Module 4
Arrays, Strings, and Pointers

Table of Contents

This module discusses arrays, strings, and pointers. Although these may seem to be three disconnected topics, they aren't. In C++ they are intertwined, and an understanding of one aids in the understanding of the others.

An array is a collection of variables of the same type that are referred to by a common name. Arrays may have from one to several dimensions, although the one-dimensional array is the most common. Arrays offer a convenient means of creating lists of related variables.

The array that you will probably use most often is the character array, because it is used to hold a character string. The C++ language does not define a built-in string data type. Instead, strings are implemented as arrays of characters. This approach to strings allows greater power and flexibility than are available in languages that use a distinct string type.

A pointer is an object that contains a memory address. Typically, a pointer is used to access the value of another object. Often this other object is an array. In fact, pointers and arrays are related to each other more than you might expect.

CRITICAL SKILL 4.1: Use one-dimensional arrays

A one-dimensional array is a list of related variables. Such lists are common in programming. For example, you might use a one-dimensional array to store the account numbers of the active users on a network. Another array might store the current batting averages for a baseball team. When computing the average of a list of values, you will often use an array to hold the values. Arrays are fundamental to modern programming.

The general form of a one-dimensional array declaration is

type name[size];

Here, type declares the base type of the array. The base type determines the data type of each element that makes up the array. The number of elements the array can hold is specified by size. For example, the following declares an integer array named sample that is ten elements long:

int sample[10];

An individual element within an array is accessed through an index. An index describes the position of an element within an array. In C++, all arrays have zero as the index of their first element. Because sample has ten elements, it has index values of 0 through 9. You access an array element by indexing the array using the number of the element. To index an array, specify the number of the element you want, surrounded by square brackets. Thus, the first element in sample is sample[0], and the last element is sample[9]. For example, the following program loads sample with the numbers 0 through 9:

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

int main()
{
  int sample[10]; // this reserves 10 integer elements
  int t;

  // load the array
  for(t=0; t<10; ++t) sample[t] = t;

  // display the array
  for(t=0; t<10; ++t)
    cout << "This is sample[" << t << "]: " << sample[t] << "\n";

  return 0;
}
```

Notice how **sample** is indexed.

The output from this example is shown here:

```
This is sample[0]: 0
```

```
This is sample[1]: 1
This is sample[2]: 2
This is sample[3]: 3
This is sample[4]: 4
This is sample[5]: 5
This is sample[6]: 6
This is sample[7]: 7
This is sample[8]: 8
This is sample[9]: 9
```

In C++, all arrays consist of contiguous memory locations. (That is, all array elements reside next to each other in memory.) The lowest address corresponds to the first element, and the highest address corresponds to the last element. For example, after this fragment is run:

```
int nums[5];
int i;

for(i=0; i<5; i++) nums[i] = i;
```

nums looks like this:

nums[0]	nums[1]	nums[2]	nums[3]	nums[4]
0	1	2	3	4

Arrays are common in programming because they let you deal easily with sets of related variables. Here is an example. The following program creates an array of ten elements and assigns each element a value. It then computes the average of those values and finds the minimum and the maximum value.

```
/*
   Compute the average and find the minimum
   and maximum of a set of values.
*/

#include <iostream>
using namespace std;

int main()
{
  int i, avg, min_val, max_val;
  int nums[10];

  nums[0] = 10;
  nums[1] = 18;
  nums[2] = 75;
  nums[3] = 0;
  nums[4] = 1;
  nums[5] = 56;
  nums[6] = 100;
  nums[7] = 12;
  nums[8] = -19;
  nums[9] = 88;

  // compute the average
  avg = 0;
  for(i=0; i<10; i++)
    avg += nums[i]; <-- Sum the values in nums.

  avg /= 10; <-- Calculate the average.

  cout << "Average is " << avg << '\n';

  // find minimum and maximum values
  min_val = max_val = nums[0];
  for(i=1; i<10; i++) {
    if(nums[i] < min_val) min_val = nums[i];
    if(nums[i] > max_val) max_val = nums[i];
  }

  cout << "Minimum value: " << min_val << '\n';
  cout << "Maximum value: " <<  max_val << '\n';

  return 0;
}
```

Find the minimum and maximum values in **nums**.

The output from the program is shown here:

```
Average is 34
Minimum value: -19
```

```
Maximum value: 100
```

Notice how the program cycles through the elements in the nums array. Storing the values in an array makes this process easy. As the program illustrates, the loop control variable of a for loop is used as an index. Loops such as this are very common when working with arrays.

There is an array restriction that you must be aware of. In C++, you cannot assign one array to another. For example, the following is illegal:

```
int a[10], b[10];

// ...

a = b; // error -- illegal
```

To transfer the contents of one array into another, you must assign each value individually, like this:

```
for(i=0; i < 10; i++) a[i] = b[i];
```

No Bounds Checking

C++ performs no bounds checking on arrays. This means that there is nothing that stops you from overrunning the end of an array. In other words, you can index an array of size N beyond N without generating any compile-time or runtime error messages, even though doing so will often cause catastrophic program failure. For example, the compiler will compile and run the following code without issuing any error messages even though the array crash is being overrun:

```
int crash[10], i;

for(i=0; i<100; i++) crash[i]=i;
```

In this case, the loop will iterate 100 times, even though crash is only ten elements long! This causes memory that is not part of crash to be overwritten.

Ask the Expert

Q: Since overrunning an array can lead to catastrophic failures, why doesn't C++ provide bounds checking on array operations?

A: C++ was designed to allow professional programmers to create the fastest, most efficient code possible. Toward this end, very little runtime error checking is included, because it slows (often dramatically) the execution of a program. Instead, C++ expects you, the programmer, to be responsible enough to prevent array overruns in the first place, and to add appropriate error checking on your own

as needed. Also, it is possible for you to define array types of your own that perform bounds checking if your program actually requires this feature.

If an array overrun occurs during an assignment operation, memory that is being used for other purposes, such as holding other variables, might be overwritten. If an array overrun occurs when data is being read, then invalid data will corrupt the program. Either way, as the programmer, it is your job both to ensure that all arrays are large enough to hold what the program will put in them, and to provide bounds checking whenever necessary.

1. What is a one-dimensional array?
2. The index of the first element in an array is always zero. True or false?
3. Does C++ provide bounds checking on arrays?

CRITICAL SKILL 4.2: Two-Dimensional Arrays

C++ allows multidimensional arrays. The simplest form of the multidimensional array is the two-dimensional array. A two-dimensional array is, in essence, a list of one-dimensional arrays. To declare a two-dimensional integer array twoD of size 10,20, you would write

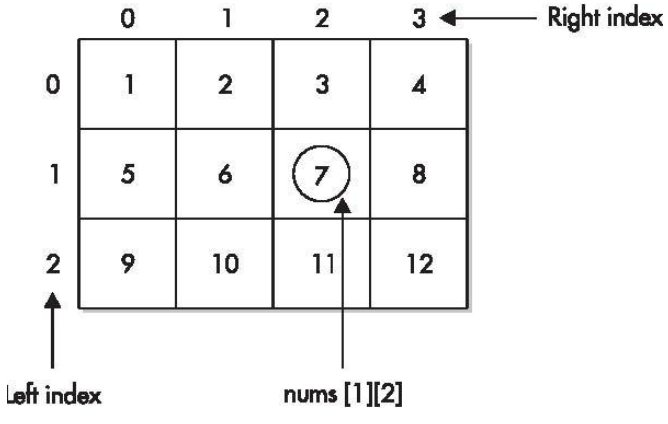
```
int twoD[10][20];
```

Pay careful attention to the declaration. Unlike some other computer languages, which use commas to separate the array dimensions, C++ places each dimension in its own set of brackets. Similarly, to access an element, specify the indices within their own set of brackets. For example, for point 3,5 of array twoD, you would use twoD[3][5].

In the next example, a two-dimensional array is loaded with the numbers 1 through 12.

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

int main()
{
  int t,i, nums[3][4];

  for(t=0; t < 3; ++t) {
    for(i=0; i < 4; ++i) {
      nums[t][i] = (t*4)+i+1; <-- Indexing nums requires two indexes.
      cout << nums[t][i] << ' ';
    }
    cout << '\n';
  }

  return 0;
}
```

