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];
```

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:

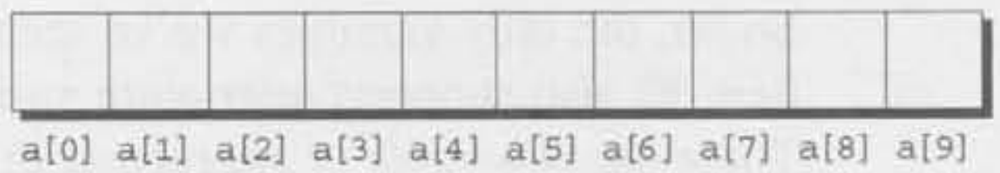
constant expressions ➤5.3

```
#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



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

lvalues ➤4.2

```
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];
```

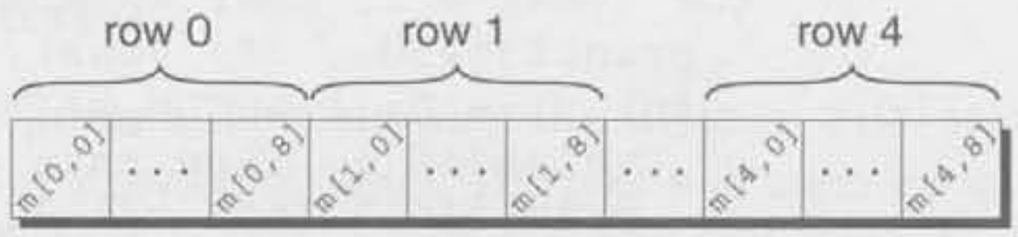
The array `m` has 5 rows and 9 columns. Both rows and columns are indexed from 0, as the following figure shows:

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 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 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).

`rand` function ➤ *26.2*

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

W 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?

W 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.

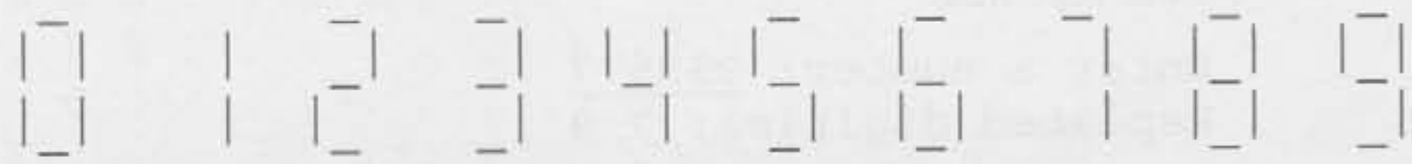
3. Write a declaration of an array named `weekend` containing seven `bool` values. Include an initializer that makes the first and last values `true`; all other values should be `false`.

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 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 $1, 2, \ldots, n^2$ 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.

9 Functions

If you have a procedure with ten parameters, you probably missed some.

We saw in Chapter 2 that a function is simply a series of statements that have been grouped together and given a name. Although the term "function" comes from mathematics, C functions don't always resemble math functions. In C, a function doesn't necessarily have arguments, nor does it necessarily compute a value. (In some programming languages, a "function" returns a value, whereas a "procedure" doesn't. C lacks this distinction.)

Functions are the building blocks of C programs. Each function is essentially a small program, with its own declarations and statements. Using functions, we can divide a program into small pieces that are easier for us—and others—to understand and modify. Functions can take some of the tedium out of programming by allowing us to avoid duplicating code that's used more than once. Moreover, functions are reusable: we can take a function that was originally part of one program and use it in others.

Our programs so far have consisted of just the `main` function. In this chapter, we'll see how to write functions other than `main`, and we'll learn more about `main` itself. Section 9.1 shows how to define and call functions. Section 9.2 then discusses function declarations and how they differ from function definitions. Next, Section 9.3 examines how arguments are passed to functions. The remainder of the chapter covers the `return` statement (Section 9.4), the related issue of program termination (Section 9.5), and recursion (Section 9.6).

9.1 Defining and Calling Functions

Before we go over the formal rules for defining a function, let's look at three simple programs that define functions.

PROGRAM Computing Averages

Suppose we often need to compute the average of two `double` values. The C library doesn't have an "average" function, but we can easily define our own. Here's what it would look like:

```
double average(double a, double b)
{
  return (a + b) / 2;
}
```

Q&A

The word `double` at the beginning is `average`'s ***return type:*** the type of data that the function returns each time it's called. The identifiers `a` and `b` (the function's ***parameters***) represent the two numbers that will be supplied when `average` is called. Each parameter must have a type (just like every variable has a type); in this example, both `a` and `b` have type `double`. (It may look odd, but the word `double` must appear twice, once for `a` and once for `b`.) A function parameter is essentially a variable whose initial value will be supplied later, when the function is called.

Every function has an executable part, called the ***body***, which is enclosed in braces. The body of `average` consists of a single `return` statement. Executing this statement causes the function to "return" to the place from which it was called; the value of `(a + b) / 2` will be the value returned by the function.

To call a function, we write the function name, followed by a list of ***arguments***. For example, `average(x, y)` is a call of the `average` function. Arguments are used to supply information to a function; in this case, `average` needs to know which two numbers to average. The effect of the call `average(x, y)` is to copy the values of `x` and `y` into the parameters `a` and `b`, and then execute the body of `average`. An argument doesn't have to be a variable; any expression of a compatible type will do, allowing us to write `average(5.1, 8.9)` or `average(x/2, y/3)`.

We'll put the call of `average` in the place where we need to use the return value. For example, we could write

```
printf("Average: %g\n", average(x, y));
```

to compute the average of `x` and `y` and then print it. This statement has the following effect:

1. The `average` function is called with `x` and `y` as arguments.
2. `x` and `y` are copied into `a` and `b`.
3. `average` executes its `return` statement, returning the average of `a` and `b`.
4. `printf` prints the value that `average` returns. (The return value of `average` becomes one of `printf`'s arguments.)

Note that the return value of `average` isn't saved anywhere; the program prints it and then discards it. If we had needed the return value later in the program, we could have captured it in a variable:

```
avg = average(x, y);
```

This statement calls `average`, then saves its return value in the variable `avg`.

Now, let's use the `average` function in a complete program. The following program reads three numbers and computes their averages, one pair at a time:

```
Enter three numbers: 3.5 9.6 10.2
Average of 3.5 and 9.6: 6.55
Average of 9.6 and 10.2: 9.9
Average of 3.5 and 10.2: 6.85
```

Among other things, this program shows that a function can be called as often as we need.

average.c

```
/* Computes pairwise averages of three numbers */

#include <stdio.h>

double average(double a, double b)
{
  return (a + b) / 2;
}

int main(void)
{
  double x, y, z;

  printf("Enter three numbers: ");
  scanf("%lf%lf%lf", &x, &y, &z);
  printf("Average of %g and %g: %g\n", x, y, average(x, y));
  printf("Average of %g and %g: %g\n", y, z, average(y, z));
  printf("Average of %g and %g: %g\n", x, z, average(x, z));

  return 0;
}
```

Notice that I've put the definition of `average` before `main`. We'll see in Section 9.2 that putting `average` after `main` causes problems.

PROGRAM

Printing a Countdown

Not every function returns a value. For example, a function whose job is to produce output may not need to return anything. To indicate that a function has no return value, we specify that its return type is `void`. (`void` is a type with no values.) Consider the following function, which prints the message `T minus` *n* `and counting`, where *n* is supplied when the function is called:

```
void print_count(int n)
{
  printf("T minus %d and counting\n", n);
}
```

`print_count` has one parameter, `n`, of type `int`. It returns nothing, so I've specified `void` as the return type and omitted the `return` statement. Since `print_count` doesn't return a value, we can't call it in the same way we call `average`. Instead, a call of `print_count` must appear in a statement by itself:

```
print_count(i);
```

Here's a program that calls `print_count` 10 times inside a loop:

countdown.c

```
/* Prints a countdown */

#include <stdio.h>

void print_count(int n)
{
  printf("T minus %d and counting\n", n);
}

int main(void)
{
  int i;

  for (i = 10; i > 0; --i)
    print_count(i);

  return 0;
}
```

Initially, `i` has the value 10. When `print_count` is called for the first time, `i` is copied into `n`, so that `n` takes on the value 10 as well. As a result, the first call of `print_count` will print

```
T minus 10 and counting
```

`print_count` then returns to the point at which it was called, which happens to be the body of a `for` statement. The `for` statement resumes where it left off, decrementing `i` to 9 and testing whether it's greater than 0. It is, so `print_count` is called again, this time printing

```
T minus 9 and counting
```

Each time `print_count` is called, `i` is different, so `print_count` will print 10 different messages.

PROGRAM Printing a Pun (Revisited)

Some functions have no parameters at all. Consider `print_pun`, which prints a bad pun each time it's called:

```
void print_pun(void)
{
  printf("To C, or not to C: that is the question.\n");
}
```

The word `void` in parentheses indicates that `print_pun` has no arguments. (This time, we're using `void` as a placeholder that means "nothing goes here.")

To call a function with no arguments, we write the function's name, followed by parentheses:

```
print_pun();
```

The parentheses *must* be present, even though there are no arguments.

Here's a tiny program that tests the `print_pun` function:

pun2.c

```
/* Prints a bad pun */

#include <stdio.h>

void print_pun(void)
{
  printf("To C, or not to C: that is the question.\n");
}

int main(void)
{
  print_pun();
  return 0;
}
```

The execution of this program begins with the first statement in `main`, which happens to be a call of `print_pun`. When `print_pun` begins to execute, it in turn calls `printf` to display a string. When `printf` returns, `print_pun` returns to `main`.

Function Definitions

Now that we've seen several examples, let's look at the general form of a ***function definition:***

function definition

return-type function-name (*parameters*)
{
  declarations
  statements
}

The return type of a function is the type of value that the function returns. The following rules govern the return type:

- Functions may not return arrays, but there are no other restrictions on the return type.
- Specifying that the return type is `void` indicates that the function doesn't return a value.

- If the return type is omitted in C89, the function is presumed to return a value of type `int`. In C99, it's illegal to omit the return type of a function.

C99

As a matter of style, some programmers put the return type *above* the function name:

```
double
average(double a, double b)
{
  return (a + b) / 2;
}
```

Putting the return type on a separate line is especially useful if the return type is lengthy, like `unsigned long int`.

After the function name comes a list of parameters. Each parameter is preceded by a specification of its type; parameters are separated by commas. If the function has no parameters, the word `void` should appear between the parentheses. *Note:* A separate type must be specified for each parameter, even when several parameters have the same type:

Q&A

```
double average(double a, b)    /*** WRONG ***/
{
  return (a + b) / 2;
}
```

The body of a function may include both declarations and statements. For example, the `average` function could be written

```
double average(double a, double b)
{
  double sum;          /* declaration */

  sum = a + b;         /* statement */
  return sum / 2;      /* statement */
}
```

Variables declared in the body of a function belong exclusively to that function; they can't be examined or modified by other functions. In C89, variable declarations must come first, before all statements in the body of a function. In C99, variable declarations and statements can be mixed, as long as each variable is declared prior to the first statement that uses the variable. (Some pre-C99 compilers also allow mixing of declarations and statements.)

C99

The body of a function whose return type is `void` (which I'll call a "`void` function") can be empty:

```
void print_pun(void)
{
}
```

Leaving the body empty may make sense during program development; we can leave room for the function without taking the time to complete it, then come back later and write the body.

Function Calls

A function call consists of a function name followed by a list of arguments, enclosed in parentheses:

```
average(x, y)
print_count(i)
print_pun()
```

If the parentheses are missing, the function won't get called:

```
print_pun;    /*** WRONG ***/
```

Q&A The result is a legal (albeit meaningless) expression statement that looks correct, but has no effect. Some compilers issue a warning such as *"statement with no effect."*

A call of a `void` function is always followed by a semicolon to turn it into a statement:

```
print_count(i);
print_pun();
```

A call of a non-`void` function, on the other hand, produces a value that can be stored in a variable, tested, printed, or used in some other way:

```
avg = average(x, y);
if (average(x, y) > 0)
  printf("Average is positive\n");
printf("The average is %g\n", average(x, y));
```

The value returned by a non-`void` function can always be discarded if it's not needed:

```
average(x, y);    /* discards return value */
```

expression statements ➤4.5

This call of `average` is an example of an expression statement: a statement that evaluates an expression but then discards the result.

Ignoring the return value of `average` is an odd thing to do, but for some functions it makes sense. The `printf` function, for example, returns the number of characters that it prints. After the following call, `num_chars` will have the value 9:

```
num_chars = printf("Hi, Mom!\n");
```

Since we're probably not interested in the number of characters printed, we'll normally discard `printf`'s return value:

```
printf("Hi, Mom!\n");    /* discards return value */
```

To make it clear that we're deliberately discarding the return value of a function, C allows us to put `(void)` before the call:

```
(void) printf("Hi, Mom!\n");
```

casting ➤7.4 What we're doing is casting (converting) the return value of `printf` to type `void`. (In C, "casting to `void`" is a polite way of saying "throwing away.") Using `(void)` makes it clear to others that you deliberately discarded the return value, not just forgot that there was one. Unfortunately, there are a great many functions in the C library whose values are routinely ignored; using `(void)` when calling them all can get tiresome, so I haven't done so in this book.

PROGRAM Testing Whether a Number Is Prime

To see how functions can make programs easier to understand, let's write a program that tests whether a number is prime. The program will prompt the user to enter a number, then respond with a message indicating whether or not the number is prime:

```
Enter a number: 34
Not prime
```

Instead of putting the prime-testing details in `main`, we'll define a separate function that returns `true` if its parameter is a prime number and `false` if it isn't. When given a number `n`, the `is_prime` function will divide `n` by each of the numbers between 2 and the square root of `n`; if the remainder is ever 0, we know that `n` isn't prime.

prime.c

```
/* Tests whether a number is prime */

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

bool is_prime(int n)
{
  int divisor;

  if (n <= 1)
    return false;
  for (divisor = 2; divisor * divisor <= n; divisor++)
    if (n % divisor == 0)
      return false;
  return true;
}

int main(void)
{
  int n;

  printf("Enter a number: ");
  scanf("%d", &n);
  if (is_prime(n))
    printf("Prime\n");
  else
    printf("Not prime\n");
```

```
  return 0;
}
```

Notice that `main` contains a variable named `n` even though `is_prime`'s parameter is also named `n`. In general, a function may declare a variable with the same name as a variable in another function. The two variables represent different locations in memory, so assigning a new value to one variable doesn't change the other. (This property extends to parameters as well.) Section 10.1 discusses this point in more detail.

As `is_prime` demonstrates, a function may have more than one `return` statement. However, we can execute just one of these statements during a given call of the function, because reaching a `return` statement causes the function to return to where it was called. We'll learn more about the `return` statement in Section 9.4.

9.2 Function Declarations

In the programs in Section 9.1, the definition of each function was always placed *above* the point at which it was called. In fact, C doesn't require that the definition of a function precede its calls. Suppose that we rearrange the `average.c` program by putting the definition of `average` *after* the definition of `main`:

```
#include <stdio.h>

int main(void)
{
  double x, y, z;

  printf("Enter three numbers: ");
  scanf("%lf%lf%lf", &x, &y, &z);
  printf("Average of %g and %g: %g\n", x, y, average(x, y));
  printf("Average of %g and %g: %g\n", y, z, average(y, z));
  printf("Average of %g and %g: %g\n", x, z, average(x, z));

  return 0;
}

double average(double a, double b)
{
  return (a + b) / 2;
}
```

When the compiler encounters the first call of `average` in `main`, it has no information about `average`: it doesn't know how many parameters `average` has, what the types of these parameters are, or what kind of value `average` returns. Instead of producing an error message, though, the compiler assumes that `average` returns an `int` value (recall from Section 9.1 that the return type of a

function is `int` by default). We say that the compiler has created an ***implicit declaration*** of the function. The compiler is unable to check that we're passing `average` the right number of arguments and that the arguments have the proper type. Instead, it performs the default argument promotions and hopes for the best. When it encounters the definition of `average` later in the program, the compiler notices that the function's return type is actually `double`, not `int`, and so we get an error message.

default argument promotions ➤9.3

One way to avoid the problem of call-before-definition is to arrange the program so that the definition of each function precedes all its calls. Unfortunately, such an arrangement doesn't always exist, and even when it does, it may make the program harder to understand by putting its function definitions in an unnatural order.

Fortunately, C offers a better solution: declare each function before calling it. A ***function declaration*** provides the compiler with a brief glimpse at a function whose full definition will appear later. A function declaration resembles the first line of a function definition with a semicolon added at the end:

function declaration

return-type function-name (*parameters*) ;

Needless to say, the declaration of a function must be consistent with the function's definition.

Q&A

Here's how our program would look with a declaration of `average` added:

```
#include <stdio.h>

double average(double a, double b);   /* DECLARATION */

int main(void)
{
  double x, y, z;

  printf("Enter three numbers: ");
  scanf("%lf%lf%lf", &x, &y, &z);
  printf("Average of %g and %g: %g\n", x, y, average(x, y));
  printf("Average of %g and %g: %g\n", y, z, average(y, z));
  printf("Average of %g and %g: %g\n", x, z, average(x, z));

  return 0;
}

double average(double a, double b)     /* DEFINITION */
{
  return (a + b) / 2;
}
```

Function declarations of the kind we've been discussing are known as ***function prototypes*** to distinguish them from an older style of function declaration in which the parentheses are left empty. A prototype provides a complete description

Q&A

of how to call a function: how many arguments to supply, what their types should be, and what type of result will be returned.

Incidentally, a function prototype doesn't have to specify the *names* of the function's parameters, as long as their *types* are present:

```
double average(double, double);
```

It's usually best not to omit parameter names, since they help document the purpose of each parameter and remind the programmer of the order in which arguQ&A ments must appear when the function is called. However, there are legitimate reasons for omitting parameter names, and some programmers prefer to do so.

