Arrays

An array is simply a number of memory locations called **array elements** or simply **elements**, each of which can store an item of data of the same given data type and which are all referenced through the same variable name.

Individual items in an array are specified by an index value which is simply an integer representing the sequence number of the elements in the array, the first having the sequence number 0, the second 1, and so on. You can also envisage the index value of an array element as being an offset from the first element in an array. The first element has an offset of 0 and therefore an index of 0, and an index value of 3 will refer to the fourth element of an array. The basic structure of an array is illustrated in Figure 1.

The name height has six elements, each storing a different value. Because there are six elements, the index values run from 0 through 5. To refer to a particular element, you write the array name, followed by the index value of the particular element between square brackets. The third element is referred to as height[2], for example. If you think of the index as being the offset from the first element, it's easy to see that the index value for the fourth element will be 3, for example.

The amount of memory required to store each element is determined by its type, and all the elements of an array are stored in a contiguous block of memory.

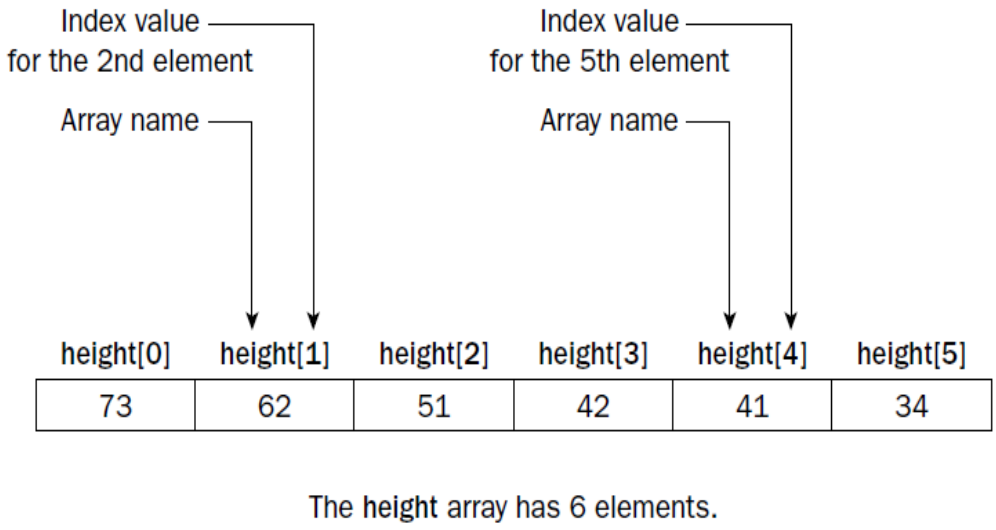


Figure 1

Declaring Arrays

You declare an array in essentially the same way as you declared the variables that you have seen up to now, the only difference being that the number of elements in the array is specified between square brackets immediately following the array name. For example, you could declare the

integer array height, shown in the previous figure, with the following declaration statement:

```
int height[6];
```

Because each int value occupies 4 bytes in memory, the whole array requires 24 bytes. Arrays can be of any size, subject to the constraints imposed by the amount of memory in the computer on which your program is running. You can declare arrays to be of any type.

Initializing Arrays

To initialize an array in its declaration, you put the initializing values separated by commas between braces, and you place the set of initial values following an equals sign after the array name. Here's an example of how you can declare and initialize an array:

```
int list[5] = { 200, 250, 300, 350, 400 };
```

The array has the name cubic_inches and has five elements that each stores a value of type int. The values in the initializing list between the braces correspond to successive index values of the array, so in this case **list [0]** has the value 200, **list [1]** the value 250, **list [2]** the value 300, and so on.

