4 Compound Types

In this chapter you'll learn about the following:

- Creating and using arrays
- Creating and using C-style strings
- Creating and using `string`-class strings
- Using the `getline()` and `get()` methods for reading strings
- Mixing string and numeric input
- Creating and using structures
- Creating and using unions
- Creating and using enumerations
- Creating and using pointers
- Managing dynamic memory with `new` and `delete`
- Creating dynamic arrays
- Creating dynamic structures
- Automatic, static, and dynamic storage
- The `vector` and `array` classes (an introduction)

Say you've developed a computer game called User-Hostile in which players match wits with a cryptic and abusive computer interface. Now you must write a program that keeps track of your monthly game sales for a five-year period. Or you want to inventory your accumulation of hacker-hero trading cards. You soon conclude that you need something more than C++'s simple basic types to meet these data requirements, and C++ offers something more—compound types. These are types built from the basic integer and floating-point types. The most far-reaching compound type is the class, that bastion of OOP toward which we are progressing. But C++ also supports several more modest compound types taken from C. The array, for example, can hold several values of the same type. A particular kind of array can hold a string, which is a series of characters. Structures can hold several values of differing types. Then there are pointers, which are variables that tell a computer where data is placed. You'll examine all these compound forms (except classes) in this chapter, take a first look at `new` and `delete` and how you can use them to manage data, and take an introductory look at the C++ `string` class, which gives you an alternative way to work with strings.

Introducing Arrays

An *array* is a data form that can hold several values, all of one type. For example, an array can hold 60 type `int` values that represent five years of game sales data, 12 `short` values that represent the number of days in each month, or 365 `float` values that indicate your food expenses for each day of the year. Each value is stored in a separate array element, and the computer stores all the elements of an array consecutively in memory.

To create an array, you use a declaration statement. An array declaration should indicate three things:

- The type of value to be stored in each element
- The name of the array
- The number of elements in the array

You accomplish this in C++ by modifying the declaration for a simple variable and adding brackets that contain the number of elements. For example, the following declaration creates an array named `months` that has 12 elements, each of which can hold a type `short` value:

```
short months[12];     // creates array of 12 short
```

Each element, in essence, is a variable that you can treat as a simple variable.

This is the general form for declaring an array:

```
typeName arrayName[arraySize];
```

The expression *`arraySize`*, which is the number of elements, must be an integer constant, such as 10 or a `const` value, or a constant expression, such as `8 * sizeof (int)`, for which all values are known at the time compilation takes place. In particular, *`arraySize`* cannot be a variable whose value is set while the program is running. However, later in this chapter you'll learn how to use the `new` operator to get around that restriction.

The Array as Compound Type

An array is called a *compound type* because it is built from some other type. (C uses the term *derived type,* but because C++ uses the term *derived* for class relationships, it had to come up with a new term.) You can't simply declare that something is an array; it always has to be an array of some particular type. There is no generalized array type. Instead, there are many specific array types, such as array of `char` or array of `long`. For example, consider this declaration:

```
float loans[20];
```

The type for `loans` is not "array"; rather, it is "array of `float`." This emphasizes that the `loans` array is built from the `float` type.

Much of the usefulness of the array comes from the fact that you can access array elements individually. The way to do this is to use a *subscript*, or an *index*, to number the elements. C++ array numbering starts with zero. (This is nonnegotiable; you have to start at zero. Pascal and BASIC users will have to adjust.) C++ uses a bracket notation with the index to specify an array element. For example, `months[0]` is the first element of the `months` array, and `months[11]` is the last element. Note that the index of the last element is one less than the size of the array (see Figure 4.1). Thus, an array declaration enables you to create a lot of variables with a single declaration, and you can then use an index to identify and access individual elements.

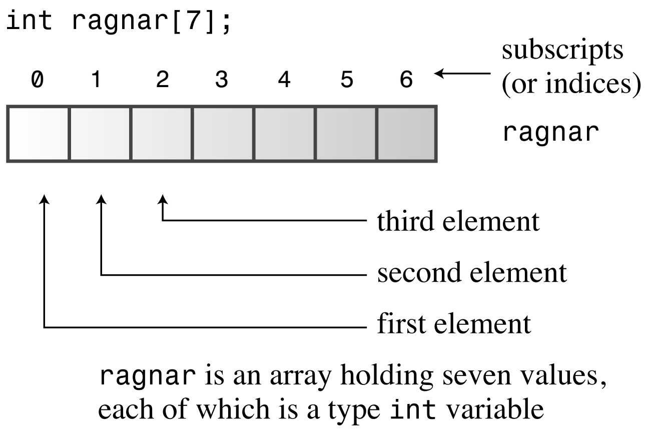


Figure 4.1 Creating an array.

The Importance of Valid Subscript Values

The compiler does not check to see if you use a valid subscript. For instance, the compiler won't complain if you assign a value to the nonexistent element `months[101]`. But that assignment could cause problems when the program runs, possibly corrupting data or code, possibly causing the program to abort. So it is your responsibility to make sure that your program uses only valid subscript values.

The yam analysis program in Listing 4.1 demonstrates a few properties of arrays, including declaring an array, assigning values to array elements, and initializing an array.

Listing 4.1 **arrayone.cpp**

```
// arrayone.cpp -- small arrays of integers
#include <iostream>
int main()
{
    using namespace std;
    int yams[3];    // creates array with three elements
    yams[0] = 7;    // assign value to first element
    yams[1] = 8;
    yams[2] = 6;

    int yamcosts[3] = {20, 30, 5}; // create, initialize array
// NOTE: If your C++ compiler or translator can't initialize
// this array, use static int yamcosts[3] instead of
// int yamcosts[3]

    cout << "Total yams = ";
    cout << yams[0] + yams[1] + yams[2] << endl;
    cout << "The package with " << yams[1] << " yams costs ";
    cout << yamcosts[1] << " cents per yam.\n";
    int total = yams[0] * yamcosts[0] + yams[1] * yamcosts[1];
    total = total + yams[2] * yamcosts[2];
    cout << "The total yam expense is " << total << " cents.\n";

    cout << "\nSize of yams array = " << sizeof yams;
    cout << " bytes.\n";
    cout << "Size of one element = " << sizeof yams[0];
    cout << " bytes.\n";
    return 0;
}
```

Here is the output from the program in Listing 4.1:

```
Total yams = 21
The package with 8 yams costs 30 cents per yam.
The total yam expense is 410 cents.

Size of yams array = 12 bytes.
Size of one element = 4 bytes.
```

Program Notes

First, the program in Listing 4.1 creates a three-element array called `yams`. Because `yams` has three elements, the elements are numbered from `0` through `2`, and `arrayone.cpp` uses index values of `0` through `2` to assign values to the three individual elements. Each individual `yam` element is an `int` with all the rights and privileges of an `int` type, so `arrayone.cpp` can, and does, assign values to elements, add elements, multiply elements, and display elements.

