10 Arrays and Pointers

You will learn about the following in this chapter:

- Keyword:

  `static`

- Operators:

  `&  *` (unary)

- How to create and initialize arrays
- Pointers (building on the basics you already know) and see how they relate to arrays
- Writing functions that process arrays
- Two-dimensional arrays

People turn to computers for tasks such as tracking monthly expenses, daily rainfall, quarterly sales, and weekly weights. Enterprises turn to computers to manage payrolls, inventory, and customer transactions. As a programmer, you inevitably have to deal with large quantities of related data. Often, arrays offer the best way to handle such data in an efficient, convenient manner. Chapter 6, “C Control Statements: Looping,” introduced arrays, and this chapter takes a more thorough look. In particular, it examines how to write array-processing functions. Such functions enable you to extend the advantages of modular programming to arrays. In doing so, you can see the intimate relationship between arrays and pointers.

Arrays

Recall that an *array* is composed of a series of elements of one data type. You use *declarations* to tell the compiler when you want an array. An *array declaration* tells the compiler how many elements the array contains and what the type is for these elements. Armed with this information, the compiler can set up the array properly. Array elements can have the same types as ordinary variables. Consider the following example of array declarations:

```
/* some array declarations */
int main(void)
```

```
{
   float candy[365];      /* array of 365 floats */
   char code[12];         /* array of 12 chars   */
   int states[50];        /* array of 50 ints    */
   ...
}
```

The brackets (`[]`) identify `candy` and the rest as arrays, and the number enclosed in the brackets indicates the number of elements in the array.

To access elements in an array, you identify an individual element by using its subscript number, also called its *index*. The numbering starts with 0. Hence, `candy[0]` is the first element of the `candy` array, and `candy[364]` is the 365th and last element.

This is rather old hat; let's learn something new.

Initialization

Arrays are often used to store data needed for a program. For example, a 12-element array can store the number of days in each month. In cases such as these, it's convenient to initialize the array at the beginning of a program. Let's see how it is done.

You know you can initialize single-valued variables (sometimes called *scalar* variables) in a declaration with expressions such as

```
int fix = 1;
float flax = PI * 2;
```

where, one hopes, `PI` was defined earlier as a macro. C extends initialization to arrays with a new syntax, as shown next:

```
int main(void)
{
    int powers[8] = {1,2,4,6,8,16,32,64}; /* ANSI C and later */
    ...
}
```

As you can see, you initialize an array by using a comma-separated list of values enclosed in braces. You can use spaces between the values and the commas, if you want. The first element (`powers[0]`) is assigned the value `1`, and so on. (If your compiler rejects this form of initialization as a syntax error, you may be suffering from a pre-ANSI compiler. Prefixing the array declaration with the keyword `static` should solve the problem. Chapter 12, "Storage Classes, Linkage, and Memory Management," discusses the meaning of this keyword.)

Listing 10.1 presents a short program that prints the number of days per month.

Listing 10.1 **The `day_mon1.c` Program**

```
/* day_mon1.c -- prints the days for each month */
#include <stdio.h>
#define MONTHS 12

int main(void)
{
    int days[MONTHS] = {31,28,31,30,31,30,31,31,30,31,30,31};
    int index;

    for (index = 0; index < MONTHS; index++)
        printf("Month %d has %2d days.\n", index +1,
               days[index]);

    return 0;
}
```

The output looks like this:

```
Month  1 has 31 days.
Month  2 has 28 days.
Month  3 has 31 days.
Month  4 has 30 days.
Month  5 has 31 days.
Month  6 has 30 days.
Month  7 has 31 days.
Month  8 has 31 days.
Month  9 has 30 days.
Month 10 has 31 days.
Month 11 has 30 days.
Month 12 has 31 days.
```

Not quite a superb program, but it's wrong only one month in every four years. The program initializes `days[]` with a list of comma-separated values enclosed in braces.

Note that this example used the symbolic constant `MONTHS` to represent the array size. This is a common and recommended practice. For example, if the world switched to a 13-month calendar, you just have to modify the `#define` statement and don't have to track down every place in the program that uses the array size.

Note Using `const` with Arrays

Sometimes you might use an array that's intended to be a read-only array. That is, the program will retrieve values from the array, but it won't try to write new values into the array. In such cases, you can, and should, use the `const` keyword when you declare and initialize the array. Therefore, a better choice for Listing 10.1 would be

```
const int days[MONTHS] = {31,28,31,30,31,30,31,31,30,31,30,31};
```

This makes the program treat each element in the array as a constant. Just as with regular variables, you should use the declaration to initialize `const` data because once it's declared `const`, you can't assign values later. Now that you know about this, we can use `const` in subsequent examples.

What if you fail to initialize an array? Listing 10.2 shows what happens.

Listing 10.2 **The `no_data.c` Program**

```
/* no_data.c -- uninitialized array */
#include <stdio.h>
#define SIZE 4
int main(void)
{
    int no_data[SIZE];  /* uninitialized array */
    int i;

    printf("%2s%14s\n",
           "i", "no_data[i]");
    for (i = 0; i < SIZE; i++)
        printf("%2d%14d\n", i, no_data[i]);

    return 0;
}
```

Here is some sample output (your results may vary):

```
i    no_data[i]
0             0
1       4204937
2       4219854
3    2147348480
```

The array members are like ordinary variables—if you don't initialize them, they might have any value. The compiler is allowed to just use whatever values were already present at those memory locations, which is why your results may vary from these.

Note Storage Class Caveat

Arrays, like other variables, can be created using different *storage classes*. Chapter 12 investigates this topic, but for now, you should be aware that the current chapter describes arrays that belong to the automatic storage class. That means they are declared inside of a function and without using the keyword `static`. All the variables and arrays used in this book, so far, are of the automatic kind.

The reason for mentioning storage classes at this point is that occasionally the different storage classes have different properties, so you can't generalize everything in this chapter to other storage classes. In particular, variables and arrays of some of the other storage classes have their contents set to `0` if they are not initialized.

The number of items in the list should match the size of the array. But what if you count wrong? Let's try the last example again, as shown in Listing 10.3, with a list that is two too short.

Listing 10.3 **The `somedata.c` Program**

```
/* some_data.c -- partially initialized array */
#include <stdio.h>
#define SIZE 4
int main(void)
{
    int some_data[SIZE] = {1492, 1066};
    int i;

    printf("%2s%14s\n",
           "i", "some_data[i]");
    for (i = 0; i < SIZE; i++)
        printf("%2d%14d\n", i, some_data[i]);

    return 0;
}
```

This time the output looks like this:

```
i  some_data[i]
0          1492
1          1066
2             0
3             0
```

As you can see, the compiler had no problem. When it ran out of values from the list, it initialized the remaining elements to `0`. That is, if you don't initialize an array at all, its elements, like uninitialized ordinary variables, get garbage values, but if you partially initialize an array, the remaining elements are set to `0`.

The compiler is not so forgiving if you have too many list values. This overgenerosity is considered an error. However, there is no need to subject yourself to the ridicule of your compiler. Instead, you can let the compiler match the array size to the list by omitting the size from the braces (see Listing 10.4).

Listing 10.4 **The `day_mon2.c` Program**

```
/* day_mon2.c -- letting the compiler count elements */
#include <stdio.h>
int main(void)
{
    const int days[] = {31,28,31,30,31,30,31,31,30,31};
    int index;
    for (index = 0; index < sizeof days / sizeof days[0]; index++)
        printf("Month %2d has %d days.\n", index +1,
               days[index]);

    return 0;
}
```

There are two main points to note in Listing 10.4:

- When you use empty brackets to initialize an array, the compiler counts the number of items in the list and makes the array that large.
- Notice what we did in the `for` loop control statement. Lacking faith (justifiably) in our ability to count correctly, we let the computer give us the size of the array. The `sizeof` operator gives the size, in bytes, of the object, or *type*, following it. So `sizeof days` is the size, in bytes, of the whole array, and `sizeof days[0]` is the size, in bytes, of one element. Dividing the size of the entire array by the size of one element tells us how many elements are in the array.

Here is the result of running this program:

```
Month  1 has 31 days.
Month  2 has 28 days.
Month  3 has 31 days.
Month  4 has 30 days.
Month  5 has 31 days.
Month  6 has 30 days.
Month  7 has 31 days.
Month  8 has 31 days.
Month  9 has 30 days.
Month 10 has 31 days.
```

Oops! We put in just 10 values, but our method of letting the program find the array size kept us from trying to print past the end of the array. This points out a potential disadvantage of automatic counting: Errors in the number of elements could pass unnoticed.

There is one more short method of initializing arrays. Because it works only for character strings, however, we will save it for the next chapter.

Designated Initializers (C99)

C99 added a new capability: *designated initializers*. This feature allows you to pick and choose which elements are initialized. Suppose, for example, that you just want to initialize the last element in an array. With traditional C initialization syntax, you also have to initialize every element preceding the last one:

```
int arr[6] = {0,0,0,0,0,212};  // traditional syntax
```

With C99, you can use an index in brackets in the initialization list to specify a particular element:

```
int arr[6] = {[5] = 212}; // initialize arr[5] to 212
```

As with regular initialization, after you initialize at least one element, the uninitialized elements are set to 0. Listing 10.5 shows a more involved example.

Listing 10.5 **The `designate.c` Program**

```
// designate.c -- use designated initializers
#include <stdio.h>
#define MONTHS 12
int main(void)
{
    int days[MONTHS] = {31,28, [4] = 31,30,31, [1] = 29};
    int i;

    for (i = 0; i < MONTHS; i++)
        printf("%2d  %d\n", i + 1, days[i]);

    return 0;
}
```

Here's the output if the compiler supports this C99 feature:

```
 1  31
 2  29
 3  0
 4  0
 5  31
 6  30
 7  31
 8  0
 9  0
10  0
11  0
12  0
```

The output reveals a couple important features of designated initializers. First, if the code follows a designated initializer with further values, as in the sequence `[4] = 31,30,31`, these further values are used to initialize the subsequent elements. That is, after initializing `days[4]` to `31`, the code initializes `days[5]` and `days[6]` to `30` and `31`, respectively. Second, if the code initializes a particular element to a value more than once, the last initialization is the one that takes effect. For example, in Listing 10.5, the start of the initialization list initializes `days[1]` to `28`, but that is overridden by the `[1] = 29` designated initialization later.

Suppose you don’t specify the array size?

```
int stuff[] = {1, [6] = 23};       // what happens?
int staff[] = {1, [6] = 4, 9, 10}; // what happens?
```

The compiler will make the array big enough to accommodate the initialization values. So `stuff` will have seven elements, numbered 0-6, and `staff` will have two more elements, or 9.

Assigning Array Values

After an array has been declared, you can *assign* values to array members by using an array index, or *subscript*. For example, the following fragment assigns even numbers to an array:

```
/* array assignment */
#include <stdio.h>
#define SIZE 50
int main(void)
{
    int counter, evens[SIZE];

    for (counter = 0; counter < SIZE; counter++)
        evens[counter] = 2 * counter;
  ...
}
```

Note that the code uses a loop to assign values element by element. C doesn’t let you assign one array to another as a unit. Nor can you use the list-in-braces form except when initializing. The following code fragment shows some forms of assignment that are not allowed:

```
/* nonvalid array assignment */
#define SIZE 5
int main(void)
{
    int oxen[SIZE] = {5,3,2,8};      /* ok here     */
    int yaks[SIZE];

    yaks = oxen;                     /*  not allowed  */
    yaks[SIZE] = oxen[SIZE];         /* out of range  */
    yaks[SIZE] = {5,3,2,8};          /* doesn't work  */
```

Recall that the last element of `oxen` is `oxen[SIZE-1]`, so `oxen[SIZE]` and `yaks[SIZE]` refer to data past the ends of the two arrays.

Array Bounds

You have to make sure you use array indices that are within bounds; that is, you have to make sure they have values valid for the array. For instance, suppose you make the following declaration:

```
int doofi[20];
```

Then it's your responsibility to make sure the program uses indices only in the range 0 through 19, because the compiler isn't required to check for you. (However, some compilers will warn you of the problem, but continue on to compile the program anyway.)

Consider the program in Listing 10.6. It creates an array with four elements and then carelessly uses index values ranging from –1 to 6.

