CHAPTER 3
STRINGS, VECTORS, AND ARRAYS

CONTENTS

Section 3.1 Namespace `using` Declarations 82
Section 3.2 Library `string` Type 84
Section 3.3 Library `vector` Type 96
Section 3.4 Introducing Iterators 106
Section 3.5 Arrays 113
Section 3.6 Multidimensional Arrays 125
Chapter Summary 131
Defined Terms 131

In addition to the built-in types covered in Chapter 2, C++ defines a rich library of abstract data types. Among the most important library types are `string`, which supports variable-length character strings, and `vector`, which defines variable-size collections. Associated with `string` and `vector` are companion types known as iterators, which are used to access the characters in a `string` or the elements in a `vector`.

The `string` and `vector` types defined by the library are abstractions of the more primitive built-in array type. This chapter covers arrays and introduces the library `vector` and `string` types.

The built-in types that we covered in Chapter 2 are defined directly by the C++ language. These types represent facilities present in most computer hardware, such as numbers or characters. The standard library defines a number of additional types of a higher-level nature that computer hardware usually does not implement directly.

In this chapter, we'll introduce two of the most important library types: `string` and `vector`. A `string` is a variable-length sequence of characters. A `vector` holds a variable-length sequence of objects of a given type. We'll also cover the built-in array type. Like other built-in types, arrays represent facilities of the hardware. As a result, arrays are less convenient to use than the library `string` and `vector` types.

Before beginning our exploration of the library types, we'll look at a mechanism for simplifying access to the names defined in the library.

3.1 Namespace `using` Declarations

Up to now, our programs have explicitly indicated that each library name we use is in the `std` namespace. For example, to read from the standard input, we write `std::cin`. These names use the scope operator (`::`) (§ 1.2, p. 8), which says that the compiler should look in the scope of the left-hand operand for the name of the right-hand operand. Thus, `std::cin` says that we want to use the name `cin` from the namespace `std`.

Referring to library names with this notation can be cumbersome. Fortunately, there are easier ways to use namespace members. The safest way is a **`using` declaration**. § 18.2.2 (p. 793) covers another way to use names from a namespace.

A `using` declaration lets us use a name from a namespace without qualifying the name with a `namespace_name::` prefix. A `using` declaration has the form

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

Once the `using` declaration has been made, we can access *name* directly:

```
#include <iostream>
// using declaration; when we use the name cin, we get the one from the namespace std
using std::cin;
int main()
{
    int i;
    cin >> i;        // ok: cin is a synonym for std::cin
    cout << i;       // error: no using declaration; we must use the full name
    std::cout << i; // ok: explicitly use cout from namepsace std
    return 0;
}
```

A Separate `using` Declaration Is Required for Each Name

Each `using` declaration introduces a single namespace member. This behavior lets us be specific about which names we're using. As an example, we'll rewrite the program from § 1.2 (p. 6) with `using` declarations for the library names it uses:

```
#include <iostream>
// using declarations for names from the standard library
using std::cin;
using std::cout; using std::endl;
int main()
{
    cout << "Enter two numbers:" << endl;
    int v1, v2;
    cin >> v1 >> v2;
    cout << "The sum of " << v1 << " and " << v2
         << " is " << v1 + v2 << endl;
    return 0;
}
```

The `using` declarations for `cin`, `cout`, and `endl` mean that we can use those names without the `std::` prefix. Recall that C++ programs are free-form, so we can put each `using` declaration on its own line or combine several onto a single line. The important part is that there must be a `using` declaration for each name we use, and each declaration must end in a semicolon.

Headers Should Not Include `using` Declarations

Code inside headers (§ 2.6.3, p. 76) ordinarily should not use `using` declarations. The reason is that the contents of a header are copied into the including program's text. If a header has a `using` declaration, then every program that includes that header gets that same `using` declaration. As a result, a program that didn't intend to use the specified library name might encounter unexpected name conflicts.

A Note to the Reader

From this point on, our examples will assume that `using` declarations have been made for the names we use from the standard library. Thus, we will refer to `cin`, not `std::cin`, in the text and in code examples.

Moreover, to keep the code examples short, we won't show the `using` declarations, nor will we show the necessary `#include` directives. Table A.1 (p. 866) in Appendix A lists the names and corresponding headers for standard library names we use in this Primer.

Readers should be aware that they must add appropriate `#include` and `using` declarations to our examples before compiling them.

Exercises Section 3.1

Exercise 3.1: Rewrite the exercises from § 1.4.1 (p. 13) and § 2.6.2 (p. 76) with appropriate `using` declarations.

3.2 Library `string` Type

A **`string`** is a variable-length sequence of characters. To use the `string` type, we must include the `string` header. Because it is part of the library, `string` is defined in the `std` namespace. Our examples assume the following code:

```
#include <string>
using std::string;
```

This section describes the most common `string` operations; § 9.5 (p. 360) will cover additional operations.

In addition to specifying the operations that the library types provide, the standard also imposes efficiency requirements on implementors. As a result, library types are efficient enough for general use.

3.2.1 Defining and Initializing `strings`

Each class defines how objects of its type can be initialized. A class may define many different ways to initialize objects of its type. Each way must be distinguished from the others either by the number of initializers that we supply, or by the types of those initializers. Table 3.1 lists the most common ways to initialize `string`s. Some examples:

```
string s1;              // default initialization; s1 is the empty string
string s2 = s1;         // s2 is a copy of s1
string s3 = "hiya";     // s3 is a copy of the string literal
string s4(10, 'c');     // s4 is cccccccccc
```

We can default initialize a `string` (§ 2.2.1, p. 44), which creates an empty `string`; that is, a `string` with no characters. When we supply a string literal (§ 2.1.3, p. 39), the characters from that literal—up to but not including the null character at the end of the literal—are copied into the newly created `string`. When we supply a count and a character, the `string` contains that many copies of the given character.

Direct and Copy Forms of Initialization

In § 2.2.1 (p. 43) we saw that C++ has several different forms of initialization. Using `string`s, we can start to understand how these forms differ from one another. When we initialize a variable using =, we are asking the compiler to **copy initialize** the object by copying the initializer on the right-hand side into the object being created. Otherwise, when we omit the =, we use **direct initialization**.

When we have a single initializer, we can use either the direct or copy form of initialization. When we initialize a variable from more than one value, such as in the initialization of `s4` above, we must use the direct form of initialization:

```
string s5 = "hiya";   // copy initialization
string s6("hiya");    // direct initialization
string s7(10, 'c');   // direct initialization; s7 is cccccccccc
```

When we want to use several values, we can indirectly use the copy form of initialization by explicitly creating a (temporary) object to copy:

```
string s8 = string(10, 'c'); // copy initialization; s8 is ccccccccc
```

The initializer of `s8`—`string(10, 'c')`—creates a `string` of the given size and character value and then copies that value into `s8`. It is as if we had written

```
string temp(10, 'c'); // temp is ccccccccc
string s8 = temp;     // copy temp into s8
```

Although the code used to initialize `s8` is legal, it is less readable and offers no compensating advantage over the way we initialized `s7`.

Table 3.1: Ways to Initialize a `string`

`string s1`	Default initialization; `s1` is the empty string.
`string s2(s1)`	`s2` is a copy of `s1`.
`string s2 = s1`	Equivalent to `s2(s1)`, `s2` is a copy of `s1`.
`string s3("value")`	`s3` is a copy of the string literal, not including the null.
`string s3 = "value"`	Equivalent to `s3("value")`, `s3` is a copy of the string literal.
`string s4(n, 'c')`	Initialize `s4` with `n` copies of the character `'c'`.

3.2.2 Operations on `strings`

Along with defining how objects are created and initialized, a class also defines the operations that objects of the class type can perform. A class can define operations that are called by name, such as the `isbn` function of our `Sales_item` class (§ 1.5.2, p. 23). A class also can define what various operator symbols, such as `<<` or `+`, mean when applied to objects of the class' type. Table 3.2 (overleaf) lists the most common `string` operations.

Reading and Writing `strings`

As we saw in Chapter 1, we use the `iostream` library to read and write values of built-in types such as `int`, `double`, and so on. We use the same IO operators to read and write `string`s:

```
// Note: #include and using declarations must be added to compile this code
int main()
{
    string s;           // empty string
    cin >> s;           // read a whitespace-separated string into s
    cout << s << endl; // write s to the output
    return 0;
}
```

Table 3.2: `string` Operations

`os << s`	Writes `s` onto output stream `os`. Returns `os`.
`is >> s`	Reads whitespace-separated string from `is` into `s`. Returns `is`.
`getline(is, s)`	Reads a line of input from `is` into `s`. Returns `is`.
`s.empty()`	Returns `true` if `s` is empty; otherwise returns `false`.
`s.size()`	Returns the number of characters in `s`.
`s[n]`	Returns a reference to the `char` at position `n` in `s`; positions start at 0.
`s1 + s2`	Returns a `string` that is the concatenation of `s1` and `s2`.
`s1 = s2`	Replaces characters in `s1` with a copy of `s2`.
`s1 == s2`	The `strings` `s1` and `s2` are equal if they contain the same characters.
`s1 != s2`	Equality is case-sensitive.
`<, <=, >, >=`	Comparisons are case-sensitive and use dictionary ordering.

This program begins by defining an empty `string` named `s`. The next line reads the standard input, storing what is read in `s`. The `string` input operator reads and discards any leading whitespace (e.g., spaces, newlines, tabs). It then reads characters until the next whitespace character is encountered.

So, if the input to this program is **`Hello World!`** (note leading and trailing spaces), then the output will be **`Hello`** with no extra spaces.

Like the input and output operations on the built-in types, the `string` operators return their left-hand operand as their result. Thus, we can chain together multiple reads or writes:

```
string s1, s2;
cin >> s1 >> s2; // read first input into s1, second into s2
cout << s1 << s2 << endl; // write both strings
```

If we give this version of the program the same input, **`Hello World!`**, our output would be "**`HelloWorld!`**"

Reading an Unknown Number of `strings`

In § 1.4.3 (p. 14) we wrote a program that read an unknown number of `int` values. We can write a similar program that reads `strings` instead:

```
int main()
{
    string word;
    while (cin >> word)          // read until end-of-file
        cout << word << endl; // write each word followed by a new line
    return 0;
}
```

In this program, we read into a `string`, not an `int`. Otherwise, the `while` condition executes similarly to the one in our previous program. The condition tests the stream after the read completes. If the stream is valid—it hasn't hit end-of-file

or encountered an invalid input—then the body of the while is executed. The body prints the value we read on the standard output. Once we hit end-of-file (or invalid input), we fall out of the while.

Using getline to Read an Entire Line

Sometimes we do not want to ignore the whitespace in our input. In such cases, we can use the **getline** function instead of the >> operator. The getline function takes an input stream and a string. This function reads the given stream up to and including the first newline and stores what it read—*not including* the newline—in its string argument. After getline sees a newline, even if it is the first character in the input, it stops reading and returns. If the first character in the input is a newline, then the resulting string is the empty string.

Like the input operator, getline returns its istream argument. As a result, we can use getline as a condition just as we can use the input operator as a condition (§ 1.4.3, p. 14). For example, we can rewrite the previous program that wrote one word per line to write a line at a time instead:

```
int main()
{
    string line;
    // read input a line at a time until end-of-file
    while (getline(cin, line))
        cout << line << endl;
    return 0;
}
```

Because line does not contain a newline, we must write our own. As usual, we use endl to end the current line and flush the buffer.

The newline that causes getline to return is discarded; the newline is *not* stored in the string.

The string empty and size Operations

The **empty** function does what one would expect: It returns a bool (§ 2.1, p. 32) indicating whether the string is empty. Like the isbn member of Sales_item (§ 1.5.2, p. 23), empty is a member function of string. To call this function, we use the dot operator to specify the object on which we want to run the empty function.

We can revise the previous program to only print lines that are not empty:

```
// read input a line at a time and discard blank lines
while (getline(cin, line))
    if (!line.empty())
        cout << line << endl;
```

The condition uses the logical NOT operator (the **! operator**). This operator returns the inverse of the bool value of its operand. In this case, the condition is true if str is not empty.

The **size** member returns the length of a string (i.e., the number of characters in it). We can use size to print only lines longer than 80 characters:

```
string line;
// read input a line at a time and print lines that are longer than 80 characters
while (getline(cin, line))
    if (line.size() > 80)
        cout << line << endl;
```

The string::size_type Type

It might be logical to expect that size returns an int or, thinking back to § 2.1.1 (p. 34), an unsigned. Instead, size returns a string::size_type value. This type requires a bit of explanation.

The string class—and most other library types—defines several companion types. These companion types make it possible to use the library types in a machine-independent manner. The type **size_type** is one of these companion types. To use the size_type defined by string, we use the scope operator to say that the name size_type is defined in the string class.

Although we don't know the precise type of string::size_type, we do know that it is an unsigned type (§ 2.1.1, p. 32) big enough to hold the size of any string. Any variable used to store the result from the string size operation should be of type string::size_type.

