World Headquarters
Jones and Bartlett Publishers
40 Tall Pine Drive
Sudbury, MA 01776
978-443-5000
info@jbpub.com
www.jbpub.com

Jones and Bartlett Publishers
Canada
2406 Nikanna Road
Mississauga, ON L5C 2W6
CANADA

Jones and Bartlett Publishers
International
Barb House, Barb Mews
London W6 7PA
UK

Library of Congress Cataloging-in-Publication Data

Dale, Nell B.
Programming and problem solving with C++ / Nell Dale, Chip Weems.– 4th ed.
p. cm.
Includes index.
ISBN 0-7637-0798-8 (pbk.)
1. C++ (Computer program language) I. Weems, Chip. II. Title.
QA76.73.C153D34 2004
005.13'3–dc22

13.4 Understanding Character Strings

Ever since Chapter 2, we have been using the `string` class to store and manipulate character strings.

```
string name;

name = "James Smith";
len = name.length();
 :
```

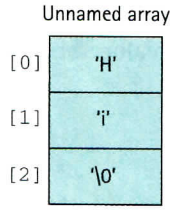
In some contexts, we think of a string as a single unit of data. In other contexts, we treat it as a group of individually accessible characters. In particular, we think of a string as a variable-length, linear collection of homogeneous components (of type `char`). Does this sound familiar? It should. As an abstraction, a string is a list of characters that, at any moment in time, has a length associated with it.

C string In C and C++, a null-terminated sequence of characters stored in a `char` array.

Thinking of a string as an ADT, how would we implement the ADT? There are many ways to implement strings. Programmers have specified and implemented their own string classes—the `string` class from the standard library, for instance. And the C++ language has its own built-in notion of a string: the **C string**. In C++, a string constant (or string literal, or literal string) is a sequence of characters enclosed by double quotes:

```
"Hi"
```

A string constant is stored as a `char` array with enough components to hold each specified character plus one more—the *null character.* The null character, which is the first character in both the ASCII and EBCDIC character sets, has internal representation 0. In C++, the escape sequence `\0` stands for the null character. When the compiler encounters the string `"Hi"` in a program, it stores the three characters 'H', 'i', and '\0' into a three-element, anonymous (unnamed) `char` array as follows:



The C string is the only kind of C++ array for which there exists an aggregate constant—the string constant. Notice that in a C++ program, the symbols `'A'` denote a single character, whereas the symbols `"A"` denote two: the character 'A' and the null character.*

In addition to C string constants, we can create C string *variables.* To do so, we explicitly declare a `char` array and store into it whatever characters we want to, finishing with the null character. Here's an example:

```
char myStr[8];   // Room for 7 significant characters plus '\0'

myStr[0] = 'H';
myStr[1] = 'i';
myStr[2] = '\0';
```

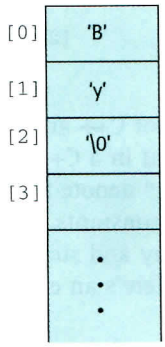
In C++, all C strings (constants or variables) are assumed to be null-terminated. This convention is agreed upon by all C++ programmers and standard library functions. The null character serves as a sentinel value; it allows algorithms to locate the end of the string. For example, here is a function that determines the length of any C string, not counting the terminating null character:

```
int StrLength( /* in */ const char str[] )

// Precondition:
//     str holds a null-terminated string
// Postcondition:
//     Function value == number of characters in str (excluding '\0')
```

*C *string* is not an official term used in C++ language manuals. Such manuals typically use the term *string.* However, we use *C string* to distinguish between the general concept of a string and the built-in array representation defined by the C and C++ languages.

```
{
    int i = 0;    // Index variable

    while (str[i] != '\0')
        i++;
    return i;
}
```

The value of `i` is the correct value for this function to return. If the array being examined is



then `i` equals 2 at loop exit. The string length is therefore 2.

The argument to the `StrLength` function can be a C string variable, as in the function call

```
cout << StrLength(myStr);
```

or it can be a string constant:

```
cout << StrLength("Hello");
```

In the first case, the base address of the `myStr` array is sent to the function, as we discussed in Chapter 12. In the second case, a base address is also sent to the function—the base address of the unnamed array that the compiler has set aside for the string constant.

There is one more thing we should say about our `StrLength` function. A C++ programmer would not actually write this function. The standard library supplies several string-processing functions, one of which is named `strlen` and does exactly what our `StrLength` function does. Later in the chapter, we look at `strlen` and other library functions.

Initializing C Strings

In Chapter 12, we showed how to initialize an array in its declaration by specifying a list of initial values within braces, like this:

```
int delta[5] = {25, -3, 7, 13, 4};
```

To initialize a C string variable in its declaration, you could use the same technique:

```
char message[8] = {'W', 'h', 'o', 'o', 'p', 's', '!', '\0'};
```

However, C++ allows a more convenient way to initialize a C string. You can simply initialize the array by using a string constant:

```
char message[8] = "Whoops!";
```

This shorthand notation is unique to C strings because there is no other kind of array for which there are aggregate constants.

