8 Arrays

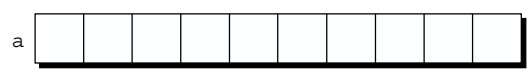
If a program manipulates a large amount of data,
it does so in a small number of ways.

So far, the only variables we've seen are ***scalar:*** capable of holding a single data item. C also supports ***aggregate*** variables, which can store collections of values. There are two kinds of aggregates in C: arrays and structures. This chapter shows how to declare and use arrays, both one-dimensional (Section 8.1) and multidimensional (Section 8.2). Section 8.3 covers C99's variable-length arrays. The focus of the chapter is on one-dimensional arrays, which play a much bigger role in C than do multidimensional arrays. Later chapters (Chapter 12 in particular) provide additional information about arrays; Chapter 16 covers structures.

8.1 One-Dimensional Arrays

An ***array*** is a data structure containing a number of data values, all of which have the same type. These values, known as ***elements***, can be individually selected by their position within the array.

The simplest kind of array has just one dimension. The elements of a one-dimensional array are conceptually arranged one after another in a single row (or column, if you prefer). Here's how we might visualize a one-dimensional array named `a`:



To declare an array, we must specify the *type* of the array's elements and the *number* of elements. For example, to declare that the array `a` has 10 elements of type `int`, we would write

```
int a[10];
```

constant expressions ➤5.3

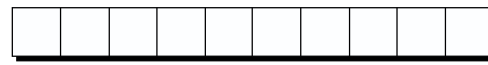
The elements of an array may be of any type; the length of the array can be specified by any (integer) constant expression. Since array lengths may need to be adjusted when the program is later changed, using a macro to define the length of an array is an excellent practice:

```
#define N 10
…
int a[N];
```

Array Subscripting

To access a particular element of an array, we write the array name followed by an integer value in square brackets (this is referred to as ***subscripting*** or ***indexing*** the array). Array elements are always numbered starting from 0, so the elements of an array of length *n* are indexed from 0 to *n* – 1. For example, if `a` is an array with 10 elements, they're designated by `a[0]`, `a[1]`, …, `a[9]`, as the following figure shows:

Q&A



lvalues ➤4.2

Expressions of the form `a[i]` are lvalues, so they can be used in the same way as ordinary variables:

```
a[0] = 1;
printf("%d\n", a[5]);
++a[i];
```

In general, if an array contains elements of type *T*, then each element of the array is treated as if it were a variable of type *T*. In this example, the elements `a[0]`, `a[5]`, and `a[i]` behave like `int` variables.

Arrays and `for` loops go hand-in-hand. Many programs contain `for` loops whose job is to perform some operation on every element in an array. Here are a few examples of typical operations on an array `a` of length `N`:

idiom
```
for (i = 0; i < N; i++)
  a[i] = 0;               /* clears a */
```

idiom
```
for (i = 0; i < N; i++)
  scanf("%d", &a[i]);     /* reads data into a */
```

idiom
```
for (i = 0; i < N; i++)
  sum += a[i];            /* sums the elements of a */
```

Notice that we must use the `&` symbol when calling `scanf` to read an array element, just as we would with an ordinary variable.

C doesn't require that subscript bounds be checked; if a subscript goes out of range, the program's behavior is undefined. One cause of a subscript going out of bounds: forgetting that an array with *n* elements is indexed from 0 to *n* – 1, not 1 to *n*. (As one of my professors liked to say, "In this business, you're always off by one." He was right, of course.) The following example illustrates a bizarre effect that can be caused by this common blunder:

```
int a[10], i;

for (i = 1; i <= 10; i++)
  a[i] = 0;
```

With some compilers, this innocent-looking `for` statement causes an infinite loop! When `i` reaches 10, the program stores 0 into `a[10]`. But `a[10]` doesn't exist, so 0 goes into memory immediately after `a[9]`. If the variable `i` happens to follow `a[9]` in memory—as might be the case—then `i` will be reset to 0, causing the loop to start over.

An array subscript may be any integer expression:

```
a[i+j*10] = 0;
```

The expression can even have side effects:

```
i = 0;
while (i < N)
  a[i++] = 0;
```

Let's trace this code. After `i` is set to 0, the `while` statement checks whether `i` is less than `N`. If it is, 0 is assigned to `a[0]`, `i` is incremented, and the loop repeats. Note that `a[++i]` wouldn't be right, because 0 would be assigned to `a[1]` during the first loop iteration.

Be careful when an array subscript has a side effect. For example, the following loop—which is supposed to copy the elements of the array `b` into the array `a`—may not work properly:

```
i = 0;
while (i < N)
  a[i] = b[i++];
```

The expression `a[i] = b[i++]` accesses the value of `i` and also modifies `i` elsewhere in the expression, which—as we saw in Section 4.4—causes undefined behavior. Of course, we can easily avoid the problem by removing the increment from the subscript:

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

PROGRAM

Reversing a Series of Numbers

Our first array program prompts the user to enter a series of numbers, then writes the numbers in reverse order:

```
Enter 10 numbers: 34 82 49 102 7 94 23 11 50 31
In reverse order: 31 50 11 23 94 7 102 49 82 34
```