C++ 11

Admittedly, it can be tedious to type string::size_type. Under the new standard, we can ask the compiler to provide the appropriate type by using auto or decltype (§ 2.5.2, p. 68):

```
auto len = line.size(); // len has type string::size_type
```

Because size returns an unsigned type, it is essential to remember that expressions that mix signed and unsigned data can have surprising results (§ 2.1.2, p. 36). For example, if n is an int that holds a negative value, then s.size() < n will almost surely evaluate as true. It yields true because the negative value in n will convert to a large unsigned value.

You can avoid problems due to conversion between unsigned and int by not using ints in expressions that use size().

Comparing strings

The string class defines several operators that compare strings. These operators work by comparing the characters of the strings. The comparisons are case-sensitive—upper- and lowercase versions of a letter are different characters.

The equality operators (== and !=) test whether two strings are equal or unequal, respectively. Two strings are equal if they are the same length and contain the same characters. The relational operators <, <=, >, >= test whether one string is less than, less than or equal to, greater than, or greater than or equal to another. These operators use the same strategy as a (case-sensitive) dictionary:

1. If two strings have different lengths and if every character in the shorter string is equal to the corresponding character of the longer string, then the shorter string is less than the longer one.
2. If any characters at corresponding positions in the two strings differ, then the result of the string comparison is the result of comparing the first character at which the strings differ.

As an example, consider the following strings:

```
string str = "Hello";
string phrase = "Hello World";
string slang  = "Hiya";
```

Using rule 1, we see that str is less than phrase. By applying rule 2, we see that slang is greater than both str and phrase.

Assignment for strings

In general, the library types strive to make it as easy to use a library type as it is to use a built-in type. To this end, most of the library types support assignment. In the case of strings, we can assign one string object to another:

```
string st1(10, 'c'), st2; // st1 is cccccccccc; st2 is an empty string
st1 = st2; // assignment: replace contents of st1 with a copy of st2
           // both st1 and st2 are now the empty string
```

Adding Two strings

Adding two strings yields a new string that is the concatenation of the left-hand followed by the right-hand operand. That is, when we use the plus operator (+) on strings, the result is a new string whose characters are a copy of those in the left-hand operand followed by those from the right-hand operand. The compound assignment operator (+=) (§ 1.4.1, p. 12) appends the right-hand operand to the left-hand string:

```
string s1  = "hello, ", s2 = "world\n";
string s3 = s1 + s2;   // s3 is hello, world\n
s1 += s2;   // equivalent to s1 = s1 + s2
```

Adding Literals and strings

As we saw in § 2.1.2 (p. 35), we can use one type where another type is expected if there is a conversion from the given type to the expected type. The string library lets us convert both character literals and character string literals (§ 2.1.3, p. 39) to strings. Because we can use these literals where a string is expected, we can rewrite the previous program as follows:

```
string s1 = "hello", s2 = "world";  // no punctuation in s1 or s2
string s3 = s1 + ", " + s2 + '\n';
```

When we mix strings and string or character literals, at least one operand to each + operator must be of string type:

```
string s4 = s1 + ", ";            // ok: adding a string and a literal
string s5 = "hello" + ", ";       // error: no string operand
string s6 = s1 + ", " + "world"; // ok: each + has a string operand
string s7 = "hello" + ", " + s2; // error: can't add string literals
```

The initializations of `s4` and `s5` involve only a single operation each, so it is easy to see whether the initialization is legal. The initialization of `s6` may appear surprising, but it works in much the same way as when we chain together input or output expressions (§ 1.2, p. 7). This initialization groups as

```
string s6 = (s1 + ", ") + "world";
```

The subexpression `s1 + ", "` returns a `string`, which forms the left-hand operand of the second + operator. It is as if we had written

```
string tmp = s1 + ", ";   // ok: + has a string operand
s6 = tmp + "world";       // ok: + has a string operand
```

On the other hand, the initialization of `s7` is illegal, which we can see if we parenthesize the expression:

```
string s7 = ("hello" + ", ") + s2;   // error: can't add string literals
```

Now it should be easy to see that the first subexpression adds two string literals. There is no way to do so, and so the statement is in error.

For historical reasons, and for compatibility with C, string literals are *not* standard library `string`s. It is important to remember that these types differ when you use string literals and library `string`s.

Exercises Section 3.2.2

Exercise 3.2: Write a program to read the standard input a line at a time. Modify your program to read a word at a time.

Exercise 3.3: Explain how whitespace characters are handled in the `string` input operator and in the `getline` function.

Exercise 3.4: Write a program to read two `string`s and report whether the `string`s are equal. If not, report which of the two is larger. Now, change the program to report whether the `string`s have the same length, and if not, report which is longer.

Exercise 3.5: Write a program to read `string`s from the standard input, concatenating what is read into one large `string`. Print the concatenated `string`. Next, change the program to separate adjacent input `string`s by a space.

3.2.3 Dealing with the Characters in a `string`

Often we need to deal with the individual characters in a `string`. We might want to check to see whether a `string` contains any whitespace, or to change the characters to lowercase, or to see whether a given character is present, and so on.

One part of this kind of processing involves how we gain access to the characters themselves. Sometimes we need to process every character. Other times we need to process only a specific character, or we can stop processing once some condition is met. It turns out that the best way to deal with these cases involves different language and library facilities.

The other part of processing characters is knowing and/or changing the characteristics of a character. This part of the job is handled by a set of library functions, described in Table 3.3 (overleaf). These functions are defined in the `cctype` header.

ADVICE: USE THE C++ VERSIONS OF C LIBRARY HEADERS

In addition to facilities defined specifically for C++, the C++ library incorporates the C library. Headers in C have names of the form *name*`.h`. The C++ versions of these headers are named `c`*name*—they remove the `.h` suffix and precede the *name* with the letter `c`. The `c` indicates that the header is part of the C library.

Hence, `cctype` has the same contents as `ctype.h`, but in a form that is appropriate for C++ programs. In particular, the names defined in the `c`*name* headers are defined inside the `std` namespace, whereas those defined in the `.h` versions are not.

Ordinarily, C++ programs should use the `c`*name* versions of headers and not the *name*`.h` versions. That way names from the standard library are consistently found in the `std` namespace. Using the `.h` headers puts the burden on the programmer to remember which library names are inherited from C and which are unique to C++.

Processing Every Character? Use Range-Based `for`

If we want to do something to every character in a `string`, by far the best approach is to use a statement introduced by the new standard: the **range `for`** statement. This statement iterates through the elements in a given sequence and performs some operation on each value in that sequence. The syntactic form is

C++ 11

```
for (declaration : expression)
    statement
```

where *expression* is an object of a type that represents a sequence, and *declaration* defines the variable that we'll use to access the underlying elements in the sequence. On each iteration, the variable in *declaration* is initialized from the value of the next element in *expression*.

A `string` represents a sequence of characters, so we can use a `string` as the *expression* in a range `for`. As a simple example, we can use a range `for` to print each character from a `string` on its own line of output:

```
string str("some string");
// print the characters in str one character to a line
for (auto c : str)       // for every char in str
    cout << c << endl;   // print the current character followed by a newline
```

The `for` loop associates the variable `c` with `str`. We define the loop control variable the same way we do any other variable. In this case, we use `auto` (§ 2.5.2,

p. 68) to let the compiler determine the type of `c`, which in this case will be `char`. On each iteration, the next character in `str` will be copied into `c`. Thus, we can read this loop as saying, "For every character `c` in the `string` `str`," do something. The "something" in this case is to print the character followed by a newline.

As a somewhat more complicated example, we'll use a range `for` and the `ispunct` function to count the number of punctuation characters in a `string`:

```
string s("Hello World!!!");
// punct_cnt has the same type that s.size returns; see § 2.5.3 (p. 70)
decltype(s.size()) punct_cnt = 0;
// count the number of punctuation characters in s
for (auto c : s)          // for every char in s
    if (ispunct(c))       // if the character is punctuation
        ++punct_cnt;      // increment the punctuation counter
cout << punct_cnt
     << " punctuation characters in " << s << endl;
```

The output of this program is

```
3 punctuation characters in Hello World!!!
```

Here we use `decltype` (§ 2.5.3, p. 70) to declare our counter, `punct_cnt`. Its type is the type returned by calling `s.size`, which is `string::size_type`. We use a range `for` to process each character in the `string`. This time we check whether each character is punctuation. If so, we use the increment operator (§ 1.4.1, p. 12) to add `1` to the counter. When the range `for` completes, we print the result.

Table 3.3: `cctype` Functions

`isalnum(c)`	`true` if `c` is a letter or a digit.
`isalpha(c)`	`true` if `c` is a letter.
`iscntrl(c)`	`true` if `c` is a control character.
`isdigit(c)`	`true` if `c` is a digit.
`isgraph(c)`	`true` if `c` is not a space but is printable.
`islower(c)`	`true` if `c` is a lowercase letter.
`isprint(c)`	`true` if `c` is a printable character (i.e., a space or a character that has a visible representation).
`ispunct(c)`	`true` if `c` is a punctuation character (i.e., a character that is not a control character, a digit, a letter, or a printable whitespace).
`isspace(c)`	`true` if `c` is whitespace (i.e., a space, tab, vertical tab, return, newline, or formfeed).
`isupper(c)`	`true` if `c` is an uppercase letter.
`isxdigit(c)`	`true` if `c` is a hexadecimal digit.
`tolower(c)`	If `c` is an uppercase letter, returns its lowercase equivalent; otherwise returns `c` unchanged.
`toupper(c)`	If `c` is a lowercase letter, returns its uppercase equivalent; otherwise returns `c` unchanged.

Using a Range for to Change the Characters in a string

If we want to change the value of the characters in a string, we must define the loop variable as a reference type (§ 2.3.1, p. 50). Remember that a reference is just another name for a given object. When we use a reference as our control variable, that variable is bound to each element in the sequence in turn. Using the reference, we can change the character to which the reference is bound.

Suppose that instead of counting punctuation, we wanted to convert a string to all uppercase letters. To do so we can use the library toupper function, which takes a character and returns the uppercase version of that character. To convert the whole string we need to call toupper on each character and put the result back in that character:

```
string s("Hello World!!!");
// convert s to uppercase
for (auto &c : s)    // for every char in s (note: c is a reference)
    c = toupper(c); //  c is a reference, so the assignment changes the char in s
cout << s << endl;
```

The output of this code is

```
HELLO WORLD!!!
```

On each iteration, c refers to the next character in s. When we assign to c, we are changing the underlying character in s. So, when we execute

```
c = toupper(c); //  c is a reference, so the assignment changes the char in s
```

we're changing the value of the character to which c is bound. When this loop completes, all the characters in str will be uppercase.

Processing Only Some Characters?

A range for works well when we need to process every character. However, sometimes we need to access only a single character or to access characters until some condition is reached. For example, we might want to capitalize only the first character or only the first word in a string.

There are two ways to access individual characters in a string: We can use a subscript or an iterator. We'll have more to say about iterators in § 3.4 (p. 106) and in Chapter 9.

The subscript operator (the **[] operator**) takes a string::size_type (§ 3.2.2, p. 88) value that denotes the position of the character we want to access. The operator returns a reference to the character at the given position.

Subscripts for strings start at zero; if s is a string with at least two characters, then s[0] is the first character, s[1] is the second, and the last character is in s[s.size() - 1].

The values we use to subscript a string must be >= 0 and < size().
The result of using an index outside this range is undefined.
By implication, subscripting an empty string is undefined.

The value in the subscript is referred to as "a subscript" or "an **index**." The index we supply can be any expression that yields an integral value. However, if our index has a signed type, its value will be converted to the unsigned type that `string::size_type` represents (§ 2.1.2, p. 36).

The following example uses the subscript operator to print the first character in a `string`:

```
if (!s.empty())              // make sure there's a character to print
    cout << s[0] << endl;    // print the first character in s
```

Before accessing the character, we check that `s` is not empty. Any time we use a subscript, we must ensure that there is a value at the given location. If `s` is empty, then `s[0]` is undefined.

So long as the `string` is not `const` (§ 2.4, p. 59), we can assign a new value to the character that the subscript operator returns. For example, we can capitalize the first letter as follows:

```
string s("some string");
if (!s.empty())              // make sure there's a character in s[0]
    s[0] = toupper(s[0]);    // assign a new value to the first character in s
```

The output of this program is

```
Some string
```

Using a Subscript for Iteration

As a another example, we'll change the first word in `s` to all uppercase:

```
// process characters in s until we run out of characters or we hit a whitespace
for (decltype(s.size()) index = 0;
     index != s.size() && !isspace(s[index]); ++index)
        s[index] = toupper(s[index]); // capitalize the current character
```

This program generates

```
SOME string
```

Our `for` loop (§ 1.4.2, p. 13) uses `index` to subscript `s`. We use `decltype` to give `index` the appropriate type. We initialize `index` to `0` so that the first iteration will start on the first character in `s`. On each iteration we increment `index` to look at the next character in `s`. In the body of the loop we capitalize the current letter.

