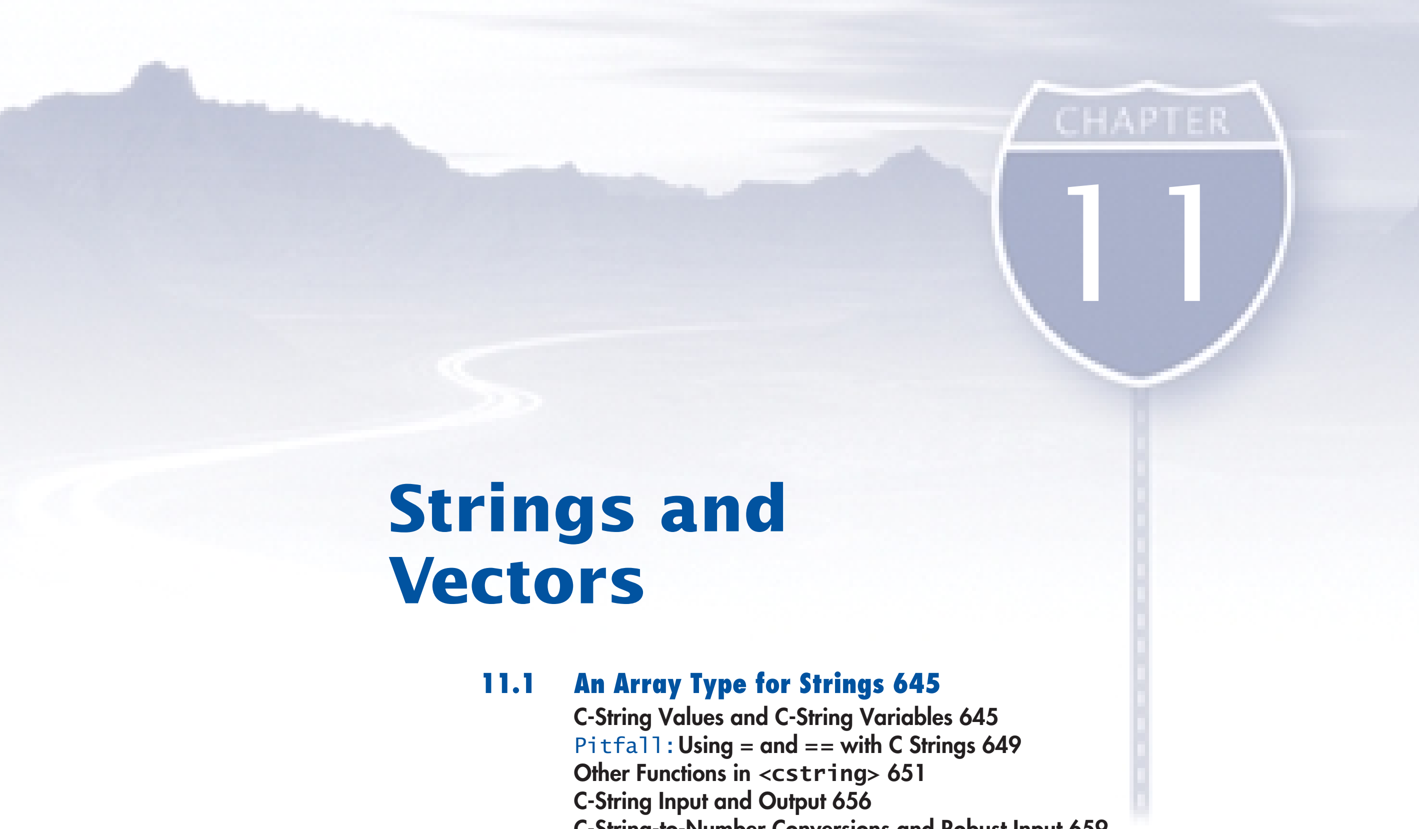

Strings and Vectors

CHAPTER 11

Strings and Vectors

Polonius: What do you read my lord?
Hamlet: Words, words, words

WILLIAM SHAKESPEARE, *HAMLET*

Introduction

This chapter discusses two topics that use arrays or are related to arrays: strings and vectors. Although strings and vectors are very closely related, this relationship is not always obvious, and no one of these topics depends on the other. The topics of strings and vectors can be covered in either order.

Sections 11.1 and 11.2 present two types whose values represent strings of characters, such as `"Hello"`. One type, discussed in Section 11.1, is just an array with base type *`char`* that stores strings of characters in the array and marks the end of the string with the null character `'\0'`. This is the older way of representing strings, which C++ inherited from the C programming language. These sorts of strings are called **C strings.** Although C strings are an older way of representing strings, it is difficult to do any sort of string processing in C++ without at least passing contact with C strings. For example, quoted strings, such as `"Hello"`, are implemented as C strings in C++.

C string

The ANSI/ISO C++ standard includes a more modern string handling facility in the form of the class `string`. The class `string` is the second string type that we will discuss in this chapter and is covered in Section 11.2.

Vectors can be thought of as arrays that can grow (and shrink) in length while your program is running. In C++, once your program creates an array, it cannot change the length of the array. Vectors serve the same purpose as arrays except that they can change length while the program is running.

Prerequisites

Sections 11.1 and 11.2, which cover strings, and Section 11.3 which covers vectors, are independent of each other. If you wish to cover vectors before strings, that is fine.

Section 11.1 on C strings uses material from Chapters 2 through 5, Chapter 7, and Sections 10.1, 10.2, and 10.3 of Chapter 10; it does not use any of the material on classes from Chapters 6, 8, or 9.

Section 11.2 on the `string` class uses Section 11.1 and material from Chapters 2 through 7 and Sections 10.1, 10.2, and 10.3 of Chapter 10.

Section 11.3 on vectors uses material from Chapters 2 through 7 and Sections 10.1, 10.2, and 10.3 of Chapter 10.

11.1 An Array Type for Strings

In everything one must consider the end.

JEAN DE LA FONTAINE, *FABLES*, BOOK III (1668)

In this section we will describe one way to represent strings of characters, which C++ has inherited from the C language. In Section 11.2 we will describe a string class that is a more modern way to represent strings. Although the string type described here may be a bit "old fashioned," it is still widely used and is an integral part of the C++ language.

C-String Values and C-String Variables

One way to represent a string is as an array with base type *char*. If the string is `"Hello"`, it is handy to represent it as an array of characters with six indexed variables: five for the five letters in `"Hello"` plus one for the character `'\0'`, which serves as an end marker. The character `'\0'` is called the **null character** and is used as an end marker because it is distinct from all the "real" characters. The end marker allows your program to read the array one character at a time and know that it should stop reading when it reads the end marker `'\0'`. A string stored in this way (as an array of characters terminated with `'\0'`) is called a **C string.**

the null character '\0'

C string

We write `'\0'` with two symbols when we write it in a program, but just like the new-line character `'\n'`, the character `'\0'` is really only a single character value. Like any other character value, `'\0'` can be stored in one variable of type *char* or one indexed variable of an array of characters.

The Null Character, '\0'

The null character, `'\0'`, is used to mark the end of a C string that is stored in an array of characters. When an array of characters is used in this way, the array is often called a C-string variable. Although the null character `'\0'` is written using two symbols, it is a single character that fits in one variable of type *char* or one indexed variable of an array of characters.

You have already been using C strings. In C++, a literal string, such as `"Hello"`, is stored as a C string, although you seldom need to be aware of this detail.

C-string variable

A **C-string variable** is just an array of characters. Thus, the following array declaration provides us with a C-string variable capable of storing a C-string value with nine or fewer characters:

```
char s[10];
```

The `10` is for the nine letters in the string plus the null character `'\0'` to mark the end of the string.

A C-string variable is a partially filled array of characters. Like any other partially filled array, a C-string variable uses positions starting at indexed variable `0` through as many as are needed. However, a C-string variable does not use an *`int`* variable to keep track of how much of the array is currently being used. *Instead, a string variable places the special symbol* `'\0'` *in the array immediately after the last character of the C string.* Thus, if `s` contains the string `"Hi Mom!"`, then the array elements are filled as shown below:

s[0]	s[1]	s[2]	s[3]	s[4]	s[5]	s[6]	s[7]	s[8]	s[9]
H	I		M	o	m	!	\0	?	?

The character `'\0'` is used as a sentinel value to mark the end of the C string. If you read the characters in the C string starting at indexed variable `s[0]`, proceed to `s[1]`, and then to `s[2]`, and so forth, you know that when you encounter the symbol `'\0'`, you have reached the end of the C string. Since the symbol `'\0'` always occupies one element of the array, the length of the longest string that the array can hold is one less than the size of the array.

C-string variables vs. arrays of characters

The thing that distinguishes a C-string variable from an ordinary array of characters is that a C-string variable must contain the null character `'\0'` at the end of the C-string value. This is a distinction in how the array is used rather than a distinction about what the array is. *A C-string variable is an array of characters, but it is used in a different way.*

initializing C-string variables

You can initialize a C-string variable when you declare it, as illustrated by the following example:

```
char my_message[20] = "Hi there.";
```

Notice that the C string assigned to the C-string variable need not fill the entire array.

C-String Variable Declaration

A **C-string variable** is the same thing as an array of characters, but it is used differently. A C-string variable is declared to be an array of characters in the usual way.

Syntax

```
char Array_Name[Maximum_C_string_Size + 1];
```

Example

```
char my_c_string[11];
```

The + 1 allows for the null character '\0', which terminates any C string stored in the array. For example, the C-string variable `my_c_string` in the above example can hold a C string that is ten or fewer characters long.