C99 C99 has adopted the rule that either a declaration or a definition of a function must be present prior to any call of the function. Calling a function for which the compiler has not yet seen a declaration or definition is an error.

9.3 Arguments

Let's review the difference between a parameter and an argument. *Parameters* appear in function *definitions;* they're dummy names that represent values to be supplied when the function is called. *Arguments* are expressions that appear in function *calls*. When the distinction between *argument* and *parameter* isn't important, I'll sometimes use *argument* to mean either.

In C, arguments are ***passed by value:*** when a function is called, each argument is evaluated and its value assigned to the corresponding parameter. Since the parameter contains a copy of the argument's value, any changes made to the parameter during the execution of the function don't affect the argument. In effect, each parameter behaves like a variable that's been initialized to the value of the matching argument.

The fact that arguments are passed by value has both advantages and disadvantages. Since a parameter can be modified without affecting the corresponding argument, we can use parameters as variables within the function, thereby reducing the number of genuine variables needed. Consider the following function, which raises a number `x` to a power `n`:

```
int power(int x, int n)
{
  int i, result = 1;

  for (i = 1; i <= n; i++)
    result = result * x;

  return result;
}
```

Since `n` is a *copy* of the original exponent, we can modify it inside the function, thus removing the need for `i`:

```
int power(int x, int n)
{
  int result = 1;

  while (n-- > 0)
    result = result * x;

  return result;
}
```

Unfortunately, C's requirement that arguments be passed by value makes it difficult to write certain kinds of functions. For example, suppose that we need a function that will decompose a `double` value into an integer part and a fractional part. Since a function can't *return* two numbers, we might try passing a pair of variables to the function and having it modify them:

```
void decompose(double x, long int_part, double frac_part)
{
  int_part = (long) x;    /* drops the fractional part of x */
  frac_part = x - int_part;
}
```

Suppose that we call the function in the following way:

```
decompose(3.14159, i, d);
```

At the beginning of the call, 3.14159 is copied into `x`, `i`'s value is copied into `int_part`, and `d`'s value is copied into `frac_part`. The statements inside `decompose` then assign 3 to `int_part` and .14159 to `frac_part`, and the function returns. Unfortunately, `i` and `d` weren't affected by the assignments to `int_part` and `frac_part`, so they have the same values after the call as they did before the call. With a little extra effort, `decompose` can be made to work, as we'll see in Section 11.4. However, we'll need to cover more of C's features first.

Argument Conversions

C allows function calls in which the types of the arguments don't match the types of the parameters. The rules governing how the arguments are converted depend on whether or not the compiler has seen a prototype for the function (or the function's full definition) prior to the call:

- ***The compiler has encountered a prototype prior to the call.*** The value of each argument is implicitly converted to the type of the corresponding parameter as if by assignment. For example, if an `int` argument is passed to a function that was expecting a `double`, the argument is converted to `double` automatically.
- ***The compiler has not encountered a prototype prior to the call.*** The compiler performs the ***default argument promotions:*** (1) `float` arguments are converted to `double`. (2) The integral promotions are performed, causing `char`

and `short` arguments to be converted to `int`. (In C99, the integer promotions are performed.)

C99

⚠

Relying on the default argument promotions is dangerous. Consider the following program:

```
#include <stdio.h>

int main(void)
{
  double x = 3.0;
  printf("Square: %d\n", square(x));

  return 0;
}

int square(int n)
{
  return n * n;
}
```

At the time `square` is called, the compiler hasn't seen a prototype yet, so it doesn't know that `square` expects an argument of type `int`. Instead, the compiler performs the default argument promotions on `x`, with no effect. Since it's expecting an argument of type `int` but has been given a `double` value instead, the effect of calling `square` is undefined. The problem can be fixed by casting `square`'s argument to the proper type:

```
printf("Square: %d\n", square((int) x));
```

Of course, a much better solution is to provide a prototype for `square` before calling it. In C99, calling `square` without first providing a declaration or definition of the function is an error.

C99

Array Arguments

Arrays are often used as arguments. When a function parameter is a one-dimensional array, the length of the array can be (and is normally) left unspecified:

Q&A

```
int f(int a[])   /* no length specified */
{
  ...
}
```

The argument can be any one-dimensional array whose elements are of the proper type. There's just one problem: how will `f` know how long the array is? Unfortunately, C doesn't provide any easy way for a function to determine the length of an array passed to it. Instead, we'll have to supply the length—if the function needs it—as an additional argument.

Although we can use the `sizeof` operator to help determine the length of an array *variable*, it doesn't give the correct answer for an array *parameter*:

```
int f(int a[])
{
  int len = sizeof(a) / sizeof(a[0]);
    /*** WRONG: not the number of elements in a ***/
  ...
}
```

Section 12.3 explains why.

The following function illustrates the use of one-dimensional array arguments. When given an array `a` of `int` values, `sum_array` returns the sum of the elements in `a`. Since `sum_array` needs to know the length of `a`, we must supply it as a second argument.

```
int sum_array(int a[], int n)
{
  int i, sum = 0;

  for (i = 0; i < n; i++)
    sum += a[i];
  return sum;
}
```

The prototype for `sum_array` has the following appearance:

```
int sum_array(int a[], int n);
```

As usual, we can omit the parameter names if we wish:

```
int sum_array(int [], int);
```

When `sum_array` is called, the first argument will be the name of an array, and the second will be its length. For example:

```
#define LEN 100

int main(void)
{
  int b[LEN], total;
  ...
  total = sum_array(b, LEN);
  ...
}
```

Notice that we don't put brackets after an array name when passing it to a function:

```
total = sum_array(b[], LEN);    /*** WRONG ***/
```

An important point about array arguments: A function has no way to check that we've passed it the correct array length. We can exploit this fact by telling the function that the array is smaller than it really is. Suppose that we've only stored 50 numbers in the b array, even though it can hold 100. We can sum just the first 50 elements by writing

```
total = sum_array(b, 50);    /* sums first 50 elements */
```

sum_array will ignore the other 50 elements. (Indeed, it won't know that they even exist!)

Be careful not to tell a function that an array argument is *larger* than it really is:

```
total = sum_array(b, 150);    /*** WRONG ***/
```

In this example, sum_array will go past the end of the array, causing undefined behavior.

Another important thing to know is that a function is allowed to change the elements of an array parameter, and the change is reflected in the corresponding argument. For example, the following function modifies an array by storing zero into each of its elements:

```
void store_zeros(int a[], int n)
{
  int i;

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

The call

```
store_zeros(b, 100);
```

will store zero into the first 100 elements of the array b. This ability to modify the elements of an array argument may seem to contradict the fact that C passes arguments by value. In fact, there's no contradiction, but I won't be able to explain why until Section 12.3.

Q&A If a parameter is a multidimensional array, only the length of the first dimension may be omitted when the parameter is declared. For example, if we revise the sum_array function so that a is a two-dimensional array, we must specify the number of columns in a, although we don't have to indicate the number of rows:

```
#define LEN 10

int sum_two_dimensional_array(int a[][LEN], int n)
{
  int i, j, sum = 0;
```

```
  for (i = 0; i < n; i++)
    for (j = 0; j < LEN; j++)
      sum += a[i][j];

  return sum;
}
```

Not being able to pass multidimensional arrays with an arbitrary number of columns can be a nuisance. Fortunately, we can often work around this difficulty by using arrays of pointers. C99's variable-length array parameters provide an even better solution to the problem.

arrays of pointers ➤13.7

C99 Variable-Length Array Parameters

C99 adds several new twists to array arguments. The first has to do with variable-length arrays (VLAs), a feature of C99 that allows the length of an array to be specified using a non-constant expression. Variable-length arrays can also be parameters, as it turns out.

variable-length arrays ➤8.3

Consider the `sum_array` function discussed earlier in this section. Here's the definition of `sum_array`, with the body omitted:

```
int sum_array(int a[], int n)
{
  ...
}
```

As it stands now, there's no direct link between `n` and the length of the array `a`. Although the function body treats `n` as `a`'s length, the actual length of the array could in fact be larger than `n` (or smaller, in which case the function won't work correctly).

Using a variable-length array parameter, we can explicitly state that `a`'s length is `n`:

```
int sum_array(int n, int a[n])
{
  ...
}
```

The value of the first parameter (`n`) specifies the length of the second parameter (`a`). Note that the order of the parameters has been switched; order is important when variable-length array parameters are used.

The following version of `sum_array` is illegal:

```
int sum_array(int a[n], int n)    /*** WRONG ***/
{
  ...
}
```

The compiler will issue an error message at `int a[n]`, because it hasn't yet seen `n`.

There are several ways to write the prototype for our new version of `sum_array`. One possibility is to make it look exactly like the function definition:

```
int sum_array(int n, int a[n]);    /* Version 1 */
```

Another possibility is to replace the array length by an asterisk (`*`):

```
int sum_array(int n, int a[*]);    /* Version 2a */
```

The reason for using the `*` notation is that parameter names are optional in function declarations. If the name of the first parameter is omitted, it wouldn't be possible to specify that the length of the array is `n`, but the `*` provides a clue that the length of the array is related to parameters that come earlier in the list:

```
int sum_array(int, int [*]);       /* Version 2b */
```

It's also legal to leave the brackets empty, as we normally do when declaring an array parameter:

```
int sum_array(int n, int a[]);     /* Version 3a */
int sum_array(int, int []);        /* Version 3b */
```

Leaving the brackets empty isn't a good choice, because it doesn't expose the relationship between `n` and `a`.

In general, the length of a variable-length array parameter can be any expression. For example, suppose that we were to write a function that concatenates two arrays `a` and `b` by copying the elements of `a`, followed by the elements of `b`, into a third array named `c`:

```
int concatenate(int m, int n, int a[m], int b[n], int c[m+n])
{
  ...
}
```

The length of `c` is the sum of the lengths of `a` and `b`. The expression used to specify the length of `c` involves two other parameters, but in general it could refer to variables outside the function or even call other functions.

Variable-length array parameters with a single dimension—as in all our examples so far—have limited usefulness. They make a function declaration or definition more descriptive by stating the desired length of an array argument. However, no additional error-checking is performed; it's still possible for an array argument to be too long or too short.

It turns out that variable-length array parameters are most useful for multidimensional arrays. Earlier in this section, we tried to write a function that sums the elements in a two-dimensional array. Our original function was limited to arrays with a fixed number of columns. If we use a variable-length array parameter, we can generalize the function to any number of columns:

```
int sum_two_dimensional_array(int n, int m, int a[n][m])
{
  int i, j, sum = 0;

  for (i = 0; i < n; i++)
    for (j = 0; j < m; j++)
      sum += a[i][j];

  return sum;
}
```

Prototypes for this function include the following:

```
int sum_two_dimensional_array(int n, int m, int a[n][m]);
int sum_two_dimensional_array(int n, int m, int a[*][*]);
int sum_two_dimensional_array(int n, int m, int a[][m]);
int sum_two_dimensional_array(int n, int m, int a[][*]);
```

C99 Using `static` in Array Parameter Declarations

C99 allows the use of the keyword `static` in the declaration of array parameters. (The keyword itself existed before C99. Section 18.2 discusses its traditional uses.)

In the following example, putting `static` in front of the number `3` indicates that the length of `a` is guaranteed to be at least 3:

```
int sum_array(int a[static 3], int n)
{
  ...
}
```

Using `static` in this way has no effect on the behavior of the program. The presence of `static` is merely a "hint" that may allow a C compiler to generate faster instructions for accessing the array. (If the compiler knows that an array will always have a certain minimum length, it can arrange to "prefetch" these elements from memory when the function is called, before the elements are actually needed by statements within the function.)

One last note about `static`: If an array parameter has more than one dimension, `static` can be used only in the first dimension (for example, when specifying the number of rows in a two-dimensional array).

C99 Compound Literals

Let's return to the original `sum_array` function one last time. When `sum_array` is called, the first argument is usually the name of an array (the one whose elements are to be summed). For example, we might call `sum_array` in the following way:

```
int b[] = {3, 0, 3, 4, 1};
total = sum_array(b, 5);
```

The only problem with this arrangement is that b must be declared as a variable and then initialized prior to the call. If b isn't needed for any other purpose, it can be mildly annoying to create it solely for the purpose of calling sum_array.

In C99, we can avoid this annoyance by using a ***compound literal:*** an unnamed array that's created "on the fly" by simply specifying which elements it contains. The following call of sum_array has a compound literal (shown in **bold**) as its first argument:

```
total = sum_array((int []){3, 0, 3, 4, 1}, 5);
```

In this example, the compound literal creates an array containing the five integers 3, 0, 3, 4, and 1. We didn't specify the length of the array, so it's determined by the number of elements in the literal. We also have the option of specifying a length explicitly: (int [4]){1, 9, 2, 1} is equivalent to (int []){1, 9, 2, 1}.

In general, a compound literal consists of a type name within parentheses, followed by a set of values enclosed by braces. A compound literal resembles a cast applied to an initializer. In fact, compound literals and initializers obey the same rules. A compound literal may contain designators, just like a designated initializer, and it may fail to provide full initialization (in which case any uninitialized elements default to zero). For example, the literal (int [10]){8, 6} has 10 elements; the first two have the values 8 and 6, and the remaining elements have the value 0.

designated initializers ➤8.1

Compound literals created inside a function may contain arbitrary expressions, not just constants. For example, we could write

```
total = sum_array((int []){2 * i, i + j, j * k}, 3);
```

where i, j, and k are variables. This aspect of compound literals greatly enhances their usefulness.

lvalues ➤4.2

A compound literal is an lvalue, so the values of its elements can be changed. If desired, a compound literal can be made "read-only" by adding the word const to its type, as in (const int []){5, 4}.

9.4 The return Statement

A non-void function must use the return statement to specify what value it will return. The return statement has the form

return statement

```
return expression ;
```

The expression is often just a constant or variable:

```
return 0;
return status;
```

conditional operator ➤5.2

More complex expressions are possible. For example, it's not unusual to see the conditional operator used in a return expression:

```
return n >= 0 ? n : 0;
```

When this statement is executed, the expression `n >= 0 ? n : 0` is evaluated first. The statement returns the value of `n` if it's not negative; otherwise, it returns 0.

If the type of the expression in a `return` statement doesn't match the function's return type, the expression will be implicitly converted to the return type. For example, if a function is declared to return an `int`, but the `return` statement contains a `double` expression, the value of the expression is converted to `int`.

`return` statements may appear in functions whose return type is `void`, provided that no expression is given:

```
return;   /* return in a void function */
```

Q&A

Putting an expression in such a `return` statement will get you a compile-time error. In the following example, the `return` statement causes the function to return immediately when given a negative argument:

```
void print_int(int i)
{
  if (i < 0)
    return;
  printf("%d", i);
}
```

If `i` is less than 0, `print_int` will return without calling `printf`.

A `return` statement may appear at the end of a `void` function:

```
void print_pun(void)
{
  printf("To C, or not to C: that is the question.\n");
  return;   /* OK, but not needed */
}
```

Using `return` is unnecessary, though, since the function will return automatically after its last statement has been executed.

If a non-`void` function reaches the end of its body—that is, it fails to execute a `return` statement—the behavior of the program is undefined if it attempts to use the value returned by the function. Some compilers will issue a warning such as *"control reaches end of non-void function"* if they detect the possibility of a non-`void` function "falling off" the end of its body.

9.5 Program Termination

Since `main` is a function, it must have a return type. Normally, the return type of `main` is `int`, which is why the programs we've seen so far have defined `main` in the following way:

```
int main(void)
{
  ...
}
```

Older C programs often omit `main`'s return type, taking advantage of the fact that it traditionally defaults to `int`:

```
main()
{
  ...
}
```

C99 Omitting the return type of a function isn't legal in C99, so it's best to avoid this practice. Omitting the word `void` in `main`'s parameter list remains legal, but—as a matter of style—it's best to be explicit about the fact that `main` has no parameters. (We'll see later that `main` sometimes *does* have two parameters, usually named `argc` and `argv`.)

`argc` and `argv` ➤*13.7*

Q&A The value returned by `main` is a status code that—in some operating systems—can be tested when the program terminates. `main` should return 0 if the program terminates normally; to indicate abnormal termination, `main` should return a value other than 0. (Actually, there's no rule to prevent us from using the return value for other purposes.) It's good practice to make sure that every C program returns a status code, even if there are no plans to use it, since someone running the program later may decide to test it.

The `exit` Function

Executing a `return` statement in `main` is one way to terminate a program. Another is calling the `exit` function, which belongs to `<stdlib.h>`. The argument passed to `exit` has the same meaning as `main`'s return value: both indicate the program's status at termination. To indicate normal termination, we'd pass `0`:

`<stdlib.h>` header ➤*26.2*

```
exit(0);                    /* normal termination */
```

Since `0` is a bit cryptic, C allows us to pass `EXIT_SUCCESS` instead (the effect is the same):

```
exit(EXIT_SUCCESS);    /* normal termination */
```

Passing `EXIT_FAILURE` indicates abnormal termination:

```
exit(EXIT_FAILURE);    /* abnormal termination */
```

`EXIT_SUCCESS` and `EXIT_FAILURE` are macros defined in `<stdlib.h>`. The values of `EXIT_SUCCESS` and `EXIT_FAILURE` are implementation-defined; typical values are 0 and 1, respectively.

As methods of terminating a program, `return` and `exit` are closely related. In fact, the statement

`return` *expression*`;`

in `main` is equivalent to

`exit(`*expression*`);`

The difference between `return` and `exit` is that `exit` causes program termination regardless of which function calls it. The `return` statement causes program termination only when it appears in the `main` function. Some programmers use `exit` exclusively to make it easier to locate all exit points in a program.

9.6 Recursion

A function is ***recursive*** if it calls itself. For example, the following function computes $n!$ recursively, using the formula $n! = n \times (n - 1)!$:

```
int fact(int n)
{
  if (n <= 1)
    return 1;
  else
    return n * fact(n - 1);
}
```

Some programming languages rely heavily on recursion, while others don't even allow it. C falls somewhere in the middle: it allows recursion, but most C programmers don't use it that often.

To see how recursion works, let's trace the execution of the statement

```
i = fact(3);
```

Here's what happens:

`fact(3)` finds that 3 is not less than or equal to 1, so it calls
  `fact(2)`, which finds that 2 is not less than or equal to 1, so it calls
    `fact(1)`, which finds that 1 *is* less than or equal to 1, so it returns 1, causing
  `fact(2)` to return $2 \times 1 = 2$, causing
`fact(3)` to return $3 \times 2 = 6$.

Notice how the unfinished calls of `fact` "pile up" until `fact` is finally passed 1. At that point, the old calls of `fact` begin to "unwind" one by one, until the original call—`fact(3)`—finally returns with the answer, 6.

Here's another example of recursion: a function that computes x^n, using the formula $x^n = x \times x^{n-1}$.