In this example, nums[0][0] will have the value 1, nums[0][1] the value 2, nums[0][2] the value 3, and so on. The value of nums[2][3] will be 12. Conceptually, the array will look like that shown here:



Two-dimensional arrays are stored in a row-column matrix, where the first index indicates the row and the second indicates the column. This means that when array elements are accessed in the order in which they are actually stored in memory, the right index changes faster than the left.

You should remember that storage for all array elements is determined at compile time. Also, the memory used to hold an array is required the entire time that the array is in existence. In the case of a two-dimensional array, you can use this formula to determine the number of bytes of memory that are needed:

bytes = number of rows × number of columns × number of bytes in type

Therefore, assuming four-byte integers, an integer array with dimensions 10,5 would have 10×5×4 (or 200) bytes allocated.

CRITICAL SKILL 4.3: Multidimensional Arrays

C++ allows arrays with more than two dimensions. Here is the general form of a multidimensional array declaration:

type name[size1][size2]...[sizeN];

For example, the following declaration creates a 4×10×3–integer array:

int multidim[4][10][3];

Arrays of more than three dimensions are not often used, due to the amount of memory required to hold them. Remember, storage for all array elements is allocated during the entire lifetime of an array. When multidimensional arrays are used, large amounts of memory can be consumed. For example, a four-dimensional character array with dimensions 10,6,9,4 would require 10×6×9×4 (or 2,160) bytes. If each array dimension is increased by a factor of 10 each (that is, 100×60×90×40), then the memory required for the array increases to 21,600,000 bytes! As you can see, large multidimensional arrays may cause a shortage of memory for other parts of your program. Thus, a program with arrays of more than two or three dimensions may find itself quickly out of memory!

Progress Check

1. Each dimension in a multidimensional array is specified with its own set of brackets. True or false?
2. Show how to declare a two-dimensional integer array called **list** with the dimensions 4×9.
3. Given **list** from the preceding question, show how to access element 2, 3.

Project 4-1 Sorting an Array

Because a one-dimensional array organizes data into an indexable linear list, it isthe perfect data structure for sorting. In this project, you will learn a simple way to sort an array. As you may know, there are a number of different sorting algorithms. The quick sort, the shaker sort, and the shell sort are just three. However, the best known, simplest, and easiest to understand sorting algorithm is called the bubble sort. While the bubble sort is not very efficient—in fact, its performance is unacceptable for sorting large arrays—it may be used effectively for sorting small ones.

Step by Step

1. Create a file called Bubble.cpp.

2. The bubble sort gets its name from the way it performs the sorting operation. It uses repeated comparison and, if necessary, exchange of adjacent elements in the array. In this process, small values move toward one end, and large ones toward the other end. The process is conceptually similar to bubbles finding their own level in a tank of water. The bubble sort operates by making several passes through the array, exchanging out-of-place elements when necessary. The number of passes required to ensure that the array is sorted is equal to one less than the number of elements in the array.

   Here is the code that forms the core of the bubble sort. The array being sorted is called nums.

```
// This is the bubble sort.
for(a=1; a<size; a++)
  for(b=size-1; b>=a; b--) {
    if(nums[b-1] > nums[b]) { // if out of order
      // exchange elements
      t = nums[b-1];
      nums[b-1] = nums[b];
      nums[b] = t;
    }
  }
```

   Notice that the sort relies on two for loops. The inner loop checks adjacent elements in the array, looking for out-of-order elements. When an out-of-order element pair is found, the two elements are exchanged. With each pass, the smallest element of those remaining moves into its proper location. The outer loop causes this process to repeat until the entire array has been sorted.

3. Here is the entire Bubble.cpp program:

```
/*
   Project 4-1
   Demonstrate the Bubble sort.
*/
#include <iostream>
#include <cstdlib>
using namespace std;

int main()
{
  int nums[10];
  int a, b, t;
  int size;

  size = 10; // number of elements to sort

  // give the array some random initial values
  for(t=0; t<size; t++) nums[t] = rand();

  // display original array
  cout << "Original array is:\n   ";
  for(t=0; t<size; t++) cout << nums[t] << ' ';
  cout << '\n';

  // This is the bubble sort.
  for(a=1; a<size; a++)
    for(b=size-1; b>=a; b--) {
      if(nums[b-1] > nums[b]) { // if out of order
        // exchange elements
        t = nums[b-1];
        nums[b-1] = nums[b];
        nums[b] = t;
      }
    }

  // display sorted array
  cout << "\nSorted array is:\n   ";
  for(t=0; t<size; t++) cout << nums[t] << ' ';

  return 0;
}
```

The output is shown here:

Original array is: 41 18467 6334 26500 19169 15724 11478 29358 26962 24464

Sorted array is: 41 6334 11478 15724 18467 19169 24464 26500 26962 29358

4. Although the bubble sort is good for small arrays, it is not efficient when used on larger ones. The best general-purpose sorting algorithm is the Quicksort. The Quicksort, however, relies on features of C++ that you have not yet learned. Also, the C++ standard library contains a function

called qsort( ) that implements a version of the Quicksort, but to use it, you will also need to know more about C++.

CRITICAL SKILL 4.4: Strings

By far the most common use for one-dimensional arrays is to create character strings. C++ supports two types of strings. The first, and most commonly used, is the null-terminated string, which is a null-terminated character array. (A null is zero.) Thus, a null-terminated string contains the characters that make up the string followed by a null. Null-terminated strings are widely used because they offer a high level of efficiency and give the programmer detailed control over string operations. When a C++ programmer uses the term string, he or she is usually referring to a null-terminated string. The second type of string defined by C++ is the string class, which is part of the C++ class library. Thus, string is not a built-in type. It provides an object-oriented approach to string handling but is not as widely used as the null-terminated string. Here, null-terminated strings are examined.