The program uses the long way to assign values to the `yam` elements. C++ also lets you initialize array elements within the declaration statement. Listing 4.1 uses this shortcut to assign values to the `yamcosts` array:

```
int yamcosts[3] = {20, 30, 5};
```

It simply provides a comma-separated list of values (the *initialization list*) enclosed in braces. The spaces in the list are optional. If you don't initialize an array that's defined inside a function, the element values remain undefined. That means the element takes on whatever value previously resided at that location in memory.

Next, the program uses the array values in a few calculations. This part of the program looks cluttered with all the subscripts and brackets. The `for` loop, coming up in Chapter 5, "Loops and Relational Expressions," provides a powerful way to deal with arrays and eliminates the need to write each index explicitly. For the time being, we'll stick to small arrays.

As you should recall, the `sizeof` operator returns the size, in bytes, of a type or data object. Note that if you use the `sizeof` operator with an array name, you get the number of bytes in the whole array. But if you use `sizeof` with an array element, you get the size, in bytes, of the element. This illustrates that `yams` is an array, but `yams[1]` is just an `int`.

Initialization Rules for Arrays

C++ has several rules about initializing arrays. They restrict when you can do it, and they determine what happens if the number of array elements doesn't match the number of values in the initializer. Let's examine these rules.

You can use the initialization form *only* when defining the array. You cannot use it later, and you cannot assign one array wholesale to another:

```
int cards[4] = {3, 6, 8, 10};       // okay
int hand[4];                        // okay
hand[4] = {5, 6, 7, 9};             // not allowed
hand = cards;                       // not allowed
```

However, you can use subscripts and assign values to the elements of an array individually.

When initializing an array, you can provide fewer values than array elements. For example, the following statement initializes only the first two elements of `hotelTips`:

```
float hotelTips[5] = {5.0, 2.5};
```

If you partially initialize an array, the compiler sets the remaining elements to zero. Thus, it's easy to initialize all the elements of an array to zero—just explicitly initialize the first element to zero and then let the compiler initialize the remaining elements to zero:

```
long totals[500] = {0};
```

Note that if you initialize to `{1}` instead of to `{0}`, just the first element is set to `1`; the rest still get set to `0`.

If you leave the square brackets (`[]`) empty when you initialize an array, the C++ compiler counts the elements for you. Suppose, for example, that you make this declaration:

```
short things[] = {1, 5, 3, 8};
```

The compiler makes `things` an array of four elements.

Letting the Compiler Do It

Often, letting the compiler count the number of elements is poor practice because its count can be different from what you think it should be. You could, for instance, accidently omit an initial value from the list. However, this approach can be a safe one for initializing a character array to a string, as you'll soon see. And if your main concern is that the program, not you, knows how large an array is, you can do something like this:

```
short things[] = {1, 5, 3, 8};
int num_elements = sizeof things / sizeof (short);
```

Whether this is useful or lazy depends on the circumstances.

C++11 Array Initialization

As Chapter 3, "Dealing with Data," mentioned, C++11 makes the brace form of initialization (list-initialization) a universal form for all types. Arrays already use list-initialization, but the C++11 version adds a few more features.

First, you can drop the `=` sign when initializing an array:

```
double earnings[4] {1.2e4, 1.6e4, 1.1e4, 1.7e4};  // okay with C++11
```

Second, you can use empty braces to set all the elements to 0:

```
unsigned int counts[10] = {};  // all elements set to 0
float balances[100] {};        // all elements set to 0
```

Third, as discussed in Chapter 3, list-initialization protects against narrowing:

```
long plifs[] = {25, 92, 3.0};               // not allowed
char slifs[4] {'h', 'i', 1122011, '\0'}; // not allowed
char tlifs[4] {'h', 'i', 112, '\0'};     // allowed
```

The first initialization fails because converting from a floating-point type to an integer type is narrowing, even if the floating-point value has only zeros after the decimal point. The second initialization fails because `1122011` is outside the range of a `char`, assuming we have an 8-bit `char`. The third succeeds because, even though `112` is an `int` value, it still is in the range of a `char`.

The C++ Standard Template Library (STL) provides an alternative to arrays called the `vector` template class, and C++11 adds an `array` template class. These alternatives are more sophisticated and flexible than the built-in array composite type. This chapter will discuss them briefly later, and Chapter 16, "The `string` Class and the Standard Template Library," discusses them more fully.

Strings

A *string* is a series of characters stored in consecutive bytes of memory. C++ has two ways of dealing with strings. The first, taken from C and often called a *C-style string*, is the first one this chapter examines. Later, this chapter discusses an alternative method based on a `string` class library.

The idea of a series of characters stored in consecutive bytes implies that you can store a string in an array of `char`, with each character kept in its own array element. Strings provide a convenient way to store text information, such as messages to the user (*"Please tell me your secret Swiss bank account number"*) or responses from the user (*"You must be joking"*). C-style strings have a special feature: The last character of every string is the *null character*. This character, written `\0`, is the character with ASCII code 0, and it serves to mark the string's end. For example, consider the following two declarations:

```
char dog[8] = { 'b', 'e', 'a', 'u', 'x', ' ', 'I', 'I'};       // not a string!
char cat[8] = {'f', 'a', 't', 'e', 's', 's', 'a', '\0'};       // a string!
```

Both of these arrays are arrays of `char`, but only the second is a string. The null character plays a fundamental role in C-style strings. For example, C++ has many functions that handle strings, including those used by `cout`. They all work by processing a string character-by-character until they reach the null character. If you ask `cout` to display a nice string like `cat` in the preceding example, it displays the first seven characters, detects the null character, and stops. But if you are ungracious enough to tell `cout` to display the `dog` array from the preceding example, which is not a string, `cout` prints the eight letters in the array and then keeps marching through memory byte-by-byte, interpreting each byte as a character to print, until it reaches a null character. Because null characters, which really are bytes set to zero, tend to be common in memory, the damage is usually contained quickly; nonetheless, you should not treat nonstring character arrays as strings.