We said in Chapter 12 that you can omit the size of an array when you initialize it in its declaration (in which case, the compiler determines its size). This feature is often used with C strings because it keeps you from having to count the number of characters. For example,

```
char promptMsg[] = "Enter a positive number:";  // Size is 25
char errMsg[] = "Value must be positive.";      // Size is 24
```

Be very careful about one thing: C++ treats initialization (in a declaration) and assignment (in an assignment statement) as two distinct operations. Different rules apply. Remember that array initialization is legal, but aggregate array assignment is not.

```
char myStr[20] = "Hello";   // OK
  :
myStr = "Howdy";            // Not allowed
```

C String Input and Output

In Chapter 12, we emphasized that C++ does not provide aggregate operations on arrays. There is no aggregate assignment, aggregate comparison, or aggregate arithmetic on arrays. We also said that aggregate input/output of arrays is not possible, with one exception. C strings are that exception. Let's look first at output.

To output the contents of an array that is *not* a C string, you aren't allowed to do this:

```
int alpha[100];
  :
cout << alpha;   // Not allowed
```

Instead, you must write a loop and print the array elements one at a time. However, aggregate output of a null-terminated `char` array (that is, a C string) is valid. The C string can be a constant (as we've been doing since Chapter 2):

```
cout << "Results are:";
```

or it can be a variable:

```
char msg[8] = "Welcome";
  :
cout << msg;
```

In both cases, the insertion operator (<<) outputs each character in the array until the null character is found. It is up to you to double-check that the terminating null character is present in the array. If not, the << operator will march through the array and into the rest of memory, printing out bytes until—just by chance—it encounters a byte whose integer value is 0.

To input C strings, we have several options. The first is to use the extraction operator (>>), which behaves exactly the same as with `string` class objects. When reading input characters into a C string variable, the >> operator skips leading whitespace characters and then reads successive characters into the array, stopping at the first trailing whitespace character (which is not consumed, but remains as the first character waiting in the input stream). The >> operator also takes care of adding the null character to the end of the string. For example, assume we have the following code:

```
char firstName[31];    // Room for 30 characters plus '\0'
char lastName[31];

cin >> firstName >> lastName;
```

If the input stream initially looks like this (where ␣ denotes a blank):

```
␣␣John␣Smith␣␣␣25
```

then our input statement stores 'J', 'o', 'h', 'n', and '\0' into `firstName[0]` through `firstName[4]`; stores 'S', 'm', 'i', 't', 'h', and '\0' into `lastName[0]` through `lastName[5]`; and leaves the input stream as

```
␣␣␣25
```

The >> operator, however, has two potential drawbacks.

1. If the array isn't large enough to hold the sequence of input characters (and the '\0'), the >> operator will continue to store characters into memory past the end of the array.
2. The >> operator cannot be used to input a string that has blanks within it. (It stops reading as soon as it encounters the first whitespace character.)

To cope with these limitations, we can use a variation of the `get` function, a member of the `istream` class. We have used the `get` function to input a single character, even if it is a whitespace character:

```
cin.get(inputChar);
```

The `get` function also can be used to input C strings, in which case the function call requires two arguments. The first is the array name and the second is an `int` expression.

```
cin.get(myStr, charCount + 1);
```

The `get` function does not skip leading whitespace characters and continues until it either has read `charCount` characters or it reaches the newline character '\n', whichever comes first. It then appends the null character to the end of the string. With the statements

```
char oneLine[81];    // Room for 80 characters plus '\0'
  :
cin.get(oneLine, 81);
```

the `get` function reads and stores an entire input line (to a maximum of 80 characters), embedded blanks and all. If the line has fewer than 80 characters, reading stops at '\n' but does not consume it. The newline character is now the first one waiting in the input stream. To read two consecutive lines worth of strings, it is necessary to consume the newline character:

```
char dummy;
  :
cin.get(string1, 81);
cin.get(dummy);          // Eat newline before next "get"
cin.get(string2, 81);
```

The first function call reads characters up to, but not including, the '\n'. If the input of `dummy` were omitted, then the input of `string2` would read *no* characters because '\n' would immediately be the first character waiting in the stream.

Finally, the `ignore` function—introduced in Chapter 4—can be useful in conjunction with the `get` function. Recall that the statement

```
cin.ignore(200, '\n');
```

says to skip at most 200 input characters but stop if a newline was read. (The newline character *is* consumed by this function.) If a program inputs a long string from the user but only wants to retain the first four characters of the response, here is a way to do it:

```
char response[5];        // Room for 4 characters plus '\0'
cin.get(response, 5);    // Input at most 4 characters
```

```
cin.ignore(100, '\n');    // Skip remaining chars up to and
                          //   including '\n'
```

The value 100 in the last statement is arbitrary. Any "large enough" number will do.