String Fundamentals

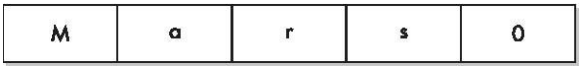
When declaring a character array that will hold a null-terminated string, you need to declare it one character longer than the largest string that it will hold. For example, if you want to declare an array str that could hold a 10-character string, here is what you would write:

```
char str[11];
```

Specifying the size as 11 makes room for the null at the end of the string. As you learned earlier in this book, C++ allows you to define string constants. A string constant is a list of characters enclosed in double quotes. Here are some examples:

“hello there” “I like C++” “Mars” ““

It is not necessary to manually add the null terminator onto the end of string constants; the C++ compiler does this for you automatically. Therefore, the string “Mars” will appear in memory like this:



The last string shown is "". This is called a null string. It contains only the null terminator and no other characters. Null strings are useful because they represent the empty string.

Reading a String from the Keyboard

The easiest way to read a string entered from the keyboard is to use a char array in a cin statement. For example, the following program reads a string entered by the user:

```
// Using cin to read a string from the keyboard.

#include <iostream>
using namespace std;

int main()
{
  char str[80];

  cout << "Enter a string: ";
  cin >> str; // read string from keyboard
  cout << "Here is your string: ";
  cout << str;
  return 0;
}
```

Read a string using **cin**.

Here is a sample run:

```
Enter a string: testing
Here is your string: testing
```

Although this program is technically correct, it will not always work the way that you expect. To see why, run the program and try entering the string "This is a test". Here is what you will see:

```
Enter a string: This is a test
Here is your string: This
```

When the program redisplays your string, it shows only the word "This", not the entire sentence. The reason for this is that the C++ I/O system stops reading a string when the first whitespace character is encountered. Whitespace characters include spaces, tabs, and newlines.

One way to solve the whitespace problem is to use another of C++'s library functions, gets( ). The general form of a call to gets( ) is

gets(array-name);

To read a string, call gets( ) with the name of the array, without any index, as its argument. Upon return from gets( ), the array will hold the string input from the keyboard. The gets( ) function will continue to read characters, including whitespace, until you enter a carriage return. The header used by gets( ) is <cstdio>.

This version of the preceding program uses gets( ) to allow the entry of strings containing spaces:

```
// Using gets() to read a string from the keyboard.

#include <iostream>
#include <cstdio>
using namespace std;

int main()
{
  char str[80];

  cout << "Enter a string: ";
  gets(str); // read a string using gets()
  cout << "Here is your string: ";
  cout << str;
  return 0;
}
```

Read a string using **gets( )**.

Here is a sample run:

```
Enter a string: This is a test
Here is your string: This is a test
```

Now, spaces are read and included in the string. One other point: Notice that in a cout statement, str can be used directly. In general, the name of a character array that holds a string can be used any place that a string constant can be used.

Keep in mind that neither cin nor gets( ) performs any bounds checking on the array that receives input. Therefore, if the user enters a string longer than the size of the array, the array will be overwritten. Later, you will learn an alternative to gets( ) that avoids this problem.

1. What is a null-terminated string?
2. To hold a string that is 8 characters long, how long must the character array be?
3. What function can be used to read a string containing spaces from the keyboard?

CRITICAL SKILL 4.5: Some String Library Functions

C++ supports a wide range of string manipulation functions. The most common are

`strcpy( )`
`strcat( )`
`strcmp( )`
`strlen( )`

The string functions all use the same header, <cstring>. Let's take a look at these functions now.

strcpy

A call to strcpy( ) takes this general form:

strcpy(to, from);

The strcpy( ) function copies the contents of the string from into to. Remember, the array that forms to must be large enough to hold the string contained in from. If it isn't, the to array will be overrun, which will probably crash your program.

strcat

A call to strcat( ) takes this form: strcat(s1, s2); The strcat( ) function appends s2 to the end of s1; s2 is unchanged. You must ensure that s1 is

large enough to hold its original contents and those of s2.

strcmp

A call to strcmp( ) takes this general form:

strcmp(s1, s2);

The strcmp( ) function compares two strings and returns 0 if they are equal. If s1 is greater than s2 lexicographically (that is, according to dictionary order), then a positive number is returned; if it is less than s2, a negative number is returned.

The key to using strcmp( ) is to remember that it returns false when the strings match.

Therefore, you will need to use the ! operator if you want something to occur when the strings

are equal. For example, the condition controlling the following if statement is true when str is

equal to "C++":

```
if(!strcmp(str, "C++") cout << "str is C++";
```

strlen

The general form of a call to strlen( ) is

strlen(s);

where s is a string. The strlen( ) function returns the length of the string pointed to by s.

A String Function Example

The following program illustrates the use of all four string functions:

```
// Demonstrate the string functions.
#include <iostream>
#include <cstdio>
#include <cstring>
using namespace std;

int main()
{
      char s1[80], s2[80];
      strcpy(s1, "C++");
      strcpy(s2, " is power programming.");
      cout << "lengths: " << strlen(s1);
      cout << ' ' << strlen(s2) << '\n';
      if(!strcmp(s1, s2))
            cout << "The strings are equal\n";
      else cout << "not equal\n";
            strcat(s1, s2);
      cout << s1 << '\n';
      strcpy(s2, s1);
      cout << s1 << " and " << s2 << "\n";
      if(!strcmp(s1, s2))
            cout << "s1 and s2 are now the same.\n";
      return 0;
}
```

Here is the output:

```
lengths: 3 22
not equal
C++ is power programming.
C++ is power programming. and C++ is power programming.
s1 and s2 are now the same.
```

Using the Null Terminator

The fact that strings are null-terminated can often be used to simplify various operations. For example, the following program converts a string to uppercase:

```
// Convert a string to uppercase.
#include <iostream>
#include <cstring>
#include <cctype>
using namespace std;

int main()
{
  char str[80];
  int i;

  strcpy(str, "this is a test");

  for(i=0; str[i]; i++) str[i] = toupper(str[i]);
                                  This loop stops when the
  cout << str;                    null terminator is indexed.

  return 0;
}
```

The output from this program is shown here:

```
THIS IS A TEST
```

This program uses the library function toupper( ), which returns the uppercase equivalent of its character argument, to convert each character in the string. The toupper( ) function uses the header <cctype>.

Notice that the test condition of the for loop is simply the array indexed by the control variable. The reason this works is that a true value is any non-zero value. Remember, all character values are non-zero, but the null terminating the string is zero. Therefore, the loop runs until it encounters the null terminator, which causes str[i] to become zero. Because the null terminator marks the end of the string, the loop stops precisely where it is supposed to. You will see many examples that use the null terminator in a similar fashion in professionally written C++ code.

Ask the Expert

Q: Besides toupper( ), does C++ support other character-manipulation functions?

A: Yes. The C++ standard library contains several other character-manipulation functions. For example, the complement to toupper( ) is tolower( ), which returns the lowercase equivalent of its character argument. You can determine the case of a letter by using isupper( ), which returns true if the letter is uppercase, and islower( ), which returns true if the letter is lowercase. Other character functions include isalpha( ), isdigit( ), isspace( ), and ispunct( ). These functions each take a character argument and

determine the category of that argument. For example, isalpha( ) returns true if its argument is a letter of the alphabet.

Progress Check

1. What does the **strcat( )** function do?
2. What does **strcmp( )** return when it compares two equivalent strings?
3. Show how to obtain the length of a string called **mystr**.