```
int power(int x, int n)
{
  if (n == 0)
    return 1;
  else
    return x * power(x, n - 1);
}
```

The call `power(5, 3)` would be executed as follows:

`power(5, 3)` finds that 3 is not equal to 0, so it calls
  `power(5, 2)`, which finds that 2 is not equal to 0, so it calls
    `power(5, 1)`, which finds that 1 is not equal to 0, so it calls
      `power(5, 0)`, which finds that 0 *is* equal to 0, so it returns 1, causing
    `power(5, 1)` to return $5 \times 1 = 5$, causing
  `power(5, 2)` to return $5 \times 5 = 25$, causing
`power(5, 3)` to return $5 \times 25 = 125$.

Incidentally, we can condense the `power` function a bit by putting a conditional expression in the `return` statement:

```
int power(int x, int n)
{
  return n == 0 ? 1 : x * power(x, n - 1);
}
```

Both `fact` and `power` are careful to test a "termination condition" as soon as they're called. When `fact` is called, it immediately checks whether its parameter is less than or equal to 1. When `power` is called, it first checks whether its second parameter is equal to 0. All recursive functions need some kind of termination condition in order to prevent infinite recursion.

The Quicksort Algorithm

At this point, you may wonder why we're bothering with recursion; after all, neither `fact` nor `power` really needs it. Well, you've got a point. Neither function makes much of a case for recursion, because each calls itself just once. Recursion is much more helpful for sophisticated algorithms that require a function to call itself two or more times.

In practice, recursion often arises naturally as a result of an algorithm design technique known as ***divide-and-conquer***, in which a large problem is divided into smaller pieces that are then tackled by the same algorithm. A classic example of the divide-and-conquer strategy can be found in the popular sorting algorithm known as ***Quicksort***. The Quicksort algorithm goes as follows (for simplicity, we'll assume that the array being sorted is indexed from 1 to n):

1. Choose an array element e (the "partitioning element"), then rearrange the array so that elements $1, \ldots, i-1$ are less than or equal to e, element i contains e, and elements $i+1, \ldots, n$ are greater than or equal to e.
2. Sort elements $1, \ldots, i-1$ by using Quicksort recursively.
3. Sort elements $i+1, \ldots, n$ by using Quicksort recursively.

After step 1, the element e is in its proper location. Since the elements to the left of e are all less than or equal to it, they'll be in their proper places once they've been sorted in step 2; similar reasoning applies to the elements to the right of e.

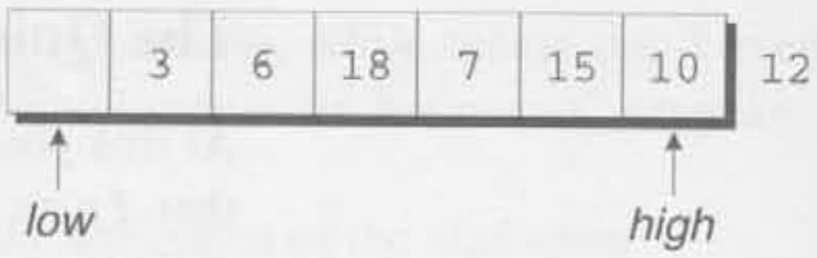
Step 1 of the Quicksort algorithm is obviously critical. There are various methods to partition an array, some much better than others. We'll use a technique

that's easy to understand but not particularly efficient. I'll first describe the partitioning algorithm informally; later, we'll translate it into C code.

The algorithm relies on two "markers" named *low* and *high*, which keep track of positions within the array. Initially, *low* points to the first element of the array and *high* points to the last element. We start by copying the first element (the partitioning element) into a temporary location elsewhere, leaving a "hole" in the array. Next, we move *high* across the array from right to left until it points to an element that's smaller than the partitioning element. We then copy the element into the hole that *low* points to, which creates a new hole (pointed to by *high*). We now move *low* from left to right, looking for an element that's larger than the partitioning element. When we find one, we copy it into the hole that *high* points to. The process repeats, with *low* and *high* taking turns, until they meet somewhere in the middle of the array. At that time, both will point to a hole; all we need do is copy the partitioning element into the hole. The following diagrams illustrate how Quicksort would sort an array of integers:

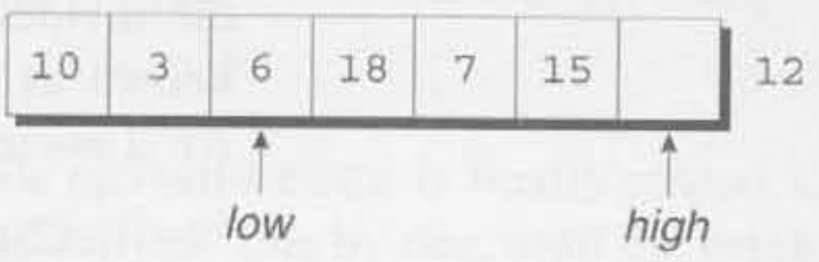
Let's start with an array containing seven elements. *low* points to the first element; *high* points to the last one.

The first element, 12, is the partitioning element. Copying it somewhere else leaves a hole at the beginning of the array.

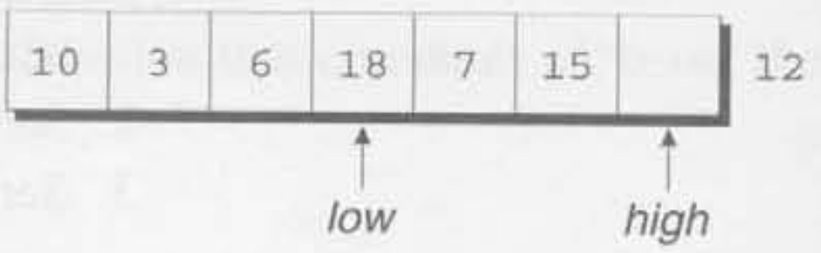


We now compare the element pointed to by *high* with 12. Since 10 is smaller than 12, it's on the wrong side of the array, so we move it to the hole and shift *low* to the right.

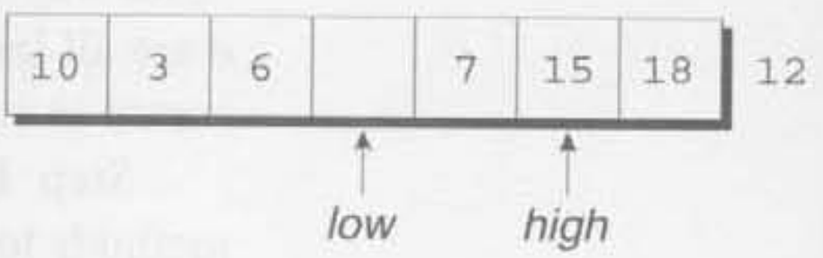
low points to the number 3, which is less than 12 and therefore doesn't need to be moved. We shift *low* to the right instead.



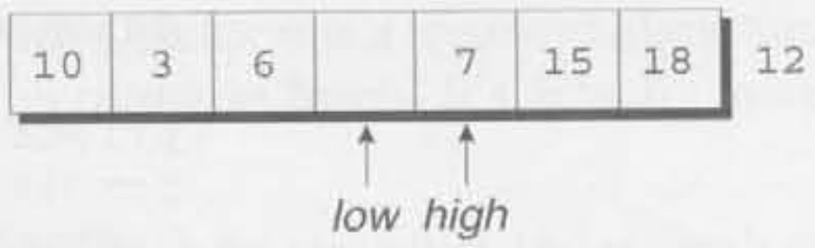
Since 6 is also less than 12, we shift *low* again.



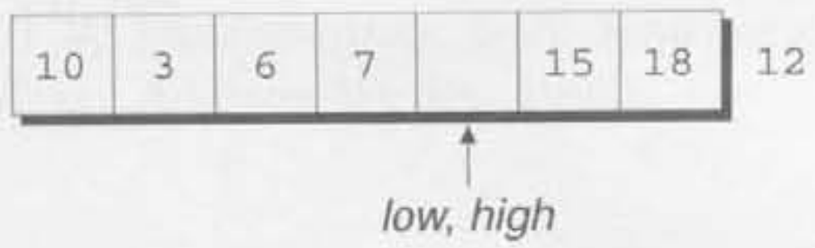
low now points to 18, which is larger than 12 and therefore out of position. After moving 18 to the hole, we shift *high* to the left.



high points to 15, which is greater than 12 and thus doesn't need to be moved. We shift *high* to the left and continue.



high points to 7, which is out of position. After moving 7 to the hole, we shift *low* to the right.



low and *high* are now equal, so we move the partitioning element to the hole.

At this point, we've accomplished our objective: all elements to the left of the partitioning element are less than or equal to 12, and all elements to the right are greater than or equal to 12. Now that the array has been partitioned, we can use Quicksort recursively to sort the first four elements of the array (10, 3, 6, and 7) and the last two (15 and 18).

PROGRAM Quicksort

Let's develop a recursive function named `quicksort` that uses the Quicksort algorithm to sort an array of integers. To test the function, we'll have `main` read 10 numbers into an array, call `quicksort` to sort the array, then print the elements in the array:

```
Enter 10 numbers to be sorted: 9 16 47 82 4 66 12 3 25 51
In sorted order: 3 4 9 12 16 25 47 51 66 82
```

Since the code for partitioning the array is a bit lengthy, I'll put it in a separate function named `split`.

qsort.c

```
/* Sorts an array of integers using Quicksort algorithm */

#include <stdio.h>

#define N 10

void quicksort(int a[], int low, int high);
int split(int a[], int low, int high);

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

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

```
  quicksort(a, 0, N - 1);

  printf("In sorted order: ");
  for (i = 0; i < N; i++)
    printf("%d ", a[i]);
  printf("\n");

  return 0;
}

void quicksort(int a[], int low, int high)
{
  int middle;

  if (low >= high) return;
  middle = split(a, low, high);
  quicksort(a, low, middle - 1);
  quicksort(a, middle + 1, high);
}

int split(int a[], int low, int high)
{
  int part_element = a[low];

  for (;;) {
    while (low < high && part_element <= a[high])
      high--;
    if (low >= high) break;
    a[low++] = a[high];

    while (low < high && a[low] <= part_element)
      low++;
    if (low >= high) break;
    a[high--] = a[low];
  }

  a[high] = part_element;
  return high;
}
```

Although this version of Quicksort works, it's not the best. There are numerous ways to improve the program's performance, including:

- ***Improving the partitioning algorithm.*** Our method isn't the most efficient. Instead of choosing the first element in the array as the partitioning element, it's better to take the median of the first element, the middle element, and the last element. The partitioning process itself can also be sped up. In particular, it's possible to avoid the `low < high` tests in the two `while` loops.
- ***Using a different method to sort small arrays.*** Instead of using Quicksort recursively all the way down to arrays with one element, it's better to use a simpler method for small arrays (those with fewer than, say, 25 elements).

- ***Making Quicksort nonrecursive.*** Although Quicksort is a recursive algorithm by nature—and is easiest to understand in recursive form—it's actually more efficient if the recursion is removed.

For details about improving Quicksort, consult a book on algorithm design, such as Robert Sedgewick's *Algorithms in C, Parts 1–4: Fundamentals, Data Structures, Sorting, Searching,* Third Edition (Boston, Mass.: Addison-Wesley, 1998).

Q & A

Q: Some C books appear to use terms other than *parameter* and *argument*. Is there any standard terminology? [p. 184]

A: As with many other aspects of C, there's no general agreement on terminology, although the C89 and C99 standards use *parameter* and *argument*. The following table should help you translate:

This book:	*Other books:*
parameter	formal argument, formal parameter
argument	actual argument, actual parameter

Keep in mind that—when no confusion would result—I sometimes deliberately blur the distinction between the two terms, using *argument* to mean either.

Q: I've seen programs in which parameter types are specified in separate declarations after the parameter list, as in the following example:

```
double average(a, b)
double a, b;
{
  return (a + b) / 2;
}
```

Is this practice legal? [p. 188]

A: This method of defining functions comes from K&R C, so you may encounter it in older books and programs. C89 and C99 support this style so that older programs will still compile. I'd avoid using it in new programs, however, for a couple of reasons.

First, functions that are defined in the older way aren't subject to the same degree of error-checking. When a function is defined in the older way—and no prototype is present—the compiler won't check that the function is called with the right number of arguments, nor will it check that the arguments have the proper types. Instead, it will perform the default argument promotions.

default argument promotions ➤9.3

Second, the C standard says that the older style is "obsolescent," meaning that its use is discouraged and that it may be dropped from C eventually.

Q: Some programming languages allow procedures and functions to be nested within each other. Does C allow function definitions to be nested?

A: No. C does not permit the definition of one function to appear in the body of another. Among other things, this restriction simplifies the compiler.

***Q: Why does the compiler allow the use of function names that aren't followed by parentheses? [p. 189]**

A: We'll see in a later chapter that the compiler treats a function name not followed by parentheses as a *pointer* to the function. Pointers to functions have legitimate uses, so the compiler can't automatically assume that a function name without parentheses is an error. The statement

pointers to functions ➤17.7

```
print_pun;
```

is legal because the compiler treats `print_pun` as a pointer and therefore an expression, making this a valid (although pointless) expression statement.

expression statements ➤4.5

***Q: In the function call `f(a, b)`, how does the compiler know whether the comma is punctuation or whether it's an operator?**

A: It turns out that the arguments in a function call can't be arbitrary expressions. Instead, they must be "assignment expressions," which can't contain commas used as operators unless they're enclosed in parentheses. In other words, in the call `f(a, b)` the comma is punctuation; in the call `f((a, b))` it's an operator.

Q: Do the names of parameters in a function prototype have to match the names given later in the function's definition? [p. 192]

A: No. Some programmers take advantage of this fact by giving long names to parameters in the prototype, then using shorter names in the actual definition. Or a French-speaking programmer might use English names in prototypes, then switch to more familiar French names in function definitions.

Q: I still don't understand why we bother with function prototypes. If we just put definitions of all the functions before `main`, we're covered, right?

A: Wrong. First, you're assuming that only `main` calls the other functions, which is unrealistic. In practice, some of the functions will call each other. If we put all function definitions above `main`, we'll have to watch their order carefully. Calling a function that hasn't been defined yet can lead to big problems.

But that's not all. Suppose that two functions call each other (which isn't as far-fetched as it may sound). No matter which function we define first, it will end up calling a function that hasn't been defined yet.

But there's still more! Once programs reach a certain size, it won't be feasible to put all the functions in one file anymore. When we reach that point, we'll need prototypes to tell the compiler about functions in other files.

Q: I've seen function declarations that omit all information about parameters:

```
double average();
```

Is this practice legal? [p. 192]

A: Yes. This declaration informs the compiler that `average` returns a `double` value but provides no information about the number and types of its parameters. (Leaving the parentheses empty doesn't necessarily mean that `average` has no parameters.)

In K&R C, this form of function declaration is the only one allowed; the form that we've been using—the function prototype, in which parameter information *is* included—was introduced in C89. The older kind of function declaration is now obsolescent, although still allowed.

Q: Why would a programmer deliberately omit parameter names in a function prototype? Isn't it easier to just leave the names? [p. 193]

A: Omitting parameter names in prototypes is typically done for defensive purposes. If a macro happens to have the same name as a parameter, the parameter name will be replaced during preprocessing, thereby damaging the prototype in which it appears. This isn't likely to be a problem in a small program written by one person but can occur in large applications written by many people.

Q: Is it legal to put a function declaration inside the body of another function?

A: Yes. Here's an example:

```
int main(void)
{
  double average(double a, double b);
  …
}
```

This declaration of `average` is valid only for the body of `main`; if other functions need to call `average`, they'll each have to declare it.

The advantage of this practice is that it's clearer to the reader which functions call which other functions. (In this example, we see that `main` will be calling `average`.) On the other hand, it can be a nuisance if several functions need to call the same function. Even worse, trying to add and remove declarations during program maintenance can be a real pain. For these reasons, I'll always put function declarations outside function bodies.

Q: If several functions have the same return type, can their declarations be combined? For example, since both `print_pun` and `print_count` have `void` as their return type, is the following declaration legal?

```
void print_pun(void), print_count(int n);
```

A: Yes. In fact, C even allows us to combine function declarations with variable declarations:

```
double x, y, average(double a, double b);
```

Combining declarations in this way usually isn't a good idea, though; it can easily cause confusion.