The new part in this loop is the condition in the `for`. That condition uses the logical AND operator (the **`&&` operator**). This operator yields `true` if both operands are `true` and `false` otherwise. The important part about this operator is that we are guaranteed that it evaluates its right-hand operand *only* if the left-hand operand is `true`. In this case, we are guaranteed that we will not subscript `s` unless we know that `index` is in range. That is, `s[index]` is executed only if `index` is not equal to `s.size()`. Because `index` is never incremented beyond the value of `s.size()`, we know that `index` will always be less than `s.size()`.

CAUTION: SUBSCRIPTS ARE UNCHECKED

When we use a subscript, we must ensure that the subscript is in range. That is, the subscript must be >= 0 and < the size() of the string. One way to simplify code that uses subscripts is *always* to use a variable of type string::size_type as the subscript. Because that type is unsigned, we ensure that the subscript cannot be less than zero. When we use a size_type value as the subscript, we need to check only that our subscript is less than value returned by size().

The library is not required to check the value of an subscript. The result of using an out-of-range subscript is undefined.

Using a Subscript for Random Access

In the previous example we advanced our subscript one position at a time to capitalize each character in sequence. We can also calculate an subscript and directly fetch the indicated character. There is no need to access characters in sequence.

As an example, let's assume we have a number between 0 and 15 and we want to generate the hexadecimal representation of that number. We can do so using a `string` that is initialized to hold the 16 hexadecimal "digits":

```
const string hexdigits = "0123456789ABCDEF";  // possible hex digits
cout << "Enter a series of numbers between 0 and 15"
     << " separated by spaces.  Hit ENTER when finished: "
     << endl;
string result;         // will hold the resulting hexify'd string
string::size_type n;   // hold numbers from the input
while (cin >> n)
    if (n < hexdigits.size())     // ignore invalid input
        result += hexdigits[n];   // fetch the indicated hex digit
cout << "Your hex number is: " << result << endl;
```

If we give this program the input

```
12 0 5 15 8 15
```

the output will be

```
Your hex number is: C05F8F
```

We start by initializing `hexdigits` to hold the hexadecimal digits `0` through `F`. We make that `string` `const` (§ 2.4, p. 59) because we do not want these values to change. Inside the loop we use the input value `n` to subscript `hexdigits`. The value of `hexdigits[n]` is the `char` that appears at position `n` in `hexdigits`. For example, if `n` is `15`, then the result is `F`; if it's `12`, the result is `C`; and so on. We append that digit to `result`, which we print once we have read all the input.

Whenever we use a subscript, we should think about how we know that it is in range. In this program, our subscript, `n`, is a `string::size_type`, which as we know is an unsigned type. As a result, we know that `n` is guaranteed to be greater than or equal to `0`. Before we use `n` to subscript `hexdigits`, we verify that it is less than the `size` of `hexdigits`.

Exercises Section 3.2.3

Exercise 3.6: Use a range `for` to change all the characters in a `string` to `X`.

Exercise 3.7: What would happen if you define the loop control variable in the previous exercise as type `char`? Predict the results and then change your program to use a `char` to see if you were right.

Exercise 3.8: Rewrite the program in the first exercise, first using a `while` and again using a traditional `for` loop. Which of the three approaches do you prefer and why?

Exercise 3.9: What does the following program do? Is it valid? If not, why not?

```
string s;
cout << s[0] << endl;
```

Exercise 3.10: Write a program that reads a string of characters including punctuation and writes what was read but with the punctuation removed.

Exercise 3.11: Is the following range `for` legal? If so, what is the type of `c`?

```
const string s = "Keep out!";
for (auto &c : s) { /* ... */ }
```

3.3 Library `vector` Type

A **`vector`** is a collection of objects, all of which have the same type. Every object in the collection has an associated index, which gives access to that object. A `vector` is often referred to as a **container** because it "contains" other objects. We'll have much more to say about containers in Part II.

To use a `vector`, we must include the appropriate header. In our examples, we also assume that an appropriate `using` declaration is made:

```
#include <vector>
using std::vector;
```

A `vector` is a **class template**. C++ has both class and function templates. Writing a template requires a fairly deep understanding of C++. Indeed, we won't see how to create our own templates until Chapter 16! Fortunately, we can use templates without knowing how to write them.

Templates are not themselves functions or classes. Instead, they can be thought of as instructions to the compiler for generating classes or functions. The process that the compiler uses to create classes or functions from templates is called **instantiation**. When we use a template, we specify what kind of class or function we want the compiler to instantiate.

For a class template, we specify which class to instantiate by supplying additional information, the nature of which depends on the template. How we specify the information is always the same: We supply it inside a pair of angle brackets following the template's name.

In the case of `vector`, the additional information we supply is the type of the objects the `vector` will hold:

```
vector<int> ivec;              // ivec holds objects of type int
vector<Sales_item> Sales_vec; // holds Sales_items
vector<vector<string>> file;  // vector whose elements are vectors
```

In this example, the compiler generates three distinct types from the `vector` template: `vector<int>`, `vector<Sales_item>`, and `vector<vector<string>>`.

`vector` is a template, not a type. Types generated from `vector` must include the element type, for example, `vector<int>`.

We can define `vectors` to hold objects of most any type. Because references are not objects (§ 2.3.1, p. 50), we cannot have a `vector` of references. However, we can have `vectors` of most other (nonreference) built-in types and most class types. In particular, we can have `vectors` whose elements are themselves `vectors`.

It is worth noting that earlier versions of C++ used a slightly different syntax to define a `vector` whose elements are themselves `vectors` (or another template type). In the past, we had to supply a space between the closing angle bracket of the outer `vector` and its element type—`vector<vector<int> >` rather than `vector<vector<int>>`.

C++ 11

Some compilers may require the old-style declarations for a `vector` of `vectors`, for example, `vector<vector<int> >`.

3.3.1 Defining and Initializing `vectors`

As with any class type, the `vector` template controls how we define and initialize `vectors`. Table 3.4 (p. 99) lists the most common ways to define `vectors`.

We can default initialize a `vector` (§ 2.2.1, p. 44), which creates an empty `vector` of the specified type:

```
vector<string> svec; // default initialization; svec has no elements
```

It might seem that an empty `vector` would be of little use. However, as we'll see shortly, we can (efficiently) add elements to a `vector` at run time. Indeed, the most common way of using `vectors` is to define an initially empty `vector` to which elements are added as their values become known at run time.

We can also supply initial value(s) for the element(s) when we define a `vector`. For example, we can copy elements from another `vector`. When we copy a `vector`, each element in the new `vector` is a copy of the corresponding element in the original `vector`. The two `vectors` must be the same type:

```
vector<int> ivec;              // initially empty
// give ivec some values
vector<int> ivec2(ivec);       // copy elements of ivec into ivec2
vector<int> ivec3 = ivec;      // copy elements of ivec into ivec3
vector<string> svec(ivec2);    // error: svec holds strings, not ints
```

List Initializing a `vector`

C++ 11

Another way to provide element values, is that under the new standard, we can list initialize (§ 2.2.1, p. 43) a `vector` from a list of zero or more initial element values enclosed in curly braces:

```
vector<string> articles = {"a", "an", "the"};
```

The resulting `vector` has three elements; the first holds the `string` `"a"`, the second holds `"an"`, and the last is `"the"`.

As we've seen, C++ provides several forms of initialization (§ 2.2.1, p. 43). In many, but not all, cases we can use these forms of initialization interchangably. So far, we have seen two examples where the form of initialization matters: when we use the copy initialization form (i.e., when we use =) (§ 3.2.1, p. 84), we can supply only a single initializer; and when we supply an in-class initializer (§ 2.6.1, p. 73), we must either use copy initialization or use curly braces. A third restriction is that we can supply a list of element values only by using list initialization in which the initializers are enclosed in curly braces. We cannot supply a list of initializers using parentheses:

```
vector<string> v1{"a", "an", "the"};  // list initialization
vector<string> v2("a", "an", "the");  // error
```

Creating a Specified Number of Elements

We can also initialize a `vector` from a count and an element value. The count determines how many elements the `vector` will have; the value provides the initial value for each of those elements:

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

Value Initialization

We can usually omit the value and supply only a size. In this case the library creates a **value-initialized** element initializer for us. This library-generated value is used to initialize each element in the container. The value of the element initializer depends on the type of the elements stored in the `vector`.

If the `vector` holds elements of a built-in type, such as `int`, then the element initializer has a value of 0. If the elements are of a class type, such as `string`, then the element initializer is itself default initialized:

```
vector<int> ivec(10);     // ten elements, each initialized to 0
vector<string> svec(10); // ten elements, each an empty string
```

There are two restrictions on this form of initialization: The first restriction is that some classes require that we always supply an explicit initializer (§ 2.2.1, p. 44). If our `vector` holds objects of a type that we cannot default initialize, then we must supply an initial element value; it is not possible to create `vector`s of such types by supplying only a size.

The second restriction is that when we supply an element count without also supplying an initial value, we must use the direct form of initialization:

```
vector<int> vi = 10;      // error: must use direct initialization to supply a size
```

Here we are using 10 to instruct `vector` how to create the `vector`—we want a `vector` with ten value-initialized elements. We are not "copying" 10 into the `vector`. Hence, we cannot use the copy form of initialization. We'll see more about how this restriction works in § 7.5.4 (p. 296).

Table 3.4: Ways to Initialize a `vector`

`vector<T> v1`	`vector` that holds objects of type `T`. Default initialization; `v1` is empty.
`vector<T> v2(v1)`	`v2` has a copy of each element in `v1`.
`vector<T> v2 = v1`	Equivalent to `v2(v1)`, `v2` is a copy of the elements in `v1`.
`vector<T> v3(n, val)`	`v3` has `n` elements with value `val`.
`vector<T> v4(n)`	`v4` has `n` copies of a value-initialized object.
`vector<T> v5{a,b,c...}`	`v5` has as many elements as there are initializers; elements are initialized by corresponding initializers.
`vector<T> v5 = {a,b,c...}`	Equivalent to `v5{a,b,c...}`.

List Initializer or Element Count?

In a few cases, what initialization means depends upon whether we use curly braces or parentheses to pass the initializer(s). For example, when we initialize a `vector<int>` from a single `int` value, that value might represent the `vector`'s size or it might be an element value. Similarly, if we supply exactly two `int` values, those values could be a size and an initial value, or they could be values for a two-element `vector`. We specify which meaning we intend by whether we use curly braces or parentheses:

```
vector<int> v1(10);      // v1 has ten elements with value 0
vector<int> v2{10};      // v2 has one element with value 10
vector<int> v3(10, 1);   // v3 has ten elements with value 1
vector<int> v4{10, 1};   // v4 has two elements with values 10 and 1
```

When we use parentheses, we are saying that the values we supply are to be used to *construct* the object. Thus, `v1` and `v3` use their initializers to determine the `vector`'s size, and its size and element values, respectively.

When we use curly braces, { . . . }, we're saying that, if possible, we want to *list initialize* the object. That is, if there is a way to use the values inside the curly braces as a list of element initializers, the class will do so. Only if it is not possible to list initialize the object will the other ways to initialize the object be considered. The values we supply when we initialize `v2` and `v4` can be used as element values. These objects are list initialized; the resulting `vectors` have one and two elements, respectively.

On the other hand, if we use braces and there is no way to use the initializers to list initialize the object, then those values will be used to construct the object. For

example, to list initialize a `vector` of `string`s, we must supply values that can be used as `string`s. In this case, there is no confusion about whether to list initialize the elements or construct a `vector` of the given size:

```
vector<string> v5{"hi"}; // list initialization: v5 has one element
vector<string> v6("hi"); // error: can't construct a vector from a string literal
vector<string> v7{10};        // v7 has ten default-initialized elements
vector<string> v8{10, "hi"}; // v8 has ten elements with value "hi"
```

Although we used braces on all but one of these definitions, only `v5` is list initialized. In order to list initialize the `vector`, the values inside braces must match the element type. We cannot use an `int` to initialize a `string`, so the initializers for `v7` and `v8` can't be element initializers. If list initialization isn't possible, the compiler looks for other ways to initialize the object from the given values.

Exercises Section 3.3.1

Exercise 3.12: Which, if any, of the following `vector` definitions are in error? For those that are legal, explain what the definition does. For those that are not legal, explain why they are illegal.

```
(a) vector<vector<int>> ivec;
(b) vector<string> svec = ivec;
(c) vector<string> svec(10, "null");
```

Exercise 3.13: How many elements are there in each of the following `vector`s? What are the values of the elements?

```
(a) vector<int> v1;           (b) vector<int> v2(10);
(c) vector<int> v3(10, 42);   (d) vector<int> v4{10};
(e) vector<int> v5{10, 42};   (f) vector<string> v6{10};
(g) vector<string> v7{10, "hi"};
```

3.3.2 Adding Elements to a `vector`

Directly initializing the elements of a `vector` is feasible only if we have a small number of known initial values, if we want to make a copy of another `vector`, or if we want to initialize all the elements to the same value. More commonly, when we create a `vector`, we don't know how many elements we'll need, or we don't know the value of those elements. Even if we do know all the values, if we have a large number of different initial element values, it can be cumbersome to specify them when we create the `vector`.