CRITICAL SKILL 4.6: Array Initialization

C++ allows arrays to be initialized. The general form of array initialization is similar to that of other variables, as shown here:

type-specifier array_name[size] = {value-list};

The value-list is a comma-separated list of values that are type compatible with the base type of the array. The first value will be placed in the first position of the array, the second value in the second position, and so on. Notice that a semicolon follows the }.

In the following example, a ten-element integer array is initialized with the numbers 1 through 10.

```
int i[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

This means that i[0] will have the value 1, and i[9] will have the value 10. Character arrays that will hold strings allow a shorthand initialization that takes this form:

char array_name[size] = “string”;

For example, the following code fragment initializes str to the string “C++”:

```
char str[4] = "C++";
```

This is the same as writing

```
char str[4] = {'C', '+', '+', '\0'};
```

Because strings in C++ must end with a null, you must make sure that the array you declare is long enough to include it. This is why str is four characters long in these examples, even though “C++” is only three. When a string constant is used, the compiler automatically supplies the null terminator.

Multidimensional arrays are initialized in the same way as one-dimensional arrays. For example, the following program initializes an array called sqrs with the numbers 1 through 10 and their squares:

```
int sqrs[10][2] = {
1, 1,
2, 4,
3, 9,
4, 16,
5, 25,
6, 36,
7, 49,
8, 64,
9, 81,
10, 100 };
```

Examine Figure 4-1 to see how the sqrs array appears in memory.

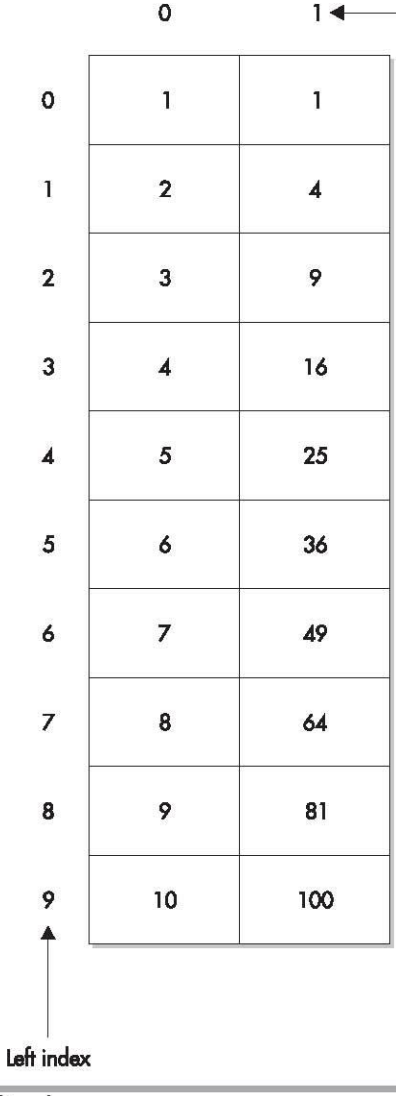


Figure 4-1 The initialized **sqrs** array

Left index

When initializing a multidimensional array, you may add braces around the initializers for each dimension. This is called subaggregate grouping. For example, here is another way to write the preceding declaration:

```
int sqrs[10][2] = {

{1, 1},

{2, 4},

{3, 9},

{4, 16},

{5, 25},

{6, 36},

{7, 49},

{8, 64},

{9, 81},

{10, 100} };
```

When using subaggregate grouping, if you don't supply enough initializers for a given group, the remaining members will automatically be set to zero.

The following program uses the sqrs array to find the square of a number entered by the user. It first looks up the number in the array and then prints the corresponding square.

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

int main() { int i, j;

int sqrs[10][2] = {

{1, 1},

{2, 4},

{3, 9},

{4, 16},

{5, 25},

{6, 36},

{7, 49},

{8, 64},
```

```
{9, 81},

{10, 100} };

cout << "Enter a number between 1 and 10: "; cin >> i;

// look up i for(j=0; j<10; j++)

if(sqrs[j][0]==i) break; cout << "The square of " << i << " is ";

cout << sqrs[j][1];

return 0; }
```

Here is a sample run:

```
Enter a number between 1 and 10: 4 The square of 4 is 16
```

Unsized Array Initializations

When declaring an initialized array, it is possible to let C++ automatically determine the array's dimension. To do this, do not specify a size for the array. Instead, the compiler determines the size by counting the number of initializers and creating an array large enough to hold them. For example,

```
int nums[] = { 1, 2, 3, 4 };
```

creates an array called nums that is four elements long that contains the values 1, 2, 3, and 4.

Because no explicit size is specified, an array such as nums is called an unsized array. Unsized arrays are quite useful. For example, imagine that you are using array initialization

to build a table of Internet addresses, as shown here:

```
char e1[16] = "www.osborne.com"; char e2[16] = "www.weather.com"; char e3[15] = "www.amazon.com";
```

As you might guess, it is very tedious to manually count the characters in each address to determine the correct array dimension. It is also error-prone because it is possible to miscount and incorrectly size the array. It is better to let the compiler size the arrays, as shown here:

```
char e1[] = "www.osborne.com"; char e2[] = "www.weather.com"; char e3[] = "www.amazon.com";
```

Besides being less tedious, the unsized array initialization method allows you to change any of the strings without fear of accidentally forgetting to resize the array.

Unsized array initializations are not restricted to one-dimensional arrays. For a multidimensional array, the leftmost dimension can be empty. (The other dimensions must be specified, however, so that the array can be properly indexed.) Using unsized array initializations, you can build tables of varying lengths, with the compiler automatically allocating enough storage for them. For example, here sqrs is declared as an unsized array:

```
int sqrs[][2] = { 1, 1, 2, 4, 3, 9, 4, 16, 5, 25, 6, 36, 7, 49, 8, 64, 9, 81, 10, 100

};
```

The advantage to this declaration over the sized version is that the table may be lengthened or shortened without changing the array dimensions.

CRITICAL SKILL 4.7: Arrays of Strings

A special form of a two-dimensional array is an array of strings. It is not uncommon in programming to use an array of strings. The input processor to a database, for instance, may verify user commands against a string array of valid commands. To create an array of strings, a two-dimensional character array is used, with the size of the left index determining the number of strings and the size of the right index specifying the maximum length of each string, including the null terminator. For example, the following declares an array of 30 strings, each having a maximum length of 79 characters plus the null terminator.

```
char str_array[30][80];
```

Accessing an individual string is quite easy: you simply specify only the left index. For example, the following statement calls gets( ) with the third string in str_array:

```
gets(str_array[2]);
```

To access an individual character within the third string, you will use a statement like this:

```
cout << str_array[2][3];
```

This displays the fourth character of the third string.

The following program demonstrates a string array by implementing a very simple computerized telephone directory. The two-dimensional array numbers holds pairs of names and numbers. To find a number, you enter the name. The number is displayed.