The `cat` array example makes initializing an array to a string look tedious—all those single quotes and then having to remember the null character. Don't worry. There is a better way to initialize a character array to a string. Just use a quoted string, called a *string constant* or *string literal*, as in the following:

```
char bird[11] = "Mr. Cheeps";     // the \0 is understood
char fish[] = "Bubbles";          // let the compiler count
```

Quoted strings always include the terminating null character implicitly, so you don't have to spell it out (see Figure 4.2). Also the various C++ input facilities for reading a string from keyboard input into a `char` array automatically add the terminating null character for you. (If, when you run the program in Listing 4.1, you discover that you have to use the keyword `static` to initialize an array, you have to use it with these `char` arrays, too.)

Of course, you should make sure the array is large enough to hold all the characters of the string, including the null character. Initializing a character array with a string constant is one case where it may be safer to let the compiler count the number of elements for you. There is no harm, other than wasted space, in making an array larger than the string. That's because functions that work with strings are guided by the location of the null character, not by the size of the array. C++ imposes no limits on the length of a string.

Caution

When determining the minimum array size necessary to hold a string, remember to include the terminating null character in your count.

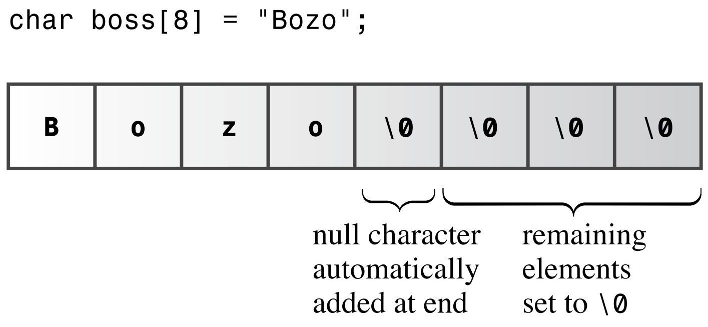

char boss[8] = "Bozo";
B
o
z
o
\0
\0
\0
\0
null character automatically added at end
remaining elements set to \0

Figure 4.2 Initializing an array to a string.

Note that a string constant (with double quotes) is not interchangeable with a character constant (with single quotes). A character constant, such as `'S'`, is a shorthand notation for the code for a character. On an ASCII system, `'S'` is just another way of writing 83. Thus, the following statement assigns the value 83 to `shirt_size`:

```
char shirt_size = 'S';          // this is fine
```

But `"S"` is not a character constant; it represents the string consisting of two characters, the `S` and the `\0` characters. Even worse, `"S"` actually represents the memory address at which the string is stored. So a statement like the following attempts to assign a memory address to `shirt_size`:

```
char shirt_size = "S";         // illegal type mismatch
```

Because an address is a separate type in C++, a C++ compiler won't allow this sort of nonsense. (We'll return to this point later in this chapter after we've discussed pointers.)

Concatenating String Literals

Sometimes a string may be too long to conveniently fit on one line of code. C++ enables you to concatenate string literals—that is, to combine two quoted strings into one. Indeed, any two string constants separated only by whitespace (spaces, tabs, and newlines) are automatically joined into one. Thus, all the following output statements are equivalent to each other:

```
cout << "I'd give my right arm to be" " a great violinist.\n";
cout << "I'd give my right arm to be a great violinist.\n";
cout << "I'd give my right ar"
"m to be a great violinist.\n";
```

Note that the join doesn't add any spaces to the joined strings. The first character of the second string immediately follows the last character, not counting `\0`, of the first string. The `\0` character from the first string is replaced by the first character of the second string.

Using Strings in an Array

The two most common ways of getting a string into an array are to initialize an array to a string constant and to read keyboard or file input into an array. Listing 4.2 demonstrates these approaches by initializing one array to a quoted string and using `cin` to place an input string into a second array. The program also uses the standard C library function `strlen()` to get the length of a string. The standard `cstring` header file (or `string.h` for older implementations) provides declarations for this and many other string-related functions.

Listing 4.2 **`strings.cpp`**

```
// strings.cpp -- storing strings in an array
#include <iostream>
#include <cstring>  // for the strlen() function
int main()
{
    using namespace std;
    const int Size = 15;
    char name1[Size];               // empty array
    char name2[Size] = "C++owboy";  // initialized array
    // NOTE: some implementations may require the static keyword
    // to initialize the array name2

    cout << "Howdy! I'm " << name2;
    cout << "! What's your name?\n";
    cin >> name1;
    cout << "Well, " << name1 << ", your name has ";
    cout << strlen(name1) << " letters and is stored\n";
    cout << "in an array of " << sizeof(name1) << " bytes.\n";
    cout << "Your initial is " << name1[0] << ".\n";
    name2[3] = '\0';                // set to null character
    cout << "Here are the first 3 characters of my name: ";
    cout << name2 << endl;
    return 0;
}
```

Here is a sample run of the program in Listing 4.2:

```
Howdy! I'm C++owboy! What's your name?
Basicman
Well, Basicman, your name has 8 letters and is stored
in an array of 15 bytes.
Your initial is B.
Here are the first 3 characters of my name: C++
```

Program Notes

What can you learn from Listing 4.2? First, note that the `sizeof` operator gives the size of the entire array, 15 bytes, but the `strlen()` function returns the size of the string stored in the array and not the size of the array itself. Also `strlen()` counts just the visible characters and not the null character. Thus, it returns a value of `8`, not `9`, for the length of `Basicman`. If `cosmic` is a string, the minimum array size for holding that string is `strlen(cosmic) + 1`.

Because `name1` and `name2` are arrays, you can use an index to access individual characters in the array. For example, the program uses `name1[0]` to find the first character in that array. Also the program sets `name2[3]` to the null character. That makes the string end after three characters, even though more characters remain in the array (see Figure 4.3).