You must not specify more initializing values than there are elements in the array, but you can include fewer. If there are fewer, the values are assigned to successive elements, starting with the first element which is the one corresponding to the index value 0. The array elements for which you didn't provide an initial value is initialized with zero. This isn't the same as supplying no initializing list. Without an initializing list, the array elements contain junk values. Also, if you include an initializing list, there must be at least one initializing value in it; otherwise the compiler generates an error message. We can illustrate this with the following rather limited example.

```
// Demonstrating array initialization
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
        int value[5] = { 7, 12, 34 };
        int junk [5];
        cout << endl;
        for (int i = 0; i < 5; i++)
                cout << setw(12) << value[i];
        cout << endl;
        for (int i = 0; i < 5; i++)
                cout << setw(12) << junk[i];
        cout << endl;
        return 0;
}
```

In this example, you declare two arrays, the first of which, value, you initialize in part, and the second, junk, you don't initialize at all. The program generates two lines of output, which on my computer look like this:

```
     7          12          34          0          0
11342688        0           1          0         -1
```

The second line (corresponding to values of junk[0] to junk[4]) may well be different on your computer.

How It Works

The first three values of the array value are the initializing values and the last two have the default value of 0. In the case of junk, all the values are spurious because you didn't provide any initial values at all. The array elements contain whatever values were left there by the program that last used these memory locations.

A convenient way to initialize a whole array to zero is simply to specify a single initializing value as 0. For example:

```
int data[100] = {0}; // Initialize all elements to zero
```

This statement declares the array data, with all one hundred elements initialized with 0. The first element is initialized by the value you have between the braces and the remaining elements are initialized to zero because you omitted values for these.

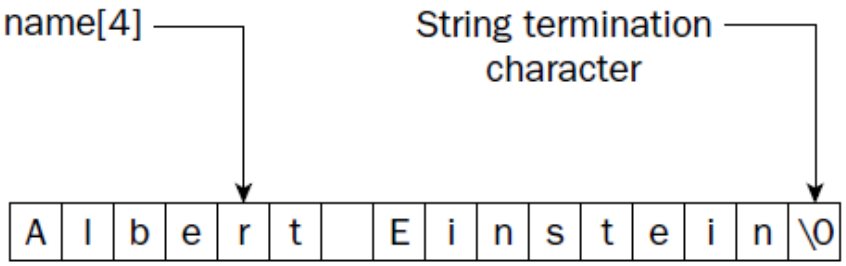
You can also omit the dimension of an array of numeric type, providing you supply initializing values. The number of elements in the array is determined by the number of initializing values you specify. For example, the array declaration

```
int value[ ] = { 2, 3, 4 };
```

defines an array with three elements that have the initial values 2, 3, and 4.

Character Arrays and String Handling

An array of type char is called a **character array** and is generally used to store a character string. A character string is a sequence of characters with a special character appended to indicate the end of the string. The string terminating character indicates the end of the string and this character is defined by the escape sequence **'\0'**, and is sometimes referred to as a **null character**, being a byte with all bits as zero. A string of this form is often referred to as a C-style string because defining a string in this way was introduced in the C language from which C++ was developed. The representation of a C-style string in memory is shown in Figure 2.



char name[] = “Albert Einstein”;

"Each character in a string occupies one byte"

Figure 2

Each character in the string occupies one byte, so together with the terminating null character, a string requires a number of bytes that is one greater than the number of characters contained in the string.

You can declare a character array and initialize it with a string literal. For example:

```
char citystring[11] = “Duhok City”;
```

Note that the terminating **‘\0’** is supplied automatically by the compiler. If you include one explicitly in the string literal, you end up with two of them. You must, however, allow for the terminating null in the number of elements that you allot to the array.

You can let the compiler work out the length of an initialized array for you, as you saw in Figure 2. Here’s another example:

```
char citystring[ ] = “Duhok City”;
```

Because the dimension is unspecified, the compiler allocates space for enough elements to hold the initializing string, plus the terminating null character. In this case it allocates **11** elements for the array **citystring**. Of course, if you want to use this array later for storing a different string, its length (including the terminating null character) must not exceed **11** bytes. In general, it is your responsibility to ensure that the array is large enough for any string you might subsequently want to store.