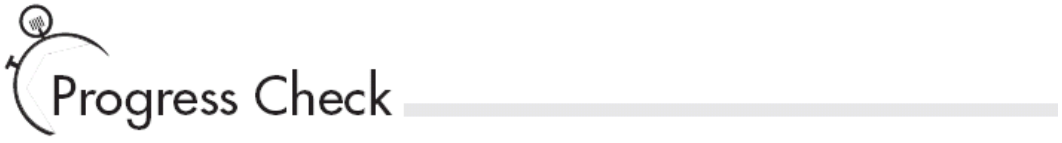
```
// A simple computerized telephone directory.
#include <iostream>
#include <cstdio>
using namespace std;

int main()
{
  int i;
  char str[80];
  char numbers[10][80] = {
    "Tom", "555-3322",
    "Mary", "555-8976",
    "Jon", "555-1037",
    "Rachel", "555-1400",
    "Sherry", "555-8873"
  };

  cout << "Enter name: ";
  cin >> str;

  for(i=0; i < 10; i += 2)
    if(!strcmp(str, numbers[i])) {
      cout << "Number is " << numbers[i+1] << "\n";
      break;
    }

  if(i == 10) cout << "Not found.\n";


  return 0;
}
```

This is an array of 10 strings, each capable of holding up to 79 characters.

Here is a sample run:

```
Enter name: Jon
Number is 555-1037
```

Notice how the for loop increments its loop control variable, i, by 2 each time through the loop. This is necessary because names and numbers alternate in the array.



1. Show how to initialize a four-element array of **int** to the values 1, 2, 3, and 4.
2. How can this initialization be rewritten?
```
char str[6] = {'H', 'e', 'l', 'l', 'o', '\0' };
```
3. Rewrite the following as an unsized array:
```
int nums[4] = {44, 55, 66, 77};
```

CRITICAL SKILL 4.8: Pointers

The pointer is one of C++'s most powerful features. It is also one of its most troublesome. Despite their potential for misuse, pointers are a crucial part of C++ programming. For example, they allow C++ to support such things as linked lists and dynamic memory allocation. They also provide one means by which a function can alter the contents of an argument. However, these and other uses of pointers will be discussed in subsequent modules. In this module, you will learn the basics about pointers and see how to manipulate them.

In a few places in the following discussions, it is necessary to refer to the size of several of C++'s basic data types. For the sake of discussion, assume that characters are one byte in length, integers are four bytes long, floats are four bytes long, and doubles have a length of eight bytes. Thus, we will be assuming a typical 32-bit environment.

What Are Pointers?

A pointer is an object that contains a memory address. Very often this address is the location of another object, such as a variable. For example, if x contains the address of y, then x is said to "point to" y.

Pointer variables must be declared as such. The general form of a pointer variable declaration is

type *var-name;

Here, type is the pointer's base type. The base type determines what type of data the pointer will be pointing to. var-name is the name of the pointer variable. For example, to declare ip to be a pointer to an int, use this declaration:

int *ip;

Since the base type of ip is int, it can be used to point to int values. Here, a float pointer is declared:

float *fp;

In this case, the base type of fp is float, which means that it can be used to point to a float value.

In general, in a declaration statement, preceding a variable name with an * causes that variable to become a pointer.

CRITICAL SKILL 4.9: The Pointer Operators

There are two special operators that are used with pointers: * and &. The & is a unary operator that returns the memory address of its operand. (Recall that a unary operator requires only one operand.) For example,

```
ptr = &total;
```

puts into ptr the memory address of the variable total. This address is the location of total in the computer's internal memory. It has nothing to do with the value of total. The operation of & can be remembered as returning "the address of" the variable it precedes. Therefore, the preceding assignment statement could be verbalized as "ptr receives the address of total." To better understand this assignment, assume that the variable total is located at address 100. Then, after the assignment takes place, ptr has the value 100.

The second operator is *, and it is the complement of &. It is a unary operator that returns the value of the variable located at the address specified by its operand. Continuing with the same example, if ptr contains the memory address of the variable total, then

```
val = *ptr;
```

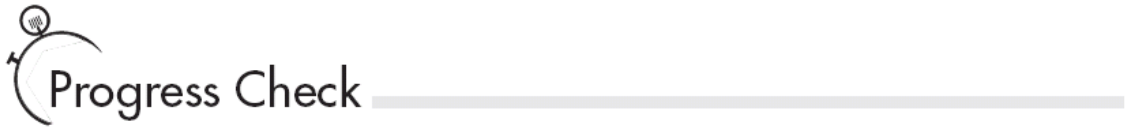
will place the value of total into val. For example, if total originally had the value 3,200, then val will have the value 3,200, because that is the value stored at location 100, the memory address that was assigned to ptr. The operation of * can be remembered as "at address." In this case, then, the statement could be read as "val receives the value at address ptr."

The following program executes the sequence of the operations just described:

```
#include <iostream> using namespace std;
int main()
{
int total;
int *ptr;
int val;
total = 3200; // assign 3,200 to total
ptr = &total; // get address of total
val = *ptr; // get value at that address
cout << "Total is: " << val << '\n';
return 0;
}
```

It is unfortunate that the multiplication symbol and the “at address” symbol are the same. This fact sometimes confuses newcomers to the C++ language. These operators have no relationship to each other. Keep in mind that both & and * have a higher precedence than any of the arithmetic operators except the unary minus, with which they have equal precedence.

The act of using a pointer is often called indirection because you are accessing one variable indirectly through another variable.



1. What is a pointer?
2. Show how to declare a **long int** pointer called **valPtr**.
3. As they relate to pointers, what do the * and **&** operators do?

The Base Type of a Pointer Is Important

In the preceding discussion, you saw that it was possible to assign val the value of total indirectly through a pointer. At this point, you may have thought of this important question: How does C++ know how many bytes to copy into val from the address pointed to by ptr? Or, more generally, how does the compiler transfer the proper number of bytes for any assignment involving a pointer? The answer is that the base type of the pointer determines the type of data upon which the pointer operates. In this case, because ptr is an int pointer, four bytes of information are copied into val (assuming a 32-bit int) from the address pointed to by ptr. However, if ptr had been a double pointer, for example, then eight bytes would have been copied.

It is important to ensure that pointer variables always point to the correct type of data. For

example, when you declare a pointer to be of type int, the compiler assumes that anything it

points to will be an integer variable. If it doesn’t point to an integer variable, then trouble is

usually not far behind! For example, the following fragment is incorrect:

```
int *p; double f; // ... p = &f; // ERROR
```

This fragment is invalid because you cannot assign a double pointer to an integer pointer. That is, &f generates a pointer to a double, but p is a pointer to an int. These two types are not compatible. (In fact, the compiler would flag an error at this point and not compile your program.)

Although two pointers must have compatible types in order for one to be assigned to

another, you can override this restriction (at your own risk) using a cast. For example, the

following fragment is now technically correct:

int *p ; double f; // ... p = (int *) &f; // Now technically OK

The cast to int * causes the double pointer to be converted to an integer pointer. However, to use a cast for this purpose is questionable practice. The reason is that the base type of a pointer determines how the compiler treats the data it points to. In this case, even though p is actually pointing to a floating-point value, the compiler still "thinks" that p is pointing to an int (because p is an int pointer).

To better understand why using a cast to assign one type of pointer to another is not usually a good idea, consider the following short program:

```
// This program will not work right.
#include <iostream>
using namespace std;

int main()
{
  double x, y;
  int *p;

  x = 123.23;
  p = (int *) &x; // use cast to assign double * to int *

  y = *p;     // What will this do?
  cout << y; // What will this print?

  return 0;
}
```

These statements won't yield the desired results.