As one example, if we need a `vector` with values from 0 to 9, we can easily use list initialization. What if we wanted elements from 0 to 99 or 0 to 999? List initialization would be too unwieldy. In such cases, it is better to create an empty `vector` and use a `vector` member named **`push_back`** to add elements at run time. The `push_back` operation takes a value and "pushes" that value as a new last element onto the "back" of the `vector`. For example:

```
vector<int> v2;          // empty vector
for (int i = 0; i != 100; ++i)
    v2.push_back(i);     // append sequential integers to v2
// at end of loop v2 has 100 elements, values 0 . . . 99
```

Even though we know we ultimately will have 100 elements, we define `v2` as empty. Each iteration adds the next sequential integer as a new element in `v2`.

We use the same approach when we want to create a `vector` where we don't know until run time how many elements the `vector` should have. For example, we might read the input, storing the values we read in the `vector`:

```
// read words from the standard input and store them as elements in a vector
string word;
vector<string> text;         // empty vector
while (cin >> word) {
    text.push_back(word);    // append word to text
}
```

Again, we start with an initially empty `vector`. This time, we read and store an unknown number of values in `text`.

KEY CONCEPT: VECTORS GROW EFFICIENTLY

The standard requires that `vector` implementations can efficiently add elements at run time. Because `vectors` grow efficiently, it is often unnecessary—and can result in poorer performance—to define a `vector` of a specific size. The exception to this rule is if *all* the elements actually need the same value. If differing element values are needed, it is usually more efficient to define an empty `vector` and add elements as the values we need become known at run time. Moreover, as we'll see in § 9.4 (p. 355), `vector` offers capabilities to allow us to further enhance run-time performance when we add elements.

Starting with an empty `vector` and adding elements at run time is distinctly different from how we use built-in arrays in C and in most other languages. In particular, if you are accustomed to using C or Java, you might expect that it would be best to define the `vector` at its expected size. In fact, the contrary is usually the case.

Programming Implications of Adding Elements to a `vector`

The fact that we can easily and efficiently add elements to a `vector` greatly simplifies many programming tasks. However, this simplicity imposes a new obligation on our programs: We must ensure that any loops we write are correct even if the loop changes the size of the `vector`.

Other implications that follow from the dynamic nature of `vectors` will become clearer as we learn more about using them. However, there is one implication that is worth noting already: For reasons we'll explore in § 5.4.3 (p. 188), we cannot use a range `for` if the body of the loop adds elements to the `vector`.

The body of a range `for` must not change the size of the sequence over which it is iterating.

Exercises Section 3.3.2

Exercise 3.14: Write a program to read a sequence of `int`s from `cin` and store those values in a `vector`.

Exercise 3.15: Repeat the previous program but read `string`s this time.

3.3.3 Other `vector` Operations

In addition to `push_back`, `vector` provides only a few other operations, most of which are similar to the corresponding operations on `string`s. Table 3.5 lists the most important ones.

We access the elements of a `vector` the same way that we access the characters in a `string`: through their position in the `vector`. For example, we can use a range `for` (§ 3.2.3, p. 91) to process all the elements in a `vector`:

```
vector<int> v{1,2,3,4,5,6,7,8,9};
for (auto &i : v)      // for each element in v (note: i is a reference)
    i *= i;            // square the element value
for (auto i : v)       // for each element in v
    cout << i << " "; // print the element
cout << endl;
```

In the first loop, we define our control variable, `i`, as a reference so that we can use `i` to assign new values to the elements in `v`. We let `auto` deduce the type of `i`. This loop uses a new form of the compound assignment operator (§ 1.4.1, p. 12). As we've seen, `+=` adds the right-hand operand to the left and stores the result in the left-hand operand. The `*=` operator behaves similarly, except that it multiplies the left- and right-hand operands, storing the result in the left-hand one. The second range `for` prints each element.

The `empty` and `size` members behave as do the corresponding `string` members (§ 3.2.2, p. 87): `empty` returns a `bool` indicating whether the `vector` has any elements, and `size` returns the number of elements in the `vector`. The `size` member returns a value of the `size_type` defined by the corresponding `vector` type.

To use `size_type`, we must name the type in which it is defined. A `vector` type *always* includes its element type (§ 3.3, p. 97):

```
vector<int>::size_type // ok
vector::size_type      // error
```

The equality and relational operators have the same behavior as the corresponding `string` operations (§ 3.2.2, p. 88). Two `vector`s are equal if they have the same number of elements and if the corresponding elements all have the same value. The relational operators apply a dictionary ordering: If the `vector`s have differing sizes, but the elements that are in common are equal, then the `vector` with fewer elements is less than the one with more elements. If the elements have

differing values, then the relationship between the vectors is determined by the relationship between the first elements that differ.

We can compare two vectors only if we can compare the elements in those vectors. Some class types, such as string, define the meaning of the equality and relational operators. Others, such as our Sales_item class, do not. The only operations Sales_item supports are those listed in § 1.5.1 (p. 20). Those operations did not include the equality or relational operators. As a result, we cannot compare two vector<Sales_item> objects.

Table 3.5: vector Operations

v.empty()	Returns true if v is empty; otherwise returns false.
v.size()	Returns the number of elements in v.
v.push_back(t)	Adds an element with value t to end of v.
v[n]	Returns a reference to the element at position n in v.
v1 = v2	Replaces the elements in v1 with a copy of the elements in v2.
v1 = {a,b,c...}	Replaces the elements in v1 with a copy of the elements in the comma-separated list.
v1 == v2 v1 != v2	v1 and v2 are equal if they have the same number of elements and each element in v1 is equal to the corresponding element in v2.
<, <=, >, >=	Have their normal meanings using dictionary ordering.

Computing a vector Index

We can fetch a given element using the subscript operator (§ 3.2.3, p. 93). As with strings, subscripts for vector start at 0; the type of a subscript is the corresponding size_type; and—assuming the vector is nonconst—we can write to the element returned by the subscript operator. In addition, as we did in § 3.2.3 (p. 95), we can compute an index and directly fetch the element at that position.

As an example, let's assume that we have a collection of grades that range from 0 through 100. We'd like to count how many grades fall into various clusters of 10. Between zero and 100 there are 101 possible grades. These grades can be represented by 11 clusters: 10 clusters of 10 grades each plus one cluster for the perfect score of 100. The first cluster will count grades of 0 through 9, the second will count grades from 10 through 19, and so on. The final cluster counts how many scores of 100 were achieved.

Clustering the grades this way, if our input is

```
42 65 95 100 39 67 95 76 88 76 83 92 76 93
```

then the output should be

```
0 0 0 1 1 0 2 3 2 4 1
```

which indicates that there were no grades below 30, one grade in the 30s, one in the 40s, none in the 50s, two in the 60s, three in the 70s, two in the 80s, four in the 90s, and one grade of 100.

We'll use a `vector` with 11 elements to hold the counters for each cluster. We can determine the cluster index for a given grade by dividing that grade by 10. When we divide two integers, we get an integer in which the fractional part is truncated. For example, 42/10 is 4, 65/10 is 6 and 100/10 is 10. Once we've computed the cluster index, we can use it to subscript our `vector` and fetch the counter we want to increment:

```
// count the number of grades by clusters of ten: 0--9, 10--19, . .. 90--99, 100
vector<unsigned> scores(11, 0); // 11 buckets, all initially 0
unsigned grade;
while (cin >> grade) {       // read the grades
    if (grade <= 100)        // handle only valid grades
        ++scores[grade/10]; // increment the counter for the current cluster
}
```

We start by defining a `vector` to hold the cluster counts. In this case, we do want each element to have the same value, so we allocate all 11 elements, each of which is initialized to `0`. The `while` condition reads the grades. Inside the loop, we check that the grade we read has a valid value (i.e., that it is less than or equal to 100). Assuming the grade is valid, we increment the appropriate counter for `grade`.

The statement that does the increment is a good example of the kind of terse code characteristic of C++ programs. This expression

```
++scores[grade/10]; // increment the counter for the current cluster
```

is equivalent to

```
auto ind = grade/10; // get the bucket index
scores[ind] = scores[ind] + 1; // increment the count
```

We compute the bucket index by dividing `grade` by 10 and use the result of the division to index `scores`. Subscripting `scores` fetches the appropriate counter for this grade. We increment the value of that element to indicate the occurrence of a score in the given range.

As we've seen, when we use a subscript, we should think about how we know that the indices are in range (§ 3.2.3, p. 95). In this program, we verify that the input is a valid grade in the range between `0` and `100`. Thus, we know that the indices we can compute are between `0` and `10`. These indices are between `0` and `scores.size() - 1`.

Subscripting Does Not Add Elements

Programmers new to C++ sometimes think that subscripting a `vector` adds elements; it does not. The following code intends to add ten elements to `ivec`:

```
vector<int> ivec;   // empty vector
for (decltype(ivec.size()) ix = 0; ix != 10; ++ix)
    ivec[ix] = ix;  // disaster: ivec has no elements
```

However, it is in error: `ivec` is an empty `vector`; there are no elements to subscript! As we've seen, the right way to write this loop is to use `push_back`:

```
for (decltype(ivec.size()) ix = 0; ix != 10; ++ix)
    ivec.push_back(ix);   //  ok: adds a new element with value ix
```

The subscript operator on `vector` (and `string`) fetches an existing element; it does *not* add an element.

CAUTION: SUBSCRIPT ONLY ELEMENTS THAT ARE KNOWN TO EXIST!

It is crucially important to understand that we may use the subscript operator (the `[]` operator) to fetch only elements that actually exist. For example,

```
vector<int> ivec;       //  empty vector
cout << ivec[0];        //  error: ivec has no elements!

vector<int> ivec2(10); //  vector with ten elements
cout << ivec2[10];      //  error: ivec2 has elements 0 . . . 9
```

It is an error to subscript an element that doesn't exist, but it is an error that the compiler is unlikely to detect. Instead, the value we get at run time is undefined.

Attempting to subscript elements that do not exist is, unfortunately, an extremely common and pernicious programming error. So-called *buffer overflow* errors are the result of subscripting elements that don't exist. Such bugs are the most common cause of security problems in PC and other applications.

A good way to ensure that subscripts are in range is to avoid subscripting altogether by using a range `for` whenever possible.

EXERCISES SECTION 3.3.3

Exercise 3.16: Write a program to print the size and contents of the `vectors` from exercise 3.13. Check whether your answers to that exercise were correct. If not, restudy § 3.3.1 (p. 97) until you understand why you were wrong.

Exercise 3.17: Read a sequence of words from `cin` and store the values a `vector`. After you've read all the words, process the `vector` and change each word to uppercase. Print the transformed elements, eight words to a line.

Exercise 3.18: Is the following program legal? If not, how might you fix it?

```
vector<int> ivec;
ivec[0] = 42;
```

Exercise 3.19: List three ways to define a `vector` and give it ten elements, each with the value 42. Indicate whether there is a preferred way to do so and why.

Exercise 3.20: Read a set of integers into a `vector`. Print the sum of each pair of adjacent elements. Change your program so that it prints the sum of the first and last elements, followed by the sum of the second and second-to-last, and so on.

3.4 Introducing Iterators

Although we can use subscripts to access the characters of a `string` or the elements in a `vector`, there is a more general mechanism—known as **iterators**—that we can use for the same purpose. As we'll see in Part II, in addition to `vector`, the library defines several other kinds of containers. All of the library containers have iterators, but only a few of them support the subscript operator. Technically speaking, a `string` is not a container type, but `string` supports many of the container operations. As we've seen `string`, like `vector` has a subscript operator. Like `vectors`, `strings` also have iterators.

Like pointers (§ 2.3.2, p. 52), iterators give us indirect access to an object. In the case of an iterator, that object is an element in a container or a character in a `string`. We can use an iterator to fetch an element and iterators have operations to move from one element to another. As with pointers, an iterator may be valid or invalid. A valid iterator either denotes an element or denotes a position one past the last element in a container. All other iterator values are invalid.

3.4.1 Using Iterators

Unlike pointers, we do not use the address-of operator to obtain an iterator. Instead, types that have iterators have members that return iterators. In particular, these types have members named **`begin`** and **`end`**. The `begin` member returns an iterator that denotes the first element (or first character), if there is one:

```
// the compiler determines the type of b and e; see § 2.5.2 (p. 68)
// b denotes the first element and e denotes one past the last element in v
auto b = v.begin(), e = v.end(); // b and e have the same type
```

The iterator returned by `end` is an iterator positioned "one past the end" of the associated container (or `string`). This iterator denotes a nonexistent element "off the end" of the container. It is used as a marker indicating when we have processed all the elements. The iterator returned by `end` is often referred to as the **off-the-end iterator** or abbreviated as "the `end` iterator." If the container is empty, `begin` returns the same iterator as the one returned by `end`.