Listing 10.6 **The `bounds.c` Program**

```
// bounds.c -- exceed the bounds of an array
#include <stdio.h>
#define SIZE 4
int main(void)
{
    int value1 = 44;
    int arr[SIZE];
    int value2 = 88;
    int i;

    printf("value1 = %d, value2 = %d\n", value1, value2);
    for (i = -1; i <= SIZE; i++)
        arr[i] = 2 * i + 1;

    for (i = -1; i < 7; i++)
        printf("%2d  %d\n", i , arr[i]);
    printf("value1 = %d, value2 = %d\n", value1, value2);
    printf("address of arr[-1]: %p\n", &arr[-1]);
    printf("address of arr[4]:  %p\n", &arr[4]);
    printf("address of value1:  %p\n", &value1);
    printf("address of value2:  %p\n", &value2);

    return 0;
}
```

The compiler doesn't check to see whether the indices are valid. The result of using a bad index is, in the language of the C standard, undefined. That means when you run the program, it might seem to work, it might work oddly, or it might abort. Here is sample output using GCC:

```
value1 = 44, value2 = 88
-1  -1
 0  1
 1  3
 2  5
 3  7
 4  9
 5  1624678494
 6  32767
value1 = 9, value2 = -1
```

```
address of arr[-1]: 0x7fff5fbff8cc
address of arr[4]:  0x7fff5fbff8e0
address of value1:  0x7fff5fbff8e0
address of value2:  0x7fff5fbff8cc
```

Note that this compiler appears to have stored `value1` just after the array and `value2` just ahead of it. (Other compilers might store the data in a different order in memory.) In this case, as shown in the output, `arr[-1]` corresponded to the same memory location as `value2`, and `arr[4]` corresponded to the same memory location as `value1`. Therefore, using out-of-bounds array indices resulted in the program altering the value of other variables. Another compiler might produce different results, including a program that aborts.

You might wonder why C allows nasty things like that to happen. It goes back to the C philosophy of trusting the programmer. Not checking bounds allows a C program to run faster. The compiler can't necessarily catch all index errors because the value of an index might not be determined until after the resulting program begins execution. Therefore, to be safe, the compiler would have to add extra code to check the value of each index during runtime, and that would slow things down. So C trusts the programmer to do the coding correctly and rewards the programmer with a faster program. Of course, not all programmers deserve that trust, and then problems can arise.

One simple thing to remember is that array numbering begins with 0. One simple habit to develop is to use a symbolic constant in the array declaration and in other places the array size is used:

```
#define SIZE 4
int main(void)
{
    int arr[SIZE];
    for (i = 0; i < SIZE; i++)
    ....
```

This helps ensure that you use the same array size consistently throughout the program.

Specifying an Array Size

So far, the examples have used integer constants when declaring arrays:

```
#define SIZE 4
int main(void)
{
    int arr[SIZE];     // symbolic integer constant
    double lots[144];  // literal integer constant
    ...
```

What else is allowed? Until the C99 standard, the answer has been that you have to use a *constant integer expression* between the brackets when declaring an array. A constant integer expression is one formed from integer constants. For this purpose, a `sizeof` expression is considered an integer constant, but (unlike the case in C++) a `const` value isn't. Also, the value of the expression must be greater than 0:

```
int n = 5;
int m = 8;
float a1[5];                // yes
float a2[5*2 + 1];          // yes
float a3[sizeof(int) + 1];  // yes
float a4[-4];               // no, size must be > 0
float a5[0];                // no, size must be > 0
float a6[2.5];              // no, size must be an integer
float a7[(int)2.5];         // yes, typecast float to int constant
float a8[n];                // not allowed before C99
float a9[m];                // not allowed before C99
```

As the comments indicate, C compilers following the C90 standard would not allow the last two declarations. As of C99, however, C does allow them, but they create a new breed of array, something called a *variable-length array*, or *VLA* for short. (C11 retreats from this bold initiative, making VLAs an optional rather than mandatory language feature.)

C99 introduced variable-length arrays primarily to allow C to become a better language for numerical computing. For instance, VLAs make it easier to convert existing libraries of FORTRAN numerical calculation routines to C. VLAs have some restrictions; for example, you can't initialize a VLA in its declaration. This chapter will return to VLAs later, after you've learned enough to understand more about the limitations of the classic C array.

Multidimensional Arrays

Tempest Cloud, a weather person who takes her subject "cirrusly," wants to analyze five years of monthly rainfall data. One of her first decisions is how to represent the data. One choice is to use 60 variables, one for each data item. (We mentioned this choice once before, and it is as senseless now as it was then.) Using an array with 60 elements would be an improvement, but it would be even nicer still if she could keep each year's data separate. She could use five arrays, each with 12 elements, but that is clumsy and could get really awkward if Tempest decides to study 50 years' worth of rainfall instead of five. She needs something better.

The better approach is to use an array of arrays. The master array would have five elements, one for each year. Each of those elements, in turn, would be a 12-element array, one for each month. Here is how to declare such an array:

```
float rain[5][12];  // array of 5 arrays of 12 floats
```

One way to view this declaration is to first look at the inner portion (the part in bold):

```
float rain[5][12];             // rain is an array of 5 somethings
```

It tells us that `rain` is an array with five elements. But what is each of those elements? Now look at the remaining part of the declaration (now in bold):

```
float rain[5] [12];  // an array of 12 floats
```

This tells us that each element is of type `float[12]`; that is, each of the five elements of `rain` is, in itself, an array of 12 `float` values.

Pursuing this logic, `rain[0]`, being the first element of `rain`, is an array of 12 `float` values. So are `rain[1]`, `rain[2]`, and so on. If `rain[0]` is an array, its first element is `rain[0][0]`, its second element is `rain[0][1]`, and so on. In short, `rain` is a five-element array of 12-element arrays of `float`, `rain[0]` is an array of 12 `float`s, and `rain[0][0]` is a `float`. To access, say, the value in row 2, column 3, use `rain[2][3]`. (Remember, array counting starts at 0, so row 2 is the third row.)

You can also visualize this `rain` array as a two-dimensional array consisting of five rows, each of 12 columns, as shown in Figure 10.1. By changing the second subscript, you move along a row, month by month. By changing the first subscript, you move vertically along a column, year by year.

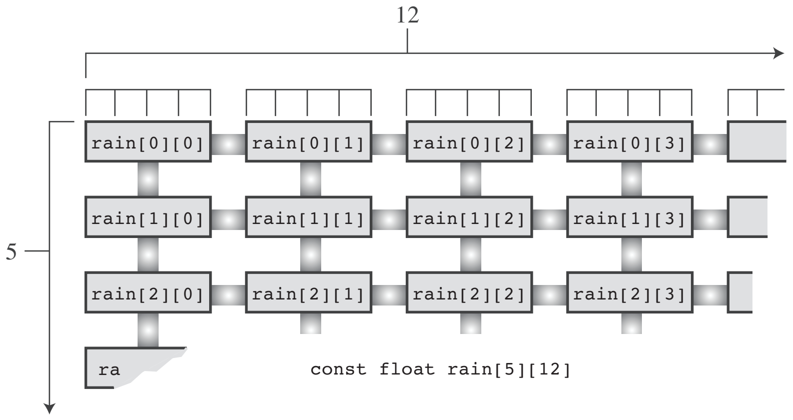

Figure 10.1 Two-dimensional array.

The two-dimensional view is merely a convenient way of visualizing an array with two indices. Internally, such an array is stored sequentially, beginning with the first 12-element array, followed by the second 12-element array, and so on.

Let's use this two-dimensional array in a weather program. The program goal is to find the total rainfall for each year, the average yearly rainfall, and the average rainfall for each month. To find the total rainfall for a year, you have to add all the data in a given row. To find the average rainfall for a given month, you have to add all the data in a given column. The two-dimensional array makes it easy to visualize and execute these activities. Listing 10.7 shows the program.

Listing 10.7 **The `rain.c` Program**

```
/* rain.c  -- finds yearly totals, yearly average, and monthly
                 average for several years of rainfall data */
#include <stdio.h>
#define MONTHS 12    // number of months in a year
#define YEARS   5    // number of years of data
int main(void)
{
 // initializing rainfall data for 2010 - 2014
    const float rain[YEARS][MONTHS] =
    {
        {4.3,4.3,4.3,3.0,2.0,1.2,0.2,0.2,0.4,2.4,3.5,6.6},
        {8.5,8.2,1.2,1.6,2.4,0.0,5.2,0.9,0.3,0.9,1.4,7.3},
        {9.1,8.5,6.7,4.3,2.1,0.8,0.2,0.2,1.1,2.3,6.1,8.4},
        {7.2,9.9,8.4,3.3,1.2,0.8,0.4,0.0,0.6,1.7,4.3,6.2},
        {7.6,5.6,3.8,2.8,3.8,0.2,0.0,0.0,0.0,1.3,2.6,5.2}
    };
    int year, month;
    float subtot, total;

    printf(" YEAR    RAINFALL  (inches)\n");
    for (year = 0, total = 0; year < YEARS; year++)
    {             // for each year, sum rainfall for each month
        for (month = 0, subtot = 0; month < MONTHS; month++)
            subtot += rain[year][month];
        printf("%5d %15.1f\n", 2010 + year, subtot);
        total += subtot; // total for all years
     }
    printf("\nThe yearly average is %.1f inches.\n\n",
            total/YEARS);
    printf("MONTHLY AVERAGES:\n\n");
    printf(" Jan  Feb  Mar  Apr  May  Jun  Jul  Aug  Sep  Oct ");
    printf(" Nov  Dec\n");

    for (month = 0; month < MONTHS; month++)
    {             // for each month, sum rainfall over years
        for (year = 0, subtot =0; year < YEARS; year++)
            subtot += rain[year][month];
        printf("%4.1f ", subtot/YEARS);
    }
    printf("\n");

    return 0;
}
```

Here is the output:

```
  YEAR    RAINFALL  (inches)
 2010           32.4
 2011           37.9
 2012           49.8
 2013           44.0
 2014           32.9

The yearly average is 39.4 inches.

MONTHLY AVERAGES:

 Jan  Feb  Mar  Apr  May  Jun  Jul  Aug  Sep  Oct  Nov  Dec
 7.3  7.3  4.9  3.0  2.3  0.6  1.2  0.3  0.5  1.7  3.6  6.7
```

As you study this program, concentrate on the initialization and on the computation scheme. The initialization is the more involved of the two, so let's look at the simpler part (the computation) first.

To find the total for a given year, keep `year` constant and let `month` go over its full range. This is the inner `for` loop of the first part of the program. Then repeat the process for the next value of `year`. This is the outer loop of the first part of the program. A nested loop structure like this one is natural for handling a two-dimensional array. One loop handles the first subscript, and the other loop handles the second subscript:

```
for (year = 0, total = 0; year < YEARS; year++)
{             // process each year
    for (month = 0, subtot = 0; month < MONTHS; month++)
        ...   // process each month
    ...       // process each year
}
```

The second part of the program has the same structure, but now it changes `year` with the inner loop and `month` with the outer. Remember, each time the outer loop cycles once, the inner loop cycles its full allotment. Therefore, this arrangement cycles through all the years before changing months. You get a five-year average for the first month, and so on:

```
for (month = 0; month < MONTHS; month++)
{             // process each month
    for (year = 0, subtot =0; year < YEARS; year++)
        ...   // process each year
    ...       // process each month
}
```

Initializing a Two-Dimensional Array

Initializing a two-dimensional array builds on the technique for initializing a one-dimensional array. First, recall that initializing a one-dimensional array looks like this:

```
sometype ar1[5] = {val1, val2, val3, val4, val5};
```

Here `val1`, `val2`, and so on are each a value appropriate for `sometype`. For example, if `sometype` were `int`, `val1` might be `7`, or if `sometype` were `double`, `val1` might be `11.34`. But `rain` is a five-element array for which each element is of type array-of-12-`float`. So, for `rain`, `val1` would be a value appropriate for initializing a one-dimensional array of `float`, such as the following:

```
{4.3,4.3,4.3,3.0,2.0,1.2,0.2,0.2,0.4,2.4,3.5,6.6}
```

That is, if `sometype` is array-of-12-`double`, `val1` is a list of 12 `double` values. Therefore, we need a comma-separated list of five of these things to initialize a two-dimensional array, such as `rain`:

```
const float rain[YEARS][MONTHS] =
{
    {4.3,4.3,4.3,3.0,2.0,1.2,0.2,0.2,0.4,2.4,3.5,6.6},
    {8.5,8.2,1.2,1.6,2.4,0.0,5.2,0.9,0.3,0.9,1.4,7.3},
    {9.1,8.5,6.7,4.3,2.1,0.8,0.2,0.2,1.1,2.3,6.1,8.4},
    {7.2,9.9,8.4,3.3,1.2,0.8,0.4,0.0,0.6,1.7,4.3,6.2},
    {7.6,5.6,3.8,2.8,3.8,0.2,0.0,0.0,0.0,1.3,2.6,5.2}
};
```

This initialization uses five embraced lists of numbers, all enclosed by one outer set of braces. The data in the first interior set of braces is assigned to the first row of the array, the data in the second interior set goes to the second row, and so on. The rules we discussed about mismatches between data and array sizes apply to each row. That is, if the first inner set of braces encloses 10 numbers, only the first 10 elements of the first row are affected. The last two elements in that row are then initialized by default to zero. If there are too many numbers, it is an error; the numbers do not get shoved into the next row.

You could omit the interior braces and just retain the two outermost braces. As long as you have the right number of entries, the effect is the same. If you are short of entries, however, the array is filled sequentially, row by row, until the data runs out. Then the remaining elements are initialized to `0`. Figure 10.2 shows both ways of initializing an array.

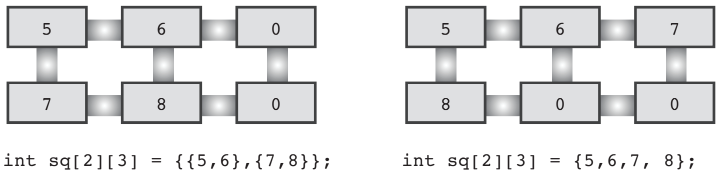

Figure 10.2 Two methods of initializing an array.