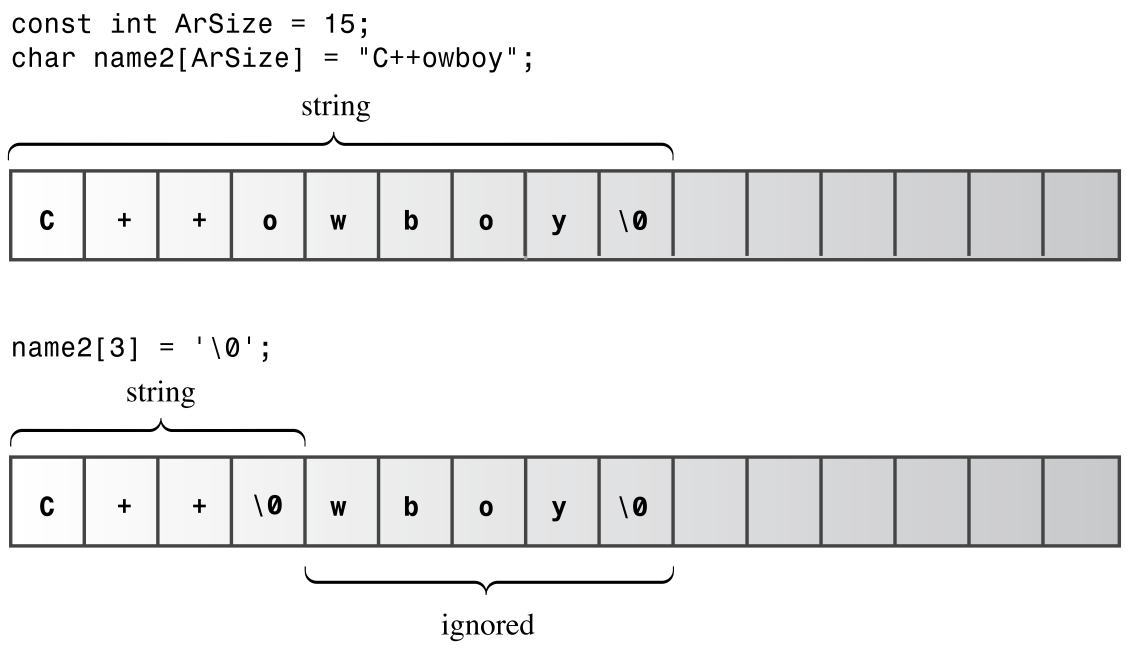


Figure 4.3 Shortening a string with `\0`.

Note that the program in Listing 4.2 uses a symbolic constant for the array size. Often the size of an array appears in several statements in a program. Using a symbolic constant to represent the size of an array simplifies revising the program to use a different array size; you just have to change the value once, where the symbolic constant is defined.

Adventures in String Input

The `strings.cpp` program has a blemish that is concealed through the often useful technique of carefully selected sample input. Listing 4.3 removes the veils and shows that string input can be tricky.

Listing 4.3 **instr1.cpp**

```
// instr1.cpp -- reading more than one string
#include <iostream>
int main()
{
    using namespace std;
    const int ArSize = 20;
    char name[ArSize];
    char dessert[ArSize];

    cout << "Enter your name:\n";
    cin >> name;
    cout << "Enter your favorite dessert:\n";
    cin >> dessert;
    cout << "I have some delicious " << dessert;
    cout << " for you, " << name << ".\n";
    return 0;
}
```

The intent of the program in Listing 4.3 is simple: Read a user's name and favorite dessert from the keyboard and then display the information. Here is a sample run:

```
Enter your name:
Alistair Dreeb
Enter your favorite dessert:
I have some delicious Dreeb for you, Alistair.
```

We didn't even get a chance to respond to the dessert prompt! The program showed it and then immediately moved on to display the final line.

The problem lies with how `cin` determines when you've finished entering a string. You can't enter the null character from the keyboard, so `cin` needs some other means for locating the end of a string. The `cin` technique is to use whitespace—spaces, tabs, and newlines—to delineate a string. This means `cin` reads just one word when it gets input for a character array. After it reads this word, `cin` automatically adds the terminating null character when it places the string into the array.

The practical result in this example is that `cin` reads `Alistair` as the entire first string and puts it into the `name` array. This leaves poor `Dreeb` still sitting in the input queue. When `cin` searches the input queue for the response to the favorite dessert question, it finds `Dreeb` still there. Then `cin` gobbles up `Dreeb` and puts it into the `dessert` array (see Figure 4.4).

Another problem, which didn't surface in the sample run, is that the input string might turn out to be longer than the destination array. Using `cin` as this example did offers no protection against placing a 30-character string in a 20-character array.

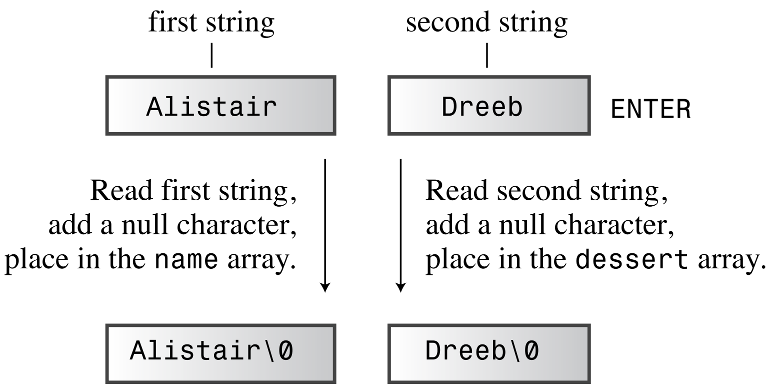

first string
second string
Alistair
Dreeb
ENTER
Read first string, add a null character, place in the name array.
Read second string, add a null character, place in the dessert array.
Alistair\0
Dreeb\0

Figure 4.4 The `cin` view of string input.

Many programs depend on string input, so it's worthwhile to explore this topic further. We'll have to draw on some of the more advanced features of `cin`, which are described in Chapter 17, "Input, Output, and Files."

Reading String Input a Line at a Time

Reading string input a word at a time is often not the most desirable choice. For instance, suppose a program asks the user to enter a city, and the user responds with **`New York`** or **`Sao Paulo`**. You would want the program to read and store the full names, not just `New` and `Sao`. To be able to enter whole phrases instead of single words as a string, you need a different approach to string input. Specifically, you need a line-oriented method instead of a word-oriented method. You are in luck, for the `istream` class, of which `cin` is an example, has some line-oriented class member functions: `getline()` and `get()`. Both read an entire input line—that is, up until a newline character. However, `getline()` then discards the newline character, whereas `get()` leaves it in the input queue. Let's look at the details, beginning with `getline()`.

Line-Oriented Input with `getline()`

The `getline()` function reads a whole line, using the newline character transmitted by the Enter key to mark the end of input. You invoke this method by using `cin.getline()` as a function call. The function takes two arguments. The first argument is the name of the target (that is, the array destined to hold the line of input), and the second argument is a limit on the number of characters to be read. If this limit is, say, 20, the function reads no more than 19 characters, leaving room to automatically add the null character at the end. The `getline()` member function stops reading input when it reaches this numeric limit or when it reads a newline character, whichever comes first.

For example, suppose you want to use `getline()` to read a name into the 20-element name array. You would use this call:

```
cin.getline(name,20);
```

This reads the entire line into the `name` array, provided that the line consists of 19 or fewer characters. (The `getline()` member function also has an optional third argument, which Chapter 17 discusses.)