If the container is empty, the iterators returned by `begin` and `end` are equal—they are both off-the-end iterators.

In general, we do not know (or care about) the precise type that an iterator has. In this example, we used `auto` to define `b` and `e` (§ 2.5.2, p. 68). As a result, these variables have whatever type is returned by the `begin` and `end` members, respectively. We'll have more to say about those types on page 108.

Iterator Operations

Iterators support only a few operations, which are listed in Table 3.6. We can compare two valid iterators using `==` or `!=`. Iterators are equal if they denote the same element or if they are both off-the-end iterators for the same container. Otherwise, they are unequal.

As with pointers, we can dereference an iterator to obtain the element denoted by an iterator. Also, like pointers, we may dereference only a valid iterator that denotes an element (§ 2.3.2, p. 53). Dereferencing an invalid iterator or an off-the-end iterator has undefined behavior.

As an example, we'll rewrite the program from § 3.2.3 (p. 94) that capitalized the first character of a `string` using an iterator instead of a subscript:

```
string s("some string");
if (s.begin() != s.end()) { //  make sure s is not empty
    auto it = s.begin();    //  it denotes the first character in s
    *it = toupper(*it);     //  make that character uppercase
}
```

As in our original program, we first check that `s` isn't empty. In this case, we do so by comparing the iterators returned by `begin` and `end`. Those iterators are equal if the `string` is empty. If they are unequl, there is at least one character in `s`.

Inside the `if` body, we obtain an iterator to the first character by assigning the iterator returned by `begin` to `it`. We dereference that iterator to pass that character to `toupper`. We also dereference `it` on the left-hand side of the assignment in order to assign the character returned from `toupper` to the first character in `s`. As in our original program, the output of this loop will be:

```
Some string
```

Table 3.6: Standard Container Iterator Operations

`*iter`	Returns a reference to the element denoted by the iterator `iter`.
`iter->mem`	Dereferences `iter` and fetches the member named `mem` from the underlying element. Equivalent to `(*iter).mem`.
`++iter`	Increments `iter` to refer to the next element in the container.
`--iter`	Decrements `iter` to refer to the previous element in the container.
`iter1 == iter2` `iter1 != iter2`	Compares two iterators for equality (inequality). Two iterators are equal if they denote the same element or if they are the off-the-end iterator for the same container.

Moving Iterators from One Element to Another

Iterators use the increment (++) operator (§ 1.4.1, p. 12) to move from one element to the next. Incrementing an iterator is a logically similar operation to incrementing an integer. In the case of integers, the effect is to "add `1`" to the integer's value. In the case of iterators, the effect is to "advance the iterator by one position."

Because the iterator returned from `end` does not denote an element, it may not be incremented or dereferenced.

Using the increment operator, we can rewrite our program that changed the case of the first word in a `string` to use iterators instead:

```
// process characters in s until we run out of characters or we hit a whitespace
for (auto it = s.begin(); it != s.end() && !isspace(*it); ++it)
    *it = toupper(*it); // capitalize the current character
```

This loop, like the one in § 3.2.3 (p. 94), iterates through the characters in `s`, stopping when we encounter a whitespace character. However, this loop accesses these characters using an iterator, not a subscript.

The loop starts by initializing `it` from `s.begin`, meaning that `it` denotes the first character (if any) in `s`. The condition checks whether `it` has reached the `end` of `s`. If not, the condition next dereferences `it` to pass the current character to `isspace` to see whether we're done. At the end of each iteration, we execute `++it` to advance the iterator to access the next character in `s`.

The body of this loop, is the same as the last statement in the previous `if`. We dereference `it` to pass the current character to `toupper` and assign the resulting uppercase letter back into the character denoted by `it`.

KEY CONCEPT: GENERIC PROGRAMMING

Programmers coming to C++ from C or Java might be surprised that we used `!=` rather than `<` in our `for` loops such as the one above and in the one on page 94. C++ programmers use `!=` as a matter of habit. They do so for the same reason that they use iterators rather than subscripts: This coding style applies equally well to various kinds of containers provided by the library.

As we've seen, only a few library types, `vector` and `string` being among them, have the subscript operator. Similarly, all of the library containers have iterators that define the `==` and `!=` operators. Most of those iterators do not have the `<` operator. By routinely using iterators and `!=`, we don't have to worry about the precise type of container we're processing.

Iterator Types

Just as we do not know the precise type of a `vector`'s or `string`'s `size_type` member (§ 3.2.2, p. 88), so too, we generally do not know—and do not need to know—the precise type of an iterator. Instead, as with `size_type`, the library types that have iterators define types named `iterator` and `const_iterator` that represent actual iterator types:

```
vector<int>::iterator it; // it can read and write vector<int> elements
string::iterator it2;     // it2 can read and write characters in a string
vector<int>::const_iterator it3; // it3 can read but not write elements
string::const_iterator it4;      // it4 can read but not write characters
```

A `const_iterator` behaves like a `const` pointer (§ 2.4.2, p. 62). Like a `const` pointer, a `const_iterator` may read but not write the element it denotes; an object of type `iterator` can both read and write. If a `vector` or `string` is `const`, we may use only its `const_iterator` type. With a nonconst `vector` or `string`, we can use either `iterator` or `const_iterator`.

TERMINOLOGY: ITERATORS AND ITERATOR TYPES

The term iterator is used to refer to three different entities. We might mean the *concept* of an iterator, or we might refer to the `iterator` *type* defined by a container, or we might refer to an *object* as an iterator.

What's important to understand is that there is a collection of types that are related conceptually. A type is an iterator if it supports a common set of actions. Those actions let us access an element in a container and let us move from one element to another.

Each container class defines a type named `iterator`; that `iterator` type supports the actions of an (conceptual) iterator.

The `begin` and `end` Operations

The type returned by `begin` and `end` depends on whether the object on which they operator is `const`. If the object is `const`, then `begin` and `end` return a `const_iterator`; if the object is not `const`, they return `iterator`:

```
vector<int> v;
const vector<int> cv;
auto it1 = v.begin();  // it1 has type vector<int>::iterator
auto it2 = cv.begin(); // it2 has type vector<int>::const_iterator
```

Often this default behavior is not what we want. For reasons we'll explain in § 6.2.3 (p. 213), it is usually best to use a `const` type (such as `const_iterator`) when we need to read but do not need to write to an object. To let us ask specifically for the `const_iterator` type, the new standard introduced two new functions named `cbegin` and `cend`:

C++ 11

```
auto it3 = v.cbegin(); // it3 has type vector<int>::const_iterator
```

As do the `begin` and `end` members, these members return iterators to the first and one past the last element in the container. However, regardless of whether the `vector` (or `string`) is `const`, they return a `const_iterator`.

Combining Dereference and Member Access

When we dereference an iterator, we get the object that the iterator denotes. If that object has a class type, we may want to access a member of that object. For example, we might have a `vector` of `string`s and we might need to know whether a given element is empty. Assuming `it` is an iterator into this `vector`, we can check whether the `string` that `it` denotes is empty as follows:

```
(*it).empty()
```

For reasons we'll cover in § 4.1.2 (p. 136), the parentheses in `(*it).empty()` are necessary. The parentheses say to apply the dereference operator to `it` and to apply the dot operator (§ 1.5.2, p. 23) to the result of dereferencing `it`. Without parentheses, the dot operator would apply to `it`, not to the resulting object:

```
(*it).empty() // dereferences it and calls the member empty on the resulting object
*it.empty()   // error: attempts to fetch the member named empty from it
              //        but it is an iterator and has no member named empty
```

The second expression is interpreted as a request to fetch the `empty` member from the object named `it`. However, `it` is an iterator and has no member named `empty`. Hence, the second expression is in error.

To simplify expressions such as this one, the language defines the arrow operator (the **`->` operator**). The arrow operator combines dereference and member access into a single operation. That is, `it->mem` is a synonym for `(*it).mem`.

For example, assume we have a `vector<string>` named `text` that holds the data from a text file. Each element in the `vector` is either a sentence or an empty `string` representing a paragraph break. If we want to print the contents of the first paragraph from `text`, we'd write a loop that iterates through `text` until we encounter an element that is empty:

```
// print each line in text up to the first blank line
for (auto it = text.cbegin();
     it != text.cend() && !it->empty(); ++it)
    cout << *it << endl;
```

We start by initializing `it` to denote the first element in `text`. The loop continues until either we process every element in `text` or we find an element that is empty. So long as there are elements and we haven't seen an empty element, we print the current element. It is worth noting that because the loop reads but does not write to the elements in `text`, we use `cbegin` and `cend` to control the iteration.

Some `vector` Operations Invalidate Iterators

In § 3.3.2 (p. 101) we noted that there are implications of the fact that `vector`s can grow dynamically. We also noted that one such implication is that we cannot add elements to a `vector` inside a range `for` loop. Another implication is that any operation, such as `push_back`, that changes the size of a `vector` potentially invalidates all iterators into that `vector`. We'll explore how iterators become invalid in more detail in § 9.3.6 (p. 353).

For now, it is important to realize that loops that use iterators should not add elements to the container to which the iterators refer.

Exercises Section 3.4.1

Exercise 3.21: Redo the first exercise from § 3.3.3 (p. 105) using iterators.

Exercise 3.22: Revise the loop that printed the first paragraph in `text` to instead change the elements in `text` that correspond to the first paragraph to all uppercase. After you've updated `text`, print its contents.

Exercise 3.23: Write a program to create a `vector` with ten `int` elements. Using an iterator, assign each element a value that is twice its current value. Test your program by printing the `vector`.

3.4.2 Iterator Arithmetic

Incrementing an iterator moves the iterator one element at a time. All the library containers have iterators that support increment. Similarly, we can use `==` and `!=` to compare two valid iterators (§ 3.4, p. 106) into any of the library container types.

Iterators for `string` and `vector` support additional operations that can move an iterator multiple elements at a time. They also support all the relational operators. These operations, which are often referred to as **iterator arithmetic**, are described in Table 3.7.

Table 3.7: Operations Supported by `vector` and `string` Iterators

`iter + n` `iter - n`	Adding (subtracting) an integral value `n` to (from) an iterator yields an iterator that many elements forward (backward) within the container. The resulting iterator must denote elements in, or one past the end of, the same container.
`iter1 += n` `iter1 -= n`	Compound-assignment for iterator addition and subtraction. Assigns to `iter1` the value of adding `n` to, or subtracting `n` from, `iter1`.
`iter1 - iter2`	Subtracting two iterators yields the number that when added to the right-hand iterator yields the left-hand iterator. The iterators must denote elements in, or one past the end of, the same container.
`>`, `>=`, `<`, `<=`	Relational operators on iterators. One iterator is less than another if it refers to an element that appears in the container before the one referred to by the other iterator. The iterators must denote elements in, or one past the end of, the same container.

Arithmetic Operations on Iterators

We can add (or subtract) an integral value and an iterator. Doing so returns an iterator positioned forward (or backward) that many elements. When we add or subtract an integral value and an iterator, the result must denote an element in the same `vector` (or `string`) or denote one past the end of the associated `vector` (or `string`). As an example, we can compute an iterator to the element nearest the middle of a `vector`:

```
// compute an iterator to the element closest to the midpoint of vi
auto mid = vi.begin() + vi.size() / 2;
```

If `vi` has 20 elements, then `vi.size()/2` is `10`. In this case, we'd set `mid` equal to `vi.begin() + 10`. Remembering that subscripts start at 0, this element is the same as `vi[10]`, the element ten past the first.

In addition to comparing two iterators for equality, we can compare `vector` and `string` iterators using the relational operators (`<`, `<=`, `>`, `>=`). The iterators must be valid and must denote elements in (or one past the end of) the same `vector` or `string`. For example, assuming `it` is an iterator into the same `vector` as `mid`, we can check whether `it` denotes an element before or after `mid` as follows:

```
if (it < mid)
    // process elements in the first half of vi
```

We can also subtract two iterators so long as they refer to elements in, or one off the end of, the same `vector` or `string`. The result is the distance between the iterators. By distance we mean the amount by which we'd have to change one iterator to get the other. The result type is a signed integral type named **`difference_type`**. Both `vector` and `string` define `difference_type`. This type is signed, because subtraction might have a negative result.

Using Iterator Arithmetic

A classic algorithm that uses iterator arithmetic is binary search. A binary search looks for a particular value in a sorted sequence. It operates by looking at the element closest to the middle of the sequence. If that element is the one we want, we're done. Otherwise, if that element is smaller than the one we want, we continue our search by looking only at elements after the rejected one. If the middle element is larger than the one we want, we continue by looking only in the first half. We compute a new middle element in the reduced range and continue looking until we either find the element or run out of elements.