Because the `rain` array holds data that should not be modified, the program uses the `const` modifier when declaring the array.

More Dimensions

Everything we have said about two-dimensional arrays can be generalized to three-dimensional arrays and further. You can declare a three-dimensional array this way:

```
int box[10][20][30];
```

You can visualize a one-dimensional array as a row of data, a two-dimensional array as a table of data, and a three-dimensional array as a stack of data tables. For example, you can visualize the `box` array as 10 two-dimensional arrays (each 20×30) stacked atop each other.

The other way to think of `box` is as an array of arrays of arrays. That is, it is a 10-element array, each element of which is a 20-element array. Each 20-element array then has elements that are 30-element arrays. Or, you can simply think of arrays in terms of the number of indices needed.

Typically, you would use three nested loops to process a three-dimensional array, four nested loops to process a four-dimensional array, and so on. We'll stick to two dimensions in our examples.

Pointers and Arrays

Pointers, as you might recall from Chapter 9, "Functions," provide a symbolic way to use addresses. Because the hardware instructions of computing machines rely heavily on addresses, pointers enable you to express yourself in a way that is close to how the machine expresses itself. This correspondence makes programs with pointers efficient. In particular, pointers offer an efficient way to deal with arrays. Indeed, as you will see, array notation is simply a disguised use of pointers.

An example of this disguised use is that an array name is also the address of the first element of the array. That is, if `flizny` is an array, the following is true:

```
flizny == &flizny[0];     // name of array is the address of the first element
```

Both `flizny` and `&flizny[0]` represent the memory address of that first element. (Recall that `&` is the address operator.) Both are *constants* because they remain fixed for the duration of the program. However, they can be assigned as values to a pointer *variable*, and you can change the value of a variable, as Listing 10.8 shows. Notice what happens to the value of a pointer when you add a number to it. (Recall that the `%p` specifier for pointers typically displays hexadecimal values.)

Listing 10.8 **The `pnt_add.c` Program**

```
// pnt_add.c -- pointer addition
#include <stdio.h>
#define SIZE 4
int main(void)
{
    short dates [SIZE];
    short * pti;
    short index;
    double bills[SIZE];
    double * ptf;

    pti = dates;    // assign address of array to pointer
    ptf = bills;
    printf("%23s %15s\n", "short", "double");
    for (index = 0; index < SIZE; index ++)
        printf("pointers + %d: %10p %10p\n",
                index, pti + index, ptf + index);

    return 0;
}
```

Here is sample output:

```
                  short          double
pointers + 0: 0x7fff5fbff8dc 0x7fff5fbff8a0
pointers + 1: 0x7fff5fbff8de 0x7fff5fbff8a8
pointers + 2: 0x7fff5fbff8e0 0x7fff5fbff8b0
pointers + 3: 0x7fff5fbff8e2 0x7fff5fbff8b8
```

The second line prints the beginning addresses of the two arrays, and the next line gives the result of adding 1 to the address, and so on. Keep in mind that the addresses are in hexadecimal, so dd is 1 more than dc and a1 is 1 more than a0. But what do we have here?

```
0x7fff5fbff8dc + 1 is 0x7fff5fbff8de?
0x7fff5fbff8a0 + 1 is 0x7fff5fbff8a8?
```

Pretty dumb? Like a fox! Our system is addressed by individual bytes, but type `short` uses 2 bytes and type `double` uses 8 bytes. What is happening here is that when you say "add 1 to a pointer," C adds one *storage unit*. For arrays, that means the address is increased to the address of the next *element*, not just the next byte (see Figure 10.3). This is one reason why you have to declare the sort of object to which a pointer points. The address is not enough because the computer needs to know how many bytes are used to store the object. (This is true even for pointers to scalar variables; otherwise, the `*pt` operation to fetch the value wouldn't work correctly.)

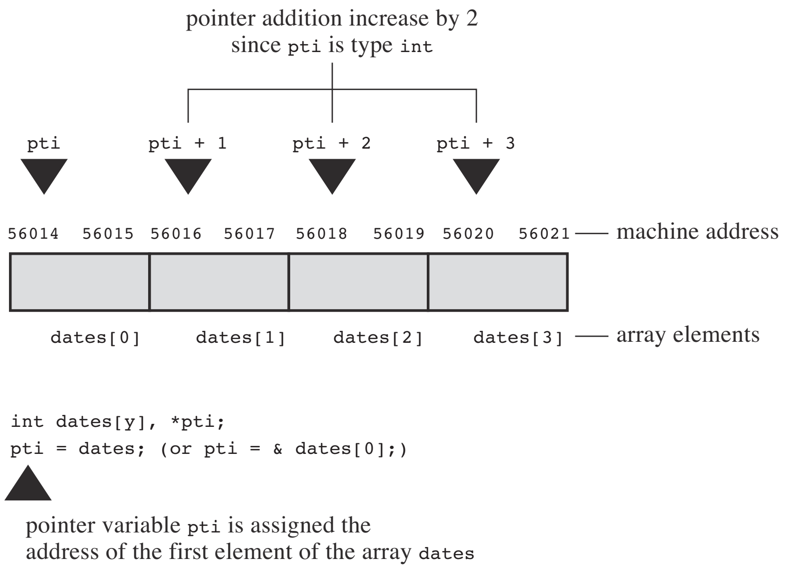

Figure 10.3 An array and pointer addition.

Now we can define more clearly what is meant by pointer-to-`int`, pointer-to-`float`, or pointer-to–any other data object:

- The value of a pointer is the address of the object to which it points. How the address is represented internally is hardware dependent. Many computers, including PCs and Macintoshes, are *byte addressable*, meaning that each byte in memory is numbered sequentially. Here, the address of a large object, such as type `double` variable, typically is the address of the first byte of the object.
- Applying the `*` operator to a pointer yields the value stored in the pointed-to object.
- Adding 1 to the pointer increases its value by the size, in bytes, of the pointed-to type.

As a result of C's cleverness, we have the following equalities:

```
dates + 2 == &date[2]        // same address
*(dates + 2) == dates[2]     // same value
```

These relationships sum up the close connection between arrays and pointers. They mean that you can use a pointer to identify an individual element of an array and to obtain its value. In essence, we have two different notations for the same thing. Indeed, the C language standard describes array notation in terms of pointers. That is, it defines `ar[n]` to mean `*(ar + n)`. You can think of the second expression as meaning, "Go to memory location `ar`, move over `n` units, and retrieve the value there."

Incidentally, don't confuse `*(dates+2)` with `*dates+2`. The indirection operator (`*`) binds more tightly (that is, has higher precedence) than `+`, so the latter means `(*dates)+2`:

```
*(dates + 2)       // value of the 3rd element of dates
*dates + 2         // 2 added to the value of the 1st element
```

The relationship between arrays and pointers means that you can often use either approach when writing a program. Listing 10.9, for instance, produces the same output as Listing 10.1 when compiled and run.

Listing 10.9 **The `day_mon3.c` Program**

```
/* day_mon3.c -- uses pointer notation */
#include <stdio.h>
#define MONTHS 12

int main(void)
{
    int days[MONTHS] = {31,28,31,30,31,30,31,31,30,31,30,31};
    int index;

    for (index = 0; index < MONTHS; index++)
        printf("Month %2d has %d days.\n", index +1,
               *(days + index));   // same as days[index]

    return 0;
}
```

Here, `days` is the address of the first element of the array, `days + index` is the address of element `days[index]`, and `*(days + index)` is the value of that element, just as `days[index]` is. The loop references each element of the array, in turn, and prints the contents of what it finds.

Is there an advantage to writing the program this way? Not really—the compiler produces the same code for either. The point to Listing 10.9 is that pointer notation and array notation are two equivalent methods. This example shows that you can use pointer notation with arrays. The reverse is also true; you can use array notation with pointers. This turns out to be important when you have a function with an array as an argument.

Functions, Arrays, and Pointers

Suppose you want to write a function that operates on an array. For example, suppose you want a function that returns the sum of the elements of an array. Suppose `marbles` is the name of an array of `int`. What would the function call look like? A reasonable guess would be this:

```
total = sum(marbles); // possible function call
```

What would the prototype be? Remember, the name of an array is the address of its first element, so the actual argument `marbles`, being the address of an `int`, should be assigned to a formal parameter that is a pointer-to-`int`:

```
int sum(int * ar);  // corresponding prototype
```

What information does `sum()` get from this argument? It gets the address of the first element of the array, and it learns that it will find an `int` at that location. Note that this information says nothing about the number of elements in the array. We're left with a couple choices of how to get that information to the function. The first choice is to code a fixed array size into the function:

```
int sum(int * ar)     // corresponding definition
{
    int i;
    int total = 0;

    for( i = 0; i < 10; i++)   // assume 10 elements
        total += ar[i];    // ar[i] the same as *(ar + i)
    return total;
}
```

Here, we make use of the fact that just as you can use pointer notation with array names, you can use array notation with a pointer. Also, recall that the `+=` operator adds the value of the operand on its right to the operand on its left. Therefore, `total` is a running sum of the array elements.

This function definition is limited; it will work only with `int` arrays of 10 elements. A more flexible approach is to pass the array size as a second argument:

```
int sum(int * ar, int n)  // more general approach
{
    int i;
    int total = 0;

    for( i = 0; i < n; i++)   // use n elements
        total += ar[i];       // ar[i] the same as *(ar + i)
    return total;
}
```

Here, the first parameter tells the function where to find the array and the type of data in the array, and the second parameter tells the function how many elements are present.

There's one more thing to tell about function parameters. In the context of a function prototype or function definition header, and *only* in that context, you can substitute `int ar[]` for `int * ar`:

```
int sum (int ar[], int n);
```

The form `int * ar` always means that `ar` is type pointer-to-`int`. The form `int ar[]` also means that `ar` is type pointer-to-`int`, but *only* when used to declare formal parameters. The idea is that the second form reminds the reader that not only does `ar` point to an `int`, it points to an `int` that's an element of an array.

Note Declaring Array Parameters

Because the name of an array is the address of the first element, an actual argument of an array name requires that the matching formal argument be a pointer. In this context, and only in this context, C interprets `int ar[]` to mean the same as `int * ar`; that is, `ar` is type pointer-to-`int`. Because prototypes allow you to omit a name, all four of the following prototypes are equivalent:

```
int sum(int *ar, int n);
int sum(int *, int);
int sum(int ar[], int n);
int sum(int [], int);
```

You can't omit names in function definitions, so, for definitions, the following two forms are equivalent:

```
int sum(int *ar, int n)
{
    // code goes here
}

int sum(int ar[], int n);
{
    // code goes here
}
```

You should be able to use any of the four prototypes with either of the two definitions shown here.

Listing 10.10 shows a program using the `sum()` function. To point out an interesting fact about array arguments, the program also prints the size of the original array and the size of the function parameter representing the array. (Use `%u` or perhaps `%lu` if your compiler doesn't support the `%zd` specifier for printing `sizeof` quantities.)

Listing 10.10 **The `sum_arr1.c` Program**

```
// sum_arr1.c -- sums the elements of an array
// use %u or %lu if %zd doesn't work
#include <stdio.h>
#define SIZE 10
int sum(int ar[], int n);
int main(void)
{
    int marbles[SIZE] = {20,10,5,39,4,16,19,26,31,20};
    long answer;

    answer = sum(marbles, SIZE);
    printf("The total number of marbles is %ld.\n", answer);
    printf("The size of marbles is %zd bytes.\n",
          sizeof marbles);

    return 0;
}

int sum(int ar[], int n)     // how big an array?
{
    int i;
    int total = 0;

    for( i = 0; i < n; i++)
        total += ar[i];
    printf("The size of ar is %zd bytes.\n", sizeof ar);

    return total;
}
```

The output on our system looks like this:

```
The size of ar is 8 bytes.
The total number of marbles is 190.
The size of marbles is 40 bytes.
```

Note that the size of `marbles` is 40 bytes. This makes sense because `marbles` contains 10 `int`s, each 4 bytes, for a total of 40 bytes. But the size of `ar` is just 8 bytes. That's because `ar` is not an array itself; it is a pointer to the first element of `marbles`. Our system uses 8 bytes for storing addresses, so the size of a pointer variable is 8 bytes. (Other systems might use a different number of bytes.) In short, in Listing 10.10, `marbles` is an array, `ar` is a pointer to the first element of `marbles`, and the C connection between arrays and pointers lets you use array notation with the pointer `ar`.

Using Pointer Parameters

A function working on an array needs to know where to start and stop. The `sum()` function uses a pointer parameter to identify the beginning of the array and an integer parameter to indicate how many elements to process. (The pointer parameter also identifies the type of data in the array.) But this is not the only way to tell a function what it needs to know. Another way to describe the array is by passing two pointers, with the first indicating where the array starts (as before) and the second where the array ends. Listing 10.11 illustrates this approach. It also uses the fact that a pointer parameter is a variable, which means that instead of using an index to indicate which element in the array to access, the function can alter the value of the pointer itself, making it point to each array element in turn.