Here is a table that summarizes the differences between the >> operator and the `get` function when reading C strings:

Statement	Skips Leading Whitespace?	Stops Reading When?
`cin >> inputStr;`	Yes	At the first trailing whitespace character (which is *not* consumed)
`cin.get(inputStr, 21);`	No	When either 20 characters are read or '\n' is encountered (which is *not* consumed)

Finally, we revisit a topic that came up in Chapter 4. Certain library functions and member functions of system-supplied classes require C strings as arguments. An example is the `ifstream` class member function named `open`. To open a file, we pass the name of the file as a C string, either a constant or a variable:

```
ifstream file1;
ifstream file2;
char     fileName[51];          // Max. 50 characters plus '\0'

file1.open("students.dat");
cin.get(fileName, 51);          // Read at most 50 characters
cin.ignore(100, '\n');          // Skip rest of input line
file2.open(fileName);
```

As discussed in Chapter 4, if our file name is contained in a `string` class object, we still can use the `open` function, *provided* we use the `string` class member function named `c_str` to convert the string to a C string:

```
ifstream inFile;
string   fileName;

cin >> fileName;
inFile.open(fileName.c_str());
```

Comparing these two code segments, you can observe a major advantage of the `string` class over C strings: A string in a `string` class object has unbounded length, whereas the length of a C string is bounded by the array size, which is fixed at compile time.

C String Library Routines

Through the header file `cstring`, the C++ standard library provides a large assortment of C string operations. In this section, we discuss three of these library functions:

strlen, which returns the length of a string; strcmp, which compares two strings using the relations less-than, equal, and greater-than; and strcpy, which copies one string to another. Here is a summary of strlen, strcmp, and strcpy:

Header File	Function	Function Value	Effect
<cstring>	strlen(str)	Integer length of str (excluding '\0')	Computes length of str
<cstring>	strcmp(str1, str2)	An integer < 0, if str1 < str2 The integer 0, if str1 = str2 An integer > 0, if str1 > str2	Compares str1 and str2
<cstring>	strcpy(toStr, fromStr)	Base address of toStr (usually ignored)	Copies fromStr (including '\0') to toStr, overwriting what was there; toStr must be large enough to hold the result

The strlen function is similar to the StrLength function we wrote earlier. It returns the number of characters in a C string prior to the terminating '\0'. Here's an example of a call to the function:

```
#include <cstring>
  :
char subject[] = "Computer Science";

cout << strlen(subject);   // Prints 16
```

The strcpy routine is important because aggregate assignment with the = operator is not allowed on C strings. In the following code fragment, we show the wrong way and the right way to perform a string copy.

```
#include <cstring>
  :
char myStr[100];
  :
myStr = "Abracadabra";           // No
strcpy(myStr, "Abracadabra");    // Yes
```

In `strcpy`'s argument list, the destination string is the one on the left, just as an assignment operation transfers data from right to left. It is the caller's responsibility to make sure that the destination array is large enough to hold the result.

The `strcpy` function is technically a value-returning function; it not only copies one C string to another, but also returns as a function value the base address of the destination array. The reason why the caller would want to use this function value is not at all obvious, and we don't discuss it here. Programmers nearly always ignore the function value and simply invoke `strcpy` as if it were a void function (as we did above). You may wish to review the Background Information box in Chapter 8 entitled "Ignoring a Function Value."

The `strcmp` function is used for comparing two strings. The function receives two C strings as parameters and compares them in *lexicographic* order (the order in which they would appear in a dictionary)—the same ordering used in comparing `string` class objects. Given the function call `strcmp(str1, str2)`, the function returns one of the following `int` values: a negative integer, if `str1 < str2` lexicographically; the value 0, if `str1 = str2`; or a positive integer, if `str1 > str2`. The precise values of the negative integer and the positive integer are unspecified. You simply test to see if the result is less than 0, 0, or greater than 0. Here is an example:

```
if (strcmp(str1, str2) < 0)    // If str1 is less than str2 ...
    :
```

We have described only three of the string-handling routines provided by the standard library. These three are the most commonly needed, but there are many more. If you are designing or maintaining programs that use C strings extensively, you should read the documentation on strings for your C++ system.

String Class or C Strings?

When working with string data, should you use a class like `string`, or should you use C strings? From the standpoints of clarity, versatility, and ease of use, there is no contest. Use a string class. The standard library `string` class provides strings of unbounded length, aggregate assignment, aggregate comparison, concatenation with the + operator, and so forth.

However, it is still useful to be familiar with C strings. Among the thousands of software products currently in use that are written in C and C++, most (but a declining percentage) use C strings to represent string data. In your next place of employment, if you are asked to modify or upgrade such software, understanding C strings is essential. Additionally, *using* a string class is one thing; *implementing* it is another. Someone must implement the class using a concrete data representation. In your employment, that someone might be you, and the underlying data representation might very well be a C string!