When you initialize a C-string variable, you can omit the array size. C++ will automatically make the size of the C-string variable 1 more than the length of the quoted string. (The one extra indexed variable is for '\0'.) For example,

```
char short_string[] = "abc";
```

is equivalent to

```
char short_string[4] = "abc";
```

Be sure you do not confuse the following initializations:

```
char short_string[] = "abc";
```

and

```
char short_string[] = {'a', 'b', 'c'};
```

They are *not equivalent*. The first of these two possible initializations places the null character '\0' in the array after the characters 'a', 'b', and 'c'. The second one does not put a '\0' anyplace in the array.

indexed variables for C-string variables

A C-string variable is an array, so it has indexed variables that can be used just like those of any other array. For example, suppose your program contains the following C-string variable declaration:

```
char our_string[5] = "Hi";
```

Initializing a C-String Variable

A C-string variable can be initialized when it is declared, as illustrated by the following example:

```
char your_string[11] = "Do Be Do";
```

Initializing in this way automatically places the null character, '\0', in the array at the end of the C string specified.

If you omit the number inside the square brackets, [], then the C-string variable will be given a size one character longer than the length of the C string. For example, the following declares my_string to have nine indexed variables (eight for the characters of the C string "Do Be Do" and one for the null character '\0'):

```
char my_string[] = "Do Be Do";
```

With our_string declared as above, your program has the following indexed variables: our_string[0], our_string[1], our_string[2], our_string[3], and our_string[4]. For example, the following will change the C-string value in our_string to a C string of the same length consisting of all 'X' characters:

```
int index = 0;
while (our_string[index] != '\0')
{
    our_string[index] = 'X';
    index++;
}
```

Do not destroy the '\0'.

When manipulating these indexed variables, you should be very careful not to replace the null character '\0' with some other value. If the array loses the value '\0', it will no longer behave like a C-string variable. For example, the following will change the array happy_string so that it no longer contains a C string:

```
char happy_string[7] = "DoBeDo";
happy_string[6] = 'Z';
```

After the above code is executed, the array happy_string will still contain the six letters in the C-string "DoBeDo", but happy_string will no longer contain the null character '\0' to mark the end of the C string. Many string-manipulating functions depend critically on the presence of '\0' to mark the end of the C-string value.

As another example, consider the previous *while* loop that changed characters in the C-string variable our_string. That *while* loop changes characters until it encounters a '\0'. If the loop never encounters a '\0', then it could change a large chunk of memory to some unwanted values, which could make your program do strange things. As a safety feature, it would be wise to rewrite that *while* loop as follows, so that if the null character '\0' is lost, the loop will not inadvertently change memory locations beyond the end of the array:

```
int index = 0;
while ( (our_string[index] != '\0') && (index < SIZE) )
{
    our_string[index] = 'X';
    index++;
}
```

SIZE is a defined constant equal to the declared size of the array our_string.

PITFALL Using = and == with C Strings

C-string values and C-string variables are not like values and variables of other data types, and many of the usual operations do not work for C strings. You cannot use a C-string variable in an assignment statement using =. If you use == to test C strings for equality, you will not get the result you expect. The reason for these problems is that C strings and C-string variables are arrays.

assigning a C-string value

Assigning a value to a C-string variable is not as simple as it is for other kinds of variables. The following is illegal:

```
char a_string[10];
a_string = "Hello";  <-- Illegal!
```

Although you can use the equal sign to assign a value to a C-string variable when the variable is declared, you cannot do it anywhere else in your program. Technically, a use of the equal sign in a declaration, as in

```
char happy_string[7] = "DoBeDo";
```

is an initialization, not an assignment. If you want to assign a value to a C-string variable, you must do something else.

There are a number of different ways to assign a value to a C-string variable. The easiest way is to use the predefined function strcpy as shown:

```
strcpy(a_string, "Hello");
```

This will set the value of a_string equal to "Hello". Unfortunately, this version of the function strcpy does not check to make sure the copying does not exceed the size of the string variable that is the first argument.

Many, but not all, versions of C++ also have a safer version of `strcpy`. This safer version is spelled `strncpy` (with an `n`). The function `strncpy` takes a third argument that gives the maximum number of characters to copy. For example:

```
char another_string[10];
strncpy(another_string, a_string_variable, 9);
```

With this `strncpy` function, at most nine characters (leaving room for `'\0'`) will be copied from the C-string variable `a_string_variable`, no matter how long the string in `a_string_variable` may be.

testing C strings for equality

You also cannot use the operator == in an expression to test whether two C strings are the same. (Things are actually much worse than that. You can use == with C strings, but it does not test for the C strings being equal. So if you use == to test two C strings for equality, you are likely to get incorrect results, but no error message!) To test whether two C strings are the same, you can use the predefined function `strcmp`. For example:

```
if (strcmp(c_string1, c_string2))
    cout << "The strings are NOT the same.";
else
    cout << "The strings are the same.";
```

Note that the function `strcmp` works differently than you might guess. The comparison is true if the strings do not match. The function `strcmp` compares the characters in the C-string arguments a character at a time. If at any point the numeric encoding of the character from `c_string1` is less than the numeric encoding of the corresponding character from `c_string2`, the testing stops, and a negative number is returned. If the character from `c_string1` is greater than the character from `c_string2`, then a positive number is returned. (Some implementations of `strcmp` return the difference of the character encodings, but you should not depend on that.) If the C strings are the same, a `0` is returned. The ordering relationship used for comparing characters is called **lexicographic** order. The important point to note is that if both strings are all in uppercase or all in lowercase, then lexicographic order is just alphabetic order.

We see that `strcmp` returns a negative value, a positive value, or zero, depending on whether the C strings compare lexicographically as less, greater, or equal. If you use `strcmp` as a Boolean expression in an *`if`* or a looping statement to test C strings for equality, then the nonzero value will be converted to *`true`* if the strings are different, and the zero will be converted to *`false`*. Be sure that you remember this inverted logic in your testing for C string equality.

C++ compilers that are compliant with the standard have a safer version of `strcmp` that has a third argument that gives the maximum number of characters to compare.

The functions `strcpy` and `strcmp` are in the library with the header file `<cstring>`, so to use them you would insert the following near the top of the file:

```
#include <cstring>
```

The functions `strcpy` and `strcmp` do not require the following or anything similar (although other parts of your program are likely to require it):[1]

```
using namespace std;
```

The `<cstring>` Library

You do not need any `include` directive or *`using`* directive in order to declare and initialize C strings. However, when processing C strings, you inevitably will use some of the predefined string functions in the library `<cstring>`. So, when using C strings, you will normally give the following include directive near the beginning of the file with your code:

```
#include <cstring>
```

Other Functions in `<cstring>`

Display 11.1 contains a few of the most commonly used functions from the library with the header file `<cstring>`. To use them you insert the following near the top of the file:

```
#include <cstring>
```

Like the functions `strcpy` and `strcmp`, all the other functions in `<cstring>` also do not require the following or anything similar (although other parts of your program are likely to require it):[1]

```
using namespace std;
```

We have already discussed `strcpy` and `strcmp`. The function `strlen` is easy to understand and use. For example, `strlen("dobedo")` returns `6` because there are six characters in `"dobedo"`.

[1] If you have not read Chapter 9, you should ignore this footnote. If you have read Chapter 9, the details are as follows: The definitions of `strcpy` and `strcmp`, and all other string functions in `<cstring>` are placed in the global namespace, not in the `std` namespace, and so no *`using`* directive is required.

The function `strcat` is used to concatenate two C strings; that is, to form a longer string by placing the two shorter C strings end-to-end. The first argument must be a C-string variable. The second argument can be anything that evaluates to a C-string value, such as a quoted string. The result is placed in the C-string variable that is the first argument. For example, consider the following:

```
char string_var[20] = "The rain";
strcat(string_var, "in Spain");
```

This code will change the value of `string_var` to `"The rainin Spain"`. As this example illustrates, you need to be careful to account for blanks when concatenating C strings.

If you look at the table in Display 11.1, you will see that safer, three-argument versions of the functions `strcpy`, `strcat`, and `strcmp` are available in many, but not all, versions of C++. Also, note that these three-argument versions are spelled with an added letter `n`: `strncpy`, `strncat`, and `strncmp`.

Display 11.1 Some Predefined C-String Functions in `<cstring>` (*part 1 of 2*)

Function	Description	Cautions
`strcpy(`*Target_String_Var*`,` *Src_String*`)`	Copies the C-string value *Src_String* into the C-string variable *Target_String_Var*.	Does not check to make sure *Target_String_Var* is large enough to hold the value *Src_String*.
`strncpy(`*Target_String_Var*`,` *Src_String*, *Limit*`)`	The same as the two-argument `strcpy` except that at most *Limit* characters are copied.	If *Limit* is chosen carefully, this is safer than the two-argument version of `strcpy`. Not implemented in all versions of C++.
`strcat(`*Target_String_Var*`,` *Src_String*`)`	Concatenates the C-string value *Src_String* onto the end of the C string in the C-string variable *Target_String_Var*.	Does not check to see that *Target_String_Var* is large enough to hold the result of the concatenation.

Display 11.1 Some Predefined C-String Functions in `<cstring>` (*part 2 of 2*)

`strncat(`*Target_String_Var*`,` *Src_String*, *Limit*`)`	The same as the two-argument `strcat` except that at most *Limit* characters are appended.	If *Limit* is chosen carefully, this is safer than the two-argument version of `strcat`. Not implemented in all versions of C++.
`strlen(`*Src_String*`)`	Returns an integer equal to the length of *Src_String*. (The null character, `'\0'`, is not counted in the length.)	
`strcmp(`*String_1*`,` *String_2*`)`	Returns 0 if *String_1* and *String_2* are the same. Returns a value < 0 if *String_1* is less than *String_2*. Returns a value > 0 if *String_1* is greater than *String_2* (that is, returns a nonzero value if *String_1* and *String_2* are different). The order is lexicographic.	If *String_1* equals *String_2*, this function returns 0, which converts to `false`. Note that this is the reverse of what you might expect it to return when the strings are equal.
`strncmp(`*String_1*`,` *String_2*, *Limit*`)`	The same as the two-argument `strcat` except that at most *Limit* characters are compared.	If *Limit* is chosen carefully, this is safer than the two-argument version of `strcmp`. Not implemented in all versions of C++.