Listing 4.4 modifies Listing 4.3 to use `cin.getline()` instead of a simple `cin`. Otherwise, the program is unchanged.

Listing 4.4 **`instr2.cpp`**

```
// instr2.cpp -- reading more than one word with getline
#include <iostream>
int main()
{
    using namespace std;
    const int ArSize = 20;
    char name[ArSize];
    char dessert[ArSize];

    cout << "Enter your name:\n";
    cin.getline(name, ArSize);  // reads through newline
    cout << "Enter your favorite dessert:\n";
    cin.getline(dessert, ArSize);
    cout << "I have some delicious " << dessert;
    cout << " for you, " << name << ".\n";
    return 0;
}
```

Here is some sample output for Listing 4.4:

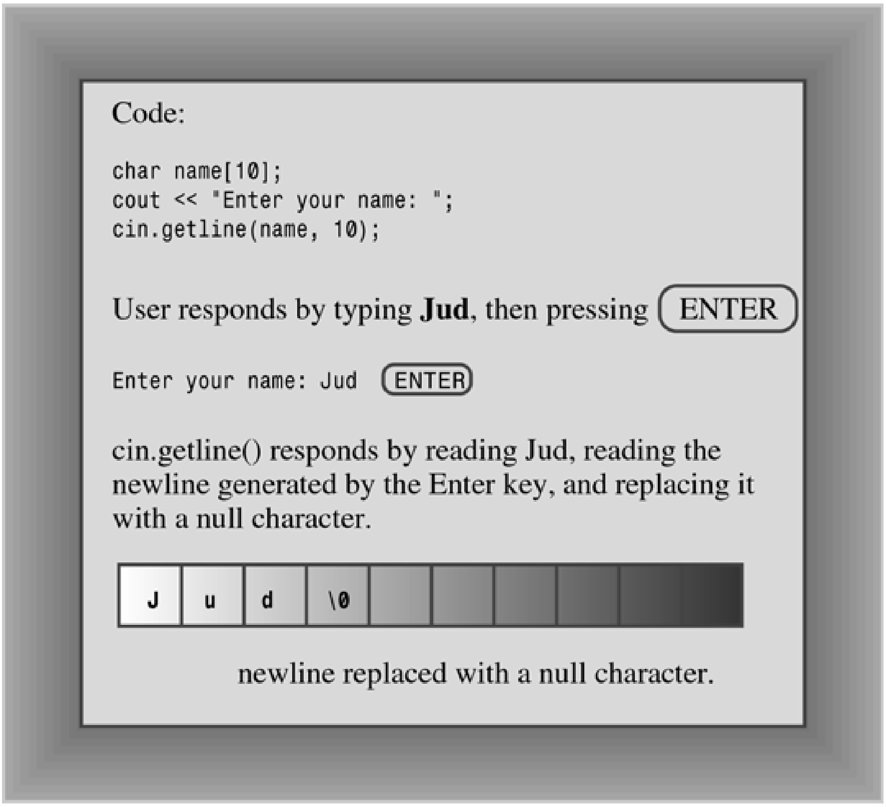
```
Enter your name:
Dirk Hammernose
Enter your favorite dessert:
Radish Torte
I have some delicious Radish Torte for you, Dirk Hammernose.
```

The program now reads complete names and delivers the user his just desserts! The `getline()` function conveniently gets a line at a time. It reads input through the newline character marking the end of the line, but it doesn't save the newline character. Instead, it replaces it with a null character when storing the string (see Figure 4.5).

Line-Oriented Input with `get()`

Let's try another approach. The `istream` class has another member function, `get()`, which comes in several variations. One variant works much like `getline()`. It takes the same arguments, interprets them the same way, and reads to the end of a line. But rather than read and discard the newline character, `get()` leaves that character in the input queue. Suppose you use two calls to `get()` in a row:

```
cin.get(name, ArSize);
cin.get(dessert, Arsize);   // a problem
```


Code:
User responds by typing **Jud**, then pressing ENTER
cin.getline() responds by reading Jud, reading the newline generated by the Enter key, and replacing it with a null character.
J
u
d
\0
newline replaced with a null character.

```
char name[10];
cout << "Enter your name: ";
cin.getline(name, 10);
```

```
Enter your name: Jud ENTER
```

Figure 4.5 `getline()` reads and replaces the newline character.

Because the first call leaves the newline character in the input queue, that newline character is the first character the second call sees. Thus, `get()` concludes that it's reached the end of line without having found anything to read. Without help, `get()` just can't get past that newline character.

Fortunately, there is help in the form of a variation of `get()`. The call `cin.get()` (with no arguments) reads the single next character, even if it is a newline, so you can use it to dispose of the newline character and prepare for the next line of input. That is, this sequence works:

```
cin.get(name, ArSize);      // read first line
cin.get();                  // read newline
cin.get(dessert, Arsize);   // read second line
```

Another way to use `get()` is to *concatenate*, or join, the two class member functions, as follows:

```
cin.get(name, ArSize).get(); // concatenate member functions
```

What makes this possible is that `cin.get(name, ArSize)` returns the `cin` object, which is then used as the object that invokes the `get()` function. Similarly, the following statement reads two consecutive input lines into the arrays `name1` and `name2`; it's equivalent to making two separate calls to `cin.getline()`:

```
cin.getline(name1, ArSize).getline(name2, ArSize);
```

Listing 4.5 uses concatenation. In Chapter 11, "Working with Classes," you'll learn how to incorporate this feature into your class definitions.

Listing 4.5 **`instr3.cpp`**

```
// instr3.cpp -- reading more than one word with get() & get()
#include <iostream>
int main()
{
    using namespace std;
    const int ArSize = 20;
    char name[ArSize];
    char dessert[ArSize];

    cout << "Enter your name:\n";
    cin.get(name, ArSize).get();    // read string, newline
    cout << "Enter your favorite dessert:\n";
    cin.get(dessert, ArSize).get();
    cout << "I have some delicious " << dessert;
    cout << " for you, " << name << ".\n";
    return 0;
}
```

Here is a sample run of the program in Listing 4.5:

```
Enter your name:
Mai Parfait
Enter your favorite dessert:
Chocolate Mousse
I have some delicious Chocolate Mousse for you, Mai Parfait.
```

One thing to note is how C++ allows multiple versions of functions, provided that they have different argument lists. If you use, say, `cin.get(name, ArSize)`, the compiler notices you're using the form that puts a string into an array and sets up the appropriate member function. If, instead, you use `cin.get()`, the compiler realizes you want the form that reads one character. Chapter 8, "Adventures in Functions," explores this feature, which is called *function overloading*.