Q: What happens if I specify a length for a one-dimensional array parameter? [p. 195]

A: The compiler ignores it. Consider the following example:

```
double inner_product(double v[3], double w[3]);
```

Other than documenting that `inner_product`'s arguments are supposed to be arrays of length 3, specifying a length doesn't buy us much. The compiler won't check that the arguments actually have length 3, so there's no added security. In fact, the practice is misleading in that it suggests that `inner_product` can only be passed arrays of length 3, when in fact we can pass arrays of arbitrary length.

***Q: Why can the first dimension in an array parameter be left unspecified, but not the other dimensions? [p. 197]**

A: First, we need to discuss how arrays are passed in C. As Section 12.3 explains, when an array is passed to a function, the function is given a *pointer* to the first element in the array.

Next, we need to know how the subscripting operator works. Suppose that `a` is a one-dimensional array passed to a function. When we write

```
a[i] = 0;
```

the compiler generates instructions that compute the address of `a[i]` by multiplying `i` by the size of an array element and adding the result to the address that `a` represents (the pointer passed to the function). This calculation doesn't depend on the length of `a`, which explains why we can omit it when defining the function.

What about multidimensional arrays? Recall that C stores arrays in row-major order, with the elements in row 0 stored first, then the elements in row 1, and so forth. Suppose that `a` is a two-dimensional array parameter and we write

```
a[i][j] = 0;
```

The compiler generates instructions to do the following: (1) multiply `i` by the size of a single row of `a`; (2) add this result to the address that `a` represents; (3) multiply `j` by the size of an array element; and (4) add this result to the address computed in step 2. To generate these instructions, the compiler must know the size of a row in the array, which is determined by the number of columns. The bottom line: the programmer must declare the number of columns in `a`.

Q: Why do some programmers put parentheses around the expression in a `return` statement?

A: The examples in the first edition of Kernighan and Ritchie's *The C Programming Language* always have parentheses in `return` statements, even though they aren't required. Programmers (and authors of subsequent books) picked up the habit from K&R. I don't use these parentheses, since they're unnecessary and

contribute nothing to readability. (Kernighan and Ritchie apparently agree: the `return` statements in the second edition of *The C Programming Language* lack parentheses.)

Q: What happens if a non-`void` function attempts to execute a `return` statement that has no expression? [p. 202]

A: That depends on the version of C. In C89, executing a `return` statement without an expression in a non-`void` function causes undefined behavior (but only if the program attempts to use the value returned by the function). In C99, such a statement is illegal and should be detected as an error by the compiler.

Q: How can I test `main`'s return value to see if a program has terminated normally? [p. 203]

A: That depends on your operating system. Many operating systems allow this value to be tested within a "batch file" or "shell script" that contains commands to run several programs. For example, the line

```
if errorlevel 1 command
```

in a Windows batch file will execute *command* if the last program terminated with a status code greater than or equal to 1.

In UNIX, each shell has its own method for testing the status code. In the Bourne shell, the variable `$?` contains the status of the last program run. The C shell has a similar variable, but its name is `$status`.

Q: Why does my compiler produce a *"control reaches end of non-void function"* warning when it compiles `main`?

A: The compiler has noticed that `main`, despite having `int` as its return type, doesn't have a `return` statement. Putting the statement

```
return 0;
```

at the end of `main` will keep the compiler happy. Incidentally, this is good practice even if your compiler doesn't object to the lack of a `return` statement.

C99 When a program is compiled using a C99 compiler, this warning shouldn't occur. In C99, it's OK to "fall off" the end of `main` without returning a value; the standard states that `main` automatically returns 0 in this situation.

Q: With regard to the previous question: Why not just define `main`'s return type to be `void`?

A: Although this practice is fairly common, it's illegal according to the C89 standard. Even if it weren't illegal, it wouldn't be a good idea, since it presumes that no one will ever test the program's status upon termination.

C99 C99 opens the door to legalizing this practice, by allowing `main` to be declared "in some other implementation-defined manner" (with a return type other than `int` or parameters other than those specified by the standard). However, any such usage isn't portable, so it's best to declare `main`'s return type to be `int`.

Q: Is it legal for a function `f1` to call a function `f2`, which then calls `f1`?

A: Yes. This is just an indirect form of recursion in which one call of `f1` leads to another. (But make sure that either `f1` or `f2` eventually terminates!)

Exercises

Section 9.1

1. The following function, which computes the area of a triangle, contains two errors. Locate the errors and show how to fix them. (*Hint:* There are no errors in the formula.)

```
double triangle_area(double base, height)
double product;
{
  product = base * height;
  return product / 2;
}
```

W 2. Write a function `check(x, y, n)` that returns 1 if both `x` and `y` fall between 0 and `n` – 1, inclusive. The function should return 0 otherwise. Assume that `x`, `y`, and `n` are all of type `int`.

3. Write a function `gcd(m, n)` that calculates the greatest common divisor of the integers `m` and `n`. (Programming Project 2 in Chapter 6 describes Euclid's algorithm for computing the GCD.)

W 4. Write a function `day_of_year(month, day, year)` that returns the day of the year (an integer between 1 and 366) specified by the three arguments.

5. Write a function `num_digits(n)` that returns the number of digits in `n` (a positive integer). *Hint:* To determine the number of digits in a number *n*, divide it by 10 repeatedly. When *n* reaches 0, the number of divisions indicates how many digits *n* originally had.

W 6. Write a function `digit(n, k)` that returns the k^{th} digit (from the right) in `n` (a positive integer). For example, `digit(829, 1)` returns 9, `digit(829, 2)` returns 2, and `digit(829, 3)` returns 8. If `k` is greater than the number of digits in `n`, have the function return 0.

7. Suppose that the function `f` has the following definition:

```
int f(int a, int b) { ... }
```

Which of the following statements are legal? (Assume that `i` has type `int` and `x` has type `double`.)

```
(a) i = f(83, 12);
(b) x = f(83, 12);
(c) i = f(3.15, 9.28);
(d) x = f(3.15, 9.28);
(e) f(83, 12);
```

Section 9.2

W 8. Which of the following would be valid prototypes for a function that returns nothing and has one `double` parameter?

```
(a) void f(double x);
```

(b) `void f(double);`
(c) `void f(x);`
(d) `f(double x);`

Section 9.3

*9. What will be the output of the following program?

```
#include <stdio.h>

void swap(int a, int b);

int main(void)
{
  int i = 1, j = 2;

  swap(i, j);
  printf("i = %d, j = %d\n", i, j);
  return 0;
}

void swap(int a, int b)
{
  int temp = a;
  a = b;
  b = temp;
}
```

W 10. Write functions that return the following values. (Assume that `a` and `n` are parameters, where `a` is an array of `int` values and `n` is the length of the array.)
(a) The largest element in `a`.
(b) The average of all elements in `a`.
(c) The number of positive elements in `a`.

11. Write the following function:

```
float compute_GPA(char grades[], int n);
```

The `grades` array will contain letter grades (`A`, `B`, `C`, `D`, or `F`, either upper-case or lower-case); `n` is the length of the array. The function should return the average of the grades (assume that `A` = 4, `B` = 3, `C` = 2, `D` = 1, and `F` = 0).

12. Write the following function:

```
double inner_product(double a[], double b[], int n);
```

The function should return `a[0] * b[0] + a[1] * b[1] + ... + a[n-1] * b[n-1]`.

13. Write the following function, which evaluates a chess position:

```
int evaluate_position(char board[8][8]);
```

`board` represents a configuration of pieces on a chessboard, where the letters `K`, `Q`, `R`, `B`, `N`, `P` represent White pieces, and the letters `k`, `q`, `r`, `b`, `n`, and `p` represent Black pieces. `evaluate_position` should sum the values of the White pieces (`Q` = 9, `R` = 5, `B` = 3, `N` = 3, `P` = 1). It should also sum the values of the Black pieces (done in a similar way). The function will return the difference between the two numbers. This value will be positive if White has an advantage in material and negative if Black has an advantage.

Section 9.4

14. The following function is supposed to return `true` if any element of the array `a` has the value 0 and `false` if all elements are nonzero. Sadly, it contains an error. Find the error and show how to fix it:

```
bool has_zero(int a[], int n)
{
  int i;

  for (i = 0; i < n; i++)
    if (a[i] == 0)
      return true;
    else
      return false;
}
```

W 15. The following (rather confusing) function finds the median of three numbers. Rewrite the function so that it has just one `return` statement.

```
double median(double x, double y, double z)
{
  if (x <= y)
    if (y <= z) return y;
    else if (x <= z) return z;
    else return x;
  if (z <= y) return y;
  if (x <= z) return x;
  return z;
}
```

Section 9.6

16. Condense the `fact` function in the same way we condensed `power`.

W 17. Rewrite the `fact` function so that it's no longer recursive.

18. Write a recursive version of the `gcd` function (see Exercise 3). Here's the strategy to use for computing `gcd(m, n)`: If `n` is 0, return `m`. Otherwise, call `gcd` recursively, passing `n` as the first argument and `m % n` as the second.

W *19. Consider the following "mystery" function:

```
void pb(int n)
{
  if (n != 0) {
    pb(n / 2);
    putchar('0' + n % 2);
  }
}
```

Trace the execution of the function by hand. Then write a program that calls the function, passing it a number entered by the user. What does the function do?

Programming Projects

1. Write a program that asks the user to enter a series of integers (which it stores in an array), then sorts the integers by calling the function `selection_sort`. When given an array with n elements, `selection_sort` must do the following:

   1. Search the array to find the largest element, then move it to the last position in the array.
   2. Call itself recursively to sort the first $n - 1$ elements of the array.

2. Modify Programming Project 5 from Chapter 5 so that it uses a function to compute the amount of income tax. When passed an amount of taxable income, the function will return the tax due.

3. Modify Programming Project 9 from Chapter 8 so that it includes the following functions:

```
void generate_random_walk(char walk[10][10]);
void print_array(char walk[10][10]);
```

`main` first calls `generate_random_walk`, which initializes the array to contain `'.'` characters and then replaces some of these characters by the letters `A` through `Z`, as described in the original project. `main` then calls `print_array` to display the array on the screen.

4. Modify Programming Project 16 from Chapter 8 so that it includes the following functions:

```
void read_word(int counts[26]);
bool equal_array(int counts1[26], int counts2[26]);
```

`main` will call `read_word` twice, once for each of the two words entered by the user. As it reads a word, `read_word` will use the letters in the word to update the `counts` array, as described in the original project. (`main` will declare two arrays, one for each word. These arrays are used to track how many times each letter occurs in the words.) `main` will then call `equal_array`, passing it the two arrays. `equal_array` will return `true` if the elements in the two arrays are identical (indicating that the words are anagrams) and `false` otherwise.

5. Modify Programming Project 17 from Chapter 8 so that it includes the following functions:

```
void create_magic_square(int n, char magic_square[n][n]);
void print_magic_square(int n, char magic_square[n][n]);
```

After obtaining the number n from the user, `main` will call `create_magic_square`, passing it an $n \times n$ array that is declared inside `main`. `create_magic_square` will fill the array with the numbers 1, 2, ..., n^2 as described in the original project. `main` will then call `print_magic_square`, which will display the array in the format described in the original project. *Note:* If your compiler doesn't support variable-length arrays, declare the array in `main` to be 99×99 instead of $n \times n$ and use the following prototypes instead:

```
void create_magic_square(int n, char magic_square[99][99]);
void print_magic_square(int n, char magic_square[99][99]);
```

6. Write a function that computes the value of the following polynomial:

$3x^5 + 2x^4 - 5x^3 - x^2 + 7x - 6$

Write a program that asks the user to enter a value for x, calls the function to compute the value of the polynomial, and then displays the value returned by the function.

7. The `power` function of Section 9.6 can be made faster by having it calculate x^n in a different way. We first notice that if n is a power of 2, then x^n can be computed by squaring. For example, x^4 is the square of x^2, so x^4 can be computed using only two multiplications instead of three. As it happens, this technique can be used even when n is not a power of 2. If n is even, we use the formula $x^n = (x^{n/2})^2$. If n is odd, then $x^n = x \times x^{n-1}$. Write a recursive function that computes x^n. (The recursion ends when $n = 0$, in which case the function returns 1.) To test your function, write a program that asks the user to enter values for x and n, calls `power` to compute x^n, and then displays the value returned by the function.

8. Write a program that simulates the game of craps, which is played with two dice. On the first roll, the player wins if the sum of the dice is 7 or 11. The player loses if the sum is 2, 3,

or 12. Any other roll is called the "point" and the game continues. On each subsequent roll, the player wins if he or she rolls the point again. The player loses by rolling 7. Any other roll is ignored and the game continues. At the end of each game, the program will ask the user whether or not to play again. When the user enters a response other than `y` or `Y`, the program will display the number of wins and losses and then terminate.

```
You rolled: 8
Your point is 8
You rolled: 3
You rolled: 10
You rolled: 8
You win!

Play again? y

You rolled: 6
Your point is 6
You rolled: 5
You rolled: 12
You rolled: 3
You rolled: 7
You lose!

Play again? y

You rolled: 11
You win!

Play again? n

Wins: 2  Losses: 1
```

Write your program as three functions: `main`, `roll_dice`, and `play_game`. Here are the prototypes for the latter two functions:

```
int roll_dice(void);
bool play_game(void);
```

`roll_dice` should generate two random numbers, each between 1 and 6, and return their sum. `play_game` should play one craps game (calling `roll_dice` to determine the outcome of each dice roll); it will return `true` if the player wins and `false` if the player loses. `play_game` is also responsible for displaying messages showing the results of the player's dice rolls. `main` will call `play_game` repeatedly, keeping track of the number of wins and losses and displaying the "you win" and "you lose" messages. *Hint:* Use the `rand` function to generate random numbers. See the `deal.c` program in Section 8.2 for an example of how to call `rand` and the related `srand` function.

10 Program Organization

As Will Rogers would have said, "There is no such thing as a free variable."

Having covered functions in Chapter 9, we're ready to confront several issues that arise when a program contains more than one function. The chapter begins with a discussion of the differences between local variables (Section 10.1) and external variables (Section 10.2). Section 10.3 then considers blocks (compound statements containing declarations). Section 10.4 tackles the scope rules that apply to local names, external names, and names declared in blocks. Finally, Section 10.5 suggests a way to organize function prototypes, function definitions, variable declarations, and the other parts of a C program.

10.1 Local Variables

A variable declared in the body of a function is said to be ***local*** to the function. In the following function, `sum` is a local variable:

```
int sum_digits(int n)
{
  int sum = 0;    /* local variable */

  while (n > 0) {
    sum += n % 10;
    n /= 10;
  }

  return sum;
}
```

By default, local variables have the following properties:

- ***Automatic storage duration.*** The ***storage duration*** (or ***extent***) of a variable is the portion of program execution during which storage for the variable exists. Storage for a local variable is "automatically" allocated when the enclosing function is called and deallocated when the function returns, so the variable is said to have ***automatic storage duration.*** A local variable doesn't retain its value when its enclosing function returns. When the function is called again, there's no guarantee that the variable will still have its old value.
- ***Block scope.*** The ***scope*** of a variable is the portion of the program text in which the variable can be referenced. A local variable has ***block scope:*** it is visible from its point of declaration to the end of the enclosing function body. Since the scope of a local variable doesn't extend beyond the function to which it belongs, other functions can use the same name for other purposes.

Section 18.2 covers these and other related concepts in more detail.

C99 Since C99 doesn't require variable declarations to come at the beginning of a function, it's possible for a local variable to have a very small scope. In the following example, the scope of `i` doesn't begin until the line on which it's declared, which could be near the end of the function body:

```
void f(void)
{
  ...
  int i;
  ...
}
```

scope of `i`

Static Local Variables

Putting the word `static` in the declaration of a local variable causes it to have ***static storage duration*** instead of automatic storage duration. A variable with static storage duration has a permanent storage location, so it retains its value throughout the execution of the program. Consider the following function:

```
void f(void)
{
  static int i;    /* static local variable */
  ...
}
```

Since the local variable `i` has been declared `static`, it occupies the same memory location throughout the execution of the program. When `f` returns, `i` won't

Q&A lose its value.

A static local variable still has block scope, so it's not visible to other functions. In a nutshell, a static variable is a place to hide data from other functions but retain it for future calls of the same function.

Parameters

Parameters have the same properties—automatic storage duration and block scope—as local variables. In fact, the only real difference between parameters and local variables is that each parameter is initialized automatically when a function is called (by being assigned the value of the corresponding argument).

10.2 External Variables

Passing arguments is one way to transmit information to a function. Functions can also communicate through ***external variables***—variables that are declared outside the body of any function.

The properties of external variables (or ***global variables,*** as they're sometimes called) are different from those of local variables:

- ***Static storage duration.*** External variables have static storage duration, just like local variables that have been declared `static`. A value stored in an external variable will stay there indefinitely.
- ***File scope.*** An external variable has ***file scope:*** it is visible from its point of declaration to the end of the enclosing file. As a result, an external variable can be accessed (and potentially modified) by all functions that follow its declaration.

Example: Using External Variables to Implement a Stack

To illustrate how external variables might be used, let's look at a data structure known as a ***stack***. (Stacks are an abstract concept, not a C feature; they can be implemented in most programming languages.) A stack, like an array, can store multiple data items of the same type. However, the operations on a stack are limited: we can either ***push*** an item onto the stack (add it to one end—the "stack top") or ***pop*** it from the stack (remove it from the same end). Examining or modifying an item that's not at the top of the stack is forbidden.

One way to implement a stack in C is to store its items in an array, which we'll call `contents`. A separate integer variable named `top` marks the position of the stack top. When the stack is empty, `top` has the value 0. To push an item on the stack, we simply store the item in `contents` at the position indicated by `top`, then increment `top`. Popping an item requires decrementing `top`, then using it as an index into `contents` to fetch the item that's being popped.