.

C-String Arguments and Parameters

A C-string variable is an array, so a C-string parameter to a function is simply an array parameter.

As with any array parameter, whenever a function changes the value of a C-string parameter, it is safest to include an additional *`int`* parameter giving the declared size of the C-string variable.

On the other hand, if a function only uses the value in a C-string argument but does not change that value, then there is no need to include another parameter to give either the declared size of the C-string variable or the amount of the C-string variable array that is filled. The null character `'\0'` can be used to detect the end of the C-string value that is stored in the C-string variable.

SELF-TEST EXERCISES

1 Which of the following declarations are equivalent?

```
char string_var[10] = "Hello";
char string_var[10] = {'H', 'e', 'l', 'l', 'o', '\0'};
char string_var[10] = {'H', 'e', 'l', 'l', 'o'};
char string_var[6] = "Hello";
char string_var[] = "Hello";
```

2 What C string will be stored in `singing_string` after the following code is run?

```
char singing_string[20] = "DoBeDo";
strcat(singing_string, " to you");
```

Assume that the code is embedded in a complete and correct program and that an `include` directive for `<cstring>` is in the program file.

3 What (if anything) is wrong with the following code?

```
char string_var[] = "Hello";
strcat(string_var, " and Good-bye.");
cout << string_var;
```

Assume that the code is embedded in a complete program, and that an `include` directive for `<cstring>` is in the program file.

4 Suppose the function `strlen` (which returns the length of its string argument) was not already defined for you. Give a function definition for `strlen`. Note that `strlen` has only one argument, which is a C string. Do not add additional arguments; they are not needed.

5 What is the length (maximum) of a string that can be placed in the string variable declared by the following declaration? Explain.

```
char s[6];
```

6 How many characters are in each of the following character and string constants?

a. `'\n'`
b. `'n'`
c. `"Mary"`
d. `"M"`
e. `"Mary\n"`

7 Since character strings are just arrays of *char*, why does the text caution you not to confuse the following declaration and initialization?

```
char short_string[] = "abc";
char short_string[] = { 'a', 'b', 'c'};
```

8 Given the following declaration and initialization of the string variable, write a loop to assign 'X' to all positions of this string variable, keeping the length the same.

```
char our_string[15] = "Hi there!";
```

9 Given the declaration of a C-string variable, where SIZE is a defined constant:

```
char our_string[SIZE];
```

The C-string variable our_string has been assigned in code not shown here. For correct C-string variables, the following loop reassigns all positions of our_string the value 'X', leaving the length the same as before. Assume this code fragment is embedded in an otherwise complete and correct program. Answer the questions following this code fragment:

```
int index = 0;
while (our_string[index] != '\0')
{
   our_string[index] = 'X';
   index++;
}
```

a. Explain how this code can destroy the contents of memory beyond the end of the array.
b. Modify this loop to protect against inadvertently changing memory beyond the end of the array.

10 Write code using a library function to copy the string constant "Hello" into the string variable declared below. Be sure to #include the necessary header file to get the declaration of the function you use.

```
char a_string[10];
```

11 What string will be output when this code is run? (Assume, as always, that this code is embedded in a complete, correct program.)

```
char song[10] = "I did it ";
char franks_song[20];
strcpy ( franks_song, song );
strcat ( franks_song, "my way!");
cout << franks_song << endl;
```

12 What is the problem (if any) with this code?

```
char a_string[20] = "How are you? ";
strcat(a_string, "Good, I hope.");
```

C-String Input and Output

C strings can be output using the insertion operator <<. In fact, we have already been doing so with quoted strings. You can use a C-string variable in the same way. For example,

```
cout << news << " Wow.\n";
```

where `news` is a C-string variable.

It is possible to fill a C-string variable using the input operator >>, but there is one thing to keep in mind. As for all other types of data, all whitespace (blanks, tabs, and line breaks) are skipped when C strings are read this way. Moreover, each reading of input stops at the next space or line break. For example, consider the following code:

```
char a[80], b[80];
cout << "Enter some input:\n";
cin >> a >> b;
cout << a << b << "END OF OUTPUT\n";
```

When embedded in a complete program, this code produces a dialogue like the following:

```
Enter some input:
Do be do to you!
DobeEND OF OUTPUT
```

The C-string variables `a` and `b` each receive only one word of the input: `a` receives the C-string value `"Do"` because the input character following **`Do`** is a blank; `b` receives `"be"` because the input character following **`be`** is a blank.

If you want your program to read an entire line of input, you can use the extraction operator >> to read the line one word at a time. This can be tedious and it still will not read the blanks in the line. There is an easy way to read an entire line of input and place the resulting C string into a C-string variable: Just use the predefined member function `getline`, which is a member function of every input stream (such as `cin` or a file input stream). The function `getline` has two arguments. The first argument is a C-string variable to receive the input and the second is an integer that typically is the declared size of the C-string variable. The second argument tells the

getline

maximum number of array elements in the C-string variable that `getline` will be allowed to fill with characters. For example, consider the following code:

```
char a[80];
cout << "Enter some input:\n";
cin.getline(a, 80);
cout << a << "END OF OUTPUT\n";
```

When embedded in a complete program, this code produces a dialogue like the following:

```
Enter some input:
Do be do to you!
Do be do to you!END OF OUTPUT
```

With the function `cin.getline`, the entire line is read. The reading ends when the line ends, even though the resulting C string may be shorter than the maximum number of characters specified by the second argument.

When `getline` is executed, the reading stops after the number of characters given by the second argument have been filled in the C-string array, even if the end of the line has not been reached. For example, consider the following code:

```
char short_string[5];
cout << "Enter some input:\n";
cin.getline(short_string, 5);
cout << short_string << "END OF OUTPUT\n";
```

When embedded in a complete program, this code produces a dialogue like the following:

```
Enter some input:
dobedowap
dobeEND OF OUTPUT
```

Notice that four, not five, characters are read into the C-string variable `short_string`, even though the second argument is `5`. This is because the null character `'\0'` fills one array position. Every C string is terminated with the null character when it is stored in a C-string variable, and this always consumes one array position.

input/output with files

The C-string input and output techniques we illustrated for `cout` and `cin` work the same way for input and output with files. The input stream `cin` can be replaced by an input stream that is connected to a file. The output stream `cout` can be replaced by an output stream that is connected to a file. (File I/O is discussed in Chapter 5.)

getline

The member function `getline` can be used to read a line of input and place the C string of characters on that line into a C-string variable.

Syntax

```
cin.getline(String_Var, Max_Characters + 1);
```

One line of input is read from the stream *Input_Stream*, and the resulting C string is placed in *String_Var*. If the line is more than *Max_Characters* long, then only the first *Max_Characters* on the line are read. (The +1 is needed because every C string has the null character `'\0'` added to the end of the C string and so the string stored in *String_Var* is one longer than the number of characters read in.)

Example

```
char one_line[80];
cin.getline(one_line, 80);
```

(You can use an input stream connected to a text file in place of `cin`.)

SELF-TEST EXERCISES

13 Consider the following code (and assume it is embedded in a complete and correct program and then run):

```
char a[80], b[80];
cout << "Enter some input:\n";
cin >> a >> b;
cout << a << '-' << b << "END OF OUTPUT\n";
```

If the dialogue begins as follows, what will be the next line of output?

```
Enter some input:

The
   time is now.
```

14 Consider the following code (and assume it is embedded in a complete and correct program and then run):

```
char my_string[80];
cout << "Enter a line of input:\n";
cin.getline(my_string, 6);
cout << my_string << "<END OF OUTPUT";
```

If the dialogue begins as follows, what will be the next line of output?

```
Enter a line of input:
May the hair on your toes grow long and curly.
```

C-String-to-Number Conversions and Robust Input

The C string `"1234"` and the number `1234` are not the same things. The first is a sequence of characters; the second is a number. In everyday life, we write them the same way and blur this distinction, but in a C++ program this distinction cannot be ignored. If you want to do arithmetic, you need `1234`, not `"1234"`. If you want to add a comma to the numeral for one thousand two hundred thirty four, then you want to change the C string `"1234"` to the C string `"1,234"`. When designing numeric input, it is often useful to read the input as a string of characters, edit the string, and then convert the string to a number. For example, if you want your program to read an amount of money, the input may or may not begin with a dollar sign. If your program is reading percentages, the input may or may not have a percent sign at the end. If your program reads the input as a string of characters, it can store the string in a C-string variable and remove any unwanted characters, leaving only a C string of digits. Your program then needs to convert this C string of digits to a number, which can easily be done with the predefined function `atoi`.

`atoi`

The function `atoi` takes one argument that is a C string and returns the *`int`* value that corresponds to that C string. For example, `atoi("1234")` returns the integer `1234`. If the argument does not correspond to an *`int`* value, then `atoi` returns `0`. For example, `atoi("#37")` returns `0`, because the character `'#'` is not a digit. You pronounce `atoi` as "A to I," which is an abbreviation of "alphabetic to integer." The function `atoi` is in the library with header file `cstdlib`, so any program that uses it must contain the following directive:

```
#include <cstdlib>
```

`atol`

If your numbers are too large to be values of type *`int`*, you can convert them from C strings to values of type *`long`*. The function `atol` performs the same conversion as the function `atoi` except that `atol` returns values of type *`long`* and thus can accommodate larger integer values (on systems where this is a concern).

`read_and_clean`

Display 11.2 contains the definition of a function called `read_and_clean` that reads a line of input and discards all characters other than the digits `'0'` through `'9'`. The function then uses the function `atoi` to convert the "cleaned up" C string of digits to an integer value. As the demonstration program indicates, you can use this function to read money amounts and it will not matter whether the user included a dollar sign or not. Similarly, you can read percentages and it will not matter whether the user types in a percent sign or not. Although the output makes it look as if the function `read_and_clean` simply removes some symbols, more than that is happening. The value produced is a true *`int`* value that can be used in a program as a number; it is not a C string of characters.