Why use `get()` instead of `getline()` at all? First, older implementations may not have `getline()`. Second, `get()` lets you be a bit more careful. Suppose, for example, you used `get()` to read a line into an array. How can you tell if it read the whole line rather than stopped because the array was filled? Look at the next input character. If it is a newline character, then the whole line was read. If it is not a newline character, then there is still more input on that line. Chapter 17 investigates this technique. In short, `getline()` is a little simpler to use, but `get()` makes error checking simpler. You can use either one to read a line of input; just keep the slightly different behaviors in mind.

Empty Lines and Other Problems

What happens after `getline()` or `get()` reads an empty line? The original practice was that the next input statement picked up where the last `getline()` or `get()` left off. However, the current practice is that after `get()` (but not `getline()`) reads an empty line, it sets something called the *failbit*. The implications of this act are that further input is blocked, but you can restore input with the following command:

```
cin.clear();
```

Another potential problem is that the input string could be longer than the allocated space. If the input line is longer than the number of characters specified, both `getline()` and `get()` leave the remaining characters in the input queue. However, `getline()` additionally sets the failbit and turns off further input.

Chapters 5, 6, and 17 investigate these properties and how to program around them.

Mixing String and Numeric Input

Mixing numeric input with line-oriented string input can cause problems. Consider the simple program in Listing 4.6.

Listing 4.6 **numstr.cpp**

```
// numstr.cpp -- following number input with line input
#include <iostream>
int main()
{
    using namespace std;
    cout << "What year was your house built?\n";
    int year;
    cin >> year;
    cout << "What is its street address?\n";
    char address[80];
    cin.getline(address, 80);
    cout << "Year built: " << year << endl;
    cout << "Address: " << address << endl;
    cout << "Done!\n";
    return 0;
}
```

Running the program in Listing 4.6 would look something like this:

```
What year was your house built?
1966
What is its street address?
Year built: 1966
Address
Done!
```

You never get the opportunity to enter the address. The problem is that when `cin` reads the year, it leaves the newline generated by the Enter key in the input queue. Then `cin.getline()` reads the newline as an empty line and assigns a null string to the `address` array. The fix is to read and discard the newline before reading the address. This can be done several ways, including by using `get()` with a `char` argument or with no argument, as described in the preceding example. You can make these calls separately:

```
cin >> year;
cin.get();   // or cin.get(ch);
```

Or you can concatenate the calls, making use of the fact that the expression `cin >> year` returns the `cin` object:

```
(cin >> year).get();  // or (cin >> year).get(ch);
```

If you make one of these changes to Listing 4.6, it works properly:

```
What year was your house built?
1966
What is its street address?
43821 Unsigned Short Street
Year built: 1966
Address: 43821 Unsigned Short Street
Done!
```

C++ programs frequently use pointers instead of arrays to handle strings. We'll take up that aspect of strings after talking a bit about pointers. Meanwhile, let's take a look at a more recent way to handle strings: the C++ `string` class.

Introducing the `string` Class

The ISO/ANSI C++98 Standard expanded the C++ library by adding a `string` class. So now, instead of using a character array to hold a string, you can use a type `string` variable (or object, to use C++ terminology). As you'll see, the `string` class is simpler to use than the array and also provides a truer representation of a string as a type.

To use the `string` class, a program has to include the `string` header file. The `string` class is part of the `std` namespace, so you have to provide a `using` directive or declaration or else refer to the class as `std::string`. The class definition hides the array nature of a string and lets you treat a string much like an ordinary variable. Listing 4.7 illustrates some of the similarities and differences between `string` objects and character arrays.

Listing 4.7 **strtype1.cpp**

```
// strtype1.cpp -- using the C++ string class
#include <iostream>
#include <string>              // make string class available
int main()
{
    using namespace std;
    char charr1[20];            // create an empty array
    char charr2[20] = "jaguar"; // create an initialized array
    string str1;                // create an empty string object
    string str2 = "panther";    // create an initialized string

    cout << "Enter a kind of feline: ";
    cin >> charr1;
    cout << "Enter another kind of feline: ";
    cin >> str1;                // use cin for input
    cout << "Here are some felines:\n";
    cout << charr1 << " " << charr2 << " "
         << str1 << " " << str2 // use cout for output
         << endl;
    cout << "The third letter in " << charr2 << " is "
         << charr2[2] << endl;
    cout << "The third letter in " << str2 << " is "
         << str2[2] << endl;    // use array notation

    return 0;
}
```

Here is a sample run of the program in Listing 4.7:

```
Enter a kind of feline: ocelot
Enter another kind of feline: tiger
Here are some felines:
ocelot jaguar tiger panther
The third letter in jaguar is g
The third letter in panther is n
```

You should learn from this example that, in many ways, you can use a string object in the same manner as a character array:

- You can initialize a `string` object to a C-style string.
- You can use `cin` to store keyboard input in a `string` object.
- You can use `cout` to display a `string` object.
- You can use array notation to access individual characters stored in a `string` object.

The main difference between `string` objects and character arrays shown in Listing 4.7 is that you declare a `string` object as a simple variable, not as an array:

```
string str1;                // create an empty string object
string str2 = "panther";    // create an initialized string
```

The class design allows the program to handle the sizing automatically. For instance, the declaration for `str1` creates a `string` object of length zero, but the program automatically resizes `str1` when it reads input into `str1`:

```
cin >> str1;                // str1 resized to fit input
```

This makes using a `string` object both more convenient and safer than using an array. Conceptually, one thinks of an array of `char` as a collection of `char` storage units used to store a string but of a `string` class variable as a single entity representing the string.

C++11 String Initialization

As you might expect by now, C++11 enables list-initialization for C-style strings and `string` objects:

```
char first_date[] = {"Le Chapon Dodu"};
char second_date[] {"The Elegant Plate"};
string third_date = {"The Bread Bowl"};
string fourth_date {"Hank's Fine Eats"};
```

Assignment, Concatenation, and Appending

The `string` class makes some operations simpler than is the case for arrays. For example, you can't simply assign one array to another. But you can assign one string object to another:

```
char charr1[20];            // create an empty array
char charr2[20] = "jaguar"; // create an initialized array
string str1;                // create an empty string object
string str2 = "panther";    // create an initialized string
charr1 = charr2;            // INVALID, no array assignment
str1 = str2;                // VALID, object assignment ok
```

The `string` class simplifies combining strings. You can use the `+` operator to add two `string` objects together and the `+=` operator to tack on a string to the end of an existing `string` object. Continuing with the preceding code, we have the following possibilities:

```
string str3;
str3 = str1 + str2;         // assign str3 the joined strings
str1 += str2;               // add str2 to the end of str1
```

Listing 4.8 illustrates these usages. Note that you can add and append C-style strings as well as `string` objects to a `string` object.