Based on this outline, here's a program fragment (not a complete program) that declares the `contents` and `top` variables for a stack and provides a set of functions that represent operations on the stack. All five functions need access to the `top` variable, and two functions need access to `contents`, so we'll make `contents` and `top` external.

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

#define STACK_SIZE 100

/* external variables */
int contents[STACK_SIZE];
int top = 0;

void make_empty(void)
{
  top = 0;
}

bool is_empty(void)
{
  return top == 0;
}

bool is_full(void)
{
  return top == STACK_SIZE;
}

void push(int i)
{
  if (is_full())
    stack_overflow();
  else
    contents[top++] = i;
}

int pop(void)
{
  if (is_empty())
    stack_underflow();
  else
    return contents[--top];
}
```

Pros and Cons of External Variables

External variables are convenient when many functions must share a variable or when a few functions share a large number of variables. In most cases, however, it's better for functions to communicate through parameters rather than by sharing variables. Here's why:

- If we change an external variable during program maintenance (by altering its type, say), we'll need to check every function in the same file to see how the change affects it.

- If an external variable is assigned an incorrect value, it may be difficult to identify the guilty function. It's like trying to solve a murder committed at a crowded party—there's no easy way to narrow the list of suspects.
- Functions that rely on external variables are hard to reuse in other programs. A function that depends on external variables isn't self-contained; to reuse the function, we'll have to drag along any external variables that it needs.

Many C programmers rely far too much on external variables. One common abuse: using the same external variable for different purposes in different functions. Suppose that several functions need a variable named `i` to control a `for` statement. Instead of declaring `i` in each function that uses it, some programmers declare it at the top of the program, thereby making the variable visible to all functions. This practice is poor not only for the reasons listed earlier, but also because it's misleading; someone reading the program later may think that the uses of the variable are related, when in fact they're not.

When you use external variables, make sure they have meaningful names. (Local variables don't always need meaningful names: it's often hard to think of a better name than `i` for the control variable in a `for` loop.) If you find yourself using names like `i` and `temp` for external variables, that's a clue that perhaps they should really be local variables.

Making variables external when they should be local can lead to some rather frustrating bugs. Consider the following example, which is supposed to display a 10 × 10 arrangement of asterisks:

```
int i;

void print_one_row(void)
{
  for (i = 1; i <= 10; i++)
    printf("*");
}

void print_all_rows(void)
{
  for (i = 1; i <= 10; i++) {
    print_one_row();
    printf("\n");
  }
}
```

Instead of printing 10 rows, `print_all_rows` prints only one row. When `print_one_row` returns after being called the first time, `i` will have the value 11. The `for` statement in `print_all_rows` then increments `i` and tests whether it's less than or equal to 10. It's not, so the loop terminates and the function returns.

PROGRAM

Guessing a Number

To get more experience with external variables, we'll write a simple game-playing program. The program generates a random number between 1 and 100, which the user attempts to guess in as few tries as possible. Here's what the user will see when the program is run:

```
Guess the secret number between 1 and 100.

A new number has been chosen.
Enter guess: 55
Too low; try again.
Enter guess: 65
Too high; try again.
Enter guess: 60
Too high; try again.
Enter guess: 58
You won in 4 guesses!

Play again? (Y/N) y

A new number has been chosen.
Enter guess: 78
Too high; try again.
Enter guess: 34
You won in 2 guesses!

Play again? (Y/N) n
```

This program will need to carry out several different tasks: initializing the random number generator, choosing a secret number, and interacting with the user until the correct number is picked. If we write a separate function to handle each task, we might end up with the following program.

guess.c

```
/* Asks user to guess a hidden number */

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define MAX_NUMBER 100

/* external variable */
int secret_number;

/* prototypes */
void initialize_number_generator(void);
void choose_new_secret_number(void);
void read_guesses(void);

int main(void)
{
  char command;
```

```
  printf("Guess the secret number between 1 and %d.\n\n",
         MAX_NUMBER);
  initialize_number_generator();
  do {
    choose_new_secret_number();
    printf("A new number has been chosen.\n");
    read_guesses();
    printf("Play again? (Y/N) ");
    scanf(" %c", &command);
    printf("\n");
  } while (command == 'y' || command == 'Y');

  return 0;
}

/**********************************************************
 * initialize_number_generator: Initializes the random    *
 *                              number generator using    *
 *                              the time of day.          *
 **********************************************************/
void initialize_number_generator(void)
{
  srand((unsigned) time(NULL));
}

/**********************************************************
 * choose_new_secret_number: Randomly selects a number    *
 *                           between 1 and MAX_NUMBER and *
 *                           stores it in secret_number.  *
 **********************************************************/
void choose_new_secret_number(void)
{
  secret_number = rand() % MAX_NUMBER + 1;
}

/**********************************************************
 * read_guesses: Repeatedly reads user guesses and tells  *
 *               the user whether each guess is too low,  *
 *               too high, or correct. When the guess is  *
 *               correct, prints the total number of      *
 *               guesses and returns.                     *
 **********************************************************/
void read_guesses(void)
{
  int guess, num_guesses = 0;

  for (;;) {
    num_guesses++;
    printf("Enter guess: ");
    scanf("%d", &guess);
    if (guess == secret_number) {
      printf("You won in %d guesses!\n\n", num_guesses);
      return;
    } else if (guess < secret_number)
```

```
      printf("Too low; try again.\n");
    else
      printf("Too high; try again.\n");
  }
}
```

time function ➤26.3
srand function ➤26.2
rand function ➤26.2

For random number generation, the `guess.c` program relies on the `time`, `srand`, and `rand` functions, which we first used in `deal.c` (Section 8.2). This time, we're scaling the return value of `rand` so that it falls between 1 and `MAX_NUMBER`.

Although `guess.c` works fine, it relies on an external variable. We made `secret_number` external so that both `choose_new_secret_number` and `read_guesses` could access it. If we alter `choose_new_secret_number` and `read_guesses` just a little, we should be able to move `secret_number` into the `main` function. We'll modify `choose_new_secret_number` so that it returns the new number, and we'll rewrite `read_guesses` so that `secret_number` can be passed to it as an argument.

Here's our new program, with changes in **bold**:

guess2.c

```
/* Asks user to guess a hidden number */

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define MAX_NUMBER 100

/* prototypes */
void initialize_number_generator(void);
int new_secret_number(void);
void read_guesses(int secret_number);

int main(void)
{
  char command;
  int secret_number;

  printf("Guess the secret number between 1 and %d.\n\n",
         MAX_NUMBER);
  initialize_number_generator();
  do {
    secret_number = new_secret_number();
    printf("A new number has been chosen.\n");
    read_guesses(secret_number);
    printf("Play again? (Y/N) ");
    scanf(" %c", &command);
    printf("\n");
  } while (command == 'y' || command == 'Y');

  return 0;
}
```

```
/*************************************************************
 * initialize_number_generator: Initializes the random       *
 *                              number generator using       *
 *                              the time of day.             *
 *************************************************************/
void initialize_number_generator(void)
{
  srand((unsigned) time(NULL));
}

/*************************************************************
 * new_secret_number: Returns a randomly chosen number       *
 *                    between 1 and MAX_NUMBER.              *
 *************************************************************/
int new_secret_number(void)
{
  return rand() % MAX_NUMBER + 1;
}

/*************************************************************
 * read_guesses: Repeatedly reads user guesses and tells     *
 *               the user whether each guess is too low,     *
 *               too high, or correct. When the guess is     *
 *               correct, prints the total number of         *
 *               guesses and returns.                        *
 *************************************************************/
void read_guesses(int secret_number)
{
  int guess, num_guesses = 0;

  for (;;) {
    num_guesses++;
    printf("Enter guess: ");
    scanf("%d", &guess);
    if (guess == secret_number) {
      printf("You won in %d guesses!\n\n", num_guesses);
      return;
    } else if (guess < secret_number)
      printf("Too low; try again.\n");
    else
      printf("Too high; try again.\n");
  }
}
```

10.3 Blocks

In Section 5.2, we encountered compound statements of the form

{ *statements* }

It turns out that C allows compound statements to contain declarations as well:

block { *declarations statements* }

I'll use the term ***block*** to describe such a compound statement. Here's an example of a block:

```
if (i > j) {
  /* swap values of i and j */
  int temp = i;
  i = j;
  j = temp;
}
```

By default, the storage duration of a variable declared in a block is automatic: storage for the variable is allocated when the block is entered and deallocated when the block is exited. The variable has block scope; it can't be referenced outside the block. A variable that belongs to a block can be declared `static` to give it static storage duration.

The body of a function is a block. Blocks are also useful inside a function body when we need variables for temporary use. In our last example, we needed a variable temporarily so that we could swap the values of `i` and `j`. Putting temporary variables in blocks has two advantages: (1) It avoids cluttering the declarations at the beginning of the function body with variables that are used only briefly. (2) It reduces name conflicts. In our example, the name `temp` can be used elsewhere in the same function for different purposes—the `temp` variable is strictly local to the block in which it's declared.

C99 C99 allows variables to be declared anywhere within a block, just as it allows variables to be declared anywhere within a function.

10.4 Scope

In a C program, the same identifier may have several different meanings. C's scope rules enable the programmer (and the compiler) to determine which meaning is relevant at a given point in the program.

Here's the most important scope rule: When a declaration inside a block names an identifier that's already visible (because it has file scope or because it's declared in an enclosing block), the new declaration temporarily "hides" the old one, and the identifier takes on a new meaning. At the end of the block, the identifier regains its old meaning.

Consider the (somewhat extreme) example at the top of the next page, in which the identifier `i` has four different meanings:

- In Declaration 1, `i` is a variable with static storage duration and file scope.

```
int i;                  /* Declaration 1 */

void f(int i)           /* Declaration 2 */
{
  i = 1;
}

void g(void)
{
  int i = 2;            /* Declaration 3 */
  if (i > 0) {
    int i;              /* Declaration 4 */
    i = 3;
  }
  i = 4;
}

void h(void)
{
  i = 5;
}
```

- In Declaration 2, `i` is a parameter with block scope.
- In Declaration 3, `i` is an automatic variable with block scope.
- In Declaration 4, `i` is also automatic and has block scope.

`i` is used five times. C's scope rules allow us to determine the meaning of `i` in each case:

- The `i = 1` assignment refers to the parameter in Declaration 2, not the variable in Declaration 1, since Declaration 2 hides Declaration 1.
- The `i > 0` test refers to the variable in Declaration 3, since Declaration 3 hides Declaration 1 and Declaration 2 is out of scope.
- The `i = 3` assignment refers to the variable in Declaration 4, which hides Declaration 3.
- The `i = 4` assignment refers to the variable in Declaration 3. It can't refer to Declaration 4, which is out of scope.
- The `i = 5` assignment refers to the variable in Declaration 1.

10.5 Organizing a C Program

Now that we've seen the major elements that make up a C program, it's time to develop a strategy for their arrangement. For now, we'll assume that a program

always fits into a single file. Chapter 15 shows how to organize a program that's split over several files.

So far, we've seen that a program may contain the following:

Preprocessing directives such as `#include` and `#define`
Type definitions
Declarations of external variables
Function prototypes
Function definitions

C imposes only a few rules on the order of these items: A preprocessing directive doesn't take effect until the line on which it appears. A type name can't be used until it's been defined. A variable can't be used until it's declared. Although C isn't as picky about functions, I strongly recommend that every function be defined or declared prior to its first call. (C99 makes this a requirement anyway.)

C99

There are several ways to organize a program so that these rules are obeyed. Here's one possible ordering:

`#include` directives
`#define` directives
Type definitions
Declarations of external variables
Prototypes for functions other than `main`
Definition of `main`
Definitions of other functions

It makes sense to put `#include` directives first, since they bring in information that will likely be needed in several places within the program. `#define` directives create macros, which are generally used throughout the program. Putting type definitions above the declarations of external variables is logical, since the declarations of these variables may refer to the type names just defined. Declaring external variables next makes them available to all the functions that follow. Declaring all functions except for `main` avoids the problems that arise when a function is called before the compiler has seen its prototype. This practice also makes it possible to arrange the function definitions in any order whatsoever: alphabetically by function name or with related functions grouped together, for example. Defining `main` before the other functions makes it easier for a reader to locate the program's starting point.

A final suggestion: Precede each function definition by a boxed comment that gives the name of the function, explains its purpose, discusses the meaning of each parameter, describes its return value (if any), and lists any side effects it has (such as modifying external variables).

PROGRAM Classifying a Poker Hand

To show how a C program might be organized, let's attempt a program that's a little more complex than our previous examples. The program will read and classify

a poker hand. Each card in the hand will have both 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 won't allow the use of jokers, and we'll assume that aces are high. The program will read a hand of five cards, then classify the hand into one of the following categories (listed in order from best to worst):

straight flush (both a straight and a flush)
four-of-a-kind (four cards of the same rank)
full house (a three-of-a-kind and a pair)
flush (five cards of the same suit)
straight (five cards with consecutive ranks)
three-of-a-kind (three cards of the same rank)
two pairs
pair (two cards of the same rank)
high card (any other hand)

If a hand falls into two or more categories, the program will choose the best one.

For input purposes, we'll abbreviate ranks and suits as follows (letters may be either upper- or lower-case):

Ranks: `2 3 4 5 6 7 8 9 t j q k a`
Suits: `c d h s`

If the user enters an illegal card or tries to enter the same card twice, the program will ignore the card, issue an error message, and then request another card. Entering the number 0 instead of a card will cause the program to terminate.

A session with the program will have the following appearance:

```
Enter a card: 2s
Enter a card: 5s
Enter a card: 4s
Enter a card: 3s
Enter a card: 6s
Straight flush

Enter a card: 8c
Enter a card: as
Enter a card: 8c
Duplicate card; ignored.
Enter a card: 7c
Enter a card: ad
Enter a card: 3h
Pair

Enter a card: 6s
Enter a card: d2
Bad card; ignored.
Enter a card: 2d
Enter a card: 9c
Enter a card: 4h
Enter a card: ts
```

```
High card

Enter a card: 0
```

From this description of the program, we see that it has three tasks:

Read a hand of five cards.
Analyze the hand for pairs, straights, and so forth.
Print the classification of the hand.

We'll divide the program into three functions—`read_cards`, `analyze_hand`, and `print_result`—that perform these three tasks. `main` does nothing but call these functions inside an endless loop. The functions will need to share a fairly large amount of information, so we'll have them communicate through external variables. `read_cards` will store information about the hand into several external variables. `analyze_hand` will then examine these variables, storing its findings into other external variables for the benefit of `print_result`.

Based on this preliminary design, we can begin to sketch an outline of the program:

```
/* #include directives go here */

/* #define directives go here */

/* declarations of external variables go here */

/* prototypes */
void read_cards(void);
void analyze_hand(void);
void print_result(void);

/**********************************************************
 * main: Calls read_cards, analyze_hand, and print_result *
 *       repeatedly.                                      *
 **********************************************************/
int main(void)
{
  for (;;) {
    read_cards();
    analyze_hand();
    print_result();
  }
}

/**********************************************************
 * read_cards:  Reads the cards into external variables;  *
 *              checks for bad cards and duplicate cards. *
 **********************************************************/
void read_cards(void)
{
  ...
}
```

```
/**********************************************************
 * analyze_hand: Determines whether the hand contains a   *
 *               straight, a flush, four-of-a-kind,       *
 *               and/or three-of-a-kind; determines the   *
 *               number of pairs; stores the results into *
 *               external variables.                      *
 **********************************************************/
void analyze_hand(void)
{
  ...
}

/**********************************************************
 * print_result: Notifies the user of the result, using   *
 *               the external variables set by            *
 *               analyze_hand.                            *
 **********************************************************/
void print_result(void)
{
  ...
}
```

The most pressing question that remains is how to represent the hand of cards. Let's see what operations `read_cards` and `analyze_hand` will perform on the hand. During the analysis of the hand, `analyze_hand` will need to know how many cards are in each rank and each suit. This suggests that we use two arrays, `num_in_rank` and `num_in_suit`. The value of `num_in_rank[r]` will be the number of cards with rank `r`, and the value of `num_in_suit[s]` will be the number of cards with suit `s`. (We'll encode ranks as numbers between 0 and 12, and suits as numbers between 0 and 3.) We'll also need a third array, `card_exists`, so that `read_cards` can detect duplicate cards. Each time `read_cards` reads a card with rank `r` and suit `s`, it checks whether the value of `card_exists[r][s]` is `true`. If so, the card was previously entered; if not, `read_cards` assigns `true` to `card_exists[r][s]`.

Both the `read_cards` function and the `analyze_hand` function will need access to the `num_in_rank` and `num_in_suit` arrays, so I'll make them external variables. The `card_exists` array is used only by `read_cards`, so it can be local to that function. As a rule, variables should be made external only if necessary.

Having decided on the major data structures, we can now finish the program:

poker.c

```
/* Classifies a poker hand */

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

#define NUM_RANKS 13
#define NUM_SUITS 4
#define NUM_CARDS 5
```

```
/* external variables */
int num_in_rank[NUM_RANKS];
int num_in_suit[NUM_SUITS];
bool straight, flush, four, three;
int pairs;   /* can be 0, 1, or 2 */

/* prototypes */
void read_cards(void);
void analyze_hand(void);
void print_result(void);

/*************************************************************
 * main: Calls read_cards, analyze_hand, and print_result    *
 *       repeatedly.                                         *
 *************************************************************/
int main(void)
{
  for (;;) {
    read_cards();
    analyze_hand();
    print_result();
  }
}