Listing 10.11 **The `sum_arr2.c` Program**

```
/* sum_arr2.c -- sums the elements of an array */
#include <stdio.h>
#define SIZE 10
int sump(int * start, int * end);
int main(void)
{
    int marbles[SIZE] = {20,10,5,39,4,16,19,26,31,20};
    long answer;

    answer = sump(marbles, marbles + SIZE);
    printf("The total number of marbles is %ld.\n", answer);

    return 0;
}

/* use pointer arithmetic   */
int sump(int * start, int * end)
{
    int total = 0;

    while (start < end)
    {
        total += *start; // add value to total
        start++;         // advance pointer to next element
    }

    return total;
}
```

The pointer `start` begins by pointing to the first element of `marbles`, so the assignment expression `total +=*start` adds the value of the first element (20) to `total`. Then the expression `start++` increments the pointer variable `start` so that it points to the next element in the array. Because `start` points to type `int`, C increments the value of `start` by the size of `int`.

Note that the `sump()` function uses a different method from `sum()` to end the summation loop. The `sum()` function uses the number of elements as a second argument, and the loop uses that value as part of the loop test:

```
for( i = 0; i < n; i++)
```

The `sump()` function, however, uses a second pointer to end the loop:

```
while (start < end)
```

Because the test is for inequality, the last element processed is the one just before the element pointed to by `end`. This means that `end` actually points to the location after the final element in the array. C guarantees that when it allocates space for an array, a pointer to the first location after the end of the array is a valid pointer. That makes constructions such as this one valid, because the final value that `start` gets in the loop is `end`. Note that using this "past-the-end" pointer makes the function call neat:

```
answer = sump(marbles, marbles + SIZE);
```

Because indexing starts at `0`, `marbles + SIZE` points to the next element after the end. If `end` pointed to the last element instead of to one past the end, you would have to use the following code instead:

```
answer = sump(marbles, marbles + SIZE - 1);
```

Not only is this code less elegant in appearance, it's harder to remember, so it is more likely to lead to programming errors. By the way, although C guarantees that the pointer `marbles + SIZE` is a valid pointer, it makes no guarantees about `marbles[SIZE]`, the value stored at that location, so a program should not attempt to access that location.

You can also condense the body of the loop to one line:

```
total += *start++;
```

The unary operators `*` and `++` have the same precedence but associate from right to left. This means the `++` applies to `start`, not to `*start`. That is, the pointer is incremented, not the value pointed to. The use of the postfix form (`start++` rather than `++start`) means that the pointer is not incremented until after the pointed-to value is added to `total`. If the program used `*++start`, the order would be increment the pointer, then use the value pointed to. If the program used `(*start)++`, however, it would use the value of `start` and then increment the value, not the pointer. That would leave the pointer pointing to the same element, but the element would contain a new number. Although the `*start++` notation is commonly used, the `*(start++)` notation is clearer. Listing 10.12 illustrates these niceties of precedence.

Listing 10.12 **The `order.c` Program**

```
/* order.c -- precedence in pointer operations */
#include <stdio.h>
int data[2] = {100, 200};
int moredata[2] = {300, 400};
int main(void)
{
    int * p1, * p2, * p3;

    p1 = p2 = data;
    p3 = moredata;
    printf("  *p1 = %d,   *p2 = %d,     *p3 = %d\n",
              *p1     ,   *p2     ,     *p3);
    printf("*p1++ = %d, *++p2 = %d, (*p3)++ = %d\n",
            *p1++     , *++p2     , (*p3)++);
    printf("  *p1 = %d,   *p2 = %d,     *p3 = %d\n",
              *p1     ,   *p2     ,     *p3);

    return 0;
}
```

Here is its output:

```
  *p1 = 100,   *p2 = 100,     *p3 = 300
*p1++ = 100, *++p2 = 200, (*p3)++ = 300
  *p1 = 200,   *p2 = 200,     *p3 = 301
```

The only operation that altered an array value is `(*p3)++`. The other two operations caused `p1` and `p2` to advance to point to the next array element.

Comment: Pointers and Arrays

As you have seen, functions that process arrays actually use pointers as arguments, but you do have a choice between array notation and pointer notation for writing array-processing functions. Using array notation, as in Listing 10.10, makes it more obvious that the function is working with arrays. Also, array notation has a more familiar look to programmers versed in other languages, such as FORTRAN, Pascal, Modula-2, or BASIC. Other programmers might be more accustomed to working with pointers and might find the pointer notation, such as that in Listing 10.11, more natural.

As far as C goes, the two expressions `ar[i]` and `*(ar+i)` are equivalent in meaning. Both work if `ar` is the name of an array, and both work if `ar` is a pointer variable. However, using an expression such as `ar++` only works if `ar` is a pointer variable.

Pointer notation, particularly when used with the increment operator, is closer to machine language and, with some compilers, leads to more efficient code. However, many programmers believe that the programmer's main concerns should be correctness and clarity and that code optimization should be left to the compiler.

Pointer Operations

Just what can you do with pointers? C offers several basic operations you can perform on pointers, and the next program demonstrates eight of these possibilities. To show the results of each operation, the program prints the value of the pointer (which is the address to which it points), the value stored in the pointed-to address, and the address of the pointer itself. (If your compiler doesn't support the `%p` specifier, try `%u` or perhaps `%lu` for printing the addresses. If it doesn't support the `%td` specifier, used for address differences, try `%d` or perhaps `%ld`.)

Listing 10.13 shows eight basic operations that can be performed with pointer variables. In addition to these operations, you can use the relational operators to compare pointers.

Listing 10.13 **The `ptr_ops.c` Program**

```
// ptr_ops.c -- pointer operations
#include <stdio.h>
int main(void)
{
    int urn[5] = {100,200,300,400,500};
    int * ptr1, * ptr2, *ptr3;

    ptr1 = urn;         // assign an address to a pointer
    ptr2 = &urn[2];     // ditto
                        // dereference a pointer and take
                        // the address of a pointer
    printf("pointer value, dereferenced pointer, pointer address:\n");
    printf("ptr1 = %p, *ptr1 =%d, &ptr1 = %p\n",
           ptr1, *ptr1, &ptr1);

    // pointer addition
    ptr3 = ptr1 + 4;
    printf("\nadding an int to a pointer:\n");
    printf("ptr1 + 4 = %p, *(ptr4 + 3) = %d\n",
            ptr1 + 4, *(ptr1 + 3));
    ptr1++;            // increment a pointer
    printf("\nvalues after ptr1++:\n");
    printf("ptr1 = %p, *ptr1 =%d, &ptr1 = %p\n",
           ptr1, *ptr1, &ptr1);
    ptr2--;            // decrement a pointer
    printf("\nvalues after --ptr2:\n");
    printf("ptr2 = %p, *ptr2 = %d, &ptr2 = %p\n",
           ptr2, *ptr2, &ptr2);
    --ptr1;            // restore to original value
    ++ptr2;            // restore to original value
    printf("\nPointers reset to original values:\n");
    printf("ptr1 = %p, ptr2 = %p\n", ptr1, ptr2);
                        // subtract one pointer from another
    printf("\nsubtracting one pointer from another:\n");
    printf("ptr2 = %p, ptr1 = %p, ptr2 - ptr1 = %td\n",
            ptr2, ptr1, ptr2 - ptr1);
                       // subtract an integer from a pointer
    printf("\nsubtracting an int from a pointer:\n");
    printf("ptr3 = %p, ptr3 - 2 = %p\n",
            ptr3,  ptr3 - 2);

    return 0;
}
```

Here is the output on one system:

```
pointer value, dereferenced pointer, pointer address:
ptr1 = 0x7fff5fbff8d0, *ptr1 =100, &ptr1 = 0x7fff5fbff8c8

adding an int to a pointer:
ptr1 + 4 = 0x7fff5fbff8e0, *(ptr4 + 3) = 400

values after ptr1++:
ptr1 = 0x7fff5fbff8d4, *ptr1 =200, &ptr1 = 0x7fff5fbff8c8

values after --ptr2:
ptr2 = 0x7fff5fbff8d4, *ptr2 = 200, &ptr2 = 0x7fff5fbff8c0

Pointers reset to original values:
ptr1 = 0x7fff5fbff8d0, ptr2 = 0x7fff5fbff8d8

subtracting one pointer from another:
ptr2 = 0x7fff5fbff8d8, ptr1 = 0x7fff5fbff8d0, ptr2 - ptr1 = 2

subtracting an int from a pointer:
ptr3 = 0x7fff5fbff8e0, ptr3 - 2 = 0x7fff5fbff8d8
```

The following list describes the basic operations that can be performed with or on pointer variables:

- **Assignment—**You can assign an address to a pointer. The assigned value can be, for example, an array name, a variable preceded by address operator (`&`), or another second pointer. In the example, `ptr1` is assigned the address of the beginning of the array `urn`. This address happens to be memory cell number `0x7fff5fbff8d0`. The variable `ptr2` gets the address of the third and last element, `urn[2]`. Note that the address should be compatible with the pointer type. That is, you can't assign the address of a `double` to a pointer-to-`int`, at least not without making an ill-advised type cast. C99/C11 enforces this rule.
- **Value finding (dereferencing)—**The `*` operator gives the value stored in the pointed-to location. Therefore, `*ptr1` is initially `100`, the value stored at location `0x7fff5fbff8d0`.
- **Taking a pointer address—**Like all variables, a pointer variable has an address and a value. The `&` operator tells you where the pointer itself is stored. In this example, `ptr1` is stored in memory location `0x7fff5fbff8c8`. The content of that memory cell is `0x7fff5fbff8d0`, the address of `urn`. So `&pt1` is a pointer to `pt1`, which, in turn, is a pointer to `urn[0]`.

- **Adding an integer to a pointer**—You can use the + operator to add an integer to a pointer or a pointer to an integer. In either case, the integer is multiplied by the number of bytes in the pointed-to type, and the result is added to the original address. This makes `ptr1 + 4` the same as `&urn[4]`. The result of addition is undefined if it lies outside of the array into which the original pointer points, except that the address one past the end element of the array is guaranteed to be valid.
- **Incrementing a pointer**—Incrementing a pointer to an array element makes it move to the next element of the array. Therefore, `ptr1++` increases the numerical value of `ptr1` by `4` (4 bytes per `int` on our system) and makes `ptr1` point to `urn[1]` (see Figure 10.4, which uses simplified addresses). Now `ptr1` has the value `0x7fff5fbff8d4` (the next array address), and `*ptr1` has the value `200` (the value of `urn[1]`). Note that the address of `ptr1` itself remains `0x7fff5fbff8c8`. After all, a variable doesn't move around just because it changes value!

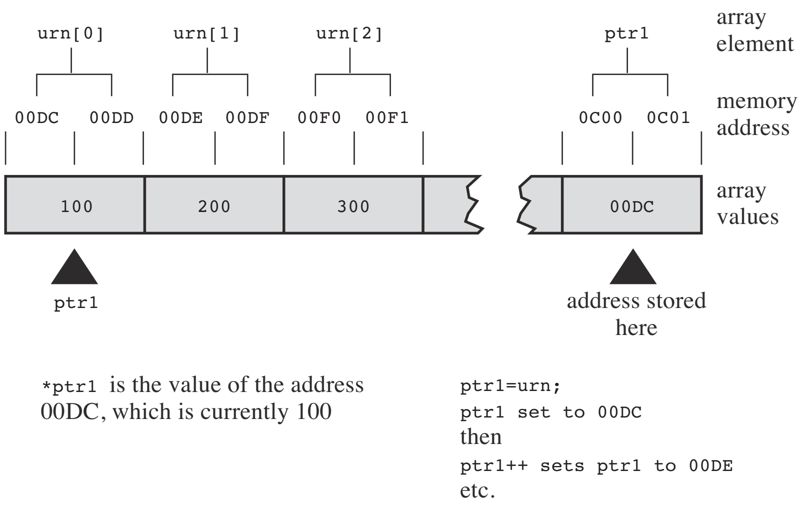

Figure 10.4 Incrementing a type `int` pointer.

- **Subtracting an integer from a pointer**—You can use the `-` operator to subtract an integer from a pointer; the pointer has to be the first operand and the integer value the second operand. The integer is multiplied by the number of bytes in the pointed-to type, and the result is subtracted from the original address. This makes `ptr3 - 2` the same as `&urn[2]` because `ptr3` points to `&urn[4]`. The result of subtraction is undefined if it lies outside of the array into which the original pointer points, except that the address one past the end element of the array is guaranteed to be valid.
- **Decrementing a pointer**—Of course, you can also decrement a pointer. In this example, decrementing `ptr2` makes it point to the second array element instead of the third. Note that you can use both the prefix and postfix forms of the increment and decrement

operators. Also note that both `ptr1` and `ptr2` wind up pointing to the same element, `urn[1]`, before they get reset.