Listing 4.8 **strtype2.cpp**

```
// strtype2.cpp -- assigning, adding, and appending
#include <iostream>
#include <string>               // make string class available
int main()
{
    using namespace std;
    string s1 = "penguin";
    string s2, s3;

    cout << "You can assign one string object to another: s2 = s1\n";
    s2 = s1;
    cout << "s1: " << s1 << ", s2: " << s2 << endl;
    cout << "You can assign a C-style string to a string object.\n";
    cout << "s2 = \"buzzard\"\n";
    s2 = "buzzard";
    cout << "s2: " << s2 << endl;
    cout << "You can concatenate strings: s3 = s1 + s2\n";
    s3 = s1 + s2;
    cout << "s3: " << s3 << endl;
    cout << "You can append strings.\n";
    s1 += s2;
    cout <<"s1 += s2 yields s1 = " << s1 << endl;
    s2 += " for a day";
    cout <<"s2 += \" for a day\" yields s2 = " << s2 << endl;

    return 0;
}
```

Recall that the escape sequence `\"` represents a double quotation mark that is used as a literal character rather than as marking the limits of a string. Here is the output from the program in Listing 4.8:

```
You can assign one string object to another: s2 = s1
s1: penguin, s2: penguin
You can assign a C-style string to a string object.
s2 = "buzzard"
s2: buzzard
You can concatenate strings: s3 = s1 + s2
s3: penguinbuzzard
You can append strings.
s1 += s2 yields s1 = penguinbuzzard
s2 += " for a day" yields s2 = buzzard for a day
```

More `string` Class Operations

Even before the string class was added to C++, programmers needed to do things like assign strings. For C-style strings, they used functions from the C library for these tasks. The `cstring` header file (formerly `string.h`) supports these functions. For example, you can use the `strcpy()` function to copy a string to a character array, and you can use the `strcat()` function to append a string to a character array:

```
strcpy(charr1, charr2);  // copy charr2 to charr1
strcat(charr1, charr2);  // append contents of charr2 to char1
```

Listing 4.9 compares techniques used with `string` objects with techniques used with character arrays.

Listing 4.9 **`strtype3.cpp`**

```
// strtype3.cpp -- more string class features
#include <iostream>
#include <string>              // make string class available
#include <cstring>             // C-style string library
int main()
{
    using namespace std;
    char charr1[20];
    char charr2[20] = "jaguar";
    string str1;
    string str2 = "panther";

    // assignment for string objects and character arrays
    str1 = str2;                // copy str2 to str1
    strcpy(charr1, charr2);     // copy charr2 to charr1

    // appending for string objects and character arrays
    str1 += " paste";           // add paste to end of str1
    strcat(charr1, " juice");   // add juice to end of charr1

    // finding the length of a string object and a C-style string
    int len1 = str1.size();     // obtain length of str1
    int len2 = strlen(charr1);  // obtain length of charr1

    cout << "The string " << str1 << " contains "
         << len1 << " characters.\n";
    cout << "The string " << charr1 << " contains "
         << len2 << " characters.\n";

    return 0;
}
```

Here is the output:

```
The string panther paste contains 13 characters.
The string jaguar juice contains 12 characters.
```

The syntax for working with string objects tends to be simpler than using the C string functions. This is especially true for more complex operations. For example, the C library equivalent of

```
str3 = str1 + str2;
```

is this:

```
strcpy(charr3, charr1);
strcat(charr3, charr2);
```

Furthermore, with arrays, there is always the danger of the destination array being too small to hold the information, as in this example:

```
char site[10] = "house";
strcat(site, " of pancakes");  // memory problem
```

The `strcat()` function would attempt to copy all 12 characters into the `site` array, thus overrunning adjacent memory. This might cause the program to abort, or the program might continue running but with corrupted data. The `string` class, with its automatic resizing as necessary, avoids this sort of problem. The C library does provide cousins to `strcat()` and `strcpy()`, called `strncat()` and `strncpy()`, that work more safely by taking a third argument to indicate the maximum allowed size of the target array, but using them adds another layer of complexity in writing programs.

Notice the different syntax used to obtain the number of characters in a string:

```
int len1 = str1.size();      // obtain length of str1
int len2 = strlen(charr1);  // obtain length of charr1
```

The `strlen()` function is a regular function that takes a C-style string as its argument and that returns the number of characters in the string. The `size()` function basically does the same thing, but the syntax for it is different. Instead of appearing as a function argument, `str1` precedes the function name and is connected to it with a dot. As you saw with the `put()` method in Chapter 3, this syntax indicates that `str1` is an object and that `size()` is a class method. A method is a function that can be invoked only by an object belonging to the same class as the method. In this particular case, `str1` is a `string` object, and `size()` is a `string` method. In short, the C functions use a function argument to identify which string to use, and the C++ `string` class objects use the object name and the dot operator to indicate which string to use.

More on `string` Class I/O

As you've seen, you can use `cin` with the `>>` operator to read a `string` object and `cout` with the `<<` operator to display a `string` object using the same syntax you use with a

C-style string. But reading a line at a time instead of a word at time uses a different syntax. Listing 4.10 shows this difference.

Listing 4.10 **`strtype4.cpp`**

```
// strtype4.cpp -- line input
#include <iostream>
#include <string>               // make string class available
#include <cstring>              // C-style string library
int main()
{
    using namespace std;
    char charr[20];
    string str;

    cout << "Length of string in charr before input: "
         << strlen(charr) << endl;
    cout << "Length of string in str before input: "
         << str.size() << endl;
    cout << "Enter a line of text:\n";
    cin.getline(charr, 20);     // indicate maximum length
    cout << "You entered: " << charr << endl;
    cout << "Enter another line of text:\n";
    getline(cin, str);          // cin now an argument; no length specifier
    cout << "You entered: " << str << endl;
    cout << "Length of string in charr after input: "
         << strlen(charr) << endl;
    cout << "Length of string in str after input: "
         << str.size() << endl;

    return 0;
}
```

Here's a sample run of the program in Listing 4.10:

```
Length of string in charr before input: 27
Length of string in str before input: 0
Enter a line of text:
peanut butter
You entered: peanut butter
Enter another line of text:
blueberry jam
You entered: blueberry jam
Length of string in charr after input: 13
Length of string in str after input: 13
```

Note that the program says the length of the string in the array `charr` before input is 27, which is larger than the size of the array! Two things are going on here. The first is that the contents of an uninitialized array are undefined. The second is that the `strlen()` function works by starting at the first element of the array and counting bytes until it reaches a null character. In this case, the first null character doesn't appear until several bytes *after* the end of the array. Where the first null character appears in uninitialized data is essentially random, so you very well could get a different numeric result using this program.