The function `read_and_clean` shown in Display 11.2 will delete any nondigits from the string typed in, but it cannot check that the remaining digits will yield the number the user has in mind. The user should be given a chance to look at the final value and see whether it is correct. If the value is not correct, the user should be given a chance to reenter the input. In Display 11.3 we have used the function `read_and_clean` in another function called `get_int`, which will accept anything the user types and will allow the user to reenter the input until she or he is satisfied with the number that is computed from the input string. It is a very robust input procedure. (The function `get_int` is an improved version of the function of the same name given in Display 5.7.)

`get_int`

The functions `read_and_clean` in Display 11.2 and `get_int` in Display 11.3 are samples of the various input functions you can design by reading numeric input as a string value. Programming Project 3 at the end of this chapter asks you to define a function similar to `get_int` that reads in a number of type *`double`*, as opposed to a number of type *`int`*. To write that function, it would be nice to have a predefined function that converts a string value to a number of type *`double`*. Fortunately, the predefined function `atof`, which is also in the library with header file `cstdlib`, does

`atof`

C-String-to-Number Functions

The functions `atoi`, `atol`, and `atof` can be used to convert a C string of digits to the corresponding numeric value. The functions `atoi` and `atol` convert C strings to integers. The only difference between `atoi` and `atol` is that `atoi` returns a value of type *`int`* while `atol` returns a value of type *`long`*. The function `atof` converts a C string to a value of type *`double`*. If the C-string argument (to either function) is such that the conversion cannot be made, then the function returns zero. For example

```
int x = atoi("657");
```

sets the value of x to 657, and

```
double y = atof("12.37");
```

sets the value of y to `12.37`.

Any program that uses `atoi` or `atof` must contain the following directive:

```
#include <cstdlib>
```

Display 11.2 C Strings to Integers (*part 1 of 2*)

```
//Demonstrates the function read_and_clean.
#include <iostream>
#include <cstdlib>
#include <cctype>

void read_and_clean(int& n);
//Reads a line of input. Discards all symbols except the digits. Converts
//the C string to an integer and sets n equal to the value of this integer.

void new_line( );
//Discards all the input remaining on the current input line.
//Also discards the '\n' at the end of the line.

int main( )
{
    using namespace std;
    int n;
    char ans;
    do
    {
        cout << "Enter an integer and press Return: ";
        read_and_clean(n);
        cout << "That string converts to the integer " << n << endl;
        cout << "Again? (yes/no): ";
        cin >> ans;
        new_line( );
    } while ( (ans != 'n') && (ans != 'N') );
    return 0;
}
```

Display 11.2 C-Strings to Integers (*part 2 of 2*)

```
//Uses iostream, cstdlib, and cctype:
void read_and_clean(int& n)
{
    using namespace std;
    const int ARRAY_SIZE = 6;
    char digit_string[ARRAY_SIZE];

    char next;
    cin.get(next);
    int index = 0;
    while (next != '\n')
    {
        if ( (isdigit(next)) && (index < ARRAY_SIZE - 1) )
        {
            digit_string[index] = next;
            index++;
        }
        cin.get(next);
    }
    digit_string[index] = '\0';

    n = atoi(digit_string);
}

//Uses iostream:
void new_line( )
 {
    using namespace std;
    <The rest of the definition of new_line is given in Display 5.7.>
```

Sample Dialogue

```
Enter an integer and press Return: $ 100
That string converts to the integer 100
Again? (yes/no): yes
Enter an integer and press Return: 100
That string converts to the integer 100
Again? (yes/no): yes
Enter an integer and press Return: 99%
That string converts to the integer 99
Again? (yes/no): yes
Enter an integer and press Return: 23% &&5 *12
That string converts to the integer 23512
Again? (yes/no): no
```

Display 11.3 Robust Input Function (*part 1 of 2*)

```
//Demonstration program for improved version of get_int.
#include <iostream>
#include <cstdlib>
#include <cctype>

void read_and_clean(int& n);
//Reads a line of input. Discards all symbols except the digits. Converts
//the C string to an integer and sets n equal to the value of this integer.

void new_line( );
//Discards all the input remaining on the current input line.
//Also discards the '\n' at the end of the line.

void get_int(int& input_number);
//Gives input_number a value that the user approves of.

int main( )
{
    using namespace std;
    int input_number;
    get_int(input_number);
    cout << "Final value read in = " << input_number << endl;
    return 0;
}

//Uses iostream and read_and_clean:
void get_int(int& input_number)
{
    using namespace std;
    char ans;
    do
    {
        cout << "Enter input number: ";
        read_and_clean(input_number);
        cout << "You entered " << input_number
             << " Is that correct? (yes/no): ";
        cin >> ans;
        new_line( );
    } while ((ans != 'y') && (ans != 'Y'));
}
```

Display 11.3 Robust Input Function (*part 2 of 2*)

```
//Uses iostream, cstdlib, and cctype:
void read_and_clean(int& n)
```

<The rest of the definition of `read_and_clean` is given in Display 11.2.>

```
//Uses iostream:
void new_line( )
```

<The rest of the definition of `new_line` is given in Display 11.2.>

Sample Dialogue

```
Enter input number: $57
You entered 57 Is that correct? (yes/no): no
Enter input number: $77*5xa
You entered 775 Is that correct? (yes/no): no
Enter input number: 77
You entered 77 Is that correct? (yes/no): no
Enter input number: $75
You entered 75 Is that correct? (yes/no): yes
Final value read in = 75
```

just that. For example, `atof("9.99")` returns the value `9.99` of type *`double`*. If the argument does not correspond to a number of type *`double`*, then `atof` returns `0.0`. You pronounce `atof` as "A to F," which is an abbreviation of "alphabetic to floating point." Recall that numbers with a decimal point are often called *floating-point* numbers because of the way the computer handles the decimal point when storing these numbers in memory.

11.2 The Standard `string` Class

I try to catch every sentence, every word you and I say, and quickly lock all these sentences and words away in my literary storehouse because they might come in handy.

ANTON CHEKHOV, *THE SEAGULL*

In Section 11.1 we introduced C strings. These C strings were simply arrays of characters terminated with the null character `'\0'`. In order to manipulate these C strings, you needed to worry about all the details of handling arrays. For example, when you want to add characters to a C string and there is not enough room in the array, you must create another array to hold this longer string of characters. In short, C strings require the programmer to keep track of all the low-level details of how the C strings are stored in memory. This is a lot of extra work and a source of programmer errors. The latest ANSI/ISO standard for C++ specified that C++ must now also have a class `string` that allows the programmer to treat strings as a basic data type without needing to worry about implementation details. In this section we introduce you to this `string` type.

Introduction to the Standard Class `string`

The class `string` is defined in the library whose name is also `<string>`, and the definitions are placed in the `std` namespace. So, in order to use the class `string`, your code must contain the following (or something more or less equivalent):

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

The class `string` allows you to treat `string` values and `string` expressions very much like values of a simple type. You can use the = operator to assign a value to a `string` variable, and you can use the + sign to concatenate two strings. For example, suppose `s1`, `s2`, and `s3` are objects of type `string` and both `s1` and `s2` have string values. Then `s3` can be set equal to the concatenation of the string value in `s1` followed by the string value in `s2` as follows:

+ does concatenation

```
s3 = s1 + s2;
```

There is no danger of s3 being too small for its new string value. If the sum of the lengths of `s1` and `s2` exceeds the capacity of `s3`, then more space is automatically allocated for `s3`.

As we noted earlier in this chapter, quoted strings are really C strings and so they are not literally of type `string`. However, C++ provides automatic type casting of quoted strings to values of type `string`. So, you can use quoted strings as if they were literal values of type `string`, and we (and most others) will often refer to quoted strings as if they were values of type `string`. For example,

```
s3 = "Hello Mom!";
```

sets the value of the `string` variable `s3` to a `string` object with the same characters as in the C string `"Hello Mom!"`.

constructors

The class `string` has a default constructor that initializes a `string` object to the empty string. The class `string` also has a second constructor that takes one argument that is a standard C string and so can be a quoted string. This second constructor initializes the `string` object to a value that represents the same string as its C-string argument. For example,

```
string phrase;
string noun("ants");
```

The first line declares the string variable `phrase` and initializes it to the empty string. The second line declares `noun` to be of type string and initializes it to a string value equivalent to the C string `"ants"`. Most programmers when talking loosely would say that "`noun` is initialized to `"ants"`," but there really is a type conversion here. The quoted string `"ants"` is a C string, not a value of type `string`. The variable `noun` receives a `string` value that has the same characters as `"ants"` in the same order as `"ants"`, but the `string` value is not terminated with the null character `'\0'`. In fact, in theory at least, you do not know or care whether the `string` value of `noun` is even stored in an array, as opposed to some other data structure.

There is an alternate notation for declaring a `string` variable and invoking a constructor. The following two lines are exactly equivalent:

```
string noun("ants");
string noun = "ants";
```

These basic details about the class `string` are illustrated in Display 11.4. Note that, as illustrated there, you can output `string` values using the operator <<.

Consider the following line from Display 11.4:

```
phrase = "I love " + adjective + " " + noun + "!";
```

converting C-string constants to the type `string`

C++ must do a lot of work to allow you to concatenate strings in this simple and natural fashion. The string constant `"I love "` is not an object of type `string`. A string constant like `"I love "` is stored as a C string (in other words, as a null-terminated array of characters). When C++ sees `"I love "` as an argument to +, it finds the definition (or overloading) of + that applies to a value such as `"I love "`. There are overloadings of the + operator that have a C string on the left and a `string` on the right, as well as the reverse of this positioning. There is even a version that has a C string on both sides of the + and produces a `string` object as the value returned. Of course, there is also the overloading you expect, with the type `string` for both operands.