Our strategy will be to store the numbers in an array as they're read, then go through the array backwards, printing the elements one by one. In other words, we won't actually reverse the elements in the array, but we'll make the user think we did.

reverse.c

```
/* Reverses a series of numbers */

#include <stdio.h>

#define N 10

int main(void)
{
  int a[N], i;

  printf("Enter %d numbers: ", N);
  for (i = 0; i < N; i++)
    scanf("%d", &a[i]);

  printf("In reverse order:");
  for (i = N - 1; i >= 0; i--)
    printf(" %d", a[i]);
  printf("\n");

  return 0;
}
```

This program shows just how useful macros can be in conjunction with arrays. The macro `N` is used four times in the program: in the declaration of `a`, in the `printf` that displays a prompt, and in both `for` loops. Should we later decide to change the size of the array, we need only edit the definition of `N` and recompile the program. Nothing else will need to be altered; even the prompt will still be correct.

Array Initialization

An array, like any other variable, can be given an initial value at the time it's declared. The rules are somewhat tricky, though, so we'll cover some of them now and save others until later.

initializers ➤ 18.5

The most common form of ***array initializer*** is a list of constant expressions enclosed in braces and separated by commas:

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

If the initializer is *shorter* than the array, the remaining elements of the array are given the value 0:

```
int a[10] = {1, 2, 3, 4, 5, 6};
  /* initial value of a is {1, 2, 3, 4, 5, 6, 0, 0, 0, 0} */
```

Using this feature, we can easily initialize an array to all zeros:

```
int a[10] = {0};
  /* initial value of a is {0, 0, 0, 0, 0, 0, 0, 0, 0, 0} */
```

It's illegal for an initializer to be completely empty, so we've put a single `0` inside the braces. It's also illegal for an initializer to be *longer* than the array it initializes.

If an initializer is present, the length of the array may be omitted:

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

The compiler uses the length of the initializer to determine how long the array is. The array still has a fixed number of elements (10, in this example), just as if we had specified the length explicitly.

C99 Designated Initializers

It's often the case that relatively few elements of an array need to be initialized explicitly; the other elements can be given default values. Consider the following example:

```
int a[15] = {0, 0, 29, 0, 0, 0, 0, 0, 0, 7, 0, 0, 0, 0, 48};
```

We want element 2 of the array to be 29, element 9 to be 7, and element 14 to be 48, but the other values are just zero. For a large array, writing an initializer in this fashion is tedious and error-prone (what if there were 200 zeros between two of the nonzero values?).

C99's ***designated initializers*** can be used to solve this problem. Here's how we could redo the previous example using a designated initializer:

```
int a[15] = {[2] = 29, [9] = 7, [14] = 48};
```

Each number in brackets is said to be a ***designator.***

Besides being shorter and easier to read (at least for some arrays), designated initializers have another advantage: the order in which the elements are listed no longer matters. Thus, our previous example could also be written in the following way:

```
int a[15] = {[14] = 48, [9] = 7, [2] = 29};
```

Designators must be integer constant expressions. If the array being initialized has length n, each designator must be between 0 and $n - 1$. However, if the length of the array is omitted, a designator can be any nonnegative integer. In the latter case, the compiler will deduce the length of the array from the largest designator.

In the following example, the fact that 23 appears as a designator will force the array to have length 24:

```
int b[] = {[5] = 10, [23] = 13, [11] = 36, [15] = 29};
```

An initializer may use both the older (element-by-element) technique and the newer (designated) technique:

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

Q&A This initializer specifies that the array's first three elements will be 5, 1, and 9. Element 4 will have the value 3. The two elements after element 4 will be 7 and 2. Finally, element 8 will have the value 6. All elements for which no value is specified will default to zero.

PROGRAM Checking a Number for Repeated Digits

Our next program checks whether any of the digits in a number appear more than once. After the user enters a number, the program prints either `Repeated digit` or `No repeated digit`:

```
Enter a number: 28212
Repeated digit
```

The number 28212 has a repeated digit (2); a number like 9357 doesn't.

The program uses an array of Boolean values to keep track of which digits appear in a number. The array, named `digit_seen`, is indexed from 0 to 9 to correspond to the 10 possible digits. Initially, every element of the array is false. (The initializer for `digit_seen` is `{false}`, which only initializes the first element of the array. However, the compiler will automatically make the remaining elements zero, which is equivalent to false.)

When given a number `n`, the program examines `n`'s digits one at a time, storing each into the `digit` variable and then using it as an index into `digit_seen`. If `digit_seen[digit]` is true, then `digit` appears at least twice in `n`. On the other hand, if `digit_seen[digit]` is false, then `digit` has not been seen before, so the program sets `digit_seen[digit]` to `true` and keeps going.

repdigit.c

```
/* Checks numbers for repeated digits */

#include <stdbool.h>   /* C99 only */
#include <stdio.h>

int main(void)
{
  bool digit_seen[10] = {false};
  int digit;
  long n;

  printf("Enter a number: ");
  scanf("%ld", &n);
```

```
  while (n > 0) {
    digit = n % 10;
    if (digit_seen[digit])
      break;
    digit_seen[digit] = true;
    n /= 10;
  }

  if (n > 0)
    printf("Repeated digit\n");
  else
    printf("No repeated digit\n");

  return 0;
}
```

C99

<stdbool.h> header ➤21.5

This program uses the names `bool`, `true`, and `false`, which are defined in C99's `<stdbool.h>` header. If your compiler doesn't support this header, you'll need to define these names yourself. One way to do so is to put the following lines above the `main` function:

```
#define true 1
#define false 0
typedef int bool;
```

Notice that `n` has type `long`, allowing the user to enter numbers up to 2,147,483,647 (or more, on some machines).