Also note that the length of the string in `str` before input is 0. That's because an uninitialized `string` object is automatically set to zero size.

This is the code for reading a line into an array:

```
cin.getline(charr, 20);
```

The dot notation indicates that the `getline()` function is a class method for the `istream` class. (Recall that `cin` is an `istream` object.) As mentioned earlier, the first argument indicates the destination array, and the second argument is the array size, which `getline()` used to avoid overrunning the array.

This is the code for reading a line into a `string` object:

```
getline(cin,str);
```

There is no dot notation, which indicates that *this* `getline()` is *not* a class method. So it takes `cin` as an argument that tells it where to find the input. Also there isn't an argument for the size of the string because the `string` object automatically resizes to fit the string.

So why is one `getline()` an `istream` class method and the other `getline()` not? The `istream` class was part of C++ long before the `string` class was added. So the `istream` design recognizes basic C++ types such as `double` and `int`, but it is ignorant of the `string` type. Therefore, there are `istream` class methods for processing `double`, `int`, and the other basic types, but there are no `istream` class methods for processing `string` objects.

Because there are no `istream` class methods for processing string objects, you might wonder why code like this works:

```
cin >> str;  // read a word into the str string object
```

It turns out that code like the following does (in disguised notation) use a member function of the `istream` class:

```
cin >> x;  // read a value into a basic C++ type
```

But the `string` class equivalent uses a friend function (also in disguised notation) of the `string` class. You'll have to wait until Chapter 11 to see what a friend function is and how this technique works. In the meantime, you can use `cin` and `cout` with `string` objects and not worry about the inner workings.

Other Forms of String Literals

C++, recall, has the `wchar_t` type in addition to `char`. And C++11 adds the `char16_t` and `char32_t` types. It's possible to create arrays of these types and string literals of these types. C++ uses the `L`, `u`, and `U` prefixes, respectively, for string literals of these types. Here's an example of how they can be used:

```
wchar_t title[] = L"Chief Astrogator";  // w_char string
char16_t name[] = u"Felonia Ripova";    // char_16 string
char32_t car[] = U"Humber Super Snipe"; // char_32 string
```

C++11 also supports an encoding scheme for Unicode characters called UTF-8. In this scheme a given character may be stored in anywhere from one 8-bit unit, or octet, to four 8-bit units, depending on the numeric value. C++ uses the `u8` prefix to indicate string literals of that type.

Another C++11 addition is the raw string. In a raw string, characters simply stand for themselves. For example, the sequence `\n` is not interpreted as representing the newline character; instead, it is two ordinary characters, a backslash and an `n`, and it would display as those two characters onscreen. As another example, you can use a simple `"` inside a string instead of the more awkward `\"` we used in Listing 4.8. Of course, if you allow a `"` inside a string literal, you no longer can use it to delimit the ends of a string. Therefore, raw strings use `"(` and `)"` as delimiters, and they use an `R` prefix to identify them as raw strings:

```
cout << R"(Jim "King" Tutt uses "\n" instead of endl.)" << '\n';
```

This would display the following:

```
Jim "King" Tutt uses \n instead of endl.
```

The standard string literal equivalent would be this:

```
cout << "Jim \"King\" Tutt uses \" \\n\" instead of endl." << '\n';
```

Here we had to use `\\` to display `\` because a single `\` is interpreted as the first character of an escape sequence.

If you press the Enter or Return key while typing a raw string, that not only moves the cursor to the next line onscreen, it also places a carriage return character in the raw string.

What if you want to display the combination `)"` in a raw string? (Who wouldn't?) Won't the compiler interpret the first occurrence of `)"` as the end of the string? Yes, it will. But the raw string syntax allows you to place additional characters between the opening `"` and `(`. This implies that the same additional characters must appear between the final `)` and `"`. So a raw string beginning with `R"+*(` must terminate with `)+*"`. Thus, the statement

```
cout << R"+*("(Who wouldn't?)", she whispered.)+*" << endl;
```

would display the following:

```
"(Who wouldn't?)", she whispered.
```

In short, the default delimiters of `"(` and `)"` have been replaced with `"+*(` and `)+*"`. You can use any of the members of the basic character set as part of the delimiter other than the space, the left parenthesis, the right parenthesis, the backslash, and control characters such as a tab or a newline.

The `R` prefix can be combined with the other string prefixes to produce raw strings of `wchar_t` and so on. It can be either the first or the last part of a compound prefix: `Ru`, `UR`, and so on.

Now let's go on to another compound type—the structure.

Introducing Structures

Suppose you want to store information about a basketball player. You might want to store his or her name, salary, height, weight, scoring average, free-throw percentage, assists, and so on. You'd like some sort of data form that could hold all this information in one unit. An array won't do. Although an array can hold several items, each item has to be the same type. That is, one array can hold 20 `int`s and another can hold 10 `float`s, but a single array can't store `int`s in some elements and `float`s in other elements.

The answer to your desire (the one about storing information about a basketball player) is the C++ structure. A *structure* is a more versatile data form than an array because a single structure can hold items of more than one data type. This enables you to unify your data representation by storing all the related basketball information in a single structure variable. If you want to keep track of a whole team, you can use an array of structures. The structure type is also a stepping stone to that bulwark of C++ OOP, the class. Learning a little about structures now takes you that much closer to the OOP heart of C++.

A structure is a user-definable type, with a structure declaration serving to define the type's data properties. After you define the type, you can create variables of that type. Thus, creating a structure is a two-part process. First, you define a structure description that describes and labels the different types of data that can be stored in a structure. Then you can create structure variables, or, more generally, structure data objects, that follow the description's plan.

For example, suppose that Bloataire, Inc., wants to create a type to describe members of its product line of designer inflatables. In particular, the type should hold the name of the item, its volume in cubic feet, and its selling price. Here is a structure description that meets those needs:

```
struct inflatable   // structure declaration
{
    char name[20];
    float volume;
    double price;
};
```

The keyword `struct` indicates that the code defines the layout for a structure. The identifier `inflatable` is the name, or *tag*, for this form; this makes `inflatable` the name