C++ did not really need to provide all those overloading cases for +. If these overloadings were not provided, C++ would look for a constructor that could perform a type conversion to convert the C string `"I love "` to a value for which + did apply. In this case, the constructor with the one C-string parameter would perform just such a conversion. However, the extra overloadings are presumably more efficient.

The class `string` is often thought of as a modern replacement for C strings. However, in C++ you cannot easily avoid also using C strings when you program with the class `string`.

Display 11.4 Program Using the Class `string`

```
//Demonstrates the standard class string.
#include <iostream>
#include <string>
using namespace std;

int main( )
{
    string phrase;
    string adjective("fried"), noun("ants");
    string wish = "Bon appetite!";

    phrase = "I love " + adjective + " " + noun + "!";
    cout << phrase << endl
         << wish << endl;

    return 0;
}
```

Initialized to the empty string

Two ways of initializing a string variable

Sample Dialogue

```
I love fried ants!
Bon appetite!
```

I/O with the Class `string`

You can use the insertion operator << and `cout` to output `string` objects just as you do for data of other types. This is illustrated in Display 11.4. Input with the class `string` is a bit more subtle.

The extraction operator >> and `cin` works the same for `string` objects as for other data, but remember that the extraction operator ignores initial whitespace and stops reading when it encounters more whitespace. This is as true for strings as it is for other data. For example, consider the following code;

```
string s1, s2;
cin >> s1;
cin >> s2;
```

The Class `string`

The class `string` can be used to represents values that are strings of characters. The class `string` provides more versatile string representation than the C strings discussed in Section 11.1.

The class `string` is defined in the library that is also named `<string>`, and its definition is placed in the `std` namespace. So, programs that use the class `string` should contain the following (or something more or less equivalent):

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

The class `string` has a default constructor that initializes the `string` object to the empty string and a constructor that takes a C string as an argument and initializes the `string` object to a value that represents the string given as the argument. For example:

```
string s1, s2("Hello");
```

If the user types in

```
May the hair on your toes grow long and curly!
```

then `s1` will receive the value `"May"` with any leading (or trailing) whitespace deleted. The variable `s2` receives the string `"the"`. Using the extraction operator `>>` and `cin`, you can only read in words; you cannot read in a line or other string that contains a blank. Sometimes this is exactly what you want, but sometimes it is not at all what you want.

`getline`

If you want your program to read an entire line of input into a variable of type string, you can use the function `getline`. The syntax for using `getline` with `string` objects is a bit different from what we described for C strings in Section 11.1. You do not use `cin.getline`; instead, you make `cin` the first argument to `getline`.[2] (Thus, this version of `getline` is not a member function.)

```
string line;
cout << "Enter a line of input:\n";
getline(cin, line);
cout << line << "END OF OUTPUT\n";
```

[2] This is a bit ironic, since the class `string` was designed using more modern object-oriented techniques, and the notation it uses for `getline` is the old fashioned, less object-oriented notation. This is an accident of history. This `getline` function was defined after the `iostream` library was already in use, so the designers had little choice but to make this `getline` a standalone function.

When embedded in a complete program, this code produces a dialogue like the following:

```
Enter some input:
Do be do to you!
Do be do to you!END OF OUTPUT
```

If there were leading or training blanks on the line, then they too would be part of the string value read by `getline`. This version of `getline` is in the library `<string>`. You can use a stream object connected to a text file in place of `cin` to do input from a file using `getline`.

You cannot use `cin` and `>>` to read in a blank character. If you want to read one character at a time, you can use `cin.get`, which we discussed in Chapter 5. The function `cin.get` reads values of type *`char`*, not of type `string`, but it can be helpful when handling `string` input. Display 11.5 contains a program that illustrates both `getline` and `cin.get` used for `string` input. The significance of the function `new_line` is explained in the Pitfall subsection entitled "Mixing `cin >> variable;` and `getline`."

SELF-TEST EXERCISES

15 Consider the following code (and assume that it is embedded in a complete and correct program and then run):

```
string s1, s2;
cout << "Enter a line of input:\n";
cin >> s1 >> s2;
cout << s1 << "*" << s2 << "<END OF OUTPUT";
```

If the dialogue begins as follows, what will be the next line of output?

```
Enter a line of input:
A string is a joy forever!
```

16 Consider the following code (and assume that it is embedded in a complete and correct program and then run):

```
string s;
cout << "Enter a line of input:\n";
getline(cin, s);
cout << s << "<END OF OUTPUT";
```

If the dialogue begins as follows, what will be the next line of output?

```
Enter a line of input:
A string is a joy forever!
```

Display 11.5 Program Using the Class `string` (*part 1 of 2*)

```
//Demonstrates getline and cin.get.
#include <iostream>
#include <string>

void new_line( );

int main( )
{
    using namespace std;

    string first_name, last_name, record_name;
    string motto = "Your records are our records.";

    cout << "Enter your first and last name:\n";
    cin >> first_name >> last_name;
    new_line( );

    record_name = last_name + ", " + first_name;
    cout << "Your name in our records is: ";
    cout << record_name << endl;

    cout << "Our motto is\n"
         << motto << endl;
    cout << "Please suggest a better (one-line) motto:\n";
    getline(cin, motto);
    cout << "Our new motto will be:\n";
    cout << motto << endl;

    return 0;
}
```

Display 11.5 Program Using the Class string (*part 2 of 2*)

```
//Uses iostream:
void new_line( )
{
    using namespace std;

    char next_char;
    do
    {
        cin.get(next_char);
    } while (next_char != '\n');
}
```

Sample Dialogue

```
Enter your first and last name:
  B'Elanna Torres
Your name in our records is: Torres, B'Elanna
Our motto is
Your records are our records.
Please suggest a better (one-line) motto:
Our records go where no records dared to go before.
Our new motto will be:
Our records go where no records dared to go before.
```

I/O with string Objects

You can use the insertion operator << with cout to output string objects. You can input a string with the extraction operator >> and cin. When using >> for input, the code reads in a string delimited with whitespace. You can use the function getline to input an entire line of text into a string object.

Examples

```
string greeting("Hello"), response, next_word;
cout << greeting << endl;
getline(cin, response);
cin >> next_word;
```

Programming TIP More Versions of getline

So far, we have described the following way of using `getline`:

```
string line;
cout << "Enter a line of input:\n";
getline(cin, line);
```

This version stops reading when it encounters the end-of-line marker `'\n'`. There is a version that allows you to specify a a different character to use as a stopping signal. For example, the following will stop when the first question mark is encountered:

```
string line;
cout << "Enter some input:\n";
getline(cin, line, '?');
```

It makes sense to use `getline` as if it were a *void* function, but it actually returns a reference to its first argument, which is `cin` in the above code. Thus, the following will read in a line of text into `s1` and a string of nonwhitespace characters into `s2`:

```
string s1, s2;
getline(cin, s1) >> s2;
```

The invocation `getline(cin, s1)` returns a reference to `cin`, so that after the invocation of `getline`, the next thing to happen is equivalent to

```
cin >> s2;
```

This kind of use of `getline` seems to have been designed for use in a C++ quiz show rather than to meet any actual programming need, but it can come in handy sometimes.

PITFALL Mixing cin >> variable; and getline

Take care in mixing input using `cin >> variable;` with input using `getline`. For example, consider the following code:

```
int n;
string line;
cin >> n;
getline(cin, line);
```

getline for Objects of the Class string

The getline function for string objects has two versions:

```
istream& getline(istream& ins, string& str_var,
                               char delimiter);
```

and

```
istream& getline(istream& ins, string& str_var);
```

The first version of this function reads characters from the istream object given as the first argument (always cin in this chapter), inserting the characters into the string variable str_var until an instance of the delimiter character is encountered. The delimiter character is removed from the input and discarded. The second version uses '\n' for the default value of delimiter; otherwise, it works the same.

These getline functions return their first argument (always cin in this chapter), but they are usually used as if they were *void* functions.

When this code reads the following input, you might expect the value of n to be set to 42 and the value of line to be set to a string value representing "Hello hitchhiker.":

```
42
Hello hitchhiker.
```

However, while n is indeed set to the value of 42, line is set equal to the empty string. What happened?

Using cin >> n skips leading whitespace on the input, but leaves the rest of the line, in this case just '\n', for the next input. A statement like

```
cin >> n;
```

always leaves something on the line for a following getline to read (even if it is just the '\n'). In this case, the getline see the '\n' and stops reading, so getline reads an empty string. If you find your program appearing to mysteriously ignore input data, see if you have mixed these two kinds of input. You may need to use either the new_line function from Display 11.5 or the function ignore from the library iostream. For example,

```
cin.ignore(1000, '\n');
```

With these arguments, a call to the ignore member function will read and discard the entire rest of the line up to and including the '\n' (or until it discards 1,000 characters if it does not find the end of the line after 1,000 characters).

There can be other baffling problems with programs that use cin with both >> and getline. Moreover, these problems can come and go as you move from one C++ compiler to another. When all else fails, or if you want to be certain of portability, you can resort to character-by-character input using cin.get.

These problems can occur with any of the versions of getline that we discuss in this chapter.

String Processing with the Class string

The class string allows you to perform the same operations that you can perform with the C strings we discussed in Section 11.1 and more.

You can access the characters in a string object in the same way that you access array elements, so string objects have all the advantages of arrays of characters plus a number of advantages that arrays do not have, such as automatically increasing their capacity.

If last_name is the name of a string object, then last_name[i] gives access to the ith character in the string represented by last_name. This use of array square brackets is illustrated in Display 11.6.

length

Display 11.6 also illustrates the member function length. Every string object has a member function named length that takes no arguments and returns the length of the string represented by the string object. Thus, a string object not only can be used like an array, but the length member function makes it behave like a partially filled array that automatically keeps track of how many positions are occupied.