We can do a binary search using iterators as follows:

```
// text must be sorted
// beg and end will denote the range we're searching
auto beg = text.begin(), end = text.end();
auto mid = text.begin() + (end - beg)/2; // original midpoint
// while there are still elements to look at and we haven't yet found sought
while (mid != end && *mid != sought) {
    if (sought < *mid)      // is the element we want in the first half?
        end = mid;          // if so, adjust the range to ignore the second half
    else                    // the element we want is in the second half
        beg = mid + 1;      // start looking with the element just after mid
    mid = beg + (end - beg)/2;  // new midpoint
}
```

We start by defining three iterators: `beg` will be the first element in the range, `end` one past the last element, and `mid` the element closest to the middle. We initialize these iterators to denote the entire range in a `vector<string>` named `text`.

Our loop first checks that the range is not empty. If `mid` is equal to the current value of `end`, then we've run out of elements to search. In this case, the condition fails and we exit the `while`. Otherwise, `mid` refers to an element and we check whether `mid` denotes the one we want. If so, we're done and we exit the loop.

If we still have elements to process, the code inside the `while` adjusts the range by moving `end` or `beg`. If the element denoted by `mid` is greater than `sought`, we know that if `sought` is in `text`, it will appear before the element denoted by `mid`. Therefore, we can ignore elements after `mid`, which we do by assigning `mid` to `end`. If `*mid` is smaller than `sought`, the element must be in the range of elements after the one denoted by `mid`. In this case, we adjust the range by making `beg` denote the element just after `mid`. We already know that `mid` is not the one we want, so we can eliminate it from the range.

At the end of the `while`, `mid` will be equal to `end` or it will denote the element for which we are looking. If `mid` equals `end`, then the element was not in `text`.

Exercises Section 3.4.2

Exercise 3.24: Redo the last exercise from § 3.3.3 (p. 105) using iterators.

Exercise 3.25: Rewrite the grade clustering program from § 3.3.3 (p. 104) using iterators instead of subscripts.

Exercise 3.26: In the binary search program on page 112, why did we write `mid = beg + (end - beg) / 2;` instead of `mid = (beg + end) /2;`?

3.5 Arrays

An array is a data structure that is similar to the library `vector` type (§ 3.3, p. 96) but offers a different trade-off between performance and flexibility. Like a `vector`, an array is a container of unnamed objects of a single type that we access by position. Unlike a `vector`, arrays have fixed size; we cannot add elements to an array. Because arrays have fixed size, they sometimes offer better run-time performance for specialized applications. However, that run-time advantage comes at the cost of lost flexibility.

If you don't know exactly how many elements you need, use a `vector`.

3.5.1 Defining and Initializing Built-in Arrays

Arrays are a compound type (§ 2.3, p. 50). An array declarator has the form `a[d]`, where `a` is the name being defined and `d` is the dimension of the array. The dimension specifies the number of elements and must be greater than zero. The number of elements in an array is part of the array's type. As a result, the dimension must be known at compile time, which means that the dimension must be a constant expression (§ 2.4.4, p. 65):

```
unsigned cnt = 42;           // not a constant expression
constexpr unsigned sz = 42; // constant expression
                             // constexpr see § 2.4.4 (p. 66)
int arr[10];               // array of ten ints
int *parr[sz];             // array of 42 pointers to int
string bad[cnt];           // error: cnt is not a constant expression
string strs[get_size()]; // ok if get_size is constexpr, error otherwise
```

By default, the elements in an array are default initialized (§ 2.2.1, p. 43).

As with variables of built-in type, a default-initialized array of built-in type that is defined inside a function will have undefined values.

When we define an array, we must specify a type for the array. We cannot use `auto` to deduce the type from a list of initializers. As with `vector`, arrays hold objects. Thus, there are no arrays of references.

Explicitly Initializing Array Elements

We can list initialize (§ 3.3.1, p. 98) the elements in an array. When we do so, we can omit the dimension. If we omit the dimension, the compiler infers it from the number of initializers. If we specify a dimension, the number of initializers must not exceed the specified size. If the dimension is greater than the number of initializers, the initializers are used for the first elements and any remaining elements are value initialized (§ 3.3.1, p. 98):

```
const unsigned sz = 3;
int ia1[sz] = {0,1,2};          //  array of three ints with values 0, 1, 2
int a2[] = {0, 1, 2};           //  an array of dimension 3
int a3[5] = {0, 1, 2};          //  equivalent to a3[] = {0, 1, 2, 0, 0}
string a4[3] = {"hi", "bye"}; //  same as a4[] = {"hi", "bye", ""}
int a5[2] = {0,1,2};            //  error: too many initializers
```

Character Arrays Are Special

Character arrays have an additional form of initialization: We can initialize such arrays from a string literal (§ 2.1.3, p. 39). When we use this form of initialization, it is important to remember that string literals end with a null character. That null character is copied into the array along with the characters in the literal:

```
char a1[] = {'C', '+', '+'};        //  list initialization, no null
char a2[] = {'C', '+', '+', '\0'}; //  list initialization, explicit null
char a3[] = "C++";                  //  null terminator added automatically
const char a4[6] = "Daniel";        //  error: no space for the null!
```

The dimension of `a1` is 3; the dimensions of `a2` and `a3` are both 4. The definition of `a4` is in error. Although the literal contains only six explicit characters, the array size must be at least seven—six to hold the literal and one for the null.

No Copy or Assignment

We cannot initialize an array as a copy of another array, nor is it legal to assign one array to another:

```
int a[] = {0, 1, 2}; //  array of three ints
int a2[] = a;        //  error: cannot initialize one array with another
a2 = a;              //  error: cannot assign one array to another
```

Some compilers allow array assignment as a **compiler extension**. It is usually a good idea to avoid using nonstandard features. Programs that use such features, will not work with a different compiler.

Understanding Complicated Array Declarations

Like `vectors`, arrays can hold objects of most any type. For example, we can have an array of pointers. Because an array is an object, we can define both pointers and references to arrays. Defining arrays that hold pointers is fairly straightforward, defining a pointer or reference to an array is a bit more complicated:

```
int *ptrs[10];              // ptrs is an array of ten pointers to int
int &refs[10] = /* ? */;    // error: no arrays of references
int (*Parray)[10] = &arr;   // Parray points to an array of ten ints
int (&arrRef)[10] = arr;    // arrRef refers to an array of ten ints
```

By default, type modifiers bind right to left. Reading the definition of `ptrs` from right to left (§ 2.3.3, p. 58) is easy: We see that we're defining an array of size 10, named `ptrs`, that holds pointers to `int`.

Reading the definition of `Parray` from right to left isn't as helpful. Because the array dimension follows the name being declared, it can be easier to read array declarations from the inside out rather than from right to left. Reading from the inside out makes it much easier to understand the type of `Parray`. We start by observing that the parentheses around `*Parray` mean that `Parray` is a pointer. Looking right, we see that `Parray` points to an array of size 10. Looking left, we see that the elements in that array are `int`s. Thus, `Parray` is a pointer to an array of ten `int`s. Similarly, `(&arrRef)` says that `arrRef` is a reference. The type to which it refers is an array of size 10. That array holds elements of type `int`.

Of course, there are no limits on how many type modifiers can be used:

```
int *(&arry)[10] = ptrs; // arry is a reference to an array of ten pointers
```

Reading this declaration from the inside out, we see that `arry` is a reference. Looking right, we see that the object to which `arry` refers is an array of size 10. Looking left, we see that the element type is pointer to `int`. Thus, `arry` is a reference to an array of ten pointers.

It can be easier to understand array declarations by starting with the array's name and reading them from the inside out.

Exercises Section 3.5.1

Exercise 3.27: Assuming `txt_size` is a function that takes no arguments and returns an `int` value, which of the following definitions are illegal? Explain why.

```
unsigned buf_size = 1024;
(a) int ia[buf_size];      (b) int ia[4 * 7 - 14];
(c) int ia[txt_size()];    (d) char st[11] = "fundamental";
```

Exercise 3.28: What are the values in the following arrays?

```
string sa[10];
int ia[10];
int main() {
    string sa2[10];
    int    ia2[10];
}
```

Exercise 3.29: List some of the drawbacks of using an array instead of a `vector`.

3.5.2 Accessing the Elements of an Array

As with the library `vector` and `string` types, we can use a range `for` or the subscript operator to access elements of an array. As usual, the indices start at 0. For an array of ten elements, the indices are 0 through 9, not 1 through 10.

When we use a variable to subscript an array, we normally should define that variable to have type **`size_t`**. `size_t` is a machine-specific unsigned type that is guaranteed to be large enough to hold the size of any object in memory. The `size_t` type is defined in the `cstddef` header, which is the C++ version of the `stddef.h` header from the C library.

With the exception that arrays are fixed size, we use arrays in ways that are similar to how we use `vectors`. For example, we can reimplement our grading program from § 3.3.3 (p. 104) to use an array to hold the cluster counters:

```
// count the number of grades by clusters of ten: 0--9, 10--19, . . . 90--99, 100
unsigned scores[11] = {};  // 11 buckets, all value initialized to 0
unsigned grade;
while (cin >> grade) {
    if (grade <= 100)
        ++scores[grade/10]; // increment the counter for the current cluster
}
```

The only obvious difference between this program and the one on page 104 is the declaration of `scores`. In this program `scores` is an array of 11 `unsigned` elements. The not so obvious difference is that the subscript operator in this program is the one that is defined as part of the language. This operator can be used on operands of array type. The subscript operator used in the program on page 104 was defined by the library `vector` template and applies to operands of type `vector`.

As in the case of `string` or `vector`, it is best to use a range `for` when we want to traverse the entire array. For example, we can print the resulting `scores` as follows:

```
for (auto i : scores)        // for each counter in scores
    cout << i << " ";        // print the value of that counter
cout << endl;
```

Because the dimension is part of each array type, the system knows how many elements are in `scores`. Using a range `for` means that we don't have to manage the traversal ourselves.

Checking Subscript Values

As with `string` and `vector`, it is up to the programmer to ensure that the subscript value is in range—that is, that the index value is equal to or greater than zero and less than the size of the array. Nothing stops a program from stepping across an array boundary except careful attention to detail and thorough testing of the code. It is possible for programs to compile and execute yet still be fatally wrong.

The most common source of security problems are buffer overflow bugs. Such bugs occur when a program fails to check a subscript and mistakenly uses memory outside the range of an array or similar data structure.

Exercises Section 3.5.2

Exercise 3.30: Identify the indexing errors in the following code:

```
constexpr size_t array_size = 10;
int ia[array_size];
for (size_t ix = 1; ix <= array_size; ++ix)
      ia[ix] = ix;
```

Exercise 3.31: Write a program to define an array of ten `int`s. Give each element the same value as its position in the array.

Exercise 3.32: Copy the array you defined in the previous exercise into another array. Rewrite your program to use `vector`s.

Exercise 3.33: What would happen if we did not initialize the `scores` array in the program on page 116?

3.5.3 Pointers and Arrays

In C++ pointers and arrays are closely intertwined. In particular, as we'll see, when we use an array, the compiler ordinarily converts the array to a pointer.

Normally, we obtain a pointer to an object by using the address-of operator (§ 2.3.2, p. 52). Generally speaking, the address-of operator may be applied to any object. The elements in an array are objects. When we subscript an array, the result is the object at that location in the array. As with any other object, we can obtain a pointer to an array element by taking the address of that element:

```
string nums[] = {"one", "two", "three"};  // array of strings
string *p = &nums[0];   // p points to the first element in nums
```

However, arrays have a special property—in most places when we use an array, the compiler automatically substitutes a pointer to the first element:

```
string *p2 = nums;       // equivalent to p2 = &nums[0]
```

In most expressions, when we use an object of array type, we are really using a pointer to the first element in that array.

There are various implications of the fact that operations on arrays are often really operations on pointers. One such implication is that when we use an array as an initializer for a variable defined using `auto` (§ 2.5.2, p. 68), the deduced type is a pointer, not an array:

```
int ia[] = {0,1,2,3,4,5,6,7,8,9}; // ia is an array of ten ints
auto ia2(ia); // ia2 is an int* that points to the first element in ia
ia2 = 42;     // error: ia2 is a pointer, and we can't assign an int to a pointer
```

Although `ia` is an array of ten `int`s, when we use `ia` as an initializer, the compiler treats that initialization as if we had written

```
auto ia2(&ia[0]);  // now it's clear that ia2 has type int*
```

It is worth noting that this conversion does not happen when we use `decltype` (§ 2.5.3, p. 70). The type returned by `decltype(ia)` is array of ten `int`s:

```
// ia3 is an array of ten ints
decltype(ia) ia3 = {0,1,2,3,4,5,6,7,8,9};
ia3 = p;     // error: can't assign an int* to an array
ia3[4] = i; // ok: assigns the value of i to an element in ia3
```

Pointers Are Iterators

Pointers that address elements in an array have additional operations beyond those we described in § 2.3.2 (p. 52). In particular, pointers to array elements support the same operations as iterators on `vector`s or `string`s (§ 3.4, p. 106). For example, we can use the increment operator to move from one element in an array to the next:

```
int arr[] = {0,1,2,3,4,5,6,7,8,9};
int *p = arr; // p points to the first element in arr
++p;          // p points to arr[1]
```

Just as we can use iterators to traverse the elements in a `vector`, we can use pointers to traverse the elements in an array. Of course, to do so, we need to obtain pointers to the first and one past the last element. As we've just seen, we can obtain a pointer to the first element by using the array itself or by taking the address-of the first element. We can obtain an off-the-end pointer by using another special property of arrays. We can take the address of the nonexistent element one past the last element of an array:

```
int *e = &arr[10]; // pointer just past the last element in arr
```

Here we used the subscript operator to index a nonexisting element; `arr` has ten elements, so the last element in `arr` is at index position 9. The only thing we can do with this element is take its address, which we do to initialize `e`. Like an off-the-end iterator (§ 3.4.1, p. 106), an off-the-end pointer does not point to an element. As a result, we may not dereference or increment an off-the-end pointer.