Using the `sizeof` Operator with Arrays

The `sizeof` operator can determine the size of an array (in bytes). If `a` is an array of 10 integers, then `sizeof(a)` is typically 40 (assuming that each integer requires four bytes).

We can also use `sizeof` to measure the size of an array element, such as `a[0]`. Dividing the array size by the element size gives the length of the array:

```
sizeof(a) / sizeof(a[0])
```

Some programmers use this expression when the length of the array is needed. To clear the array `a`, for example, we could write

```
for (i = 0; i < sizeof(a) / sizeof(a[0]); i++)
  a[i] = 0;
```

With this technique, the loop doesn't have to be modified if the array length should change at a later date. Using a macro to represent the array length has the same advantage, of course, but the `sizeof` technique is slightly better, since there's no macro name to remember (and possibly get wrong).

One minor annoyance is that some compilers produce a warning message for the expression `i < sizeof(a) / sizeof(a[0])`. The variable `i` probably has

type `int` (a signed type), whereas `sizeof` produces a value of type `size_t` (an unsigned type). We know from Section 7.4 that comparing a signed integer with an unsigned integer is a dangerous practice, although in this case it's safe because both `i` and `sizeof(a) / sizeof(a[0])` have nonnegative values. To avoid a warning, we can add a cast that converts `sizeof(a) / sizeof(a[0])` to a signed integer:

```
for (i = 0; i < (int) (sizeof(a) / sizeof(a[0])); i++)
  a[i] = 0;
```

Writing `(int) (sizeof(a) / sizeof(a[0]))` is a bit unwieldy; defining a macro that represents it is often helpful:

```
#define SIZE ((int) (sizeof(a) / sizeof(a[0])))

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

If we're back to using a macro, though, what's the advantage of `sizeof`? We'll answer that question in a later chapter (the trick is to add a parameter to the macro).

parameterized macros ➤ 14.3

PROGRAM Computing Interest

Our next program prints a table showing the value of $100 invested at different rates of interest over a period of years. The user will enter an interest rate and the number of years the money will be invested. The table will show the value of the money at one-year intervals—at that interest rate and the next four higher rates—assuming that interest is compounded once a year. Here's what a session with the program will look like:

```
Enter interest rate: 6
Enter number of years: 5

Years      6%      7%      8%      9%     10%
  1     106.00  107.00  108.00  109.00  110.00
  2     112.36  114.49  116.64  118.81  121.00
  3     119.10  122.50  125.97  129.50  133.10
  4     126.25  131.08  136.05  141.16  146.41
  5     133.82  140.26  146.93  153.86  161.05
```

Clearly, we can use a `for` statement to print the first row. The second row is a little trickier, since its values depend on the numbers in the first row. Our solution is to store the first row in an array as it's computed, then use the values in the array to compute the second row. Of course, this process can be repeated for the third and later rows. We'll end up with two `for` statements, one nested inside the other. The outer loop will count from 1 to the number of years requested by the user. The inner loop will increment the interest rate from its lowest value to its highest value.

interest.c

```
/* Prints a table of compound interest */

#include <stdio.h>

#define NUM_RATES ((int) (sizeof(value) / sizeof(value[0])))
#define INITIAL_BALANCE 100.00

int main(void)
{
  int i, low_rate, num_years, year;
  double value[5];

  printf("Enter interest rate: ");
  scanf("%d", &low_rate);
  printf("Enter number of years: ");
  scanf("%d", &num_years);

  printf("\nYears");
  for (i = 0; i < NUM_RATES; i++) {
    printf("%6d%%", low_rate + i);
    value[i] = INITIAL_BALANCE;
  }
  printf("\n");

  for (year = 1; year <= num_years; year++) {
    printf("%3d    ", year);
    for (i = 0; i < NUM_RATES; i++) {
      value[i] += (low_rate + i) / 100.0 * value[i];
      printf("%7.2f", value[i]);
    }
    printf("\n");
  }

  return 0;
}
```

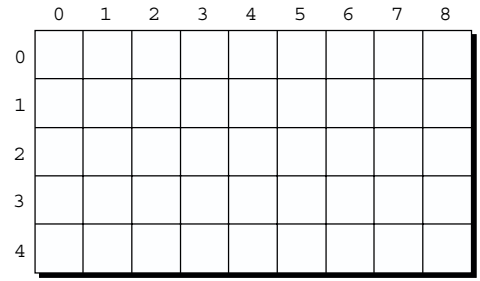
Note the use of `NUM_RATES` to control two of the `for` loops. If we later change the size of the `value` array, the loops will adjust automatically.

8.2 Multidimensional Arrays

An array may have any number of dimensions. For example, the following declaration creates a two-dimensional array (a *matrix,* in mathematical terminology):