When used with an object of the class string, the array square brackets do not check for illegal indexes. If you use an illegal index (that is, an index that is greater than or equal to the length of the string in the object), then the results are unpredictable but are bound to be bad. You may just get strange behavior without any error message that tells you that the problem is an illegal index value.

There is a member function named at that does check for illegal index values. The member function named at behaves basically the same as the square brackets, except for two points: You use function notation with at, so instead of a[i], you use a.at(i); and the at member function checks to see if i evaluates to an illegal index. If the value of i in a.at(i) is an illegal index, then you should get a run-time error message telling you what is wrong. In the following two example code fragments, the attempted access is out of range, yet the first of these probably will

Display 11.6 A string Object Can Behave Like an Array

```
//Demonstrates using a string object as if it were an array.
#include <iostream>
#include <string>
using namespace std;

int main( )
{
    string first_name, last_name;

    cout << "Enter your first and last name:\n";
    cin >> first_name >> last_name;

    cout << "Your last name is spelled:\n";
    int i;
    for (i = 0; i < last_name.length( ); i++)
    {
        cout << last_name[i] << " ";
        last_name[i] = '-';
    }
    cout << endl;
    for (i = 0; i < last_name.length( ); i++)
        cout << last_name[i] << " "; //Places a "-" under each letter.
    cout << endl;

    cout << "Good day " << first_name << endl;
    return 0;
}
```

Sample Dialogue

```
Enter your first and last name:
John Crichton
Your last name is spelled:
C r i c h t o n
- - - - - - - -
Good day John
```

not produce an error message, although it will be accessing a nonexistent indexed variable:

```
string str("Mary");
cout << str[6] << endl;
```

The second example, however, will cause the program to terminate abnormally, so you at least know that something is wrong:

```
string str("Mary");
cout << str.at(6) << endl;
```

But be warned that some systems give very poor error messages when `a.at(i)` has an illegal index `i`.

You can change a single character in the string by assigning a *`char`* value to the indexed variable, such as `str[i]`. This may also be done with the member function `at`. For example, to change the third character in the `string` object `str` to `'X'`, you can use either of the following code fragments:

```
str.at(2)='X';
```

or

```
str[2]='X';
```

As in an ordinary array of characters, character positions for objects of type `string` are indexed starting with `0`, so the third character in a string is in index position `2`.

Display 11.7 gives a partial list of the member functions of the class `string`.

In many ways objects of the class `string` are better behaved than the C strings we introduced in Section 11.1. In particular, the == operator on objects of the `string` class returns a result that corresponds to our intuitive notion of strings being equal—namely, it returns *`true`* if the two string contain the same characters in the same order, and returns *`false`* otherwise. Similarly, the comparison operators <, >, <=, >= compare string objects using lexicographic ordering. (Lexicographic ordering is alphabetic ordering using the order of symbols given in the ASCII character set in Appendix 3. If the strings consist of all letters and are both either all uppercase or all lowercase letters, then for this case lexicographic ordering is the same as everyday alphabetical ordering.)

= and == Are Different for `strings` and C Strings

The operators =, ==, !=, <, >, <=, >=, when used with the standard C++ type `string`, produce results that correspond to our intuitive notion of how strings compare. They do not misbehave as they do with the C strings, as we discussed in Section 11.1.

Display 11.7 Member Functions of the Standard Class string

Example	**Remarks**
Constructors	
`string str;`	Default constructor creates empty `string` object `str`.
`string str("sample");`	Creates a `string` object with data `"sample"`.
`string str(a_string);`	Creates a `string` object `str` that is a copy of `a_string`; `a_string` is an object of the class `string`.
Element access	
`str[i]`	Returns read/write reference to character in `str` at index `i`. Does not check for illegal index.
`str.at(i)`	Returns read/write reference to character in `str` at index `i`. Same as `str[i]`, but this version checks for illegal index.
`str.substr(position, length)`	Returns the substring of the calling object starting at `position` and having `length` characters.
Assignment/modifiers	
`str1 = str2;`	Initializes `str1` to `str2`'s data,
`str1 += str2;`	Character data of `str2` is concatenated to the end of `str1`.
`str.empty( )`	Returns *`true`* if `str` is an empty string; *`false`* otherwise.
`str1 + str2`	Returns a string that has `str2`'s data concatenated to the end of `str1`'s data.
`str.insert(pos, str2);`	Inserts `str2` into `str` beginning at position pos.
`str.remove(pos, length);`	Removes substring of size `length`, starting at position pos.
Comparison	
`str1 == str2  str1 != str2`	Compare for equality or inequality; returns a Boolean value.

`str1 < str2   str1 > str2`
`str1 <= str2  str1 >= str2`
	Four comparisons. All are lexicographical comparisons.

Finds	
`str.find(str1)`	Returns index of the first occurrence of `str1` in `str`.
`str.find(str1, pos)`	Returns index of the first occurrence of string `str1` in `str`; the search starts at position pos.
`str.find_first_of(str1, pos)`	Returns the index of the first instance in `str` of any character in `str1`, starting the search at position pos.

`str.find_first_not_of`
`(str1, pos)`
	Returns the index of the first instance in `str` of any character not in `str1`, starting the search at position pos.

Programming EXAMPLE
Palindrome Testing

A palindrome is a string that reads the same front to back as it does back to front. The program in Display 11.8 tests an input string to see if it is a palindrome. Our palindrome test will disregard all spaces and punctuations and will consider upper- and lowercase versions of a letter to be the same when deciding if something is a palindrome. Some palindrome examples are as follows:

```
Able was I ere I saw Elba.
I Love Me, Vol. I.
Madam, I'm Adam.
A man, a plan, a canal, Panama.
Rats live on no evil star.
radar
deed
mom
racecar
```

The `remove_punct` function is of interest in that it uses the `string` member functions `substr` and `find`. The member function `substr` extracts a substring of the calling object, given the position and length of the desired substring. The first three lines of `remove_punct` declare variables for use in the function. The *`for`* loop runs through the characters of the parameter `s` one at a time and tries to `find` them in the `punct` string. To do this, a `string` that is the substring of `s`, of length `1` at each character position, is extracted. The position of this substring in the `punct` string is determined using the `find` member function. If this one-character `string` is not in the `punct` string, then the one-character `string` is concatenated to the `no_punct` string that is to be returned.

SELF-TEST EXERCISES

17 Consider the following code:

```
string s1, s2("Hello");
cout << "Enter a line of input:\n";
cin >> s1;
if (s1 == s2)
    cout << "Equal\n";
else
    cout << "Not equal\n";
```

If the dialogue begins as follows, what will be the next line of output?

```
Enter a line of input:
Hello friend!
```

Display 11.8 Palindrome Testing Program (*part 1 of 4*)

```
//Test for palindrome property.
#include <iostream>
#include <string>
#include <cctype>
using namespace std;

void swap(char& v1, char& v2);
//Interchanges the values of v1 and v2.

string reverse(const string& s);
//Returns a copy of s but with characters in reverse order.

string remove_punct(const string& s, const string& punct);
//Returns a copy of s with any occurrences of characters
//in the string punct removed.

string make_lower(const string& s);
//Returns a copy of s that has all uppercase
//characters changed to lowercase, other characters unchanged.

bool is_pal(const string& s);
//Returns true if s is a palindrome, false otherwise.

int main( )
{
    string str;
    cout << "Enter a candidate for palindrome test\n"
         << "followed by pressing Return.\n";
    getline(cin, str);

    if (is_pal(str))
         cout << "\"" << str + "\" is a palindrome.";
    else
         cout << "\"" << str + "\" is not a palindrome.";
    cout << endl;

    return 0;
}
```

Display 11.8 Palindrome Testing Program (*part 2 of 4*)

```
void swap(char& v1, char& v2)
{
    char temp = v1;
    v1 = v2;
    v2 = temp;
}


string reverse(const string& s)
{
    int start = 0;
    int end = s.length( );
    string temp(s);

    while (start < end)
    {
       end--;
       swap(temp[start], temp[end]);
       start++;
    }

    return temp;
}


//Uses <cctype> and <string>
string make_lower(const string& s)
{
    string temp(s);
    for (int i = 0; i < s.length( ); i++)
        temp[i] = tolower(s[i]);

    return temp;
}
```

Display 11.8 Palindrome Testing Program (*part 3 of 4*)

```
string remove_punct(const string& s, const string& punct)
{
    string no_punct; //initialized to empty string
    int s_length = s.length( );
    int punct_length = punct.length( );

    for (int i = 0; i < s_length; i++)
    {
         string a_char = s.substr(i,1); //A one-character string
         int location = punct.find(a_char, 0);
         //Find location of successive characters
         //of src in punct.

      if (location < 0 || location >= punct_length)
         no_punct = no_punct + a_char; //a_char not in punct, so keep it
    }

    return no_punct;
}


//uses functions make_lower, remove_punct.
bool is_pal(const string& s)
{
    string punct(",;:.?!'\" "); //includes a blank
    string str(s);
    str = make_lower(str);
    string lower_str = remove_punct(str, punct);

    return (lower_str == reverse(lower_str));
}
```

Display 11.8 Palindrome Testing Program (*part 4 of 4*)

Sample Dialogues

```
Enter a candidate for palindrome test
followed by pressing Return.
Madam, I'm Adam.
"Madam, I'm Adam." is a palindrome.
```

```
Enter a candidate for palindrome test
followed by pressing Return.
Radar
"Radar" is a palindrome.
```

```
Enter a candidate for palindrome test
followed by pressing Return.
Am I a palindrome?
"Am I a palindrome?" is not a palindrome.
```

18 What is the output produced by the following code?

```
string s1, s2("Hello");
s1 = s2;
s2[0] = 'J';
cout << s1 << " " << s2;
```

Converting between string Objects and C Strings

You have already seen that C++ will perform an automatic type conversion to allow you to store a C string in a variable of type `string`. For example, the following will work fine:

```
char a_c_string[] = "This is my C string.";
string string_variable;
string_variable = a_c_string;
```

However, the following will produce a compiler error message:

```
a_c_string = string_variable; //ILLEGAL
```

The following is also illegal:

```
strcpy(a_c_string, string_variable); //ILLEGAL
```

`strcpy` cannot take a `string` object as its second argument, and there is no automatic conversion of `string` objects to C strings, which is the problem we cannot seem to get away from.