- **Differencing—**You can find the difference between two pointers. Normally, you do this for two pointers to elements that are in the same array to find out how far apart the elements are. The result is in the same units as the type size. For example, in the output from Listing 10.13, `ptr2 - ptr1` has the value `2`, meaning that these pointers point to objects separated by two `int`s, not by 2 bytes. Subtraction is guaranteed to be a valid operation as long as both pointers point into the same array (or possibly to a position one past the end). Applying the operation to pointers to two different arrays might produce a value or could lead to a runtime error.
- **Comparisons—**You can use the relational operators to compare the values of two pointers, provided the pointers are of the same type.

Note that there are two forms of subtraction. You can subtract one pointer from another to get an integer, and you can subtract an integer from a pointer and get a pointer.

There are some cautions to remember when incrementing or decrementing a pointer. The computer does not keep track of whether a pointer still points to an array element. C guarantees that, given an array, a pointer to any array element, or to the position after the last element, is a valid pointer. But the effect of incrementing or decrementing a pointer beyond these limits is undefined. Also, you can dereference a pointer to any array element. However, even though a pointer to one past the end element is valid, it's not guaranteed that such a one-past-the-end pointer can be dereferenced.

Dereferencing an Uninitialized Pointer

Speaking of cautions, there is one rule you should burn into your memory: Do not dereference an uninitialized pointer. For example, consider the following:

```
int * pt;  // an uninitialized pointer
*pt = 5;   // a terrible error
```

Why is this so bad? The second line means store the value `5` in the location to which `pt` points. But `pt`, being uninitialized, has a random value, so there is no knowing where the `5` will be placed. It might go somewhere harmless, it might overwrite data or code, or it might cause the program to crash. Remember, creating a pointer only allocates memory to store the pointer itself; it doesn't allocate memory to store data. Therefore, before you use a pointer, it should be assigned a memory location that has already been allocated. For example, you can assign the address of an existing variable to the pointer. (This is what happens when you use a function with a pointer parameter.) Or you can use the `malloc()` function, as discussed in Chapter 12, to allocate memory first. Anyway, to drive the point home, do not dereference an uninitialized pointer!

```
double * pd;  // uninitialized pointer
*pd = 2.4;    // DON'T DO IT
```

Given

```
int urn[3];
int * ptr1, * ptr2;
```

the following are some valid and invalid statements:

Valid	**Invalid**
`ptr1++;`	`urn++;`
`ptr2 = ptr1 + 2;`	`ptr2 = ptr2 + ptr1;`
`ptr2 = urn + 1;`	`ptr2 = urn * ptr1;`

The valid operations open many possibilities. C programmers create arrays of pointers, pointers to functions, arrays of pointers to pointers, arrays of pointers to functions, and so on. Relax, though—we'll stick to the basic uses we have already unveiled. The first basic use for pointers is to communicate information to and from functions. You already know that you must use pointers if you want a function to affect variables in the calling function. The second use is in functions designed to manipulate arrays. Let's look at another programming example using functions and arrays.

Protecting Array Contents

When you write a function that processes a fundamental type, such as `int`, you have a choice of passing the `int` by value or of passing a pointer-to-`int`. The usual rule is to pass quantities by value unless the program needs to alter the value, in which case you pass a pointer. Arrays don't give you that choice; you *must* pass a pointer. The reason is efficiency. If a function passed an array by value, it would have to allocate enough space to hold a copy of the original array and then copy all the data from the original array to the new array. It is much quicker to pass the address of the array and have the function work with the original data.

This technique can cause problems. The reason C ordinarily passes data by value is to preserve the integrity of the data. If a function works with a copy of the original data, it won't accidentally modify the original data. But, because array-processing functions do work with the original data, they *can* modify the array. Sometimes that's desirable. For example, here's a function that adds the same value to each member of an array:

```
void add_to(double ar[], int n, double val)
{
    int i;
    for( i = 0; i < n; i++)
        ar[i] += val;
}
```

Therefore, the function call

```
add_to(prices, 100, 2.50);
```

causes each element in the `prices` array to be replaced by a value larger by 2.5; this function modifies the contents of the array. It can do so because, by working with pointers, the function uses the original data.

Other functions, however, do not have the intent of modifying data. The following function, for example, is intended to find the sum of the array's contents; it shouldn't change the array. However, because `ar` is really a pointer, a programming error could lead to the original data being corrupted. Here, for example, the expression `ar[i]++` results in each element having 1 added to its value:

```
int sum(int ar[], int n)  // faulty code
{
    int i;
    int total = 0;

    for( i = 0; i < n; i++)
        total += ar[i]++;   // error increments each element
    return total;
}
```

Using `const` with Formal Parameters

With K&R C, the only way to avoid this sort of error is to be vigilant. Since ANSI C, there is an alternative. If a function's intent is that it not change the contents of the array, use the keyword `const` when declaring the formal parameter in the prototype and in the function definition. For example, the prototype and definition for `sum()` should look like this:

```
int sum(const int ar[], int n);  /* prototype  */

int sum(const int ar[], int n)   /* definition */
{
    int i;
    int total = 0;

    for( i = 0; i < n; i++)
        total += ar[i];
    return total;
}
```

This tells the compiler that the function should treat the array pointed to by `ar` as though the array contains constant data. Then, if you accidentally use an expression such as `ar[i]++`, the compiler can catch it and generate an error message, telling you that the function is attempting to alter constant data.

It's important to understand that using `const` this way does not require that the original array *be* constant; it just says that the function has to treat the array *as though* it were constant. Using `const` this way provides the protection for arrays that passing by value provides for fundamental types; it prevents a function from modifying data in the calling function. In general, if you write a function intended to modify an array, don't use `const` when declaring the array parameter. If you write a function not intended to modify an array, do use `const` when declaring the array parameter.

In the program shown in Listing 10.14, one function displays an array and one function multiplies each element of an array by a given value. Because the first function should not alter the array, it uses `const`. Because the second function has the intent of modifying the array, it doesn't use `const`.

Listing 10.14 **The `arf.c` Program**

```
/* arf.c -- array functions */
#include <stdio.h>
#define SIZE 5
void show_array(const double ar[], int n);
void mult_array(double ar[], int n, double mult);
int main(void)
{
    double dip[SIZE] = {20.0, 17.66, 8.2, 15.3, 22.22};

    printf("The original dip array:\n");
    show_array(dip, SIZE);
    mult_array(dip, SIZE, 2.5);
    printf("The dip array after calling mult_array():\n");
    show_array(dip, SIZE);

    return 0;
}

/* displays array contents */
void show_array(const double ar[], int n)
{
    int i;

    for (i = 0; i < n; i++)
        printf("%8.3f ", ar[i]);
    putchar('\n');
}

/* multiplies each array member by the same multiplier */
void mult_array(double ar[], int n, double mult)
{
    int i;
    for (i = 0; i < n; i++)
        ar[i] *= mult;
}
```

Here is the output:

```
The original dip array:
  20.000   17.660    8.200   15.300   22.220
The dip array after calling mult_array():
  50.000   44.150   20.500   38.250   55.550
```

Note that both functions are type `void`. The `mult_array()` function does provide new values to the `dip` array, but not by using the `return` mechanism.

More About `const`

Earlier, you saw that you can use `const` to create symbolic constants:

```
const double PI = 3.14159;
```

That was something you could do with the `#define` directive, too, but `const` additionally lets you create constant arrays, constant pointers, and pointers to constants.

Listing 10.4 showed how to use the `const` keyword to protect an array:

```
#define MONTHS 12
...
const int days[MONTHS] = {31,28,31,30,31,30,31,31,30,31,30,31};
```

If the program code subsequently tries to alter the array, you'll get a compile-time error message:

```
days[9] = 44;    /* compile error */
```

Pointers to constants can't be used to change values. Consider the following code:

```
double rates[5] = {88.99, 100.12, 59.45, 183.11, 340.5};
const double * pd = rates;    // pd points to beginning of the array
```

The second line of code declares that the type `double` value to which `pd` points is a `const`. That means you can't use `pd` to change pointed-to values:

```
*pd = 29.89;      // not allowed
pd[2] = 222.22;   // not allowed
rates[0] = 99.99; // allowed because rates is not const
```

Whether you use pointer notation or array notation, you are not allowed to use `pd` to change the value of pointed-to data. Note, however, that because `rates` was not declared as a constant, you can still use `rates` to change values. Also, note that you can make `pd` point somewhere else:

```
pd++;       /* make pd point to rates[1] -- allowed */
```

A pointer-to-constant is normally used as a function parameter to indicate that the function won't use the pointer to change data. For example, the `show_array()` function from Listing 10.14 could have been prototyped as

```
void show_array(const double *ar, int n);
```

There are some rules you should know about pointer assignments and `const`. First, it's valid to assign the address of either constant data or non-constant data to a pointer-to-constant:

```
double rates[5] = {88.99, 100.12, 59.45, 183.11, 340.5};
const double locked[4] = {0.08, 0.075, 0.0725, 0.07};
const double * pc = rates;    // valid
pc = locked;                  // valid
pc = &rates[3];               // valid
```

However, only the addresses of non-constant data can be assigned to regular pointers:

```
double rates[5] = {88.99, 100.12, 59.45, 183.11, 340.5};
const double locked[4] = {0.08, 0.075, 0.0725, 0.07};
double * pnc = rates;          // valid
pnc = locked;                  // not valid
pnc = &rates[3];               // valid
```

This is a reasonable rule. Otherwise, you could use the pointer to change data that was supposed to be constant.

A practical consequence of these rules is that a function such as `show_array()` can accept the names of regular arrays *and* of constant arrays as actual arguments, because either can be assigned to a pointer-to-constant:

```
show_array(rates, 5);    // valid
show_array(locked, 4);   // valid
```

Therefore, using `const` in a function parameter definition not only protects data, it also allows the function to work with arrays that have been declared `const`.

A function such as `mult_array()`, however, shouldn't be passed the name of a constant array as an argument:

```
mult_array(rates, 5, 1.2);    // valid
mult_array(locked, 4, 1.2);   // bad idea
```

What the C standard says is that an attempt to modify `const` data, such as `locked`, using a non-`const` identifier, such as the `mult_array()` formal argument `ar`, results in undefined behavior.

There are more possible uses of `const`. For example, you can declare and initialize a pointer so that it can't be made to point elsewhere. The trick is the placement of the keyword `const`:

```
double rates[5] = {88.99, 100.12, 59.45, 183.11, 340.5};
double * const pc = rates;    // pc points to beginning of the array
pc = &rates[2];               // not allowed to point elsewhere
*pc = 92.99;                  // ok -- changes rates[0]
```

Such a pointer can still be used to change values, but it can point only to the location originally assigned to it.

Finally, you can use `const` twice to create a pointer that can neither change where it's pointing nor change the value to which it points:

```
double rates[5] = {88.99, 100.12, 59.45, 183.11, 340.5};
const double * const pc = rates;
pc = &rates[2];               // not allowed
*pc = 92.99;                  // not allowed
```

Pointers and Multidimensional Arrays

How do pointers relate to multidimensional arrays? And why would you want to know? Functions that work with multidimensional arrays do so with pointers, so you need some further pointer background before working with such functions. As to the first question, let's look at some examples now to find the answer. To simplify the discussion, let's use a small array. Suppose you have this declaration:

```
int zippo[4][2];  /* an array of arrays of ints */
```

Then `zippo`, being the name of an array, is the address of the first element of the array. In this case, the first element of `zippo` is itself an array of two `int`s, so `zippo` is the address of an array of two `int`s. Let's analyze that further in terms of pointer properties:

- Because `zippo` is the address of the array's first element, `zippo` has the same value as `&zippo[0]`. Next, `zippo[0]` is itself an array of two integers, so `zippo[0]` has the same value as `&zippo[0][0]`, the address of its first element, an `int`. In short, `zippo[0]` is the address of an `int`-sized object, and `zippo` is the address of a two-`int`-sized object. Because both the integer and the array of two integers begin at the same location, both `zippo` and `zippo[0]` have the same numeric value.
- Adding 1 to a pointer or address yields a value larger by the size of the referred-to object. In this respect, `zippo` and `zippo[0]` differ, because `zippo` refers to an object two `int`s in size, and `zippo[0]` refers to an object one `int` in size. Therefore, `zippo + 1` has a different value from `zippo[0] + 1`.
- Dereferencing a pointer or an address (applying the `*` operator or else the `[]` operator with an index) yields the value represented by the referred-to object. Because `zippo[0]` is the address of its first element, (`zippo[0][0]`), `*(zippo[0])` represents the value stored

in `zippo[0][0]`, an `int` value. Similarly, `*zippo` represents the value of its first element, `zippo[0]`, but `zippo[0]` itself is the address of an `int`. It's the address `&zippo[0][0]`, so `*zippo` is `&zippo[0][0]`. Applying the dereferencing operator to both expressions implies that `**zippo` equals `*&zippo[0][0]`, which reduces to `zippo[0][0]`, an `int`. In short, `zippo` is the address of an address and must be dereferenced twice to get an ordinary value. An address of an address or a pointer of a pointer is an example of *double indirection*.

Clearly, increasing the number of array dimensions increases the complexity of the pointer view. At this point, most students of C begin realizing why pointers are considered one of the more difficult aspects of the language. You might want to study the preceding points carefully and see how they are illustrated in Listing 10.15, which displays some address values and array contents.