String Input

The <iostream> header file contains definitions of a number of functions for reading characters from the keyboard. The one that you’ll look at here is the function getline(), which reads a sequence of characters entered through the keyboard and stores it in a character array as a string terminated by \0. You typically use the getline() function statements like this:

```
const int MAX = 80; // Maximum string length including \0
char name[MAX]; // Array to store a string
cin.getline(name, MAX, ‘\n’); // Read input line as a string
```

These statements first declare a char array name with **MAX** elements and then read characters from **cin** using the function getline(). The source of the data, **cin**, is written as shown, with a period separating it from the function name. The significance of the arguments to the getline() function is shown in Figure 3.

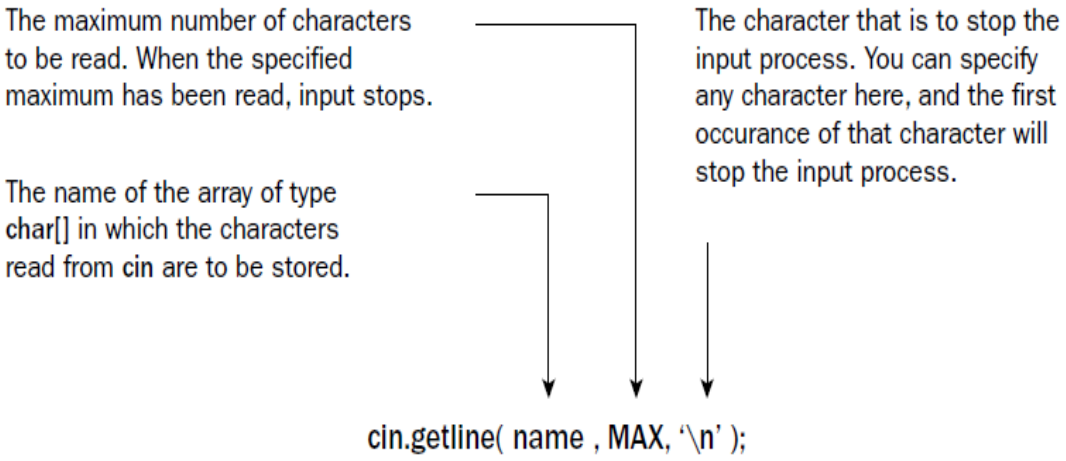


Figure 3

Because the last argument to the **getline()** function is **'\n'** (newline or end line character) and the second argument is **MAX**, characters are read from **cin** until the **'\n'** character is read, or when **MAX – 1** characters have been read, whichever occurs first. The maximum number of characters read is **MAX – 1** rather than **MAX** to allow for the **'\0'** character to be appended to the sequence of characters stored in the array. The **'\n'** character is generated when you press the ***Return*** key on your keyboard and is therefore usually the most convenient character to end input. You can, however, specify something else by changing the last argument. The **'\n'** isn't stored in the input array name, but as we said, a **'\0'** is added at the end of the input string in the array.

```
// Counting string characters
#include <iostream>
using namespace std;
int main()
{
        const int MAX = 80; // Maximum array dimension
        char buffer[MAX]; // Input buffer
        int count = 0; // Character count

        cout << "Enter a string of less than 80 characters:\n";
        cin.getline(buffer, MAX, '\n'); // Read a string until \n

        while (buffer[count] != '\0') // Increment count as long as
                count++;              // the current character is not null

        cout << endl<< "The string \"" << buffer<< "\" has " <<
        count << " characters.";

        cout << endl;
        return 0;
}
```

Typical output from this program is as follows:

Enter a string of less than 80 characters:
College of Science Dept. of Computer Science

The string "College of Science Dept. of Computer Science" has 44 characters.

How It Works

This program declares a character array buffer and reads a character string into the array from the keyboard after displaying a prompt for the input. Reading from the keyboard ends when the user presses ***Return***, or when **MAX-1** characters have been read.

A while loop is used to count the number of characters read. The loop continues as long as the current character referenced with **buffer[count]** is not **'\0'**. The only action in the loop is to increment count for each non-null character.

There is also a library function, **strlen()**, that can save you the trouble of coding it yourself. If you use it, you need to include the **<cstring>** header file in your program with an #include directive like this:

#include <cstring>