/*************************************************************
 * read_cards: Reads the cards into the external             *
 *             variables num_in_rank and num_in_suit;        *
 *             checks for bad cards and duplicate cards.     *
 *************************************************************/
void read_cards(void)
{
  bool card_exists[NUM_RANKS][NUM_SUITS];
  char ch, rank_ch, suit_ch;
  int rank, suit;
  bool bad_card;
  int cards_read = 0;

  for (rank = 0; rank < NUM_RANKS; rank++) {
    num_in_rank[rank] = 0;
    for (suit = 0; suit < NUM_SUITS; suit++)
      card_exists[rank][suit] = false;
  }

for (suit = 0; suit < NUM_SUITS; suit++)
  num_in_suit[suit] = 0;

while (cards_read < NUM_CARDS) {
  bad_card = false;

  printf("Enter a card: ");

  rank_ch = getchar();
  switch (rank_ch) {
```

```
      case '0':           exit(EXIT_SUCCESS);
      case '2':           rank = 0; break;
      case '3':           rank = 1; break;
      case '4':           rank = 2; break;
      case '5':           rank = 3; break;
      case '6':           rank = 4; break;
      case '7':           rank = 5; break;
      case '8':           rank = 6; break;
      case '9':           rank = 7; break;
      case 't': case 'T': rank = 8; break;
      case 'j': case 'J': rank = 9; break;
      case 'q': case 'Q': rank = 10; break;
      case 'k': case 'K': rank = 11; break;
      case 'a': case 'A': rank = 12; break;
      default:            bad_card = true;
    }

    suit_ch = getchar();
    switch (suit_ch) {
      case 'c': case 'C': suit = 0; break;
      case 'd': case 'D': suit = 1; break;
      case 'h': case 'H': suit = 2; break;
      case 's': case 'S': suit = 3; break;
      default:            bad_card = true;
    }

    while ((ch = getchar()) != '\n')
      if (ch != ' ') bad_card = true;

    if (bad_card)
      printf("Bad card; ignored.\n");
    else if (card_exists[rank][suit])
      printf("Duplicate card; ignored.\n");
    else {
      num_in_rank[rank]++;
      num_in_suit[suit]++;
      card_exists[rank][suit] = true;
      cards_read++;
    }
  }
}

/**********************************************************
 * analyze_hand: Determines whether the hand contains a   *
 *               straight, a flush, four-of-a-kind,       *
 *               and/or three-of-a-kind; determines the   *
 *               number of pairs; stores the results into *
 *               the external variables straight, flush,  *
 *               four, three, and pairs.                  *
 **********************************************************/
void analyze_hand(void)
{
  int num_consec = 0;
  int rank, suit;
```

```
  straight = false;
  flush = false;
  four = false;
  three = false;
  pairs = 0;

  /* check for flush */
  for (suit = 0; suit < NUM_SUITS; suit++)
    if (num_in_suit[suit] == NUM_CARDS)
      flush = true;

  /* check for straight */
  rank = 0;
  while (num_in_rank[rank] == 0) rank++;
  for (; rank < NUM_RANKS && num_in_rank[rank] > 0; rank++)
    num_consec++;
  if (num_consec == NUM_CARDS) {
    straight = true;
    return;
  }

  /* check for 4-of-a-kind, 3-of-a-kind, and pairs */
  for (rank = 0; rank < NUM_RANKS; rank++) {
    if (num_in_rank[rank] == 4) four = true;
    if (num_in_rank[rank] == 3) three = true;
    if (num_in_rank[rank] == 2) pairs++;
  }
}

/**********************************************************
 * print_result: Prints the classification of the hand,   *
 *               based on the values of the external      *
 *               variables straight, flush, four, three,  *
 *               and pairs.                               *
 **********************************************************/
void print_result(void)
{
  if (straight && flush) printf("Straight flush");
  else if (four)         printf("Four of a kind");
  else if (three &&
           pairs == 1)   printf("Full house");
  else if (flush)        printf("Flush");
  else if (straight)     printf("Straight");
  else if (three)        printf("Three of a kind");
  else if (pairs == 2)   printf("Two pairs");
  else if (pairs == 1)   printf("Pair");
  else                   printf("High card");

  printf("\n\n");
}
```

Notice the use of the `exit` function in `read_cards` (in case `'0'` of the first `switch` statement). `exit` is convenient for this program because of its ability to terminate execution from anywhere in the program.

Q & A

Q: What impact do local variables with static storage duration have on recursive functions? [p. 220]

A: When a function is called recursively, fresh copies are made of its automatic variables for each call. This doesn't occur for static variables, though. Instead, all calls of the function share the *same* static variables.

Q: In the following example, `j` is initialized to the same value as `i`, but there are two variables named `i`:

```
int i = 1;

void f(void)
{
  int j = i;
  int i = 2;
  ...
}
```

Is this code legal? If so, what is `j`'s initial value, 1 or 2?

A: The code is indeed legal. The scope of a local variable doesn't begin until its declaration. Therefore, the declaration of `j` refers to the external variable named `i`. The initial value of `j` will be 1.

Exercises

Section 10.4

W 1. The following program outline shows only function definitions and variable declarations.

```
int a;

void f(int b)
{
  int c;
}

void g(void)
{
  int d;
  {
    int e;
  }
}

int main(void)
{
  int f;
}
```

For each of the following scopes, list all variable and parameter names visible in that scope:

(a) The `f` function
(b) The `g` function
(c) The block in which `e` is declared
(d) The `main` function

2. The following program outline shows only function definitions and variable declarations.

```
int b, c;

void f(void)
{
  int b, d;
}

void g(int a)
{
  int c;
  {
    int a, d;
  }
}

int main(void)
{
  int c, d;
}
```

For each of the following scopes, list all variable and parameter names visible in that scope. If there's more than one variable or parameter with the same name, indicate which one is visible.

(a) The `f` function
(b) The `g` function
(c) The block in which `a` and `d` are declared
(d) The `main` function

*3. Suppose that a program has only one function (`main`). How many different variables named `i` could this program contain?

Programming Projects

1. Modify the stack example of Section 10.2 so that it stores characters instead of integers. Next, add a `main` function that asks the user to enter a series of parentheses and/or braces, then indicates whether or not they're properly nested:

```
Enter parentheses and/or braces: (){}{()})
Parentheses/braces are nested properly
```

Hint: As the program reads characters, have it push each left parenthesis or left brace. When it reads a right parenthesis or brace, have it pop the stack and check that the item popped is a matching parenthesis or brace. (If not, the parentheses/braces aren't nested properly.) When the program reads the new-line character, have it check whether the stack is empty; if so, the parentheses/braces are matched. If the stack *isn't* empty (or if `stack_underflow` is ever

called), the parentheses/braces aren't matched. If `stack_overflow` is called, have the program print the message `Stack overflow` and terminate immediately.

2. Modify the `poker.c` program of Section 10.5 by moving the `num_in_rank` and `num_in_suit` arrays into `main`, which will pass them as arguments to `read_cards` and `analyze_hand`.

W 3. Remove the `num_in_rank`, `num_in_suit`, and `card_exists` arrays from the `poker.c` program of Section 10.5. Have the program store the cards in a 5 × 2 array instead. Each row of the array will represent a card. For example, if the array is named `hand`, then `hand[0][0]` will store the rank of the first card and `hand[0][1]` will store the suit of the first card.

4. Modify the `poker.c` program of Section 10.5 by having it recognize an additional category, "royal flush" (ace, king, queen, jack, ten of the same suit). A royal flush ranks higher than all other hands.

W 5. Modify the `poker.c` program of Section 10.5 by allowing "ace-low" straights (ace, two, three, four, five).

6. Some calculators (notably those from Hewlett-Packard) use a system of writing mathematical expressions known as Reverse Polish Notation (RPN). In this notation, operators are placed *after* their operands instead of *between* their operands. For example, 1 + 2 would be written 1 2 + in RPN, and 1 + 2 * 3 would be written 1 2 3 * +. RPN expressions can easily be evaluated using a stack. The algorithm involves reading the operators and operands in an expression from left to right, performing the following actions:

   When an operand is encountered, push it onto the stack.

   When an operator is encountered, pop its operands from the stack, perform the operation on those operands, and then push the result onto the stack.

   Write a program that evaluates RPN expressions. The operands will be single-digit integers. The operators are +, –, *, /, and =. The = operator causes the top stack item to be displayed; afterwards, the stack is cleared and the user is prompted to enter another expression. The process continues until the user enters a character that is not an operator or operand:

```
Enter an RPN expression: 1 2 3 * + =
Value of expression: 7
Enter an RPN expression: 5 8 * 4 9 - / =
Value of expression: -8
Enter an RPN expression: q
```

   If the stack overflows, the program will display the message `Expression is too complex` and terminate. If the stack underflows (because of an expression such as 1 2 + +), the program will display the message `Not enough operands in expression` and terminate. *Hints:* Incorporate the stack code from Section 10.2 into your program. Use `scanf(" %c", &ch)` to read the operators and operands.

7. Write a program that prompts the user for a number and then displays the number, using characters to simulate the effect of a seven-segment display:

```
Enter a number: 491-9014
```

```
 _   _       _   _   _       _   _
|_| |_|   | |_| |_  |_|   | |_| |_|
  |  _|   | |_|  _| |_|   | |   |
```

   Characters other than digits should be ignored. Write the program so that the maximum number of digits is controlled by a macro named `MAX_DIGITS`, which has the value 10. If

the number contains more than this number of digits, the extra digits are ignored. *Hints:* Use two external arrays. One is the `segments` array (see Exercise 6 in Chapter 8), which stores data representing the correspondence between digits and segments. The other array, `digits`, will be an array of characters with 4 rows (since each segmented digit is four characters high) and `MAX_DIGITS * 4` columns (digits are three characters wide, but a space is needed between digits for readability). Write your program as four functions: `main`, `clear_digits_array`, `process_digit`, and `print_digits_array`. Here are the prototypes for the latter three functions:

```
void clear_digits_array(void);
void process_digit(int digit, int position);
void print_digits_array(void);
```

`clear_digits_array` will store blank characters into all elements of the `digits` array. `process_digit` will store the seven-segment representation of `digit` into a specified position in the `digits` array (positions range from 0 to `MAX_DIGITS – 1`). `print_digits_array` will display the rows of the `digits` array, each on a single line, producing output such as that shown in the example.

11 Pointers

The 11th commandment was "Thou Shalt Compute"
or "Thou Shalt Not Compute"—I forget which.

Pointers are one of C's most important—and most often misunderstood—features. Because of their importance, we'll devote three chapters to pointers. In this chapter, we'll concentrate on the basics; Chapters 12 and 17 cover more advanced uses of pointers.

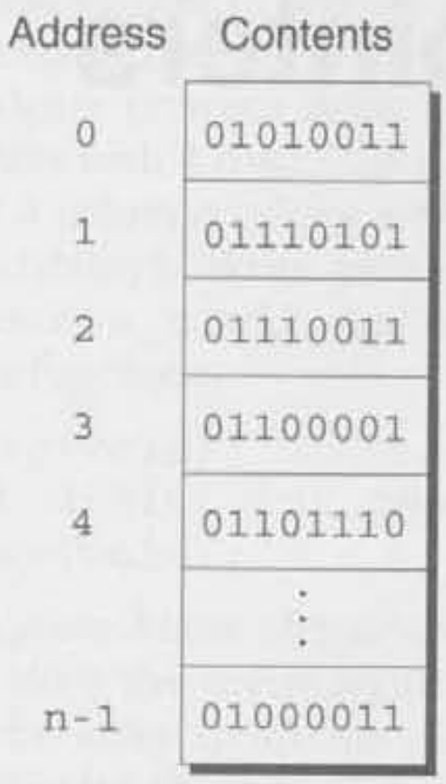
We'll start with a discussion of memory addresses and their relationship to pointer variables (Section 11.1). Section 11.2 then introduces the address and indirection operators. Section 11.3 covers pointer assignment. Section 11.4 explains how to pass pointers to functions, while Section 11.5 discusses returning pointers from functions.

11.1 Pointer Variables

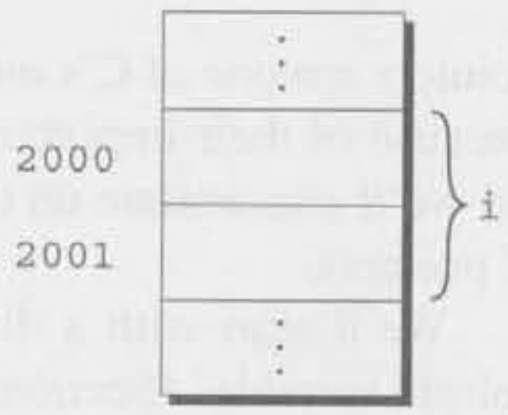
The first step in understanding pointers is visualizing what they represent at the machine level. In most modern computers, main memory is divided into ***bytes***, with each byte capable of storing eight bits of information:

Each byte has a unique ***address*** to distinguish it from the other bytes in memory. If there are n bytes in memory, we can think of addresses as numbers that range from 0 to $n - 1$ (see the figure at the top of the next page).

An executable program consists of both code (machine instructions corresponding to statements in the original C program) and data (variables in the original program). Each variable in the program occupies one or more bytes of memory;



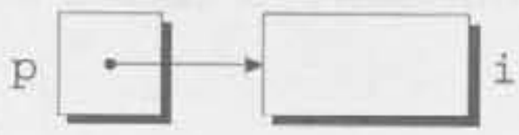
the address of the first byte is said to be the address of the variable. In the following figure, the variable `i` occupies the bytes at addresses 2000 and 2001, so `i`'s address is 2000:



Here's where pointers come in. Although addresses are represented by numbers, their range of values may differ from that of integers, so we can't necessarily store them in ordinary integer variables. We can, however, store them in special ***pointer variables***. When we store the address of a variable `i` in the pointer variable `p`, we say that `p` "points to" `i`. In other words, a pointer is nothing more than an address, and a pointer variable is just a variable that can store an address.

Q&A

Instead of showing addresses as numbers in our examples, I'll use a simpler notation. To indicate that a pointer variable `p` stores the address of a variable `i`, I'll show the contents of `p` as an arrow directed toward `i`:



Declaring Pointer Variables

A pointer variable is declared in much the same way as an ordinary variable. The only difference is that the name of a pointer variable must be preceded by an asterisk:

```
int *p;
```

This declaration states that p is a pointer variable capable of pointing to ***objects*** of type int. I'm using the term *object* instead of *variable* since—as we'll see in Chapter 17—p might point to an area of memory that doesn't belong to a variable. (Be aware that "object" will have a different meaning when we discuss program design in Chapter 19.)

abstract objects ➤ 19.1

Pointer variables can appear in declarations along with other variables:

```
int i, j, a[10], b[20], *p, *q;
```

In this example, i and j are ordinary integer variables, a and b are arrays of integers, and p and q are pointers to integer objects.

C requires that every pointer variable point only to objects of a particular type (the ***referenced type***):

```
int *p;       /* points only to integers   */
double *q;    /* points only to doubles    */
char *r;      /* points only to characters */
```

There are no restrictions on what the referenced type may be. In fact, a pointer variable can even point to another pointer.

pointers to pointers ➤ 17.6

11.2 The Address and Indirection Operators

C provides a pair of operators designed specifically for use with pointers. To find the address of a variable, we use the & (address) operator. If x is a variable, then &x is the address of x in memory. To gain access to the object that a pointer points to, we use the * (***indirection***) operator. If p is a pointer, then *p represents the object to which p currently points.

The Address Operator

Declaring a pointer variable sets aside space for a pointer but doesn't make it point to an object:

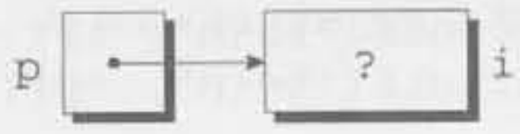
```
int *p;   /* points nowhere in particular */
```

It's crucial to initialize p before we use it. One way to initialize a pointer variable is to assign it the address of some variable—or, more generally, lvalue—using the & operator:

lvalues ➤ 4.2

```
int i, *p;
...
p = &i;
```

By assigning the address of i to the variable p, this statement makes p point to i:



It's also possible to initialize a pointer variable at the time we declare it:

```
int i;
int *p = &i;
```

Q&A

We can even combine the declaration of `i` with the declaration of `p`, provided that `i` is declared first:

```
int i, *p = &i;
```

The Indirection Operator

Once a pointer variable points to an object, we can use the `*` (indirection) operator to access what's stored in the object. If `p` points to `i`, for example, we can print the value of `i` as follows:

```
printf("%d\n", *p);
```

Q&A

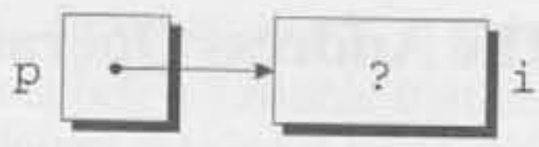
`printf` will display the *value* of `i`, not the *address* of `i`.

The mathematically inclined reader may wish to think of `*` as the inverse of `&`. Applying `&` to a variable produces a pointer to the variable; applying `*` to the pointer takes us back to the original variable:

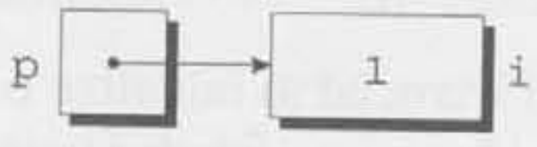
```
j = *&i;    /* same as j = i; */
```

As long as `p` points to `i`, `*p` is an ***alias*** for `i`. Not only does `*p` have the same value as `i`, but changing the value of `*p` also changes the value of `i`. (`*p` is an lvalue, so assignment to it is legal.) The following example illustrates the equivalence of `*p` and `i`; diagrams show the values of `p` and `i` at various points in the computation.

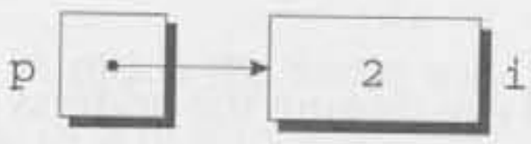
```
p = &i;
```