Here is the output produced by the program. (You might see a different value.)

1.37439e+009

This value is clearly not 123.23! Here is why. In the program, p (which is an integer pointer) has been assigned the address of x (which is a double). Thus, when y is assigned the value pointed to by p, y receives only four bytes of data (and not the eight required for a double value), because p is an integer pointer. Therefore, the cout statement displays not 123.23, but a garbage value instead.

Assigning Values through a Pointer

You can use a pointer on the left-hand side of an assignment statement to assign a value to the location pointed to by the pointer. Assuming that p is an int pointer, this assigns the value 101 to the location pointed to by p.

*p = 101;

You can verbalize this assignment like this: "At the location pointed to by p, assign the value 101." To increment or decrement the value at the location pointed to by a pointer, you can use a statement like this:

(*p)++;

The parentheses are necessary because the * operator has lower precedence than does the ++ operator.

The following program demonstrates an assignment through a pointer:

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

int main()
{
  int *p, num;

  p = &num;

  *p = 100; <-- Assign num the value 100 through p.
  cout << num << ' ';
  (*p)++; <-- Increment num through p.
  cout << num << ' ';
  (*p)--; <-- Decrement num through p.
  cout << num << '\n';

  return 0;
}
```

The output from the program is shown here:

100 101 100

CRITICAL SKILL 4.10: Pointer Expressions

Pointers can be used in most C++ expressions. However, some special rules apply. Remember also that you may need to surround some parts of a pointer expression with parentheses in order to ensure that the outcome is what you desire.

Pointer Arithmetic

There are only four arithmetic operators that can be used on pointers: ++, – –, +, and –. To understand what occurs in pointer arithmetic, let p1 be an int pointer with a current value of 2,000 (that is, it contains the address 2,000). Assuming 32-bit integers, after the expression

p1++;

the contents of p1 will be 2,004, not 2,001! The reason for this is that each time p1 is incremented, it will point to the next int. The same is true of decrements. For example, again assuming that p1 has the value 2000, the expression

```
p1--;
```

causes p1 to have the value 1996.

Generalizing from the preceding example, the following rules apply to pointer arithmetic. Each time that a pointer is incremented, it will point to the memory location of the next element of its base type. Each time it is decremented, it will point to the location of the previous element of its base type. In the case of character pointers, an increment or decrement will appear as "normal" arithmetic because characters are one byte long. However, every other type of pointer will increase or decrease by the length of its base type.

You are not limited to only increment and decrement operations. You can also add or subtract integers to or from pointers. The expression

```
p1 = p1 + 9;
```

makes p1 point to the ninth element of p1's base type, beyond the one to which it is currently pointing.

Although you cannot add pointers, you can subtract one pointer from another (provided they are both of the same base type). The remainder will be the number of elements of the base type that separate the two pointers.

Other than addition and subtraction of a pointer and an integer, or the subtraction of two pointers, no other arithmetic operations can be performed on pointers. For example, you cannot add or subtract float or double values to or from pointers.

To graphically see the effects of pointer arithmetic, execute the next short program. It creates an int pointer (i) and a double pointer (f). It then adds the values 0 through 9 to these pointers and displays the results. Observe how each address changes, relative to its base type, each time the loop is repeated. (For most 32-bit compilers, i will increase by 4s and f will increase by 8s.) Notice that when using a pointer in a cout statement, its address is automatically displayed in the addressing format applicable to the CPU and environment.

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

int main()
{
  int *i, j[10];
  double *f, g[10];
  int x;

  i = j;
  f = g;

  for(x=0; x<10; x++)
    cout << i+x << ' ' << f+x << '\n';
```

Display the addresses produced by adding **x** to each pointer.

```
  return 0;
}
```

Here is a sample run. (The precise values you see may differ from these.)

```
0012FE5C 09012FE84
0012FE60 0012FE8C
0012FE64 0012FE94
0012FE68 0012FE9C
0012FE6C 0012FEA4
0012FE70 0012FEAC
0012FE74 0012FEB4
0012FE78 0012FEBC
0012FE7C 0012FEC4
0012FE80 0012FECC
```

Pointer Comparisons

Pointers may be compared using the relational operators, such as ==, <, and >. In general, for the outcome of a pointer comparison to be meaningful, the two pointers must have some relationship to each other. For example, both may point to elements within the same array. (You will see an example of this in Project 4-2.) There is, however, one other type of pointer comparison: any pointer can be compared to the null pointer, which is zero.

1. All pointer arithmetic is performed relative to the _____ _____ of the pointer.
2. Assuming that type **double** is 8 bytes long, when a **double** pointer is incremented, by how much is its value increased?
3. In general, two pointers can be meaningfully compared in what case?

CRITICAL SKILL 4.11: Pointers and Arrays

In C++, there is a close relationship between pointers and arrays. In fact, frequently a pointer and an array are interchangeable. Consider this fragment:

```
char str[80]; char *p1;

p1 = str;
```

Here, str is an array of 80 characters and p1 is a character pointer. However, it is the third line that is of interest. In this line, p1 is assigned the address of the first element in the str array. (That is, after the assignment, p1 will point to str[0].) Here's why: In C++, using the name of an array without an index generates a pointer to the first element in the array. Thus, the assignment

p1 = str;

assigns the address of str[0] to p1. This is a crucial point to understand: When an unindexed

array name is used in an expression, it yields a pointer to the first element in the array. Since, after the assignment, p1 points to the beginning of str, you can use p1 to access

elements in the array. For example, if you want to access the fifth element in str, you can use

str[4]

or

*(p1+4)

Both statements obtain the fifth element. Remember, array indices start at zero, so when str is indexed, a 4 is used to access the fifth element. A 4 is also added to the pointer p1 to get the fifth element, because p1 currently points to the first element of str.

The parentheses surrounding p1+4 are necessary because the * operation has a higher priority than the + operation. Without them, the expression would first find the value pointed to by p1 (the first location in the array) and then add 4 to it. In effect, C++ allows two methods of accessing array elements: pointer arithmetic and array indexing. This is important because pointer arithmetic can sometimes be faster than array indexing—especially when you are accessing an array in strictly sequential order. Since speed is often a consideration in programming, the use of pointers to access array elements is very common in C++ programs. Also, you can sometimes write tighter code by using pointers instead of array indexing.

Here is an example that demonstrates the difference between using array indexing and pointer arithmetic to access the elements of an array. We will create two versions of a program that reverse the case of letters within a string. The first version uses array indexing. The second uses pointer arithmetic. The first version is shown here:

// Reverse case using array indexing. #include <iostream> #include <cctype> using namespace std;

int main()

{

```
int i;
char str[80] = "This Is A Test";
cout << "Original string: " << str << "\n";
for(i = 0; str[i]; i++) {
        if(isupper(str[i]))
                str[i] = tolower(str[i]);
        else if(islower(str[i]))
                str[i] = toupper(str[i]);
}
cout << "Inverted-case string: " << str;
return 0;
```

}

The output from the program is shown here:

Original string: This Is A Test Inverted-case string: tHIS iS a tEST

Notice that the program uses the isupper( ) and islower( ) library functions to determine the case of a letter. The isupper( ) function returns true when its argument is an uppercase letter; islower( ) returns true when its argument is a lowercase letter. Inside the for loop, str is indexed, and the case of each letter is checked and changed. The loop iterates until the null terminating str is indexed. Since a null is zero (false), the loop stops.