Listing 10.15 **The `zippo1.c` Program**

```
/* zippo1.c --  zippo info */
#include <stdio.h>
int main(void)
{
    int zippo[4][2] = { {2,4}, {6,8}, {1,3}, {5, 7} };

    printf("   zippo = %p,    zippo + 1 = %p\n",
              zippo,         zippo + 1);
    printf("zippo[0] = %p, zippo[0] + 1 = %p\n",
            zippo[0],      zippo[0] + 1);
    printf("  *zippo = %p,   *zippo + 1 = %p\n",
             *zippo,        *zippo + 1);
    printf("zippo[0][0] = %d\n", zippo[0][0]);
    printf("  *zippo[0] = %d\n", *zippo[0]);
    printf("    **zippo = %d\n", **zippo);
    printf("      zippo[2][1] = %d\n", zippo[2][1]);
    printf("*(*(zippo+2) + 1) = %d\n", *(*(zippo+2) + 1));

    return 0;
}
```

Here is the output for one system:

```
   zippo = 0x0064fd38,    zippo + 1 = 0x0064fd40
zippo[0] = 0x0064fd38, zippo[0] + 1 = 0x0064fd3c
  *zippo = 0x0064fd38,   *zippo + 1 = 0x0064fd3c
zippo[0][0] = 2
  *zippo[0] = 2
```

```
    **zippo = 2
      zippo[1][2] = 3
*(*(zippo+1) + 2) = 3
```

Other systems might display different address values and address formats, but the relationships will be the same as described here. The output shows that the address of the two-dimensional array, `zippo`, and the address of the one-dimensional array, `zippo[0]`, are the same. Each is the address of the corresponding array's first element, and this is the same numerically as `&zippo[0][0]`.

Nonetheless, there is a difference. On our system, `int` is 4 bytes. As discussed earlier, `zippo[0]` points to a 4-byte data object. Adding 1 to `zippo[0]` should produce a value larger by 4, which it does. (In hex, `38 + 4` is `3c`.) The name `zippo` is the address of an array of two `int`s, so it identifies an 8-byte data object. Therefore, adding 1 to `zippo` should produce an address 8 bytes larger, which it does. (In hex, `40` is `8` larger than `38`.)

The program shows that `zippo[0]` and `*zippo` are identical, and they should be. Next, it shows that the name of a two-dimensional array has to be dereferenced twice to get a value stored in the array. This can be done by using the indirection operator (`*`) twice or by using the bracket operator (`[ ]`) twice. (It also can be done by using one `*` and one set of `[ ]`, but let's not get carried away by all the possibilities.)

In particular, note that the pointer notation equivalent of `zippo[2][1]` is `*(*(zippo+2) + 1)`. You probably should make the effort at least once in your life to break this down. Let's build up the expression in steps:

`zippo`	←the address of the first two-`int` element
`zippo+2`	←the address of the third two-`int` element
`*(zippo+2)`	←the third element, a two-`int` array, hence the address of its first element, an `int`
`*(zippo+2) + 1`	←the address of the second element of the two-`int` array, also an `int`
`*(*(zippo+2) + 1)`	←the value of the second `int` in the third row (`zippo[2][1]`)

The point of the baroque display of pointer notation is not that you can use it instead of the simpler `zippo[2][1]` but that, if you happen to have a pointer to a two-dimensional array and want to extract a value, you can use the simpler array notation rather than pointer notation.

Figure 10.5 provides another view of the relationships among array addresses, array contents, and pointers.

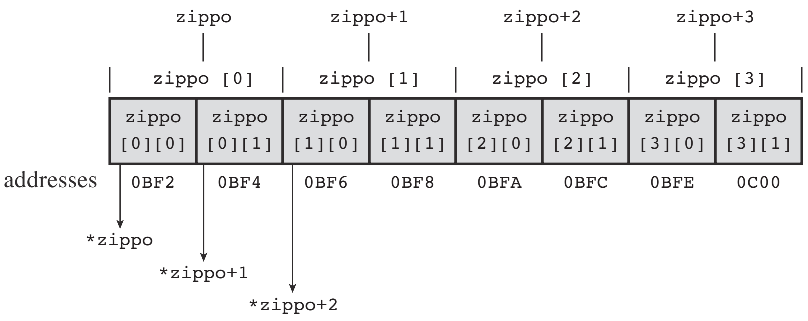

Figure 10.5 An array of arrays.

Pointers to Multidimensional Arrays

How would you declare a pointer variable `pz` that can point to a two-dimensional array such as `zippo`? Such a pointer could be used, for example, in writing a function to deal with `zippo`-like arrays. Will the type pointer-to-`int` suffice? No. That type is compatible with `zippo[0]`, which points to a single `int`. But `zippo` is the address of its first element, which is an array of two `int`s. Hence, `pz` must point to an array of two `int`s, not to a single `int`. Here is what you can do:

```
int (* pz)[2];  // pz points to an array of 2 ints
```

This statement says that `pz` is a pointer to an array of two `int`s. Why the parentheses? Well, `[]` has a higher precedence than `*`. Therefore, with a declaration such as

```
int * pax[2];  // pax is an array of two pointers-to-int
```

you apply the brackets first, making `pax` an array of two somethings. Next, you apply the `*`, making `pax` an array of two pointers. Finally, use the `int`, making `pax` an array of two pointers to `int`. This declaration creates *two* pointers to single `int`s, but the original version uses parentheses to apply the `*` first, creating *one* pointer to an array of two `int`s. Listing 10.16 shows how you can use such a pointer just like the original array.

Listing 10.16 **The `zippo2.c` Program**

```
/* zippo2.c --  zippo info via a pointer variable */
#include <stdio.h>
int main(void)
{
    int zippo[4][2] = { {2,4}, {6,8}, {1,3}, {5, 7} };
    int (*pz)[2];
    pz = zippo;

    printf("   pz = %p,    pz + 1 = %p\n",
               pz,         pz + 1);
    printf("pz[0] = %p, pz[0] + 1 = %p\n",
            pz[0],      pz[0] + 1);
    printf("  *pz = %p,   *pz + 1 = %p\n",
              *pz,        *pz + 1);
    printf("pz[0][0] = %d\n", pz[0][0]);
    printf("  *pz[0] = %d\n", *pz[0]);
    printf("    **pz = %d\n", **pz);
    printf("      pz[2][1] = %d\n", pz[2][1]);
    printf("*(*(pz+2) + 1) = %d\n", *(*(pz+2) + 1));

    return 0;
}
```

Here is the new output:

```
pz = 0x0064fd38,    pz + 1 = 0x0064fd40
pz[0] = 0x0064fd38, pz[0] + 1 = 0x0064fd3c
  *pz = 0x0064fd38,   *pz + 1 = 0x0064fd3c
pz[0][0] = 2
  *pz[0] = 2
    **pz = 2
      pz[2][1] = 3
*(*(pz+2) + 1) = 3
```

Again, you might get different addresses, but the relationships will be the same. As promised, you can use notation such as `pz[2][1]`, even though `pz` is a pointer, not an array name. More generally, you can represent individual elements by using array notation or pointer notation with either an array name or a pointer:

```
zippo[m][n] == *(*(zippo + m) + n)
pz[m][n] == *(*(pz + m) + n)
```

Pointer Compatibility

The rules for assigning one pointer to another are tighter than the rules for numeric types. For example, you can assign an `int` value to a `double` variable without using a type conversion, but you can't do the same for pointers to these two types:

```
int n = 5;
double x;
int * p1 = &n;
double * pd    = &x;
x = n;              // implicit type conversion
pd = p1;            // compile-time error
```

These restrictions extend to more complex types. Suppose we have the following declarations:

```
int * pt;
int (*pa)[3];
int ar1[2][3];
int ar2[3][2];
int **p2;       // a pointer to a pointer
```

Then we have the following:

```
pt = &ar1[0][0];  // both pointer-to-int
pt = ar1[0];      // both pointer-to-int
pt = ar1;         // not valid
pa = ar1;         // both pointer-to-int[3]
pa = ar2;         // not valid
p2 = &pt;         // both pointer-to-int *
*p2 = ar2[0];     // both pointer-to-int
p2 = ar2;         // not valid
```

Notice that the nonvalid assignments all involve two pointers that don't point to the same type. For example, `pt` points to a single `int`, but `ar1` points to an array of three `int`s. Similarly, `pa` points to an array of two `int`s, so it is compatible with `ar1`, but not with `ar2`, which points to an array of two `int`s.

The last two examples are somewhat tricky. The variable `p2` is a pointer-to-pointer-to-`int`, whereas `ar2` is a pointer-to-array-of-two-`int`s (or, more concisely, pointer-to-`int[2]`). So `p2` and `ar2` are of different types, and you can't assign `ar2` to `p2`. But `*p2` is type pointer-to-`int`, making it compatible with `ar2[0]`. Recall that `ar2[0]` is a pointer to its first element, `ar2[0][0]`, making `ar2[0]` type pointer-to-`int` also.

In general, multiple indirection is tricky. For instance, consider the next snippet of code:

```
int x = 20;
const int y = 23;
int * p1 = &x;
const int * p2 = &y;
const int ** pp2;
p1 = p2;   // not safe -- assigning const to non-const
p2 = p1;   // valid    -- assigning non-const to const
pp2 = &p1; // not safe -- assigning nested pointer types
```

As we saw earlier, assigning a `const` pointer to a non-`const` pointer is not safe, because you could use the new pointer to alter `const` data. While the code would compile, perhaps with a warning, the effect of executing the code is undefined. But assigning a non-`const` pointer to a `const` pointer is okay, provided that you're dealing with just one level of indirection:

```
p2 = p1;   // valid     -- assigning non-const to const
```

But such assignments no longer are safe when you go to two levels of indirection. For instance, you could do something like this:

```
const int **pp2;
int *p1;
const int n = 13;
pp2 = &p1; // allowed, but const qualifier disregarded
*pp2 = &n; // valid, both const, but sets p1 to point at n
*p1 = 10;  // valid, but tries to change const n
```

What happens? As mentioned before, the standard says the effect of altering const data using a non-`const` pointer is undefined. For instance, compiling a short program with this code using gcc in Terminal (OS X's access to the underlying Unix system) led to `n` ending up with the value 13, but using clang in the same environment led to a value of 10. Both compilers did warn about incompatible pointer types. You can, of course, ignore the warnings, but you'd best not rely upon the results of running the program.

C `const` and C++ `const`

C and C++ use `const` similarly, but not identically. One difference is that C++ allows using a `const` integer value to declare an array size and C is more restrictive. Another is that C++ has stricter rules about pointer assignments:

```
const int y;
const int * p2 = &y;
int * p1;
p1 = p2;   // error in C++, possible warning in C
```

In C++ you are not allowed to assign a `const` pointer to a non-`const` pointer. In C, you can make this assignment, but the behavior is undefined if you try to use `p1` to alter `y`.

Functions and Multidimensional Arrays

If you want to write functions that process two-dimensional arrays, you need to understand pointers well enough to make the proper declarations for function arguments. In the function body itself, you can usually get by with array notation.

Let's write a function to deal with two-dimensional arrays. One possibility is to use a `for` loop to apply a one-dimensional array function to each row of the two-dimensional array. That is, you could do something like the following:

```
int junk[3][4] = { {2,4,5,8}, {3,5,6,9}, {12,10,8,6} };
int i, j;
int total = 0;
for (i = 0; i < 3 ; i++)
    total += sum(junk[i], 4);  // junk[i] -- one-dimensional array
```

Remember, if `junk` is a two-dimensional array, `junk[i]` is a one-dimensional array, which you can visualize as being one row of the two-dimensional array. Here, the `sum()` function calculates the subtotal of each row of the two-dimensional array, and the `for` loop adds up these subtotals.

However, this approach loses track of the column-and-row information. In this application (summing all), that information is unimportant, but suppose each row represented a year and each column a month. Then you might want a function to, say, total up individual columns. In that case, the function should have the row and column information available. This can be accomplished by declaring the right kind of formal variable so that the function can pass the array properly. In this case, the array `junk` is an array of three arrays of four `int`s. As the earlier discussion pointed out, that means `junk` is a pointer to an array of four `int`s. You can declare a function parameter of this type like this:

```
void somefunction( int (* pt)[4] );
```

Alternatively, if (and only if) `pt` is a formal parameter to a function, you can declare it as follows:

```
void somefunction( int pt[][4] );
```

Note that the first set of brackets is empty. The empty brackets identify `pt` as being a pointer. Such a variable can then be used in the same way as `junk`. That is what we have done in the next example, shown in Listing 10.17. Notice that the listing exhibits three equivalent alternatives for the prototype syntax.