Using these pointers we can write a loop to print the elements in `arr` as follows:

```
for (int *b = arr; b != e; ++b)
    cout << *b << endl; // print the elements in arr
```

The Library `begin` and `end` Functions

C++ 11

Although we can compute an off-the-end pointer, doing so is error-prone. To make it easier and safer to use pointers, the new library includes two functions, named `begin` and `end`. These functions act like the similarly named container members (§ 3.4.1, p. 106). However, arrays are not class types, so these functions are not member functions. Instead, they take an argument that is an array:

```
int ia[] = {0,1,2,3,4,5,6,7,8,9}; // ia is an array of ten ints
int *beg = begin(ia); // pointer to the first element in ia
int *last = end(ia);  // pointer one past the last element in ia
```

`begin` returns a pointer to the first, and `end` returns a pointer one past the last element in the given array: These functions are defined in the `iterator` header.

Using `begin` and `end`, it is easy to write a loop to process the elements in an array. For example, assuming `arr` is an array that holds `int` values, we might find the first negative value in `arr` as follows:

```
// pbeg points to the first and pend points just past the last element in arr
int *pbeg = begin(arr),  *pend = end(arr);
// find the first negative element, stopping if we've seen all the elements
while (pbeg != pend && *pbeg >= 0)
    ++pbeg;
```

We start by defining two `int` pointers named `pbeg` and `pend`. We position `pbeg` to denote the first element and `pend` to point one past the last element in `arr`. The `while` condition uses `pend` to know whether it is safe to dereference `pbeg`. If `pbeg` does point at an element, we dereference and check whether the underlying element is negative. If so, the condition fails and we exit the loop. If not, we increment the pointer to look at the next element.

A pointer "one past" the end of a built-in array behaves the same way as the iterator returned by the `end` operation of a `vector`. In particular, we may not dereference or increment an off-the-end pointer.

Pointer Arithmetic

Pointers that address array elements can use all the iterator operations listed in Table 3.6 (p. 107) and Table 3.7 (p. 111). These operations—dereference, increment, comparisons, addition of an integral value, subtraction of two pointers—have the same meaning when applied to pointers that point at elements in a built-in array as they do when applied to iterators.

When we add (or subtract) an integral value to (or from) a pointer, the result is a new pointer. That new pointer points to the element the given number ahead of (or behind) the original pointer:

```
constexpr size_t sz = 5;
int arr[sz] = {1,2,3,4,5};
int *ip = arr;      // equivalent to int *ip = &arr[0]
int *ip2 = ip + 4; // ip2 points to arr[4], the last element in arr
```

The result of adding `4` to `ip` is a pointer that points to the element four elements further on in the array from the one to which `ip` currently points.

The result of adding an integral value to a pointer must be a pointer to an element in the same array, or a pointer just past the end of the array:

```
// ok: arr is converted to a pointer to its first element; p points one past the end of arr
int *p = arr + sz; // use caution -- do not dereference!
int *p2 = arr + 10; // error: arr has only 5 elements; p2 has undefined value
```

When we add `sz` to `arr`, the compiler converts `arr` to a pointer to the first element in `arr`. When we add `sz` to that pointer, we get a pointer that points `sz` positions

(i.e., 5 positions) past the first one. That is, it points one past the last element in arr. Computing a pointer more than one past the last element is an error, although the compiler is unlikely to detect such errors.

As with iterators, subtracting two pointers gives us the distance between those pointers. The pointers must point to elements in the same array:

```
auto n = end(arr) - begin(arr); //  n is 5, the number of elements in arr
```

The result of subtracting two pointers is a library type named **ptrdiff_t**. Like size_t, the ptrdiff_t type is a machine-specific type and is defined in the cstddef header. Because subtraction might yield a negative distance, ptrdiff_t is a signed integral type.

We can use the relational operators to compare pointers that point to elements of an array, or one past the last element in that array. For example, we can traverse the elements in arr as follows:

```
int *b = arr, *e = arr + sz;
while (b < e) {
    //  use *b
    ++b;
}
```

We cannot use the relational operators on pointers to two unrelated objects:

```
int i = 0, sz = 42;
int *p = &i, *e = &sz;
//  undefined:  p and e are unrelated; comparison is meaningless!
while (p < e)
```

Although the utility may be obscure at this point, it is worth noting that pointer arithmetic is also valid for null pointers (§ 2.3.2, p. 53) and for pointers that point to an object that is not an array. In the latter case, the pointers must point to the same object, or one past that object. If p is a null pointer, we can add or subtract an integral constant expression (§ 2.4.4, p. 65) whose value is 0 to p. We can also subtract two null pointers from one another, in which case the result is 0.

Interaction between Dereference and Pointer Arithmetic

The result of adding an integral value to a pointer is itself a pointer. Assuming the resulting pointer points to an element, we can dereference the resulting pointer:

```
int ia[] = {0,2,4,6,8};  //  array with 5 elements of type int
int last = *(ia + 4);  //  ok: initializes last to 8, the value of ia[4]
```

The expression *(ia + 4) calculates the address four elements past ia and dereferences the resulting pointer. This expression is equivalent to writing ia[4].

Recall that in § 3.4.1 (p. 109) we noted that parentheses are required in expressions that contain dereference and dot operators. Similarly, the parentheses around this pointer addition are essential. Writing

```
last = *ia + 4;    //  ok: last = 4, equivalent to ia[0] + 4
```

means dereference ia and add 4 to the dereferenced value. We'll cover the reasons for this behavior in § 4.1.2 (p. 136).

Subscripts and Pointers

As we've seen, in most places when we use the name of an array, we are really using a pointer to the first element in that array. One place where the compiler does this transformation is when we subscript an array. Given

```
int ia[] = {0,2,4,6,8};  // array with 5 elements of type int
```

if we write `ia[0]`, that is an expression that uses the name of an array. When we subscript an array, we are really subscripting a pointer to an element in that array:

```
int i = ia[2];  // ia is converted to a pointer to the first element in ia
                // ia[2] fetches the element to which (ia + 2) points
int *p = ia;    // p points to the first element in ia
i = *(p + 2);   // equivalent to i = ia[2]
```

We can use the subscript operator on any pointer, as long as that pointer points to an element (or one past the last element) in an array:

```
int *p = &ia[2];  // p points to the element indexed by 2
int j = p[1];     // p[1] is equivalent to *(p + 1),
                  // p[1] is the same element as ia[3]
int k = p[-2];    // p[-2] is the same element as ia[0]
```

This last example points out an important difference between arrays and library types such as `vector` and `string` that have subscript operators. The library types force the index used with a subscript to be an unsigned value. The built-in subscript operator does not. The index used with the built-in subscript operator can be a negative value. Of course, the resulting address must point to an element in (or one past the end of) the array to which the original pointer points.

Unlike subscripts for `vector` and `string`, the index of the built-in subscript operator is not an `unsigned` type.

EXERCISES SECTION 3.5.3

Exercise 3.34: Given that `p1` and `p2` point to elements in the same array, what does the following code do? Are there values of `p1` or `p2` that make this code illegal?

```
p1 += p2 - p1;
```

Exercise 3.35: Using pointers, write a program to set the elements in an array to zero.

Exercise 3.36: Write a program to compare two arrays for equality. Write a similar program to compare two `vectors`.

3.5.4 C-Style Character Strings

Although C++ supports C-style strings, they should not be used by C++ programs. C-style strings are a surprisingly rich source of bugs and are the root cause of many security problems. They're also harder to use!

Character string literals are an instance of a more general construct that C++ inherits from C: **C-style character strings**. C-style strings are not a type. Instead, they are a convention for how to represent and use character strings. Strings that follow this convention are stored in character arrays and are **null terminated**. By null-terminated we mean that the last character in the string is followed by a null character (`'\0'`). Ordinarily we use pointers to manipulate these strings.

C Library String Functions

The Standard C library provides a set of functions, listed in Table 3.8, that operate on C-style strings. These functions are defined in the `cstring` header, which is the C++ version of the C header `string.h`.

The functions in Table 3.8 do not verify their string parameters.

The pointer(s) passed to these routines must point to null-terminated array(s):

```
char ca[] = {'C', '+', '+'};   // not null terminated
cout << strlen(ca) << endl;    // disaster: ca isn't null terminated
```

In this case, `ca` is an array of `char` but is not null terminated. The result is undefined. The most likely effect of this call is that `strlen` will keep looking through the memory that follows `ca` until it encounters a null character.

Table 3.8: C-Style Character String Functions

`strlen(p)`	Returns the length of `p`, *not counting the null*.
`strcmp(p1, p2)`	Compares `p1` and `p2` for equality. Returns `0` if `p1 == p2`, a positive value if `p1 > p2`, a negative value if `p1 < p2`.
`strcat(p1, p2)`	Appends `p2` to `p1`. Returns `p1`.
`strcpy(p1, p2)`	Copies `p2` into `p1`. Returns `p1`.

Comparing Strings

Comparing two C-style strings is done quite differently from how we compare library `strings`. When we compare two library `strings`, we use the normal relational or equality operators:

```
string s1 = "A string example";
string s2 = "A different string";
if (s1 < s2)  // false: s2 is less than s1
```

Using these operators on similarly defined C-style strings compares the pointer values, not the strings themselves:

```
const char ca1[] = "A string example";
const char ca2[] = "A different string";
if (ca1 < ca2)  // undefined: compares two unrelated addresses
```

Remember that when we use an array, we are really using a pointer to the first element in the array (§ 3.5.3, p. 117). Hence, this condition actually compares two `const char*` values. Those pointers do not address the same object, so the comparison is undefined.

To compare the strings, rather than the pointer values, we can call `strcmp`. That function returns 0 if the strings are equal, or a positive or negative value, depending on whether the first string is larger or smaller than the second:

```
if (strcmp(ca1, ca2) < 0) // same effect as string comparison s1 < s2
```

Caller Is Responsible for Size of a Destination String

Concatenating or copying C-style strings is also very different from the same operations on library `string`s. For example, if we wanted to concatenate the two `string`s `s1` and `s2` defined above, we can do so directly:

```
// initialize largeStr as a concatenation of s1, a space, and s2
string largeStr = s1 + " " + s2;
```

Doing the same with our two arrays, `ca1` and `ca2`, would be an error. The expression `ca1 + ca2` tries to add two pointers, which is illegal and meaningless.

Instead we can use `strcat` and `strcpy`. However, to use these functions, we must pass an array to hold the resulting string. The array we pass *must* be large enough to hold the generated string, including the null character at the end. The code we show here, although a common usage pattern, is fraught with potential for serious error:

```
// disastrous if we miscalculated the size of largeStr
strcpy(largeStr, ca1);      // copies ca1 into largeStr
strcat(largeStr, " ");      // adds a space at the end of largeStr
strcat(largeStr, ca2);      // concatenates ca2 onto largeStr
```

The problem is that we can easily miscalculate the size needed for `largeStr`. Moreover, any time we change the values we want to store in `largeStr`, we have to remember to double-check that we calculated its size correctly. Unfortunately, programs similar to this code are widely distributed. Programs with such code are error-prone and often lead to serious security leaks.

For most applications, in addition to being safer, it is also more efficient to use library `string`s rather than C-style strings.

EXERCISES SECTION 3.5.4

Exercise 3.37: What does the following program do?

```
const char ca[] = {'h', 'e', 'l', 'l', 'o'};
const char *cp = ca;
while (*cp) {
    cout << *cp << endl;
    ++cp;
}
```

Exercise 3.38: In this section, we noted that it was not only illegal but meaningless to try to add two pointers. Why would adding two pointers be meaningless?

Exercise 3.39: Write a program to compare two `string`s. Now write a program to compare the values of two C-style character strings.

Exercise 3.40: Write a program to define two character arrays initialized from string literals. Now define a third character array to hold the concatenation of the two arrays. Use `strcpy` and `strcat` to copy the two arrays into the third.

3.5.5 Interfacing to Older Code

Many C++ programs predate the standard library and do not use the `string` and `vector` types. Moreover, many C++ programs interface to programs written in C or other languages that cannot use the C++ library. Hence, programs written in modern C++ may have to interface to code that uses arrays and/or C-style character strings. The C++ library offers facilities to make the interface easier to manage.