To obtain the C string corresponding to a `string` object, you must perform an explicit conversion. This can be done with the `string` member function `c_str( )`. The correct version of the copying we have been trying to do is the following:

`c_str( )`

```
strcpy(a_c_string, string_variable.c_str( )); //Legal;
```

Note that you need to use the `strcpy` function to do the copying. The member function `c_str( )` returns the C string corresponding to the `string` calling object. As we noted earlier in this chapter, the assignment operator does not work with C strings. So, just in case you thought the following might work, we should point out that it too is illegal.

```
a_c_string = string_variable.c_str( ); //ILLEGAL
```

11.3 Vectors

"Well, I'll eat it," said Alice, "and if it makes me grow larger, I can reach the key; and if it makes me grow smaller, I can creep under the door; so either way I'll get into the garden. . . ."

LEWIS CARROLL, *ALICE'S ADVENTURES IN WONDERLAND*

Vectors can be thought of as arrays that can grow (and shrink) in length while your program is running. In C++, once your program creates an array, it cannot change the length of the array. Vectors serve the same purpose as arrays except that they can change length while the program is running.

You need not read the previous sections of this chapter before covering this section.

Vector Basics

Like an array, a vector has a base type, and like an array, a vector stores a collection of values of its base type. However, the syntax for a vector type and a vector variable declaration are different from the syntax for arrays.

declaring a vector variable

You declare a variable `v` for a vector with base type *int* as follows:

```
vector<int> v;
```

template class

The notation `vector<`*Base_Type*`>` is a **template class,** which means you can plug in any type for *Base_Type* and that will produce a class for vectors with that base type. You can simply think of this as specifying the base type for a vector in the same sense as you specify a base type for an array. You can use any type, including class types, as the base type for a vector. The notation `vector<int>` is a class name, and so the previous declaration of `v` as a vector of type `vector<int>` includes a call to the default constructor for the class `vector<int>`, which creates a vector object that is empty (has no elements).

Vector elements are indexed starting with `0`, the same as arrays. The array square brackets notation can be used to read or change these elements, just as with an array. For example, the following changes the value of the `i`th element of the vector `v` and then outputs that changed value. (`i` is an *int* variable.)

v[i]

```
v[i] = 42;
cout << "The answer is " << v[i];
```

There is, however, a restriction on this use of the square brackets notation with vectors that is unlike the same notation used with arrays. You can use `v[i]` to change the value of the `i`th element. However, you cannot initialize the `i`th element using `v[i]`; you can only change an element that has already been given some value. To add an element to an index position of a vector for the first time, you normally use the member function `push_back`.

push_back

You add elements to a vector in order of positions, first at position `0`, then position `1`, then `2`, and so forth. The member function `push_back` adds an element in the next available position. For example, the following gives initial values to elements `0`, `1`, and `2` of the vector sample:

```
vector<double> sample;
sample.push_back(0.0);
sample.push_back(1.1);
sample.push_back(2.2);
```

size

The number of elements in a vector is called the **size** of the vector. The member function `size` can be used to determine how many elements are in a vector. For example, after the previously shown code is executed, `sample.size( )` returns `3`. You can write out all the elements currently in the vector `sample` as follows:

```
for (int i = 0; i < sample.size( ); i++)
    cout << sample[i] << endl;
```

`size`
`unsigned int`

The function `size` returns a value of type *`unsigned int`*, not a value of type *`int`*. (The type *`unsigned int`* allows only nonnegative integer values.) This returned value should be automatically converted to type *`int`* when it needs to be of type *`int`*, but some compilers may warn you that you are using an *`unsigned int`* where an *`int`* is required. If you want to be very safe, you can always apply a type cast to convert the returned *`unsigned int`* to an *`int`* or, in cases like this *`for`* loop, use a loop control variable of type *`unsigned int`* as follows:

```
for (unsigned int i = 0; i < sample.size( ); i++)
    cout << sample[i] << endl;
```

A simple demonstration illustrating some basic vector techniques is given in Display 11.9.

There is a vector constructor that takes one integer argument and will initialize the number of positions given as the argument. For example, if you declare `v` as follows:

```
vector<int> v(10);
```

then the first ten elements are initialized to `0`, and `v.size( )` would return `10`. You can then set the value of the `i`th element using `v[i]` for values of `i` equal to `0` through `9`. In particular, the following could immediately follow the declaration:

```
for (unsigned int i = 0; i < 10; i++)
    v[i] = i;
```

To set the `i`th element, for `i` greater than or equal to `10`, you would use `push_back`.

When you use the constructor with an integer argument, vectors of numbers are initialized to the zero of the number type. If the vector base type, is a class type the default constructor is used for initialization

The vector definition is given in the library `vector`, which places it in the `std` namespace. Thus, a file that uses vectors would include the following (or something similar):

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

Display 11.9 Using a Vector

```
#include <iostream>
#include <vector>
using namespace std;

int main( )
{
    vector<int> v;
    cout << "Enter a list of positive numbers.\n"
         << "Place a negative number at the end.\n";

    int next;
    cin >> next;
    while (next > 0)
    {
        v.push_back(next);
        cout << next << " added. ";
        cout << "v.size( ) = " << v.size( ) << endl;
        cin >> next;
    }

    cout << "You entered:\n";
    for (unsigned int i = 0; i < v.size( ); i++)
        cout << v[i] << " ";
    cout << endl;

    return 0;
}
```

Sample Dialogue

```
Enter a list of positive numbers.
Place a negative number at the end.
2 4 6 8 -1
2 added. v.size( ) = 1
4 added. v.size( ) = 2
6 added. v.size( ) = 3
8 added. v.size( ) = 4
You entered:
2 4 6 8
```

Vectors

Vectors are used very much like arrays are used, but a vector does not have a fixed size. If it needs more capacity to store another element, its capacity is automatically increased. Vectors are defined in the library `<vector>`, which places them in the `std` namespace. Thus, a file that uses vectors would include the following (or something similar):

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

The vector class for a given *Base_Type* is written `vector<`*Base_Type*`>`. Two sample vector declarations are

```
vector<int> v; //default constructor
               //producing an empty vector.
vector<AClass> record(20); //vector constructor uses the
//default constructor for AClass to initialize 20 elements.
```

Elements are added to a vector using the member function `push_back`, as illustrated below:

```
v.push_back(42);
```

Once an element position has received its first element, either with `push_back` or with a constructor initialization, that element position can then be accessed using square bracket notation, just like an array element.

PITFALL Using Square Brackets beyond the Vector Size

If `v` is a vector and `i` is greater than or equal to `v.size( )`, then the element `v[i]` does not yet exist and needs to be created by using `push_back` to add elements up to and including position `i`. If you try to set `v[i]` for `i` greater than or equal to `v.size( )`, as in

```
v[i] = n;
```

then you may or may not get an error message, but your program will undoubtedly misbehave at some point.

Programming TIP Vector Assignment Is Well Behaved

The assignment operator with vectors does an element-by-element assignment to the vector on the left-hand side of the assignment operator (increasing capacity if needed and resetting the size of the vector on the left-hand side of the assignment operator). Thus, provided the assignment operator on the base type makes an independent copy of the an element of the base type, then the assignment operator on the vector will make an independent copy.

Note that for the assignment operator to produce a totally independent copy of the vector on the right-hand side of the assignment operator requires that the assignment operator on the base type make completely independent copies. The assignment operator on a vector is only as good (or bad) as the assignment operator on its base type. (Details on overloading the assignment operator for classes that need it are given in Chapter 12.)

Efficiency Issues

capacity

At any point in time a vector has a **capacity,** which is the number of elements for which it currently has memory allocated. The member function `capacity( )` can be used to find out the capacity of a vector. Do not confuse the capacity of a vector with the size of a vector. The *size* is the number of elements in a vector, while the *capacity* is the number of elements for which there is memory allocated. Typically the capacity is larger than the size, and the capacity is always greater than or equal to the size.

Whenever a vector runs out of capacity and needs room for an additional member, the capacity is automatically increased. The exact amount of the increase is implementation dependent, but always allows for more capacity than is immediately needed. A commonly used implementation scheme is for the capacity to double whenever it needs to increase. Since increasing capacity is a complex task, this approach of reallocating capacity in large chunks is more efficient than allocating numerous small chunks.

Size and Capacity

The **size** of a vector is the number of elements in the vector. The **capacity** of a vector is the number of elements for which it currently has memory allocated. For a vector v, the size and capacity can be recovered with the member functions `v.size( )` and `v.capacity( )`.

You can completely ignore the capacity of a vector and that will have no effect on what your program does. However, if efficiency is an issue, you may want to manage capacity yourself and not simply accept the default behavior of doubling capacity whenever more is needed. You can use the member function `reserve` to explicitly increase the capacity of a vector. For example,

```
v.reserve(32);
```

sets the capacity to at least 32 elements, and

```
v.reserve(v.size( ) + 10);
```

sets the capacity to at least 10 more than the number of elements currently in the vector. Note that you can rely on `v.reserve` to increase the capacity of a vector, but it does not necessarily decrease the capacity of a vector if the argument is smaller than the current capacity.

You can change the size of a vector using the member function `resize`. For example, the following resizes a vector to 24 elements:

```
v.resize(24);
```

If the previous size was less than 24, then the new elements are initialized as we described for the constructor with an integer argument. If the previous size was greater than 24, then all but the first 24 elements are lost. The capacity is automatically increased if need be. Using `resize` and `reserve`, you can shrink the size and capacity of a vector when there is no longer any need for some elements or some capacity.