```
i = 1;
```




```
printf("%d\n", i);     /* prints 1 */
printf("%d\n", *p);    /* prints 1 */
*p = 2;
```




```
printf("%d\n", i);     /* prints 2 */
printf("%d\n", *p);    /* prints 2 */
```

Never apply the indirection operator to an uninitialized pointer variable. If a pointer variable `p` hasn't been initialized, attempting to use the value of `p` in any way causes undefined behavior. In the following example, the call of `printf` may print garbage, cause the program to crash, or have some other effect:

```
int *p;
printf("%d", *p);    /*** WRONG ***/
```

Assigning a value to `*p` is particularly dangerous. If `p` happens to contain a valid memory address, the following assignment will attempt to modify the data stored at that address:

```
int *p;
*p = 1;    /*** WRONG ***/
```

If the location modified by this assignment belongs to the program, it may behave erratically; if it belongs to the operating system, the program will most likely crash. Your compiler may issue a warning that `p` is uninitialized, so pay close attention to any warning messages you get.

11.3 Pointer Assignment

C allows the use of the assignment operator to copy pointers, provided that they have the same type. Suppose that `i`, `j`, `p`, and `q` have been declared as follows:

```
int i, j, *p, *q;
```

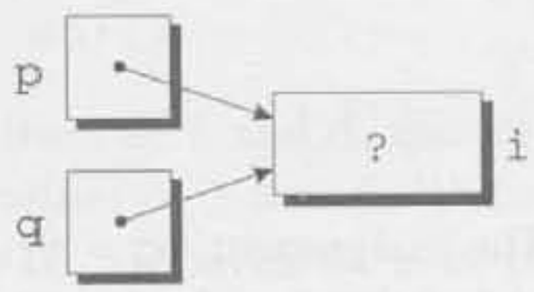
The statement

```
p = &i;
```

is an example of pointer assignment; the address of `i` is copied into `p`. Here's another example of pointer assignment:

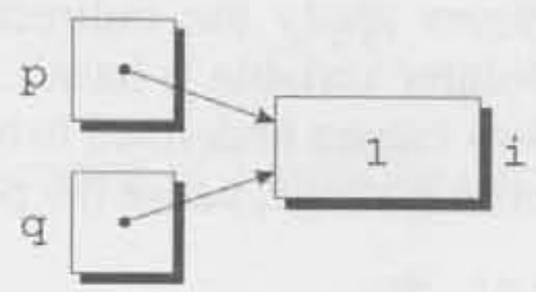
```
q = p;
```

This statement copies the contents of `p` (the address of `i`) into `q`, in effect making `q` point to the same place as `p`:

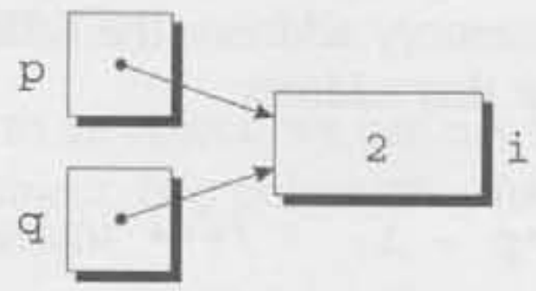


Both `p` and `q` now point to `i`, so we can change `i` by assigning a new value to either `*p` or `*q`:

```
*p = 1;
```




```
*q = 2;
```



Any number of pointer variables may point to the same object.

Be careful not to confuse

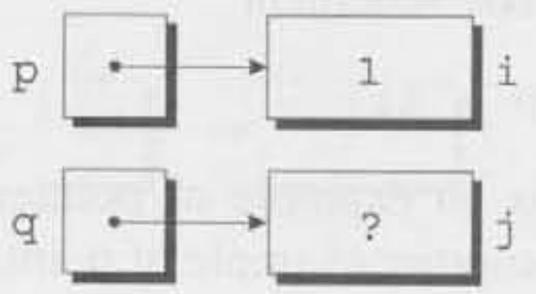
```
q = p;
```

with

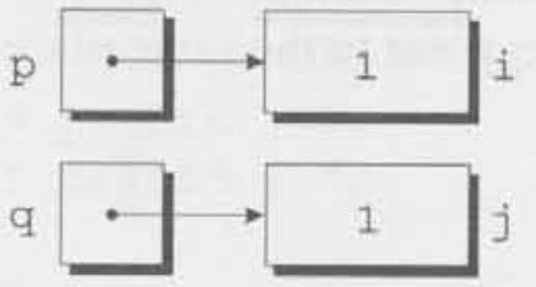
```
*q = *p;
```

The first statement is a pointer assignment; the second isn't, as the following example shows:

```
p = &i;
q = &j;
i = 1;
```




```
*q = *p;
```



The assignment `*q = *p` copies the value that `p` points to (the value of `i`) into the object that `q` points to (the variable `j`).

11.4 Pointers as Arguments

So far, we've managed to avoid a rather important question: What are pointers good for? There's no single answer to that question, since pointers have several distinct uses in C. In this section, we'll see how a pointer to a variable can be useful as a function argument. We'll discover other uses for pointers in Section 11.5 and in Chapters 12 and 17.

We saw in Section 9.3 that a variable supplied as an argument in a function call is protected against change, because C passes arguments by value. This property of C can be a nuisance if we want the function to be able to modify the variable. In Section 9.3, we tried—and failed—to write a `decompose` function that could modify two of its arguments.

Pointers offer a solution to this problem: instead of passing a variable `x` as the argument to a function, we'll supply `&x`, a pointer to `x`. We'll declare the corresponding parameter `p` to be a pointer. When the function is called, `p` will have the value `&x`, hence `*p` (the object that `p` points to) will be an alias for `x`. Each appearance of `*p` in the body of the function will be an indirect reference to `x`, allowing the function both to read `x` and to modify it.

To see this technique in action, let's modify the `decompose` function by declaring the parameters `int_part` and `frac_part` to be pointers. The definition of `decompose` will now look like this:

```
void decompose(double x, long *int_part, double *frac_part)
{
  *int_part = (long) x;
  *frac_part = x - *int_part;
}
```

The prototype for `decompose` could be either

```
void decompose(double x, long *int_part, double *frac_part);
```

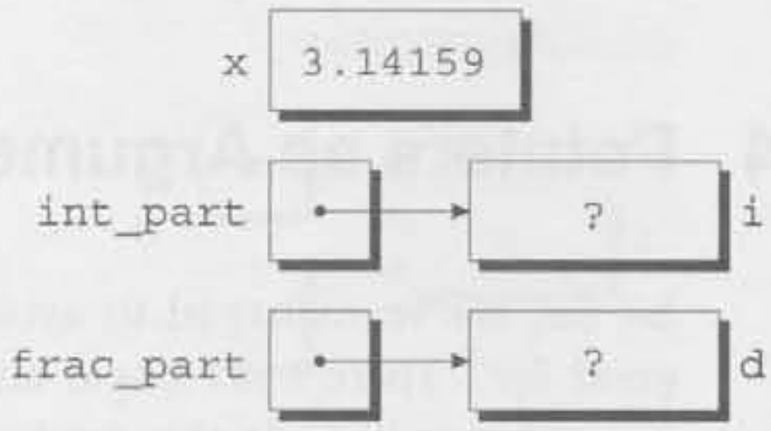
or

```
void decompose(double, long *, double *);
```

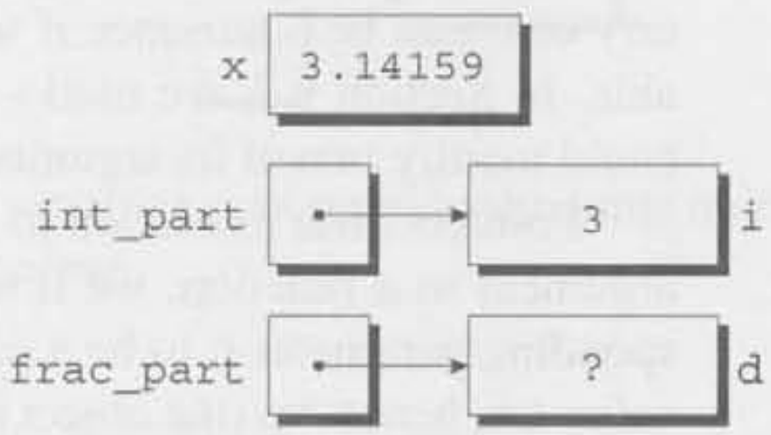
We'll call `decompose` in the following way:

```
decompose(3.14159, &i, &d);
```

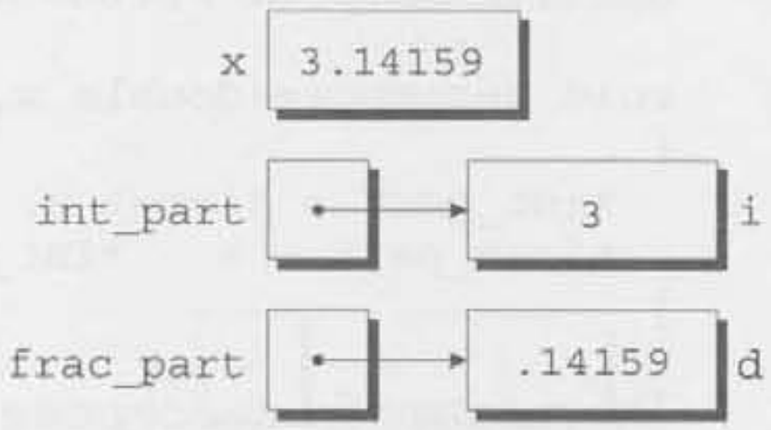
Because of the `&` operator in front of `i` and `d`, the arguments to `decompose` are *pointers* to `i` and `d`, not the *values* of `i` and `d`. When `decompose` is called, the value 3.14159 is copied into `x`, a pointer to `i` is stored in `int_part`, and a pointer to `d` is stored in `frac_part`:



The first assignment in the body of `decompose` converts the value of `x` to type `long` and stores it in the object pointed to by `int_part`. Since `int_part` points to `i`, the assignment puts the value 3 in `i`:



The second assignment fetches the value that `int_part` points to (the value of `i`), which is 3. This value is converted to type `double` and subtracted from `x`, giving .14159, which is then stored in the object that `frac_part` points to:



When `decompose` returns, `i` and `d` will have the values 3 and .14159, just as we originally wanted.

Using pointers as arguments to functions is actually nothing new; we've been doing it in calls of `scanf` since Chapter 2. Consider the following example:

```
int i;
...
scanf("%d", &i);
```

We must put the `&` operator in front of `i` so that `scanf` is given a *pointer* to `i`; that pointer tells `scanf` where to put the value that it reads. Without the `&`, `scanf` would be supplied with the *value* of `i`.

Although `scanf`'s arguments must be pointers, it's not always true that every argument needs the `&` operator. In the following example, `scanf` is passed a pointer variable:

```
int i, *p;
...
p = &i;
scanf("%d", p);
```

Since `p` contains the address of `i`, `scanf` will read an integer and store it in `i`. Using the `&` operator in the call would be wrong:

```
scanf("%d", &p);    /*** WRONG ***/
```

`scanf` would read an integer and store it in `p` instead of in `i`.

Failing to pass a pointer to a function when one is expected can have disastrous results. Suppose that we call `decompose` without the `&` operator in front of `i` and `d`:

```
decompose(3.14159, i, d);
```

`decompose` is expecting pointers as its second and third arguments, but it's been given the *values* of `i` and `d` instead. `decompose` has no way to tell the difference, so it will use the values of `i` and `d` as though they were pointers. When `decompose` stores values in `*int_part` and `*frac_part`, it will attempt to change unknown memory locations instead of modifying `i` and `d`.

If we've provided a prototype for `decompose` (as we should always do, of course), the compiler will let us know that we're attempting to pass arguments of the wrong type. In the case of `scanf`, however, failing to pass pointers often goes undetected by the compiler, making `scanf` an especially error-prone function.

PROGRAM

Finding the Largest and Smallest Elements in an Array

To illustrate how pointers are passed to functions, let's look at a function named `max_min` that finds the largest and smallest elements in an array. When we call `max_min`, we'll pass it pointers to two variables; `max_min` will then store its answers in these variables. `max_min` has the following prototype:

```
void max_min(int a[], int n, int *max, int *min);
```

A call of `max_min` might have the following appearance:

```
max_min(b, N, &big, &small);
```

`b` is an array of integers; `N` is the number of elements in `b`. `big` and `small` are ordinary integer variables. When `max_min` finds the largest element in `b`, it stores the value in `big` by assigning it to `*max`. (Since `max` points to `big`, an assignment to `*max` will modify the value of `big`.) `max_min` stores the smallest element of `b` in `small` by assigning it to `*min`.

To test `max_min`, we'll write a program that reads 10 numbers into an array, passes the array to `max_min`, and prints the results:

```
Enter 10 numbers: 34 82 49 102 7 94 23 11 50 31
Largest: 102
Smallest: 7
```

Here's the complete program:

maxmin.c

```
/* Finds the largest and smallest elements in an array */

#include <stdio.h>

#define N 10

void max_min(int a[], int n, int *max, int *min);

int main(void)
{
  int b[N], i, big, small;

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

  max_min(b, N, &big, &small);

  printf("Largest: %d\n", big);
  printf("Smallest: %d\n", small);

  return 0;
}

void max_min(int a[], int n, int *max, int *min)
{
  int i;

  *max = *min = a[0];
  for (i = 1; i < n; i++) {
    if (a[i] > *max)
      *max = a[i];
    else if (a[i] < *min)
      *min = a[i];
  }
}
```

Using `const` to Protect Arguments

When we call a function and pass it a pointer to a variable, we normally assume that the function will modify the variable (otherwise, why would the function require a pointer?). For example, if we see a statement like

```
f(&x);
```

in a program, we'd probably expect `f` to change the value of `x`. It's possible, though, that `f` merely needs to examine the value of `x`, not change it. The reason for the pointer might be efficiency: passing the value of a variable can waste time and space if the variable requires a large amount of storage. (Section 12.3 covers this point in more detail.)

We can use the word `const` to document that a function won't change an object whose address is passed to the function. `const` goes in the parameter's declaration, just before the specification of its type:

Q&A

```
void f(const int *p)
{
  *p = 0;    /*** WRONG ***/
}
```

This use of `const` indicates that `p` is a pointer to a "constant integer." Attempting to modify `*p` is an error that the compiler will detect.

11.5 Pointers as Return Values

We can not only pass pointers to functions but also write functions that *return* pointers. Such functions are relatively common; we'll encounter several in Chapter 13.

The following function, when given pointers to two integers, returns a pointer to whichever integer is larger:

```
int *max(int *a, int *b)
{
  if (*a > *b)
    return a;
  else
    return b;
}
```

When we call `max`, we'll pass pointers to two `int` variables and store the result in a pointer variable:

```
int *p, i, j;
...
p = max(&i, &j);
```

During the call of `max`, `*a` is an alias for `i`, while `*b` is an alias for `j`. If `i` has a larger value than `j`, `max` returns the address of `i`; otherwise, it returns the address of `j`. After the call, `p` points to either `i` or `j`.

Although the `max` function returns one of the pointers passed to it as an argument, that's not the only possibility. A function could also return a pointer to an external variable or to a local variable that's been declared `static`.

Never return a pointer to an *automatic* local variable:

```
int *f(void)
{
  int i;
  …
  return &i;
}
```

The variable `i` doesn't exist once `f` returns, so the pointer to it will be invalid. Some compilers issue a warning such as "*function returns address of local variable*" in this situation.

Pointers can point to array elements, not just ordinary variables. If `a` is an array, then `&a[i]` is a pointer to element `i` of `a`. When a function has an array argument, it's sometimes useful for the function to return a pointer to one of the elements in the array. For example, the following function returns a pointer to the middle element of the array `a`, assuming that `a` has `n` elements:

```
int *find_middle(int a[], int n) {
  return &a[n/2];
}
```

Chapter 12 explores the relationship between pointers and arrays in considerable detail.

Q & A

***Q: Is a pointer always the same as an address? [p. 242]**

A: Usually, but not always. Consider a computer whose main memory is divided into ***words*** rather than bytes. A word might contain 36 bits, 60 bits, or some other number of bits. If we assume 36-bit words, memory will have the following appearance:

Address	Contents
0	001010011001010011001010011001010011
1	001110101001110101001110101001110101
2	001110011001110011001110011001110011
3	001100001001100001001100001001100001
4	001101110001101110001101110001101110
	⋮
n-1	001000011001000011001000011001000011

When memory is divided into words, each word has an address. An integer usually occupies one word, so a pointer to an integer can just be an address. However, a word can store more than one character. For example, a 36-bit word might store six 6-bit characters:

or four 9-bit characters:

For this reason, a pointer to a character may need to be stored in a different form than other pointers. A pointer to a character might consist of an address (the word in which the character is stored) plus a small integer (the position of the character within the word).

On some computers, pointers may be "offsets" rather than complete addresses. For example, CPUs in the Intel x86 family (used in many personal computers) can execute programs in several modes. The oldest of these, which dates back to the 8086 processor of 1978, is called ***real mode***. In this mode, addresses are sometimes represented by a single 16-bit number (an ***offset***) and sometimes by two 16-bit numbers (a ***segment:offset pair***). An offset isn't a true memory address; the CPU must combine it with a segment value stored in a special register. To support real mode, older C compilers often provide two kinds of pointers: ***near pointers*** (16-bit offsets) and ***far pointers*** (32-bit segment:offset pairs). These compilers usually reserve the words `near` and `far` as nonstandard keywords that can be used to declare pointer variables.

Q: If a pointer can point to *data* in a program, is it possible to have a pointer to program *code?

A: Yes. We'll cover pointers to functions in Section 17.7.

Q: It seems to me that there's an inconsistency between the declaration

```
int *p = &i;
```

and the statement

```
p = &i;
```

Why isn't `p` preceded by a `*` symbol in the statement, as it is in the declaration? [p. 244]

A: The source of the confusion is the fact that the `*` symbol can have different meanings in C, depending on the context in which it's used. In the declaration

```
int *p = &i;
```

the `*` symbol is *not* the indirection operator. Instead, it helps specify the type of `p`, informing the compiler that `p` is a *pointer* to an `int`. When it appears in a statement,

however, the * symbol performs indirection (when used as a unary operator). The statement

```
*p = &i;    /*** WRONG ***/
```

would be wrong, because it assigns the address of `i` to the object that `p` points to, not to `p` itself.

Q: Is there some way to print the address of a variable? [p. 244]

A: Any pointer, including the address of a variable, can be displayed by calling the `printf` function and using `%p` as the conversion specification. See Section 22.3 for details.

Q: The following declaration is confusing:

```
void f(const int *p);
```

Does this say that `f` can't modify `p`? [p. 251]

A: No. It says that `f` can't change the integer that `p` *points to*; it doesn't prevent `f` from changing `p` itself.