Mixing Library `string`s and C-Style Strings

In § 3.2.1 (p. 84) we saw that we can initialize a `string` from a string literal:

```
string s("Hello World");  // s holds Hello World
```

More generally, we can use a null-terminated character array anywhere that we can use a string literal:

- We can use a null-terminated character array to initialize or assign a `string`.
- We can use a null-terminated character array as one operand (but not both operands) to the `string` addition operator or as the right-hand operand in the `string` compound assignment (+=) operator.

The reverse functionality is not provided: There is no direct way to use a library `string` when a C-style string is required. For example, there is no way to initialize a character pointer from a `string`. There is, however, a `string` member function named `c_str` that we can often use to accomplish what we want:

```
char *str = s; // error: can't initialize a char* from a string
const char *str = s.c_str(); // ok
```

The name `c_str` indicates that the function returns a C-style character string. That is, it returns a pointer to the beginning of a null-terminated character array that holds the same data as the characters in the `string`. The type of the pointer is `const char*`, which prevents us from changing the contents of the array.

The array returned by `c_str` is not guaranteed to be valid indefinitely. Any subsequent use of `s` that might change the value of `s` can invalidate this array.

If a program needs continuing access to the contents of the array returned by `str()`, the program must copy the array returned by `c_str`.

Using an Array to Initialize a `vector`

In § 3.5.1 (p. 114) we noted that we cannot initialize a built-in array from another array. Nor can we initialize an array from a `vector`. However, we can use an array to initialize a `vector`. To do so, we specify the address of the first element and one past the last element that we wish to copy:

```
int int_arr[] = {0, 1, 2, 3, 4, 5};
//  ivec has six elements; each is a copy of the corresponding element in int_arr
vector<int> ivec(begin(int_arr), end(int_arr));
```

The two pointers used to construct `ivec` mark the range of values to use to initialize the elements in `ivec`. The second pointer points one past the last element to be copied. In this case, we used the library `begin` and `end` functions (§ 3.5.3, p. 118) to pass pointers to the first and one past the last elements in `int_arr`. As a result, `ivec` will have six elements each of which will have the same value as the corresponding element in `int_arr`.

The specified range can be a subset of the array:

```
//  copies three elements: int_arr[1], int_arr[2], int_arr[3]
vector<int> subVec(int_arr + 1, int_arr + 4);
```

This initialization creates `subVec` with three elements. The values of these elements are copies of the values in `int_arr[1]` through `int_arr[3]`.

ADVICE: USE LIBRARY TYPES INSTEAD OF ARRAYS

Pointers and arrays are surprisingly error-prone. Part of the problem is conceptual: Pointers are used for low-level manipulations and it is easy to make bookkeeping mistakes. Other problems arise because of the syntax, particularly the declaration syntax used with pointers.

Modern C++ programs should use `vector`s and iterators instead of built-in arrays and pointers, and use `string`s rather than C-style array-based character strings.

3.6 Multidimensional Arrays

Strictly speaking, there are no multidimensional arrays in C++. What are commonly referred to as multidimensional arrays are actually arrays of arrays. It can

Exercises Section 3.5.5

Exercise 3.41: Write a program to initialize a `vector` from an array of `int`s.

Exercise 3.42: Write a program to copy a `vector` of `int`s into an array of `int`s.

be helpful to keep this fact in mind when you use what appears to be a multidimensional array.

We define an array whose elements are arrays by providing two dimensions: the dimension of the array itself and the dimension of its elements:

```
int ia[3][4]; //  array of size 3; each element is an array of ints of size 4
//  array of size 10; each element is a 20-element array whose elements are arrays of 30 ints
int arr[10][20][30] = {0}; //  initialize all elements to 0
```

As we saw in § 3.5.1 (p. 115), we can more easily understand these definitions by reading them from the inside out. We start with the name we're defining (`ia`) and see that `ia` is an array of size 3. Continuing to look to the right, we see that the elements of `ia` also have a dimension. Thus, the elements in `ia` are themselves arrays of size 4. Looking left, we see that the type of those elements is `int`. So, `ia` is an array of size 3, each of whose elements is an array of four `int`s.

We read the definition for `arr` in the same way. First we see that `arr` is an array of size 10. The elements of that array are themselves arrays of size 20. Each of those arrays has 30 elements that are of type `int`. There is no limit on how many subscripts are used. That is, we can have an array whose elements are arrays of elements that are arrays, and so on.

In a two-dimensional array, the first dimension is usually referred to as the row and the second as the column.

Initializing the Elements of a Multidimensional Array

As with any array, we can initialize the elements of a multidimensional array by providing a bracketed list of initializers. Multidimensional arrays may be initialized by specifying bracketed values for each row:

```
int ia[3][4] = {     //  three elements; each element is an array of size 4
    {0, 1, 2, 3},    //  initializers for the row indexed by 0
    {4, 5, 6, 7},    //  initializers for the row indexed by 1
    {8, 9, 10, 11}   //  initializers for the row indexed by 2
};
```

The nested braces are optional. The following initialization is equivalent, although considerably less clear:

```
//  equivalent initialization without the optional nested braces for each row
int ia[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};
```

As is the case for single-dimension arrays, elements may be left out of the initializer list. We can initialize only the first element of each row as follows:

```
// explicitly initialize only element 0 in each row
int ia[3][4] = {{ 0 }, { 4 }, { 8 }};
```

The remaining elements are value initialized in the same way as ordinary, single-dimension arrays (§ 3.5.1, p. 114). If the nested braces were omitted, the results would be very different. This code

```
// explicitly initialize row 0; the remaining elements are value initialized
int ix[3][4] = {0, 3, 6, 9};
```

initializes the elements of the first row. The remaining elements are initialized to 0.

Subscripting a Multidimensional Array

As with any array, we can use a subscript to access the elements of a multidimensional array. To do so, we use a separate subscript for each dimension.

If an expression provides as many subscripts as there are dimensions, we get an element with the specified type. If we supply fewer subscripts than there are dimensions, then the result is the inner-array element at the specified index:

```
// assigns the first element of arr to the last element in the last row of ia
ia[2][3] = arr[0][0][0];
int (&row)[4] = ia[1]; // binds row to the second four-element array in ia
```

In the first example we supply indices for all the dimensions for both arrays. On the left-hand side, `ia[2]` returns the last row in `ia`. It does not fetch an element from that array but returns the array itself. We subscript that array, fetching element `[3]`, which is the last element in that array.

Similarly, the right-hand operand has three dimensions. We first fetch the array at index `0` from the outermost array. The result of that operation is a (multidimensional) array of size 20. We take the first element from that 20-element array, yielding an array of size 30. We then fetch the first element from that array.

In the second example, we define `row` as a reference to an array of four `int`s. We bind that reference to the second row in `ia`.

As another example, it is common to use a pair of nested `for` loops to process the elements in a multidimensional array:

```
constexpr size_t rowCnt = 3, colCnt = 4;
int ia[rowCnt][colCnt];    // 12 uninitialized elements
// for each row
for (size_t i = 0; i != rowCnt; ++i) {
    // for each column within the row
    for (size_t j = 0; j != colCnt; ++j) {
        // assign the element's positional index as its value
        ia[i][j] = i * colCnt + j;
    }
}
```

The outer `for` loops through each of the array elements in `ia`. The inner `for` loops through the elements of those interior arrays. In this case, we set the value of each element as its index in the overall array.

Using a Range `for` with Multidimensional Arrays

Under the new standard we can simplify the previous loop by using a range `for`:

```
size_t cnt = 0;
for (auto &row : ia)          // for every element in the outer array
    for (auto &col : row) {   // for every element in the inner array
        col = cnt;            // give this element the next value
        ++cnt;                // increment cnt
    }
```

This loop gives the elements of `ia` the same values as the previous loop, but this time we let the system manage the indices for us. We want to change the value of the elements, so we declare our control variables, `row` and `col`, as references (§ 3.2.3, p. 93). The first `for` iterates through the elements in `ia`. Those elements are arrays of size 4. Thus, the type of `row` is a reference to an array of four `int`s. The second `for` iterates through one of those 4-element arrays. Hence, `col` is `int&`. On each iteration we assign the value of `cnt` to the next element in `ia` and increment `cnt`.

In the previous example, we used references as our loop control variables because we wanted to change the elements in the array. However, there is a deeper reason for using references. As an example, consider the following loop:

```
for (const auto &row : ia)   // for every element in the outer array
    for (auto col : row)     // for every element in the inner array
        cout << col << endl;
```

This loop does not write to the elements, yet we still define the control variable of the outer loop as a reference. We do so in order to avoid the normal array to pointer conversion (§ 3.5.3, p. 117). Had we neglected the reference and written these loops as:

```
for (auto row : ia)
    for (auto col : row)
```

our program would not compile. As before, the first `for` iterates through `ia`, whose elements are arrays of size 4. Because `row` is not a reference, when the compiler initializes `row` it will convert each array element (like any other object of array type) to a pointer to that array's first element. As a result, in this loop the type of `row` is `int*`. The inner `for` loop is illegal. Despite our intentions, that loop attempts to iterate over an `int*`.

To use a multidimensional array in a range `for`, the loop control variable for all but the innermost array must be references.

Pointers and Multidimensional Arrays

As with any array, when we use the name of a multidimensional array, it is automatically converted to a pointer to the first element in the array.

When you define a pointer to a multidimensional array, remember that a multidimensional array is really an array of arrays.

Because a multidimensional array is really an array of arrays, the pointer type to which the array converts is a pointer to the first inner array:

```
int ia[3][4];      // array of size 3; each element is an array of ints of size 4
int (*p)[4] = ia; // p points to an array of four ints
p = &ia[2];        // p now points to the last element in ia
```

Applying the strategy from § 3.5.1 (p. 115), we start by noting that `(*p)` says `p` is a pointer. Looking right, we see that the object to which `p` points has a dimension of size `4`, and looking left that the element type is `int`. Hence, `p` is a pointer to an array of four `int`s.

The parentheses in this declaration are essential:

```
int *ip[4];     // array of pointers to int
int (*ip)[4];   // pointer to an array of four ints
```

With the advent of the new standard, we can often avoid having to write the type of a pointer into an array by using `auto` or `decltype` (§ 2.5.2, p. 68):

C++ 11

```
// print the value of each element in ia, with each inner array on its own line
// p points to an array of four ints
for (auto p = ia; p != ia + 3; ++p) {
    // q points to the first element of an array of four ints; that is, q points to an int
    for (auto q = *p; q != *p + 4; ++q)
         cout << *q << ' ';
    cout << endl;
}
```

The outer `for` loop starts by initializing `p` to point to the first array in `ia`. That loop continues until we've processed all three rows in `ia`. The increment, `++p`, has the effect of moving `p` to point to the next row (i.e., the next element) in `ia`.

The inner `for` loop prints the values of the inner arrays. It starts by making `q` point to the first element in the array to which `p` points. The result of `*p` is an array of four `int`s. As usual, when we use an array, it is converted automatically to a pointer to its first element. The inner `for` loop runs until we've processed every element in the inner array. To obtain a pointer just off the end of the inner array, we again dereference `p` to get a pointer to the first element in that array. We then add 4 to that pointer to process the four elements in each inner array.

Of course, we can even more easily write this loop using the library `begin` and `end` functions (§ 3.5.3, p. 118):

```
// p points to the first array in ia
for (auto p = begin(ia); p != end(ia); ++p) {
    // q points to the first element in an inner array
    for (auto q = begin(*p); q != end(*p); ++q)
        cout << *q << ' ';    // prints the int value to which q points
    cout << endl;
}
```

Here we let the library determine the end pointer, and we use `auto` to avoid having to write the type returned from `begin`. In the outer loop, that type is a pointer to an array of four `int`s. In the inner loop, that type is a pointer to `int`.

Type Aliases Simplify Pointers to Multidimensional Arrays

A type alias (§ 2.5.1, p. 67) can make it easier to read, write, and understand pointers to multidimensional arrays. For example:

```
using int_array = int[4]; //  new style type alias declaration; see § 2.5.1 (p. 68)
typedef int int_array[4]; //  equivalent typedef declaration; § 2.5.1 (p. 67)
// print the value of each element in ia, with each inner array on its own line
for (int_array *p = ia; p != ia + 3; ++p) {
    for (int *q = *p; q != *p + 4; ++q)
         cout << *q << ' ';
    cout << endl;
}
```

Here we start by defining `int_array` as a name for the type "array of four `int`s." We use that type name to define our loop control variable in the outer `for` loop.

EXERCISES SECTION 3.6

Exercise 3.43: Write three different versions of a program to print the elements of `ia`. One version should use a range `for` to manage the iteration, the other two should use an ordinary `for` loop in one case using subscripts and in the other using pointers. In all three programs write all the types directly. That is, do not use a type alias, `auto`, or `decltype` to simplify the code.

Exercise 3.44: Rewrite the programs from the previous exercises using a type alias for the type of the loop control variables.

Exercise 3.45: Rewrite the programs again, this time using `auto`.

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`.