```
int m[5][9];
```

The array `m` has 5 rows and 9 columns. Both rows and columns are indexed from 0, as the following figure shows:

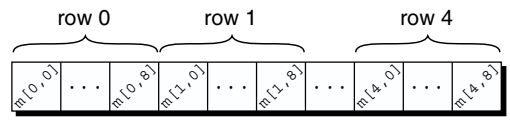


To access the element of `m` in row `i`, column `j`, we must write `m[i][j]`. The expression `m[i]` designates row `i` of `m`, and `m[i][j]` then selects element `j` in this row.

Resist the temptation to write `m[i,j]` instead of `m[i][j]`. C treats the comma as an operator in this context, so `m[i,j]` is the same as `m[j]`.

comma operator ➤6.3

Although we visualize two-dimensional arrays as tables, that's not the way they're actually stored in computer memory. C stores arrays in ***row-major order,*** with row 0 first, then row 1, and so forth. For example, here's how the `m` array is stored:



We'll usually ignore this detail, but sometimes it will affect our code.

Just as `for` loops go hand-in-hand with one-dimensional arrays, nested `for` loops are ideal for processing multidimensional arrays. Consider, for example, the problem of initializing an array for use as an identity matrix. (In mathematics, an *identity matrix* has 1's on the main diagonal, where the row and column index are the same, and 0's everywhere else.) We'll need to visit each element in the array in some systematic fashion. A pair of nested `for` loops—one that steps through every row index and one that steps through each column index—is perfect for the job:

```
#define N 10

double ident[N][N];
int row, col;

for (row = 0; row < N; row++)
  for (col = 0; col < N; col++)
    if (row == col)
      ident[row][col] = 1.0;
    else
      ident[row][col] = 0.0;
```

Multidimensional arrays play a lesser role in C than in many other programming languages, primarily because C provides a more flexible way to store multidimensional data: arrays of pointers.

arrays of pointers ➤13.7

Initializing a Multidimensional Array

We can create an initializer for a two-dimensional array by nesting one-dimensional initializers:

```
int m[5][9] = {{1, 1, 1, 1, 1, 0, 1, 1, 1},
               {0, 1, 0, 1, 0, 1, 0, 1, 0},
               {0, 1, 0, 1, 1, 0, 0, 1, 0},
               {1, 1, 0, 1, 0, 0, 0, 1, 0},
               {1, 1, 0, 1, 0, 0, 1, 1, 1}};
```

Each inner initializer provides values for one row of the matrix. Initializers for higher-dimensional arrays are constructed in a similar fashion.

C provides a variety of ways to abbreviate initializers for multidimensional arrays:

- If an initializer isn't large enough to fill a multidimensional array, the remaining elements are given the value 0. For example, the following initializer fills only the first three rows of `m`; the last two rows will contain zeros:

  ```
  int m[5][9] = {{1, 1, 1, 1, 1, 0, 1, 1, 1},
                 {0, 1, 0, 1, 0, 1, 0, 1, 0},
                 {0, 1, 0, 1, 1, 0, 0, 1, 0}};
  ```

- If an inner list isn't long enough to fill a row, the remaining elements in the row are initialized to 0:

  ```
  int m[5][9] = {{1, 1, 1, 1, 1, 0, 1, 1, 1},
                 {0, 1, 0, 1, 0, 1, 0, 1},
                 {0, 1, 0, 1, 1, 0, 0, 1},
                 {1, 1, 0, 1, 0, 0, 0, 1},
                 {1, 1, 0, 1, 0, 0, 1, 1, 1}};
  ```

- We can even omit the inner braces:

  ```
  int m[5][9] = {1, 1, 1, 1, 1, 0, 1, 1, 1,
                 0, 1, 0, 1, 0, 1, 0, 1, 0,
                 0, 1, 0, 1, 1, 0, 0, 1, 0,
                 1, 1, 0, 1, 0, 0, 0, 1, 0,
                 1, 1, 0, 1, 0, 0, 1, 1, 1};
  ```

  Once the compiler has seen enough values to fill one row, it begins filling the next.

Omitting the inner braces in a multidimensional array initializer can be risky, since an extra element (or even worse, a missing element) will affect the rest of the initializer. Leaving out the braces causes some compilers to produce a warning message such as *"missing braces around initializer."*

C99's designated initializers work with multidimensional arrays. For example, we could create a 2×2 identity matrix as follows:

```
double ident[2][2] = {[0][0] = 1.0, [1][1] = 1.0};
```

As usual, all elements for which no value is specified will default to zero.

Constant Arrays

Any array, whether one-dimensional or multidimensional, can be made "constant" by starting its declaration with the word `const`:

```
const char hex_chars[] =
  {'0', '1', '2', '3', '4', '5', '6', '7', '8', '9',
   'A', 'B', 'C', 'D', 'E', 'F'};
```

An array that's been declared `const` should not be modified by the program; the compiler will detect direct attempts to modify an element.

Declaring an array to be `const` has a couple of primary advantages. It documents that the program won't change the array, which can be valuable information for someone reading the code later. It also helps the compiler catch errors, by informing it that we don't intend to modify the array.

`const` type qualifier ➤ *18.3*

`const` isn't limited to arrays; it works with any variable, as we'll see later. However, `const` is particularly useful in array declarations, because arrays may contain reference information that won't change during program execution.

PROGRAM Dealing a Hand of Cards

Our next program illustrates both two-dimensional arrays and constant arrays. The program deals a random hand from a standard deck of playing cards. (In case you haven't had time to play games recently, each card in a standard deck has a *suit*—clubs, diamonds, hearts, or spades—and a *rank*—two, three, four, five, six, seven, eight, nine, ten, jack, queen, king, or ace.) We'll have the user specify how many cards should be in the hand:

```
Enter number of cards in hand: 5
Your hand: 7c 2s 5d as 2h
```

It's not immediately obvious how we'd write such a program. How do we pick cards randomly from the deck? And how do we avoid picking the same card twice? Let's tackle these problems separately.

`time` function ➤ *26.3*

To pick cards randomly, we'll use several C library functions. The `time` function (from `<time.h>`) returns the current time, encoded in a single number. The

`srand` function ➤ *26.2*

`srand` function (from `<stdlib.h>`) initializes C's random number generator. Passing the return value of `time` to `srand` prevents the program from dealing the

`rand` function ➤ *26.2*

same cards every time we run it. The `rand` function (also from `<stdlib.h>`) produces an apparently random number each time it's called. By using the `%` operator, we can scale the return value from `rand` so that it falls between 0 and 3 (for suits) or between 0 and 12 (for ranks).

To avoid picking the same card twice, we'll need to keep track of which cards have already been chosen. For that purpose, we'll use an array named `in_hand`

that has four rows (one for each suit) and 13 columns (one for each rank). In other words, each element in the array corresponds to one of the 52 cards in the deck. All elements of the array will be false to start with. Each time we pick a card at random, we'll check whether the element of `in_hand` corresponding to that card is true or false. If it's true, we'll have to pick another card. If it's false, we'll store `true` in that card's array element to remind us later that this card has already been picked.

Once we've verified that a card is "new"—not already selected—we'll need to translate its numerical rank and suit into characters and then display the card. To translate the rank and suit to character form, we'll set up two arrays of characters—one for the rank and one for the suit—and then use the numbers to subscript the arrays. These arrays won't change during program execution, so we may as well declare them to be `const`.

deal.c

```
/* Deals a random hand of cards */

#include <stdbool.h>   /* C99 only */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define NUM_SUITS 4
#define NUM_RANKS 13

int main(void)
{
  bool in_hand[NUM_SUITS][NUM_RANKS] = {false};
  int num_cards, rank, suit;
  const char rank_code[] = {'2','3','4','5','6','7','8',
                            '9','t','j','q','k','a'};
  const char suit_code[] = {'c','d','h','s'};

  srand((unsigned) time(NULL));

  printf("Enter number of cards in hand: ");
  scanf("%d", &num_cards);

  printf("Your hand:");
  while (num_cards > 0) {
    suit = rand() % NUM_SUITS;     /* picks a random suit */
    rank = rand() % NUM_RANKS;     /* picks a random rank */
    if (!in_hand[suit][rank]) {
      in_hand[suit][rank] = true;
      num_cards--;
      printf(" %c%c", rank_code[rank], suit_code[suit]);
    }
  }
  printf("\n");

  return 0;
}
```

Notice the initializer for the `in_hand` array:

```
bool in_hand[NUM_SUITS][NUM_RANKS] = {false};
```

Even though `in_hand` is a two-dimensional array, we can use a single pair of braces (at the risk of possibly incurring a warning from the compiler). Also, we've supplied only one value in the initializer, knowing that the compiler will fill in 0 (false) for the other elements.

8.3 Variable-Length Arrays (C99)

Section 8.1 stated that the length of an array variable must be specified by a constant expression. In C99, however, it's sometimes possible to use an expression that's *not* constant. The following modification of the `reverse.c` program (Section 8.1) illustrates this ability:

reverse2.c

```
/* Reverses a series of numbers using a variable-length
   array - C99 only */

#include <stdio.h>

int main(void)
{
  int i, n;

  printf("How many numbers do you want to reverse? ");
  scanf("%d", &n);

  int a[n];   /* C99 only - length of array depends on n */

  printf("Enter %d numbers: ", n);
  for (i = 0; i < n; i++)
    scanf("%d", &a[i]);

  printf("In reverse order:");
  for (i = n - 1; i >= 0; i--)
    printf(" %d", a[i]);
  printf("\n");

  return 0;
}
```

The array `a` in this program is an example of a ***variable-length array*** (or ***VLA*** for short). The length of a VLA is computed when the program is executed, not when the program is compiled. The chief advantage of a VLA is that the programmer doesn't have to pick an arbitrary length when declaring an array; instead, the program itself can calculate exactly how many elements are needed. If the programmer makes the choice, it's likely that the array will be too long (wasting memory) or too short (causing the program to fail). In the `reverse2.c` program, the num-

ber entered by the user determines the length of `a`; the programmer doesn't have to choose a fixed length, unlike in the original version of the program.

The length of a VLA doesn't have to be specified by a single variable. Arbitrary expressions, possibly containing operators, are also legal. For example:

```
int a[3*i+5];
int b[j+k];
```

Like other arrays, VLAs can be multidimensional:

```
int c[m][n];
```

static storage duration ➤ 18.2

The primary restriction on VLAs is that they can't have static storage duration. (We haven't yet seen any arrays with this property.) Another restriction is that a VLA may not have an initializer.

Variable-length arrays are most often seen in functions other than `main`. One big advantage of a VLA that belongs to a function `f` is that it can have a different length each time `f` is called. We'll explore this feature in Section 9.3.

Q & A

Q: Why do array subscripts start at 0 instead of 1? [p. 162]

A: Having subscripts begin at 0 simplifies the compiler a bit. Also, it can make array subscripting marginally faster.

Q: What if I want an array with subscripts that go from 1 to 10 instead of 0 to 9?

A: Here's a common trick: declare the array to have 11 elements instead of 10. The subscripts will go from 0 to 10, but you can just ignore element 0.

Q: Is it possible to use a character as an array subscript?

A: Yes, because C treats characters as integers. You'll probably need to "scale" the character before you use it as a subscript, though. Let's say that we want the `letter_count` array to keep track of a count for each letter in the alphabet. The array will need 26 elements, so we'd declare it in the following way:

```
int letter_count[26];
```

However, we can't use letters to subscript `letter_count` directly, because their integer values don't fall between 0 and 25. To scale a lower-case letter to the proper range, we can simply subtract `'a'`; to scale an upper-case letter, we'll subtract `'A'`. For example, if `ch` contains a lower-case letter, we'd write

```
letter_count[ch-'a'] = 0;
```

to clear the count that corresponds to `ch`. A minor caveat: this technique isn't completely portable, because it assumes that letters have consecutive codes. However, it works with most character sets, including ASCII.

Q: It seems like a designated initializer could end up initializing an array element more than once. Consider the following array declaration:

```
int a[] = {4, 9, 1, 8, [0] = 5, 7};
```

Is this declaration legal, and if so, what is the length of the array? [p. 166]

A: Yes, the declaration is legal. Here's how it works: as it processes an initializer list, the compiler keeps track of which array element is to be initialized next. Normally, the next element is the one following the element that was last initialized. However, when a designator appears in the list, it forces the next element be the one represented by the designator, *even if that element has already been initialized.*

Here's a step-by-step look at how the compiler will process the initializer for the array `a`:

The 4 initializes element 0; the next element to be initialized is element 1.
The 9 initializes element 1; the next element to be initialized is element 2.
The 1 initializes element 2; the next element to be initialized is element 3.
The 8 initializes element 3; the next element to be initialized is element 4.
The [0] designator causes the next element to become 0, so the 5 initializes element 0 (replacing the 4 previously stored there). The next element to be initialized is element 1.
The 7 initializes element 1 (replacing the 9 previously stored there). The next element to be initialized is element 2 (which is irrelevant since we're at the end of the list).

The net effect is the same as if we had written

```
int a[] = {5, 7, 1, 8};
```

Thus, the length of this array is four.

Q: The compiler gives me an error message if I try to copy one array into another by using the assignment operator. What's wrong?

A: Although it looks quite plausible, the assignment

```
a = b;   /* a and b are arrays */
```

is indeed illegal. The reason for its illegality isn't obvious; it has to do with the peculiar relationship between arrays and pointers in C, a topic we'll explore in Chapter 12.

The simplest way to copy one array into another is to use a loop that copies the elements, one by one:

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

memcpy function ➤23.6

Another possibility is to use the `memcpy` ("memory copy") function from the `<string.h>` header. `memcpy` is a low-level function that simply copies bytes from one place to another. To copy the array `b` into the array `a`, use `memcpy` as follows:

```
memcpy(a, b, sizeof(a));
```

Many programmers prefer `memcpy`, especially for large arrays, because it's potentially faster than an ordinary loop.

***Q: Section 6.4 mentioned that C99 doesn't allow a `goto` statement to bypass the declaration of a variable-length array. What's the reason for this restriction?**

A: The memory used to store a variable-length array is usually allocated when the declaration of the array is reached during program execution. Bypassing the declaration using a `goto` statement could result in a program accessing the elements of an array that was never allocated.

Exercises

Section 8.1

1. We discussed using the expression `sizeof(a) / sizeof(a[0])` to calculate the number of elements in an array. The expression `sizeof(a) / sizeof(`*t*`)`, where *t* is the type of `a`'s elements, would also work, but it's considered an inferior technique. Why?

2. The Q&A section shows how to use a *letter* as an array subscript. Describe how to use a *digit* (in character form) as a subscript.

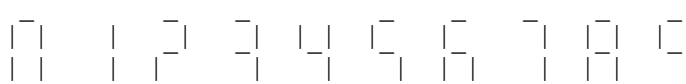
3. Write a declaration of an array named `weekend` containing seven `bool` values. Include an initializer that makes the first and last values `true`; all other values should be `false`.

4. (C99) Repeat Exercise 3, but this time use a designated initializer. Make the initializer as short as possible.

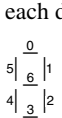
5. The Fibonacci numbers are 0, 1, 1, 2, 3, 5, 8, 13, …, where each number is the sum of the two preceding numbers. Write a program fragment that declares an array named `fib_numbers` of length 40 and fills the array with the first 40 Fibonacci numbers. *Hint:* Fill in the first two numbers individually, then use a loop to compute the remaining numbers.

Section 8.2

6. Calculators, watches, and other electronic devices often rely on seven-segment displays for numerical output. To form a digit, such devices "turn on" some of the seven segments while leaving others "off":



Suppose that we want to set up an array that remembers which segments should be "on" for each digit. Let's number the segments as follows:



Here's what the array might look like, with each row representing one digit:

```
const int segments[10][7] = {{1, 1, 1, 1, 1, 1, 0}, …};
```

I've given you the first row of the initializer; fill in the rest.

W 7. Using the shortcuts described in Section 8.2, shrink the initializer for the `segments` array (Exercise 6) as much as you can.

8. Write a declaration for a two-dimensional array named `temperature_readings` that stores one month of hourly temperature readings. (For simplicity, assume that a month has 30 days.) The rows of the array should represent days of the month; the columns should represent hours of the day.

9. Using the array of Exercise 8, write a program fragment that computes the average temperature for a month (averaged over all days of the month and all hours of the day).

10. Write a declaration for an 8×8 `char` array named `chess_board`. Include an initializer that puts the following data into the array (one character per array element):

```
r n b q k b n r
p p p p p p p p
  .   .   .   .
.   .   .   .
  .   .   .   .
.   .   .   .
P P P P P P P P
R N B Q K B N R
```

11. Write a program fragment that declares an 8×8 `char` array named `checker_board` and then uses a loop to store the following data into the array (one character per array element):

```
B R B R B R B R
R B R B R B R B
B R B R B R B R
R B R B R B R B
B R B R B R B R
R B R B R B R B
B R B R B R B R
R B R B R B R B
```

Hint: The element in row *i*, column *j*, should be the letter `B` if $i + j$ is an even number.

Programming Projects

1. Modify the `repdigit.c` program of Section 8.1 so that it shows which digits (if any) were repeated:

```
Enter a number: 939577
Repeated digit(s): 7 9
```

W 2. Modify the `repdigit.c` program of Section 8.1 so that it prints a table showing how many times each digit appears in the number:

```
Enter a number: 41271092
Digit:          0  1  2  3  4  5  6  7  8  9
Occurrences:    1  2  2  0  1  0  0  1  0  1
```

3. Modify the `repdigit.c` program of Section 8.1 so that the user can enter more than one number to be tested for repeated digits. The program should terminate when the user enters a number that's less than or equal to 0.

4. Modify the `reverse.c` program of Section 8.1 to use the expression `(int) (sizeof(a) / sizeof(a[0]))` (or a macro with this value) for the array length.

W 5. Modify the `interest.c` program of Section 8.1 so that it compounds interest *monthly* instead of *annually*. The form of the output shouldn't change; the balance should still be shown at annual intervals.

6. The prototypical Internet newbie is a fellow named B1FF, who has a unique way of writing messages. Here's a typical B1FF communiqué:

```
H3Y DUD3, C 15 R1LLY C00L!!!!!!!!!!
```

Write a "B1FF filter" that reads a message entered by the user and translates it into B1FF-speak:

```
Enter message: Hey dude, C is rilly cool
In B1FF-speak: H3Y DUD3, C 15 R1LLY C00L!!!!!!!!!!
```

Your program should convert the message to upper-case letters, substitute digits for certain letters (A→4, B→8, E→3, I→1, O→0, S→5), and then append 10 or so exclamation marks. *Hint:* Store the original message in an array of characters, then go back through the array, translating and printing characters one by one.

7. Write a program that reads a 5 × 5 array of integers and then prints the row sums and the column sums:

```
Enter row 1: 8 3 9 0 10
Enter row 2: 3 5 17 1 1
Enter row 3: 2 8 6 23 1
Enter row 4: 15 7 3 2 9
Enter row 5: 6 14 2 6 0

Row totals: 30 27 40 36 28
Column totals: 34 37 37 32 21
```

W 8. Modify Programming Project 7 so that it prompts for five quiz grades for each of five students, then computes the total score and average score for each *student*, and the average score, high score, and low score for each *quiz*.

9. Write a program that generates a "random walk" across a 10 × 10 array. The array will contain characters (all `'.'` initially). The program must randomly "walk" from element to element, always going up, down, left, or right by one element. The elements visited by the program will be labeled with the letters `A` through `Z`, in the order visited. Here's an example of the desired output:

```
A . . . . . . . . .
B C D . . . . . . .
. F E . . . . . . .
H G . . . . . . . .
I . . . . . . . . .
J . . . . . . . Z .
K . . R S T U V Y .
L M P Q . . . W X .
. N O . . . . . . .
. . . . . . . . . .
```

Hint: Use the `srand` and `rand` functions (see `deal.c`) to generate random numbers. After generating a number, look at its remainder when divided by 4. There are four possible values for the remainder—0, 1, 2, and 3—indicating the direction of the next move. Before performing a move, check that (a) it won't go outside the array, and (b) it doesn't take us to

an element that already has a letter assigned. If either condition is violated, try moving in another direction. If all four directions are blocked, the program must terminate. Here's an example of premature termination:

```
A B G H I . . . . .
. C F . J K . . . .
. D E . M L . . . .
. . . . N O . . . .
. . W X Y P Q . . .
. . V U T S R . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
```

`Y` is blocked on all four sides, so there's no place to put `Z`.

10. Modify Programming Project 8 from Chapter 5 so that the departure times are stored in an array and the arrival times are stored in a second array. (The times are integers, representing the number of minutes since midnight.) The program will use a loop to search the array of departure times for the one closest to the time entered by the user.

11. Modify Programming Project 4 from Chapter 7 so that the program labels its output:

```
Enter phone number: 1-800-COL-LECT
In numeric form: 1-800-265-5328
```

The program will need to store the phone number (either in its original form or in its numeric form) in an array of characters until it can be printed. You may assume that the phone number is no more than 15 characters long.

12. Modify Programming Project 5 from Chapter 7 so that the SCRABBLE values of the letters are stored in an array. The array will have 26 elements, corresponding to the 26 letters of the alphabet. For example, element 0 of the array will store 1 (because the SCRABBLE value of the letter A is 1), element 1 of the array will store 3 (because the SCRABBLE value of the letter B is 3), and so forth. As each character of the input word is read, the program will use the array to determine the SCRABBLE value of that character. Use an array initializer to set up the array.

13. Modify Programming Project 11 from Chapter 7 so that the program labels its output:

```
Enter a first and last name: Lloyd Fosdick
You entered the name: Fosdick, L.
```

The program will need to store the last name (but not the first name) in an array of characters until it can be printed. You may assume that the last name is no more than 20 characters long.

14. Write a program that reverses the words in a sentence:

```
Enter a sentence: you can cage a swallow can't you?
Reversal of sentence: you can't swallow a cage can you?
```

Hint: Use a loop to read the characters one by one and store them in a one-dimensional `char` array. Have the loop stop at a period, question mark, or exclamation point (the "terminating character"), which is saved in a separate `char` variable. Then use a second loop to search backward through the array for the beginning of the last word. Print the last word, then search backward for the next-to-last word. Repeat until the beginning of the array is reached. Finally, print the terminating character.

15. One of the oldest known encryption techniques is the Caesar cipher, attributed to Julius Caesar. It involves replacing each letter in a message with another letter that is a fixed number of

positions later in the alphabet. (If the replacement would go past the letter *Z*, the cipher "wraps around" to the beginning of the alphabet. For example, if each letter is replaced by the letter two positions after it, then *Y* would be replaced by *A*, and *Z* would be replaced by *B*.) Write a program that encrypts a message using a Caesar cipher. The user will enter the message to be encrypted and the shift amount (the number of positions by which letters should be shifted):

```
Enter message to be encrypted: Go ahead, make my day.
Enter shift amount (1-25): 3
Encrypted message: Jr dkhdg, pdnh pb gdb.
```

Notice that the program can decrypt a message if the user enters 26 minus the original key:

```
Enter message to be encrypted: Jr dkhdg, pdnh pb gdb.
Enter shift amount (1-25): 23
Encrypted message: Go ahead, make my day.
```

You may assume that the message does not exceed 80 characters. Characters other than letters should be left unchanged. Lower-case letters remain lower-case when encrypted, and upper-case letters remain upper-case. *Hint:* To handle the wrap-around problem, use the expression `((ch - 'A') + n) % 26 + 'A'` to calculate the encrypted version of an upper-case letter, where `ch` stores the letter and `n` stores the shift amount. (You'll need a similar expression for lower-case letters.)

16. Write a program that tests whether two words are anagrams (permutations of the same letters):

```
Enter first word: smartest
Enter second word: mattress
The words are anagrams.

Enter first word: dumbest
Enter second word: stumble
The words are not anagrams.
```

Write a loop that reads the first word, character by character, using an array of 26 integers to keep track of how many times each letter has been seen. (For example, after the word *smartest* has been read, the array should contain the values 1 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 1 2 2 0 0 0 0 0 0 0, reflecting the fact that *smartest* contains one *a*, one *e*, one *m*, one *r*, two *s*'s and two *t*'s.) Use another loop to read the second word, except this time decrementing the corresponding array element as each letter is read. Both loops should ignore any characters that aren't letters, and both should treat upper-case letters in the same way as lower-case letters. After the second word has been read, use a third loop to check whether all the elements in the array are zero. If so, the words are anagrams. *Hint:* You may wish to use functions from `<ctype.h>`, such as `isalpha` and `tolower`.

17. Write a program that prints an $n \times n$ magic square (a square arrangement of the numbers 1, 2, …, n^2 in which the sums of the rows, columns, and diagonals are all the same). The user will specify the value of n:

```
This program creates a magic square of a specified size.
The size must be an odd number between 1 and 99.
Enter size of magic square: 5
   17   24    1    8   15
   23    5    7   14   16
    4    6   13   20   22
   10   12   19   21    3
   11   18   25    2    9
```

Store the magic square in a two-dimensional array. Start by placing the number 1 in the middle of row 0. Place each of the remaining numbers 2, 3, …, n^2 by moving up one row and over one column. Any attempt to go outside the bounds of the array should "wrap around" to the opposite side of the array. For example, instead of storing the next number in row –1, we would store it in row $n - 1$ (the last row). Instead of storing the next number in column n, we would store it in column 0. If a particular array element is already occupied, put the number directly below the previously stored number. If your compiler supports variable-length arrays, declare the array to have n rows and n columns. If not, declare the array to have 99 rows and 99 columns.