Listing 10.17 The `array2d.c` Program

```
// array2d.c -- functions for 2d arrays
#include <stdio.h>
#define ROWS 3
#define COLS 4
void sum_rows(int ar[][COLS], int rows);
void sum_cols(int [][COLS], int );    // ok to omit names
int sum2d(int (*ar)[COLS], int rows); // another syntax
int main(void)
{
     int junk[ROWS][COLS] = {
            {2,4,6,8},
            {3,5,7,9},
            {12,10,8,6}
     };

     sum_rows(junk, ROWS);
     sum_cols(junk, ROWS);
     printf("Sum of all elements = %d\n", sum2d(junk, ROWS));

     return 0;
}

void sum_rows(int ar[][COLS], int rows)
{
    int r;
    int c;
    int tot;

    for (r = 0; r < rows; r++)
    {
        tot = 0;
        for (c = 0; c < COLS; c++)
            tot += ar[r][c];
        printf("row %d: sum = %d\n", r, tot);
    }
}

void sum_cols(int ar[][COLS], int rows)
{
    int r;
    int c;
    int tot;

    for (c = 0; c < COLS; c++)
    {
        tot = 0;
        for (r = 0; r < rows; r++)
            tot += ar[r][c];
        printf("col %d: sum = %d\n", c, tot);
    }
}

int sum2d(int ar[][COLS], int rows)
{
    int r;
    int c;
    int tot = 0;

    for (r = 0; r < rows; r++)
        for (c = 0; c < COLS; c++)
            tot += ar[r][c];

    return tot;
}
```

Here is the output:

```
row 0: sum = 20
row 1: sum = 24
row 2: sum = 36
col 0: sum = 17
col 1: sum = 19
col 2: sum = 21
col 3: sum = 23
Sum of all elements = 80
```

The program in Listing 10.17 passes as arguments the name `junk`, which is a pointer to the first element, a subarray, and the symbolic constant `ROWS`, representing `3`, the number of rows. Each function then treats `ar` as an array of arrays of four `int`s. The number of columns is built in to the function, but the number of rows is left open. The same function will work with, say, a 12×4 array if 12 is passed as the number of rows. That's because `rows` is the number of elements; however, because each element is an array, or row, `rows` becomes the number of rows.

Note that `ar` is used in the same fashion as `junk` is used in `main()`. This is possible because `ar` and `junk` are the same type: pointer-to-array-of-four-`int`s.

Be aware that the following declaration will not work properly:

```
int sum2(int ar[][], int rows); // faulty declaration
```

Recall that the compiler converts array notation to pointer notation. This means, for example, that `ar[1]` will become `ar+1`. For the compiler to evaluate this, it needs to know the size object to which `ar` points. The declaration

```
int sum2(int ar[][4], int rows); // valid declaration
```

says that `ar` points to an array of four `int`s (hence, to an object 16 bytes long on our system), so `ar+1` means "add 16 bytes to the address." With the empty-bracket version, the compiler would not know what to do.

You can also include a size in the other bracket pair, as shown here, but the compiler ignores it:

```
int sum2(int ar[3][4], int rows); // valid declaration, 3 ignored
```

This is convenient for those who use `typedef`s (mentioned in Chapter 5, "Operators, Expressions, and Statements," and discussed in Chapter 14, "Structures and Other Data Forms"):

```
typedef int arr4[4];              // arr4 array of 4 int
typedef arr4 arr3x4[3];           // arr3x4 array of 3 arr4
int sum2(arr3x4 ar, int rows);    // same as next declaration
int sum2(int ar[3][4], int rows); // same as next declaration
int sum2(int ar[][4], int rows);  // standard form
```

In general, to declare a pointer corresponding to an *N*-dimensional array, you must supply values for all but the leftmost set of brackets:

```
int sum4d(int ar[][12][20][30], int rows);
```

That's because the first set of brackets indicates a pointer, whereas the rest of the brackets describe the type of data object being pointed to, as the following equivalent prototype illustrates:

```
int sum4d(int (*ar)[12][20][30], int rows);  // ar a pointer
```

Here, `ar` points to a 12×20×30 array of `int`s.

Variable-Length Arrays (VLAs)

You might have noticed an oddity about functions dealing with two-dimensional arrays: You can describe the number of rows with a function parameter, but the number of columns is built in to the function. For example, look at this definition:

```
#define COLS 4
int sum2d(int ar[][COLS], int rows)
{
    int r;
    int c;
    int tot = 0;

    for (r = 0; r < rows; r++)
        for (c = 0; c < COLS; c++)
            tot += ar[r][c];
    return tot;
}
```

Next, suppose the following arrays have been declared:

```
int array1[5][4];
int array2[100][4];
int array3[2][4];
```

You can use the `sum2d()` function with any of these arrays:

```
tot = sum2d(array1, 5);   // sum a 5 x 4 array
tot = sum2d(array2, 100); // sum a 100 x 4 array
tot = sum2d(array3, 2);   // sum a 2 x 4 array
```

That's because the number of rows is passed to the `rows` parameter, a variable. But if you wanted to sum a 6×5 array, you would need to use a new function, one for which `COLS` is defined to be `5`. This behavior is a result of the fact that you have to use constants for array dimensions; therefore, you can't replace `COLS` with a variable.

If you really want to create a single function that will work with any size two-dimensional array, you can, but it's awkward to do. (You have to pass the array as a one-dimensional array and have the function calculate where each row starts.) Furthermore, this technique doesn't mesh smoothly with FORTRAN subroutines, which do allow one to specify both dimensions in a function call. FORTRAN might be a hoary old programming language, but over the decades experts in the field of numerical calculations have developed many useful computational libraries in FORTRAN. C is being positioned to take over from FORTRAN, so the ability to convert FORTRAN libraries with a minimum of fuss is useful.

This need was the primary impulse for C99 introducing variable-length arrays, which allow you to use variables when dimensioning an array. For example, you can do this:

```
int quarters = 4;
int regions = 5;
double sales[regions][quarters]; // a VLA
```

As mentioned earlier, VLAs have some restrictions. They need to have the automatic storage class, which means they are declared either in a function without using the `static` or `extern` storage class modifiers (Chapter 12) or as function parameters. Also, you can't initialize them in a declaration. Finally, under C11, VLAs are an optional feature rather than a mandatory feature, as they were under C99.

Note VLAs Do Not Change Size

The term *variable* in variable-length array does not mean that you can modify the length of the array after you create it. Once created, a VLA keeps the same size. What the term *variable* does mean is that you can use a variable when specifying the array dimensions when first creating the array.

Because VLAs are a new addition to the language, support for them is incomplete at the present. Let's look at a simple example that shows how to write a function that will sum the contents of any two-dimensional array of `int`s.

First, here's how to declare a function with a two-dimensional VLA argument:

```
int sum2d(int rows, int cols, int ar[rows][cols]);  // ar a VLA
```

Note that the first two parameters (`rows` and `cols`) are used as dimensions for declaring the array parameter `ar`. Because the `ar` declaration uses `rows` and `cols`, they have to be declared before `ar` in the parameter list. Therefore, the following prototype is in error:

```
int sum2d(int ar[rows][cols], int rows, int cols); // invalid order
```

The C99/C11 standard says you can omit names from the prototype; but in that case, you need to replace the omitted dimensions with asterisks:

```
int sum2d(int, int, int ar[*][*]);  // ar a VLA, names omitted
```

Second, here's how to define the function:

```
int sum2d(int rows, int cols, int ar[rows][cols])
{
    int r;
    int c;
    int tot = 0;

    for (r = 0; r < rows; r++)
        for (c = 0; c < cols; c++)
            tot += ar[r][c];
    return tot;
}
```

Aside from the new function header, the only difference from the classic C version of this function (Listing 10.17) is that the constant `COLS` has been replaced with the variable `cols`. The presence of the variable length array in the function header is what makes this change possible. Also, having variables that represent both the number of rows and columns lets us use the new `sum2d()` with any size of two-dimensional array of `int`s. Listing 10.18 illustrates this point. However, it does require a C compiler that implements the VLA feature. It also demonstrates that this VLA-based function can be used with either traditional C arrays or with a variable-length array.

Listing 10.18 **The `vararr2d.c` Program**

```
//vararr2d.c -- functions using VLAs
#include <stdio.h>
#define ROWS 3
#define COLS 4
int sum2d(int rows, int cols, int ar[rows][cols]);
int main(void)
{
     int i, j;
     int rs = 3;
     int cs = 10;
     int junk[ROWS][COLS] = {
            {2,4,6,8},
            {3,5,7,9},
            {12,10,8,6}
     };

     int morejunk[ROWS-1][COLS+2] = {
            {20,30,40,50,60,70},
            {5,6,7,8,9,10}
     };

     int varr[rs][cs];  // VLA
     for (i = 0; i < rs; i++)
         for (j = 0; j < cs; j++)
             varr[i][j] = i * j + j;

     printf("3x5 array\n");
     printf("Sum of all elements = %d\n",
             sum2d(ROWS, COLS, junk));

     printf("2x6 array\n");
     printf("Sum of all elements = %d\n",
             sum2d(ROWS-1, COLS+2, morejunk));

     printf("3x10 VLA\n");
     printf("Sum of all elements = %d\n",
             sum2d(rs, cs, varr));

     return 0;
}

// function with a VLA parameter
int sum2d(int rows, int cols, int ar[rows][cols])
{
    int r;
    int c;
    int tot = 0;

    for (r = 0; r < rows; r++)
        for (c = 0; c < cols; c++)
            tot += ar[r][c];

    return tot;
}
```

Here is the output:

```
3x5 array
Sum of all elements = 80
2x6 array
Sum of all elements = 315
3x10 VLA
Sum of all elements = 270
```

One point to note is that a VLA declaration in a function definition parameter list doesn't actually create an array. Just as with the old syntax, the VLA name really is a pointer. This means a function with a VLA parameter actually works with the data in the original array, and therefore has the ability to modify the array passed as an argument. The following snippet points out when a pointer is declared and when an actual array is declared:

```
    int thing[10][6];
    twoset(10,6,thing);
    ...
}
void twoset (int n, int m, int ar[n][m])  // ar a pointer to
                                          // an array of m ints
{
    int temp[n][m];     // temp an n x m array of int
    temp[0][0] = 2;     // set an element of temp to 2
    ar[0][0] = 2;       // set thing[0][0] to 2
}
```

When `twoset()` is called as shown, `ar` becomes a pointer to `thing[0]`, and `temp` is created as a 10×6 array. Because both `ar` and `thing` are pointers to `thing[0]`, `ar[0][0]` accesses the same data location as `thing[0][0]`.

Variable-length arrays also allow for dynamic memory allocation. This means you can specify the size of the array while the program is running. Regular C arrays have static memory allocation, meaning the size of the array is determined at compile time. That's because the array sizes, being constants, are known to the compiler. Chapter 12 looks at dynamic memory allocation.

`const` and Array Sizes

Can you use a `const` symbolic constant when declaring an array?

```
const int SZ = 80;
...
double ar[SZ];    // permitted?
```

For C90, the answer is no (probably). The size has to be given by an integer constant expression, which can be a combination of integer constants, such as `20`, `sizeof` expressions, and a few other things, none of which are `const`. An implementation can expand the range of what is considered an integer constant expression, so it could permit using `const`, but the code wouldn't be portable.

For C99/C11, the answer is yes, if the array could otherwise be a VLA. So the definition would have to be for an automatic storage class array declared inside a block.

Compound Literals

Suppose you want to pass a value to a function with an `int` parameter; you can pass an `int` variable, but you also can pass an `int` constant, such as `5`. Before C99, the situation for a function with an array argument was different; you could pass an array, but there was no equivalent to an array constant. C99 changed that with the addition of *compound literals*. Literals are constants that aren't symbolic. For example, `5` is a type `int` literal, `81.3` is a type `double` literal, `'Y'` is a type `char` literal, and `"elephant"` is a string literal. The committee that developed the C99 standard concluded that it would be convenient to have compound literals that could represent the contents of arrays and of structures.

For arrays, a compound literal looks like an array initialization list preceded by a type name that is enclosed in parentheses. For example, here's an ordinary array declaration:

```
int diva[2] = {10, 20};
```

And here's a compound literal that creates a nameless array containing the same two `int` values:

```
(int [2]){10, 20}     // a compound literal
```

Note that the type name is what you would get if you removed `diva` from the earlier declaration, leaving `int [2]` behind.

Just as you can leave out the array size if you initialize a named array, you can omit it from a compound literal, and the compiler will count how many elements are present:

```
(int []){50, 20, 90}     // a compound literal with 3 elements
```

Because these compound literals are nameless, you can't just create them in one statement and then use them later. Instead, you have to use them somehow when you make them. One way is to use a pointer to keep track of the location. That is, you can do something like this:

```
int * pt1;
pt1 = (int [2]) {10, 20};
```

Note that this literal constant is identified as an array of `int`s. Like the name of an array, this translates to the address of the first element, so it can be assigned to a pointer-to-`int`. You then can use the pointer later. For example, `*pt1` would be 10 in this case, and `pt1[1]` would be 20.

Another thing you could do with a compound literal is pass it as an actual argument to a function with a matching formal parameter:

```
int sum(const int ar[], int n);
...
int total3;
total3 = sum((int []){4,4,4,5,5,5}, 6);
```

Here, the first argument is a six-element array of `int`s that acts like the address of the first element, just as an array name does. This kind of use, in which you pass information to a function without having to create an array first, is a typical use for compound literals.

