first element in the array.

buffer overflow Serious programming bug that results when we use an index that is out-of-range for a container, such as a `string`, `vector`, or an array.

C-style strings Null-terminated character array. String literals are C-style strings. C-style strings are inherently error-prone.

class template A blueprint from which specific clas types can be created. To use a class template, we must specify additional information. For example, to define a `vector`, we specify the element type: `vector<int>` holds `int`s.

compiler extension Feature that is added to the language by a particular compiler. Programs that rely on compiler extensions cannot be moved easily to other compilers.

container A type whose objects hold a collection of objects of a given type. `vector` is a container type.

copy initialization Form of initialization that uses an `=`. The newly created object is a copy of the given initializer.

difference_type A `signed` integral type defined by `vector` and `string` that can hold the distance between any two iterators.

direct initialization Form of initialization that does not include an `=`.

empty Member of `string` and `vector`. Returns `bool`, which is `true` if `size` is zero, `false` otherwise.

end Member of `string` and `vector` that returns an off-the-end iterator. Also, free-standing library function that takes an array and returns a pointer one past the last element in the array.

getline Function defined in the `string` header that takes an `istream` and a `string`. The function reads the stream up to the next newline, storing what it read into the `string`, and returns the `istream`. The newline is read and discarded.

index Value used in the subscript operator to denote the element to retrieve from a `string`, `vector`, or array.

instantiation Compiler process that generates a specific template class or function.

iterator A type used to access and navigate among the elements of a container.

iterator arithmetic Operations on `vector` or `string` iterators: Adding or subtracting an integral value and an iterator yields an iterator that many elements ahead of or behind the original iterator. Subtracting one iterator from another yields the distance between them. Iterators must refer to elements in, or off-the-end of the same container.

null-terminated string String whose last character is followed by the null character (`'\0'`).

off-the-end iterator The iterator returned by `end` that refers to a nonexistent element one past the end of a container.

pointer arithmetic The arithmetic operations that can be applied to pointers. Pointers to arrays support the same operations as iterator arithmetic.

ptrdiff_t Machine-dependent signed integral type defined in the `cstddef` header that is large enough to hold the difference between two pointers into the largest possible array.

push_back Member of `vector`. Appends elements to the back of a `vector`.

range for Control statement that iterates through a specified collection of values.

size Member of `string` and `vector`. Returns the number of characters or elements, respectively. Returns a value of the `size_type` for the type.

size_t Machine-dependent unsigned integral type defined in the `cstddef` header that is large enough to hold the size of the largest possible array.

size_type Name of types defined by the `string` and `vector` classes that are capable of containing the size of any `string` or `vector`, respectively. Library classes that define `size_type` define it as an `unsigned` type.

string Library type that represents a sequence of characters.

using declarations Make a name from a namespace accessible directly.

```
using namespace::name;
```

makes *name* accessible without the *namespace*`::` prefix.

value initialization Initialization in which built-in types are initialized to zero and class types are initialized by the class's default constructor. Objects of a class type can be value initialized only if the class has a default constructor. Used to initialize a container's elements when a size, but not an element initializer, is specified. Elements are initialized as a copy of this compiler-generated value.

vector Library type that holds a collection of elements of a specified type.

++ operator The iterator types and pointers define the increment operator to "add one" by moving the iterator to refer to the next element.

[ ] operator Subscript operator. `obj[i]` yields the element at position `i` from the container object `obj`. Indices count from zero—the first element is element `0` and the last is the element indexed by `obj.size() - 1`. Subscript returns an object. If `p` is a pointer and `n` an integer, `p[n]` is a synonym for `*(p+n)`.

-> operator Arrow operator. Combines the operations of dereference and dot operators: `a->b` is a synonym for `(*a).b`.

<< operator The `string` library type defines an output operator. The `string` operator prints the characters in a `string`.

>> operator The `string` library type defines an input operator. The `string` operator reads whitespace-delimited chunks of characters, storing what is read into the right-hand (`string`) operand.

! operator Logical NOT operator. Returns the inverse of the `bool` value of its operand. Result is `true` if operand is `false` and vice versa.

&& operator Logical AND operator. Result is `true` if both operands are `true`. The right-hand operand is evaluated *only* if the left-hand operand is `true`.

|| operator Logical OR operator. Yields `true` if either operand is `true`. The right-hand operand is evaluated *only* if the left-hand operand is `false`.