Here is the same program rewritten to use pointer arithmetic:

```
// Reverse case using array indexing.
#include <iostream>
#include <cctype>
using namespace std;

int main()
{
  int i;
  char str[80] = "This Is A Test";

  cout << "Original string: " << str << "\n";

  for(i = 0; str[i]; i++) {
    if(isupper(str[i]))
      str[i] = tolower(str[i]);
    else if(islower(str[i]))
      str[i] = toupper(str[i]);
  }

  cout << "Inverted-case string: " << str;

  return 0;
}
```

In this version, p is set to the start of str. Then, inside the while loop, the letter at p is checked and changed, and then p is incremented. The loop stops when p points to the null terminator that ends str. Because of the way some C++ compilers generate code, these two programs may not be equivalent in performance. Generally, it takes more machine instructions to index an array than it does to perform arithmetic on a pointer. Consequently, in professionally written C++ code, it is common to see the pointer version used more frequently. However, as a beginning C++ programmer, feel free to use array indexing until you are comfortable with pointers.

Indexing a Pointer

As you have just seen, it is possible to access an array using pointer arithmetic. What you might find surprising is that the reverse is also true. In C++, it is possible to index a pointer as if it were an array. Here is an example. It is a third version of the case-changing program.

```
// Index a pointer as if it were an array.
#include <iostream>
#include <cctype>

using namespace std;

int main()
{
  char *p;
  int i;
  char str[80] = "This Is A Test";

  cout << "Original string: " << str << "\n";

  p = str; // assign p the address of the start of the array

  // now, index p
  for(i = 0; p[i]; i++) {
    if(isupper(p[i]))
      p[i] = tolower(p[i]);
    else if(islower(p[i]))
      p[i] = toupper(p[i]);
  }

  cout << "Inverted-case string: " << str;

  return 0;
}
```

Access **p** as if it were an array.

The program creates a char pointer called p and then assigns to that pointer the address of the first element in str. Inside the for loop, p is indexed using the normal array indexing syntax. This is perfectly valid because in C++, the statement p[i] is functionally identical to *(p+i). This further illustrates the close relationship between pointers and arrays.

Progress Check

1. Can an array be accessed through a pointer?
2. Can a pointer be indexed as if it were an array?
3. An array name used by itself, with no index, yields what?

1. Yes, an array can be accessed through a pointer.
2. Yes, a pointer can be indexed as if it were an array.
3. An array name used by itself, with no index, yields a pointer to the first element of the array.

Ask the Expert

Q: Are pointers and arrays interchangeable?

A: As the preceding few pages have shown, pointers and arrays are strongly related and are interchangeable in many cases. For example, a pointer that points to the beginning of an array can access that array using either pointer arithmetic or array-style indexing. However, pointers and arrays are not completely interchangeable. For example, consider this fragment:

```
int nums[10]; int i;

for(i=0; i<10; i++) { *nums = i; // this is OK nums++; // ERROR -- cannot modify nums

}
```

Here, nums is an array of integers. As the comments describe, while it is perfectly acceptable to apply the * operator to nums (which is a pointer operation), it is illegal to modify nums' value. The reason for this is that nums is a constant that points to the beginning of an array. Thus, you cannot increment it. More generally, while an array name without an index does generate a pointer to the beginning of an array, it cannot be changed.

Although an array name generates a pointer constant, it can still take part in pointer-style expressions, as long as it is not modified. For example, the following is a valid statement that assigns nums[3] the value 100:

```
*(nums+3) = 100; // This is OK because nums is not changed
```

String Constants

You might be wondering how string constants, like the one in the fragment shown here, are handled by C++:

```
cout << strlen("Xanadu");
```

The answer is that when the compiler encounters a string constant, it stores it in the program's string table and generates a pointer to the string. Thus, "Xanadu" yields a pointer to its entry in the string table. Therefore, the following program is perfectly valid and prints the phrase **Pointers add power to C++.**:

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

int main()
{
  char *ptr;

  ptr = "Pointers add power to C++.\n"; <-- ptr is assigned the address of this string constant.

  cout << ptr;

  return 0;
}
```

In this program, the characters that make up a string constant are stored in the string table, and ptr is assigned a pointer to the string in that table.

Since a pointer into your program's string table is generated automatically whenever a string constant is used, you might be tempted to use this fact to modify the contents of the string table. However, this is usually not a good idea because many C++ compilers create optimized tables in which one string constant may be used at two or more different places in your program. Thus, changing a string may cause undesired side effects.

Project 4-2 Reversing a String in Place

Earlier it was mentioned that comparing one pointer to another is meaningful only if the two pointers point to a common object, such as an array. Now that you understand how pointers and arrays relate, you can apply pointer comparisons to streamline some types of algorithms. In this project, you will see an example. The program developed here reverses the contents of a string, in place. Thus, instead of copying the string back-to-front into another array, it reverses the contents of the string inside the array that holds it. The program uses two pointer variables to accomplish this. One initially points to the beginning of a string, and the other initially points to the last character in the string. A loop is set up that continues to run as long as the start pointer is less than the end pointer. Each time through the loop, the characters pointed to by the pointers are swapped and the pointers are advanced. When the start pointer is greater than or equal to the end pointer, the string has been reversed.

Step by Step

1. Create a file called StrRev.cpp.
2. Begin by adding these lines to the file:

```
/*
   Project 4-2
   Reverse a string in place.
*/
#include <iostream>
#include <cstring>
using namespace std;

int main()
{
  char str[] = "this is a test";
  char *start, *end;
  int len;
  char t;
```

The string to be reversed is contained in str. The pointers start and end will be used to access the string.

3. Add these lines, which display the original string, obtain the string's length, and set the initial values for the start and end pointers:

```
cout << "Original: " << str << "\n";

len = strlen(str);

start = str; end = &str[len-1];
```

Notice that end points to the last character in the string, not the null terminator.

4. Add the code that reverses the string:

```
while(start < end) {
  // swap chars
  t = *start;
  *start = *end;
  *end = t;

  // advance pointers
  start++;
  end--;
}
```

The process works like this. As long as the start pointer points to a memory location that is less than the end pointer, the loop iterates. Inside the loop, the characters being pointed to by start and end are swapped. Then start is incremented and end is decremented. When end is greater than or equal to start, all of the characters in the string have been reversed. Since both start and end point into the same array, their comparison is meaningful.

5. Here is the complete StrRev.cpp program:

```
/*
   Project 4-2
   Reverse a string in place.
*/
#include <iostream>
#include <cstring>
using namespace std;

int main()
{
  char str[] = "this is a test";
  char *start, *end;
  int len;
  char t;

  cout << "Original: " << str << "\n";

  len = strlen(str);

  start = str;
  end = &str[len-1];

  while(start < end) {
    // swap chars
    t = *start;
    *start = *end;
    *end = t;

    // advance pointers
    start++;
    end--;
  }


  cout << "Reversed: " << str << "\n";
  return 0;
}
```

The output from the program is shown here:

```
Original: this is a test
Reversed: tset a si siht
```

Arrays of Pointers

Pointers can be arrayed like any other data type. For example, the declaration for an int pointer array of size 10 is

int *pi[10];

Here, pi is an array of ten integer pointers. To assign the address of an int variable called var to the third element of the pointer array, you would write

```
int var;

pi[2] = &var;
```

Remember, pi is an array of int pointers. The only thing that the array elements can hold are the addresses of integer values—not the values themselves. To find the value of var, you would write

```
*pi[2]
```

Like other arrays, arrays of pointers can be initialized. A common use for initialized pointer arrays is to hold pointers to strings. Here is an example that uses a two-dimensional array of character pointers to implement a small dictionary:

```
// Use a 2-D array of pointers to create a dictionary.