You can extend the technique to two-dimensional arrays, and beyond. Here, for example, is how to create a two-dimensional array of `int`s and store the address:

```
int (*pt2)[4];     // declare a pointer to an array of 4-int arrays
pt2 = (int [2][4]) { {1,2,3,-9}, {4,5,6,-8} };
```

Here, the type is `int [2][4]`, a 2×4 array of `int`s.

Listing 10.19 incorporates these examples into a complete program.

Listing 10.19 The `flc.c` Program

```
// flc.c -- funny-looking constants
#include <stdio.h>
#define COLS 4
int sum2d(const int ar[][COLS], int rows);
int sum(const int ar[], int n);
int main(void)
{
     int total1, total2, total3;
     int * pt1;
     int (*pt2)[COLS];

     pt1 = (int [2]) {10, 20};
     pt2 = (int [2][COLS]) { {1,2,3,-9}, {4,5,6,-8} };

     total1 = sum(pt1, 2);
     total2 = sum2d(pt2, 2);
     total3 = sum((int []){4,4,4,5,5,5}, 6);
     printf("total1 = %d\n", total1);
     printf("total2 = %d\n", total2);
     printf("total3 = %d\n", total3);

     return 0;
}

int sum(const int ar[], int n)
{
    int i;
    int total = 0;

    for( i = 0; i < n; i++)
        total += ar[i];

    return total;
}

int sum2d(const int ar[][COLS], int rows)
{
    int r;
    int c;
    int tot = 0;
    for (r = 0; r < rows; r++)
        for (c = 0; c < COLS; c++)
            tot += ar[r][c];

    return tot;
}
```

You'll need a compiler that accepts this C99 addition (not all do). Here is the output:

```
total1 = 30
total2 = 4
total3 = 27
```

Keep in mind that a compound literal is a means for providing values that are needed only temporarily. It has block scope, a concept covered in Chapter 12. That means its existence is not guaranteed once program execution leaves the block in which the compound literal is defined, that is, the innermost pair of braces containing the definition.

Key Concepts

When you need to store many items, all of the same kind, an array might be the answer. C refers to arrays as *derived types* because they are built on other types. That is, you don't simply declare an array. Instead, you declare an array-of-`int` or an array-of-`float`, or an array of some other type. That other type can itself be an array type, in which case, you get an array of arrays, or a two-dimensional array.

It's often advantageous to write functions to process arrays; that helps modularize a program by locating specific tasks in specific functions. It's important to realize that when you use an array name as an actual argument, you're not passing the entire array to the function; you are just passing the address of the array (hence, the corresponding formal parameter is a pointer). To process the array, the function has to know where the array is and how many elements the array has. The array address provides the "where"; the "how many" either has to be built in to the function or be passed as a separate argument. The second approach is more general so that the same function can work with arrays of different sizes.

The connection between arrays and pointers is an intimate one, and you can often represent the same operation using either array notation or pointer notation. It's this connection that allows you to use array notation in an array-processing function even though the formal parameter is a pointer, not an array.

You must specify the size of a conventional C array with a constant expression, so the size is determined at compile time. C99/C11 offers the variable-length array alternative for which the size specifier can be a variable. This allows you to delay specifying the size of a VLA until the program is running.

Summary

An *array* is a set of elements that all have the same data type. Array elements are stored sequentially in memory and are accessed by using an integer index (or *offset*). In C, the first element of an array has an index of `0`, so the final element in an array of `n` elements has an index of `n - 1`. It's your responsibility to use array indices that are valid for the array, because neither the compiler nor the running program need check for this.

To declare a simple *one-dimensional* array, use this form:

```
type name[size];
```

Here, *`type`* is the data type for each and every element, *`name`* is the name of the array, and *`size`* is the number of elements. Traditionally, C has required that *`size`* be a constant integer expression. C99/C11 allows you to use a nonconstant integer expression; in that case, the array is termed a variable-length array.

C interprets the name of an array to be the address of the first element of the array. In other terms, the name of an array is equivalent to a pointer to the first element. In general, arrays and pointers are closely connected. If `ar` is an array, then the expressions `ar[i]` and `*(ar + i)` are equivalent.

C does not enable entire arrays to be passed as function arguments, but you can pass the address of an array. The function can then use this address to manipulate the original array. If the intent of the function is not to modify the original array, you should use the `const` keyword when declaring the formal parameter representing the array. You can use either array notation or pointer notation in the called function. In either case, you're actually using a pointer variable.

Adding an integer to a pointer or incrementing a pointer changes the value of the pointer by the number of bytes of the object being pointed to. That is, if `pd` points to an 8-byte `double` value in an array, adding 1 to `pd` increases its value by 8 so that it will point to the next element of the array.

Two-dimensional arrays represent an array of arrays. For instance, the declaration

```
double sales[5][12];
```

creates an array called `sales` having five elements, each of which is an array of 12 `doubles`. The first of these one-dimensional arrays can be referred to as `sales[0]`, the second as `sales[1]`, and so on, with each being an array of 12 `doubles`. Use a second index to access a particular element in these arrays. For example, `sales[2][5]` is the sixth element of `sales[2]`, and `sales[2]` is the third element of `sales`.

The traditional C method for passing a multidimensional array to a function is to pass the array name, which is an address, to a suitably typed pointer parameter. The declaration for this pointer should specify all the dimensions of the array aside from the first; the dimension of the first parameter typically is passed as a second argument. For example, to process the previously mentioned `sales` array, the function prototype and function call would look like this:

```
void display(double ar[][12], int rows);
...
display(sales, 5);
```

Variable-length arrays provide a second syntax in which both array dimensions are passed as arguments. In this case, the function prototype and function call would look like this:

```
void display(int rows, int cols, double ar[rows][cols]);
...
display(5, 12, sales);
```

We've used `int` arrays and `double` arrays in this discussion, but the same concepts apply to other types. Character strings, however, have many special rules. This stems from the fact that the terminal null character in a string provides a way for functions to detect the end of a string without being passed a size. We will look at character strings in detail in Chapter 11, "Character Strings and String Functions."

Review Questions

You'll find answers to the review questions in Appendix A, "Answers to the Review Questions."

1. What will this program print?

```
#include <stdio.h>
int main(void)
{
  int ref[] = {8, 4, 0, 2};
  int *ptr;
  int index;

  for (index = 0, ptr = ref; index < 4; index++, ptr++)
     printf("%d %d\n", ref[index], *ptr);
  return 0;
}
```

2. In question 1, how many elements does `ref` have?

3. In question 1, `ref` is the address of what? What about `ref + 1`? What does `++ref` point to?

4. What is the value of `*ptr` and of `*(ptr + 2)` in each case?

   a.

```
int *ptr;
int torf[2][2] = {12, 14, 16};
ptr = torf[0];
```

b.

```
int * ptr;
int fort[2][2] = { {12}, {14,16} };
ptr = fort[0];
```

5. What is the value of `**ptr` and of `**(ptr + 1)` in each case?

   a.

   ```
   int (*ptr)[2];
   int torf[2][2] = {12, 14, 16};
   ptr = torf;
   ```

   b.

   ```
   int (*ptr)[2];
   int fort[2][2] = { {12}, {14,16} };
   ptr = fort;
   ```

6. Suppose you have the following declaration:

   `int grid[30][100];`.

   a. Express the address of `grid[22][56]` one way.

   b. Express the address of `grid[22][0]` two ways.

   c. Express the address of `grid[0][0]` three ways.

7. Create an appropriate declaration for each of the following variables:

   a. `digits` is an array of 10 `int`s.

   b. `rates` is an array of six `float`s.

   c. `mat` is an array of three arrays of five integers.

   d. `psa` is an array of 20 pointers to `char`.

   e. `pstr` is a pointer to an array of 20 `char`s.

8.

   a. Declare an array of six `int`s and initialize it to the values `1`, `2`, `4`, `8`, `16`, and `32`.

   b. Use array notation to represent the third element (the one with the value `4`) of the array in part a.

   c. Assuming C99/C11 rules are in effect, declare an array of 100 `int`s and initialize it so that the last element is `-1`; don't worry about the other elements.

   d. Assuming C99/C11 rules are in effect, declare an array of 100 `int`s and initialize it so that elements 5, 10, 11, 12, and 3 are `101`; don't worry about the other elements.

9. What is the index range for a 10-element array?

10. Suppose you have these declarations:

```
float rootbeer[10], things[10][5], *pf, value = 2.2;
int i = 3;
```

Identify each of the following statements as valid or invalid:

   a. `rootbeer[2] = value;`
   b. `scanf("%f", &rootbeer );`
   c. `rootbeer = value;`
   d. `printf("%f", rootbeer);`
   e. `things[4][4] = rootbeer[3];`
   f. `things[5] = rootbeer;`
   g. `pf = value;`
   h. `pf = rootbeer;`

11. Declare an 800×600 array of `int`.

12. Here are three array declarations:

```
double trots[20];
short clops[10][30];
long shots[5][10][15];
```

   a. Show a function prototype and a function call for a traditional `void` function that processes `trots` and also for a C function using a VLA.
   b. Show a function prototype and a function call for a traditional `void` function that processes `clops` and also for a C function using a VLA.
   c. Show a function prototype and a function call for a traditional `void` function that processes `shots` and also for a C function using a VLA.

13. Here are two function prototypes:

```
void show(const double ar[], int n);    // n is number of elements
void show2(const double ar2[][3], int n);  // n is number of rows
```

   a. Show a function call that passes a compound literal containing the values `8`, `3`, `9`, and `2` to the `show()` function.
   b. Show a function call that passes a compound literal containing the values `8`, `3`, and `9` as the first row and the values `5`, `4`, and `1` as the second row to the `show2()` function.

Programming Exercises

1. Modify the rain program in Listing 10.7 so that it does the calculations using pointers instead of subscripts. (You still have to declare and initialize the array.)

2. Write a program that initializes an array-of-`double` and then copies the contents of the array into three other arrays. (All four arrays should be declared in the main program.) To make the first copy, use a function with array notation. To make the second copy, use a function with pointer notation and pointer incrementing. Have the first two functions take as arguments the name of the target array, the name of the source array, and the number of elements to be copied. Have the third function take as arguments the name of the target, the name of the source, and a pointer to the element following the last element of the source. That is, the function calls would look like this, given the following declarations:

   ```
   double source[5] = {1.1, 2.2, 3.3, 4.4, 5.5};
   double target1[5];
   double target2[5];
   double target3[5];
   copy_arr(target1, source, 5);
   copy_ptr(target2, source, 5);

   copy_ptrs(target3, source, source + 5);
   ```

3. Write a function that returns the largest value stored in an array-of-`int`. Test the function in a simple program.

4. Write a function that returns the index of the largest value stored in an array-of-`double`. Test the function in a simple program.

5. Write a function that returns the difference between the largest and smallest elements of an array-of-`double`. Test the function in a simple program.

6. Write a function that reverses the contents of an array of `double` and test it in a simple program.

7. Write a program that initializes a two-dimensional array-of-`double` and uses one of the copy functions from exercise 2 to copy it to a second two-dimensional array. (Because a two-dimensional array is an array of arrays, a one-dimensional copy function can be used with each subarray.)

8. Use a copy function from Programming Exercise 2 to copy the third through fifth elements of a seven-element array into a three-element array. The function itself need

not be altered; just choose the right actual arguments. (The actual arguments need not be an array name and array size. They only have to be the address of an array element and a number of elements to be processed.)

9. Write a program that initializes a two-dimensional 3×5 array-of-`double` and uses a VLA-based function to copy it to a second two-dimensional array. Also provide a VLA-based function to display the contents of the two arrays. The two functions should be capable, in general, of processing arbitrary N×M arrays. (If you don't have access to a VLA-capable compiler, use the traditional C approach of functions that can process an N×5 array).

10. Write a function that sets each element in an array to the sum of the corresponding elements in two other arrays. That is, if array 1 has the values `2`, `4`, `5`, and `8` and array 2 has the values `1`, `0`, `4`, and `6`, the function assigns array 3 the values `3`, `4`, `9`, and `14`. The function should take three array names and an array size as arguments. Test the function in a simple program.

11. Write a program that declares a 3×5 array of `int` and initializes it to some values of your choice. Have the program print the values, double all the values, and then display the new values. Write a function to do the displaying and a second function to do the doubling. Have the functions take the array name and the number of rows as arguments.

12. Rewrite the rain program in Listing 10.7 so that the main tasks are performed by functions instead of in `main()`.

13. Write a program that prompts the user to enter three sets of five `double` numbers each. (You may assume the user responds correctly and doesn't enter non-numeric data.) The program should accomplish all of the following:

    a. Store the information in a 3×5 array.

    b. Compute the average of each set of five values.

    c. Compute the average of all the values.

    d. Determine the largest value of the 15 values.

    e. Report the results.

    Each major task should be handled by a separate function using the traditional C approach to handling arrays. Accomplish task "b" by using a function that computes and returns the average of a one-dimensional array; use a loop to call this function three times. The other tasks should take the entire array as an argument, and the functions performing tasks "c" and "d" should return the answer to the calling program.

14. Do Programming Exercise 13, but use variable-length array function parameters.