SELF-TEST EXERCISES

19 Is the following program legal? If so, what is the output?

```
#include <iostream>
#include <vector>
using namespace std;

int main( )
{
    vector<int> v(10);
    int i;

    for (i = 0; i < v.size( ); i++)
        v[i] = i;

    vector<int> copy;
    copy = v;
    v[0] = 42;
```

```
        for (i = 0; i < copy.size( ); i++)
            cout << copy[i] << " ";
        cout << endl;

        return 0;
    }
```

20 What is the difference between the size and the capacity of a vector?

CHAPTER SUMMARY

- A C-string variable is the same thing as an array of characters, but it is used in a slightly different way. A string variable uses the null character `'\0'` to mark the end of the string stored in the array.
- C-string variables usually must be treated like arrays, rather than simple variables of the kind we used for numbers and single characters. In particular, you cannot assign a C-string value to a C-string variable using the equal sign, =, and you cannot compare the values in two C-string variables using the == operator. Instead you must use special C-string functions to perform these tasks.
- The ANSI/ISO standard `<string>` library provides a fully featured class called `string` that can be used to represent strings of characters.
- Objects of the class `string` are better behaved than C strings. In particular, the assignment and equal operators, = and ==, have their intuitive meaning when used with objects of the class `string`.
- Vectors can be thought of as arrays that can grow (and shrink) in length while your program is running.

Answers to Self-Test Exercises

1 The following two are equivalent to each other (but not equivalent to any others):

```
char string_var[10] = "Hello";
char string_var[10] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

The following two are equivalent to each other (but not equivalent to any others):

```
char string_var[6] = "Hello";
char string_var[] = "Hello";
```

The following is not equivalent to any of the others:

```
char string_var[10] = {'H', 'e', 'l', 'l', 'o'};
```

2 `"DoBeDo to you"`

3 The declaration means that `string_var` has room for only six characters (including the null character `'\0'`). The function `strcat` does not check that there is room to add more characters to `string_var`, so `strcat` will write all the characters in the string `" and Good-bye."` into memory, even though that requires more memory than has been assigned to `string_var`. This means memory that should not be changed will be changed. The net effect is unpredictable, but bad.

4 If `strlen` were not already defined for you, you could use the following definition:

```
int strlen(const char str[])
//Precondition: str contains a string value terminated
//with '\0'.
//Returns the number of characters in the string str (not
//counting '\0').
{
    int index = 0;
    while (str[index] != '\0')
        index++;
    return index;
}
```

5 The maximum number of characters is five because the sixth position is needed for the null terminator (`'\0'`).

6 a. 1
b. 1
c. 5 (including the `'\0'`)
d. 2 (including the `'\0'`)
e. 6 (including the `'\0'`)

7 These are *not equivalent*. The first of these places the null character `'\0'` in the array after the characters `'a'`, `'b'`, and `'c'`. The second only assigns the successive positions `'a'`, `'b'`, and `'c'` but *does not put a* `'\0'` *anywhere*.

8
```
int index = 0;
while ( our_string[index] != '\0' )
{
    our_string[index] = 'X';
    index++;
}
```

9 a. If the C-string variable does not have a null terminator, `'\0'`, the loop can run beyond memory allocated for the C string, destroying the contents of memory there. To protect memory beyond the end of the array, change the *`while`* condition as shown in (b).

b. `while( our_string[index] != '\0' && index < SIZE )`

10
```
#include <cstring>
//needed to get the declaration of strcpy
...
strcpy(a_string, "Hello");
```

11 `I did it my way!`

12 The string `"good, I hope."` is too long for `a_string`. A chunk of memory that doesn't belong to the array `a_string` will be overwritten.

13
```
Enter some input:
The
   time is now.
The-timeEND OF OUTPUT
```

14 The complete dialogue is as follows:

```
Enter a line of input:
May the hair on your toes grow long and curly.
May t<END OF OUTPUT
```

15 `A*string<END OF OUTPUT`

16 `A string is a joy forever!<END OF OUTPUT`

17 The complete dialogue is

```
Enter a line of input:
Hello friend!
Equal
```

Remember, `cin` stops reading when it reaches a whitespace character such as a blank.

18 `Hello Jello`

19 The program is legal. The output is

```
0 1 2 3 4 5 6 7 8 9
```

Note that changing `v` does not change `copy`. A true independent copy is made with the assignment

```
  copy = v;
```

20 The size is the number of elements in a vector, whereas the capacity is the number of elements for which there is memory allocated. Typically, the capacity is larger than the size.

Programming Projects

1 Write a program that will read in a sentence of up to 100 characters and output the sentence with spacing corrected and with letters corrected for capitalization. In other words, in the output sentence, all strings of two or more blanks should be compressed to a single blank. The sentence should start with an uppercase letter but should contain no other uppercase letters. Do not worry about proper names; if their first letters are changed to lowercase, that is acceptable. Treat a line break as if it were a blank, in the sense that a line break and any number of blanks are compressed to a single blank. Assume that the sentence ends with a period and contains no other periods. For example, the input

```
the     Answer to life, the Universe, and   everything
IS 42.
```

should produce the following output:

```
The answer to life, the universe, and everything is 42.
```

2 Write a program that will read in a line of text and output the number of words in the line and the number of occurrences of each letter. Define a word to be any string of letters that is delimited at each end by either whitespace, a period, a comma, or the beginning or end of the line. You can assume that the input consists entirely of letters, whitespace, commas, and periods. When outputting the number of letters that occur in a line, be sure to count upper- and lowercase versions of a letter as the same letter. Output the letters in alphabetical order and list only those letters that do occur in the input line. For example, the input line

```
I say Hi.
```

should produce output similar to the following:

```
3 words
1 a
1 h
2 i
1 s
1 y
```

3 Give the function definition for the function with the following function declaration. Embed your definition in a suitable test program.

```
void get_double(double& input_number);
//Postcondition:input_number is given a value
//that the user approves of.
```

You can assume that the user types in the input in normal everyday notation, such as **23.789**, and does not use e-notation to type in the number. Model your definition after the definition of the function `get_int` given in Display 11.3 so that your function reads the input as characters, edits the string of characters, and converts the resulting string to a number of type *double*. You will need to define a function like `read_and_clean` that is more sophisticated than the one in Display 11.2, since it must cope with the decimal point. This is a fairly easy project. For a more difficult project, allow the user to enter the number in either the normal everyday notation, as discussed above, or in e-notation. Your function should decide whether or not the input is in e-notation by reading the input, *not* by asking the user whether she or he will use e-notation.

4 Write a program that reads a person's name in the following format: first name, then middle name or initial, and then last name. The program then outputs the name in the following format:

```
Last_Name, First_Name Middle_Initial.
```

For example, the input

```
Mary Average User
```

should produce the output:

```
User, Mary A.
```

The input

```
Mary A. User
```

should also produce the output:

```
User, Mary A.
```

Your program should work the same and place a period after the middle initial even if the input did not contain a period. Your program should allow for users who give no middle name or middle initial. In that case, the output, of course, contains no middle name or initial. For example, the input

```
Mary User
```

should produce the output

```
User, Mary
```

If you are using C strings, assume that each name is at most 20 characters long. Alternatively, use the class `string`. *Hint:* You may want to use three string variables rather than one large string variable for the input. You may find it easier to *not* use `getline`.

5 Write a program that reads in a line of text and replaces all four-letter words with the word `"love"`. For example, the input string

```
I hate you, you dodo!
```

should produce the output

```
I love you, you love!
```

Of course, the output will not always make sense. For example, the input string

```
John will run home.
```

should produce the output

```
Love love run love.
```

If the four-letter word starts with a capital letter, it should be replaced by `"Love"`, not by `"love"`. You need not check capitalization, except for the first letter of a word. A word is any string consisting of the letters of the alphabet and delimited at each end by a blank, the end of the line, or any other character that is not a letter. Your program should repeat this action until the user says to quit.

6 Write a program that can be used to train the user to use less sexist language by suggesting alternative versions of sentences given by the user. The program

will ask for a sentence, read the sentence into a `string` variable, and replace all occurrences of masculine pronouns with gender-neutral pronouns. For example, it will replace `"he"` by `"she or he"`. Thus, the input sentence

```
See an adviser, talk to him, and listen to him.
```

should produce the following suggested changed version of the sentence:

```
See an adviser, talk to her or him, and listen to her or him.
```

Be sure to preserve uppercase letters for the first word of the sentence. The pronoun `"his"` can be replaced by `"her(s)"`; your program need not decide between `"her"` and `"hers"`. Allow the user to repeat this for more sentences until the user says she or he is done.

This will be a long program that requires a good deal of patience. Your program should not replace the string `"he"` when it occurs inside another word, such as `"here"`. A word is any string consisting of the letters of the alphabet and delimited at each end by a blank, the end of the line, or any other character that is not a letter. Allow your sentences to be up to 100 characters long.

7 Write a sorting function that is similar to Display 10.12 in Chapter 10 except that it has an argument for a vector of *`int`*s rather than an array. This function will not need a parameter like `number_used` as in Display 10.12, since a vector can determine the number used with the member function `size()`. This sort function will this have only one parameter, which will be of a vector type. Use the selection sort algorithm (which was used in Display 10.12).

8 Redo Programming Project 5 from Chapter 10, but this time use vectors instead of arrays. (It may help to do the previous Programming Project first.)

9 Redo Programming Project 4 from Chapter 10, but this time use vectors instead of arrays. You should do either Programming Project 7 or 8 before doing this one. However, you will need to write your own (similar) sorting code for this project rather than using the sorting function from Programming Project 7 or 8 with no changes. You may want to use a vector with a *`struct`* as the base type.