#include <iostream>
#include <cstring>
using namespace std;

int main() {

  char *dictionary[][2] = {
    "pencil", "A writing instrument.",
    "keyboard", "An input device.",
    "rifle", "A shoulder-fired firearm.",
    "airplane", "A fixed-wing aircraft.",
    "network", "An interconnected group of computers.",
    "", ""
  };
  char word[80];
  int i;
```

This is a two-dimensional array of **char** pointers, which is used to point to pairs of strings.

```
  cout << "Enter word: ";
  cin >> word;

  for(i = 0; *dictionary[i][0]; i++) {
    if(!strcmp(dictionary[i][0], word)) {
      cout << dictionary[i][1] << "\n";
      break;
    }
  }

  if(!*dictionary[i][0])
    cout << word << " not found.\n";

  return 0;
}
```

To find a definition, **word** is searched for in **dictionary**. If a match is found, the definition is displayed.

Here is a sample run:

Enter word: network

An interconnected group of computers.

When the array dictionary is created, it is initialized with a set of words and their meanings. Recall, C++ stores all string constants in the string table associated with your program, so the array need only store pointers to the strings. The program works by testing the word entered by the user against the strings stored in the dictionary. If a match is found, the meaning is displayed. If no match is found, an error message is printed.

Notice that dictionary ends with two null strings. These mark the end of the array. Recall that a null string contains only the terminating null character. The for loop runs until the first character in a string is null. This condition is tested with this expression:

*dictionary[i][0]

The array indices specify a pointer to a string. The * obtains the character at that location. If this character is null, then the expression is false and the loop terminates. Otherwise, the expression is true and the loop continues.

The Null Pointer Convention

After a pointer is declared, but before it has been assigned, it will contain an arbitrary value. Should you try to use the pointer prior to giving it a value, you will probably crash your program. While there is no sure way to avoid using an uninitialized pointer, C++ programmers have adopted a procedure that helps prevent some errors. By convention, if a pointer contains the null (zero) value, it is assumed to point to nothing. Thus, if all unused pointers are given the null value and you avoid the use of a null pointer, you can avoid the accidental misuse of an uninitialized pointer. This is a good practice to follow.

Any type of pointer can be initialized to null when it is declared. For example, the following initializes p to null:

```
float *p = 0; // p is now a null pointer
```

To check for a null pointer, use an if statement, like one of these:

```
if(p) // succeeds if p is not null

if(!p) // succeeds if p is null
```

1. String constants used in a program are stored in a _____ _____.
2. What does this declaration create? `float *fpa[18];`
3. By convention, a pointer containing null is assumed to be unused. True or false?

CRITICAL SKILL 4.12: Multiple Indirection

A pointer to a pointer is a form of multiple indirection, or a chain of pointers. Consider Figure 4-2. As you can see, in the case of a normal pointer, the value of the pointer is the address of a value. In the case of a pointer to a pointer, the first pointer contains the address of the second pointer, which points to the location that contains the desired value.

Multiple indirection can be carried on to whatever extent desired, but there are few cases where more than a pointer to a pointer is needed, or, indeed, even wise to use. Excessive indirection is difficult to follow and prone to conceptual errors.

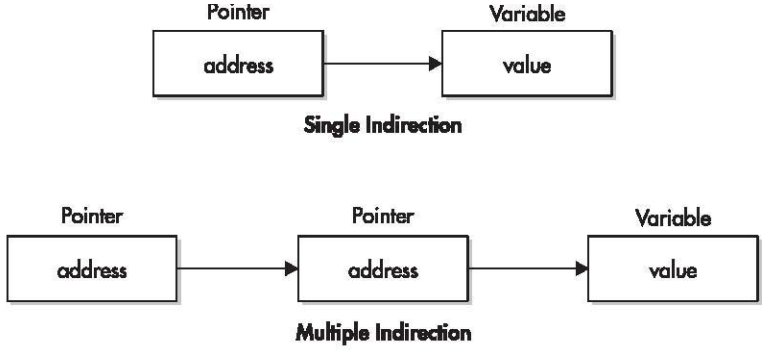


Figure 4-2 Single and multiple indirection

A variable that is a pointer to a pointer must be declared as such. This is done by placing an additional asterisk in front of its name. For example, this declaration tells the compiler that balance is a pointer to a pointer of type int:

int **balance;

It is important to understand that balance is not a pointer to an integer, but rather a pointer to an int pointer. When a target value is indirectly pointed to by a pointer to a pointer, accessing that value requires that the asterisk operator be applied twice, as is shown in this short example:

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

int main()
{
  int x, *p, **q;

  x = 10;

  p = &x;   <-- Assign p the address of x.

  q = &p;   <-- Assign q the address of p.

  cout << **q; // prints the value of x <-- Access x's value through q.
                                            Notice that two * are required.
  return 0;
}
```

Ask the Expert

Q: Given the power of pointers, I can see that their misuse could easily cause extensive damage to a program. Do you have any tips on avoiding pointer errors?

A: First, make sure that pointer variables are initialized before using them. That is, make sure that a pointer actually points to something before you attempt to use it! Second, make sure that the type of the object to which a pointer points is the same as the base type of pointer. Third, don't perform operations through null pointers. Recall that a null pointer indicates that the pointer points nowhere. Finally, don't cast pointers "just to make your code compile." Usually, pointer mismatch errors indicate that you are thinking about something incorrectly. Casting one type of pointer into another is usually needed only in unusual circumstances.

Here, p is declared as a pointer to an integer, and q as a pointer to a pointer to an integer. The cout statement will print the number 10 on the screen.

Module 4 Mastery Check

1. Show how to declare a short int array called hightemps that is 31 elements long.

2. In C++, all arrays begin indexing at _____.

3. Write a program that searches an array of ten integers for duplicate values. Have the program display each duplicate found.

4. What is a null-terminated string?

5. Write a program that prompts the user for two strings and then compares the strings for equality, but ignores case differences. Thus, “ok” and “OK” will compare as equal.

6. When using strcat( ), how large must the recipient array be?

7. In a multidimensional array, how is each index specified?

8. Show how to initialize an int array called nums with the values 5, 66, and 88.

9. What is the principal advantage of an unsized array declaration?

10. What is a pointer? What are the two pointer operators?

11. Can a pointer be indexed like an array? Can an array be accessed through a pointer?

12. Write a program that counts the uppercase letters in a string. Have it display the result.

13. What is it called when one pointer points to another pointer?

14. Of what significance is a null pointer in C++?