```
void f(const int *p)
{
  int j;

  *p = 0;    /*** WRONG ***/
  p = &j;    /* legal */
}
```

Since arguments are passed by value, assigning `p` a new value—by making it point somewhere else—won't have any effect outside the function.

***Q: When declaring a parameter of a pointer type, is it legal to put the word `const` in front of the parameter's name, as in the following example?**

```
void f(int * const p);
```

A: Yes, although the effect isn't the same as if `const` precedes `p`'s type. We saw in Section 11.4 that putting `const` *before* `p`'s type protects the object that `p` points to. Putting `const` *after* `p`'s type protects `p` itself:

```
void f(int * const p)
{
  int j;

  *p = 0;    /* legal */
  p = &j;    /*** WRONG ***/
}
```

This feature isn't used very often. Since `p` is merely a copy of another pointer (the argument when the function is called), there's rarely any reason to protect it.

An even greater rarity is the need to protect both `p` *and* the object it points to, which can be done by putting `const` both before and after `p`'s type:

```
void f(const int * const p)
{
  int j;

  *p = 0;    /*** WRONG ***/
  p = &j;    /*** WRONG ***/
}
```

Exercises

Section 11.2

1. If `i` is a variable and `p` points to `i`, which of the following expressions are aliases for `i`?

   (a) `*p` (c) `*&p` (e) `*i` (g) `*&i`
   (b) `&p` (d) `&*p` (f) `&i` (h) `&*i`

Section 11.3

W 2. If `i` is an `int` variable and `p` and `q` are pointers to `int`, which of the following assignments are legal?

   (a) `p = i;` (d) `p = &q;` (g) `p = *q;`
   (b) `*p = &i;` (e) `p = *&q;` (h) `*p = q;`
   (c) `&p = q;` (f) `p = q;` (i) `*p = *q;`

Section 11.4

3. The following function supposedly computes the sum and average of the numbers in the array `a`, which has length `n`. `avg` and `sum` point to variables that the function should modify. Unfortunately, the function contains several errors; find and correct them.

```
void avg_sum(double a[], int n, double *avg, double *sum)
{
  int i;

  sum = 0.0;
  for (i = 0; i < n; i++)
    sum += a[i];
  avg = sum / n;
}
```

W 4. Write the following function:

```
void swap(int *p, int *q);
```

When passed the addresses of two variables, `swap` should exchange the values of the variables:

```
swap(&i, &j);   /* exchanges values of i and j */
```

5. Write the following function:

```
void split_time(long total_sec, int *hr, int *min, int *sec);
```

`total_sec` is a time represented as the number of seconds since midnight. `hr`, `min`, and `sec` are pointers to variables in which the function will store the equivalent time in hours (0–23), minutes (0–59), and seconds (0–59), respectively.

W 6. Write the following function:

```
void find_two_largest(int a[], int n, int *largest,
                      int *second_largest);
```

When passed an array `a` of length `n`, the function will search `a` for its largest and second-largest elements, storing them in the variables pointed to by `largest` and `second_largest`, respectively.

7. Write the following function:

```
void split_date(int day_of_year, int year,
                int *month, int *day);
```

`day_of_year` is an integer between 1 and 366, specifying a particular day within the year designated by `year`. `month` and `day` point to variables in which the function will store the equivalent month (1–12) and day within that month (1–31).

Section 11.5

8. Write the following function:

```
int *find_largest(int a[], int n);
```

When passed an array `a` of length `n`, the function will return a pointer to the array's largest element.

Programming Projects

1. Modify Programming Project 7 from Chapter 2 so that it includes the following function:

```
void pay_amount(int dollars, int *twenties, int *tens,
                int *fives, int *ones);
```

The function determines the smallest number of $20, $10, $5, and $1 bills necessary to pay the amount represented by the `dollars` parameter. The `twenties` parameter points to a variable in which the function will store the number of $20 bills required. The `tens`, `fives`, and `ones` parameters are similar.

2. Modify Programming Project 8 from Chapter 5 so that it includes the following function:

```
void find_closest_flight(int desired_time,
                         int *departure_time,
                         int *arrival_time);
```

This function will find the flight whose departure time is closest to `desired_time` (expressed in minutes since midnight). It will store the departure and arrival times of this flight (also expressed in minutes since midnight) in the variables pointed to by `departure_time` and `arrival_time`, respectively.

3. Modify Programming Project 3 from Chapter 6 so that it includes the following function:

```
void reduce(int numerator, int denominator,
            int *reduced_numerator,
            int *reduced_denominator);
```

`numerator` and `denominator` are the numerator and denominator of a fraction. `reduced_numerator` and `reduced_denominator` are pointers to variables in which the function will store the numerator and denominator of the fraction once it has been reduced to lowest terms.

4. Modify the `poker.c` program of Section 10.5 by moving all external variables into `main` and modifying functions so that they communicate by passing arguments. The `analyze_hand` function needs to change the `straight`, `flush`, `four`, `three`, and `pairs` variables, so it will have to be passed pointers to those variables.

12 Pointers and Arrays

Optimization hinders evolution.

Chapter 11 introduced pointers and showed how they're used as function arguments and as values returned by functions. This chapter covers another application for pointers. When pointers point to array elements, C allows us to perform arithmetic—addition and subtraction—on the pointers, which leads to an alternative way of processing arrays in which pointers take the place of array subscripts.

The relationship between pointers and arrays in C is a close one, as we'll soon see. We'll exploit this relationship in subsequent chapters, including Chapter 13 (Strings) and Chapter 17 (Advanced Uses of Pointers). Understanding the connection between pointers and arrays is critical for mastering C: it will give you insight into how C was designed and help you understand existing programs. Be aware, however, that one of the primary reasons for using pointers to process arrays—efficiency—is no longer as important as it once was, thanks to improved compilers.

Section 12.1 discusses pointer arithmetic and shows how pointers can be compared using the relational and equality operators. Section 12.2 then demonstrates how we can use pointer arithmetic for processing array elements. Section 12.3 reveals a key fact about arrays—an array name can serve as a pointer to the array's first element—and uses it to show how array arguments really work. Section 12.4 shows how the topics of the first three sections apply to multidimensional arrays. Section 12.5 wraps up the chapter by exploring the relationship between pointers and variable-length arrays, a C99 feature.

12.1 Pointer Arithmetic

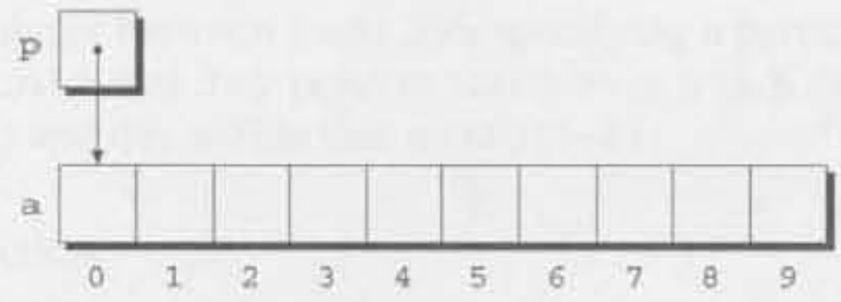
We saw in Section 11.5 that pointers can point to array elements. For example, suppose that `a` and `p` have been declared as follows:

```
int a[10], *p;
```

We can make p point to a[0] by writing

```
p = &a[0];
```

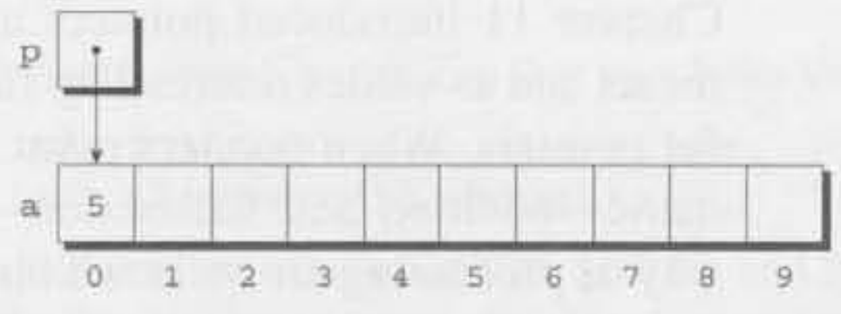
Graphically, here's what we've just done:



We can now access a[0] through p; for example, we can store the value 5 in a[0] by writing

```
*p = 5;
```

Here's our picture now:



Making a pointer p point to an element of an array a isn't particularly exciting. However, by performing ***pointer arithmetic*** (or ***address arithmetic***) on p, we can access the other elements of a. C supports three (and only three) forms of pointer arithmetic:

Adding an integer to a pointer
Subtracting an integer from a pointer
Subtracting one pointer from another

Let's take a close look at each of these operations. Our examples assume that the following declarations are in effect:

```
int a[10], *p, *q, i;
```

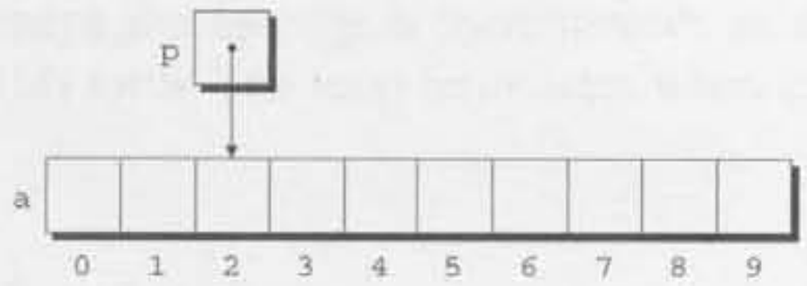
Adding an Integer to a Pointer

Adding an integer j to a pointer p yields a pointer to the element j places after the one that p points to. More precisely, if p points to the array element a[i], then p + j points to a[i+j] (provided, of course, that a[i+j] exists).

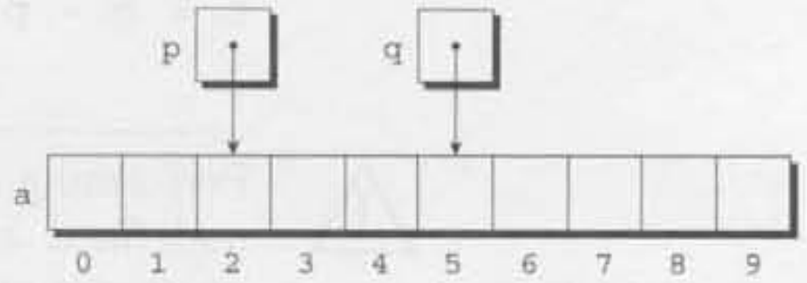
Q&A

The following example illustrates pointer addition; diagrams show the values of p and q at various points in the computation.

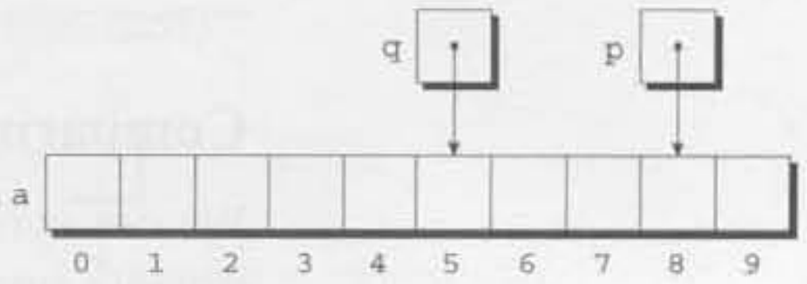
```
p = &a[2];
```




```
q = p + 3;
```



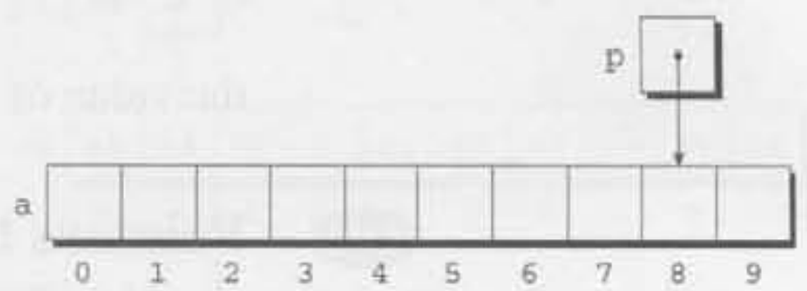

```
p += 6;
```



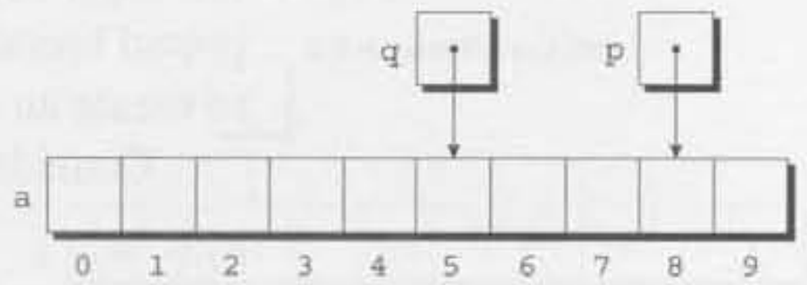
Subtracting an Integer from a Pointer

If `p` points to the array element `a[i]`, then `p - j` points to `a[i-j]`. For example:

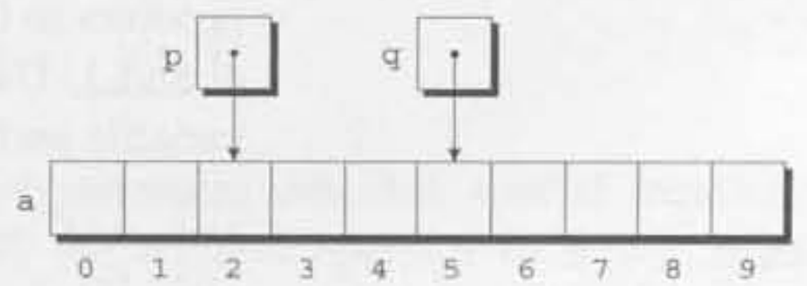
```
p = &a[8];
```




```
q = p - 3;
```



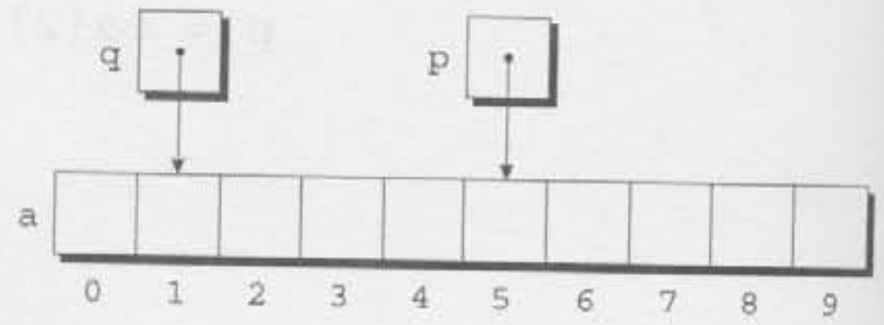

```
p -= 6;
```



Subtracting One Pointer from Another

When one pointer is subtracted from another, the result is the distance (measured in array elements) between the pointers. Thus, if `p` points to `a[i]` and `q` points to `a[j]`, then `p - q` is equal to `i - j`. For example:

```
p = &a[5];
q = &a[1];
```




```
i = p - q;    /* i is 4 */
i = q - p;    /* i is -4 */
```

Performing arithmetic on a pointer that doesn't point to an array element causes undefined behavior. Furthermore, the effect of subtracting one pointer from another is undefined unless both point to elements of the *same* array.

Comparing Pointers

We can compare pointers using the relational operators (`<`, `<=`, `>`, `>=`) and the equality operators (`==` and `!=`). Using the relational operators to compare two pointers is meaningful only when both point to elements of the same array. The outcome of the comparison depends on the relative positions of the two elements in the array. For example, after the assignments

```
p = &a[5];
q = &a[1];
```

the value of `p <= q` is 0 and the value of `p >= q` is 1.

C99 Pointers to Compound Literals

compound literals ➤9.3

It's legal for a pointer to point to an element within an array created by a compound literal. A compound literal, you may recall, is a C99 feature that can be used to create an array with no name.

Consider the following example:

```
int *p = (int []){3, 0, 3, 4, 1};
```

`p` points to the first element of a five-element array containing the integers 3, 0, 3, 4, and 1. Using a compound literal saves us the trouble of first declaring an array variable and then making `p` point to the first element of that array:

```
int a[] = {3, 0, 3, 4, 1};
int *p = &a[0];
```

12.2 Using Pointers for Array Processing

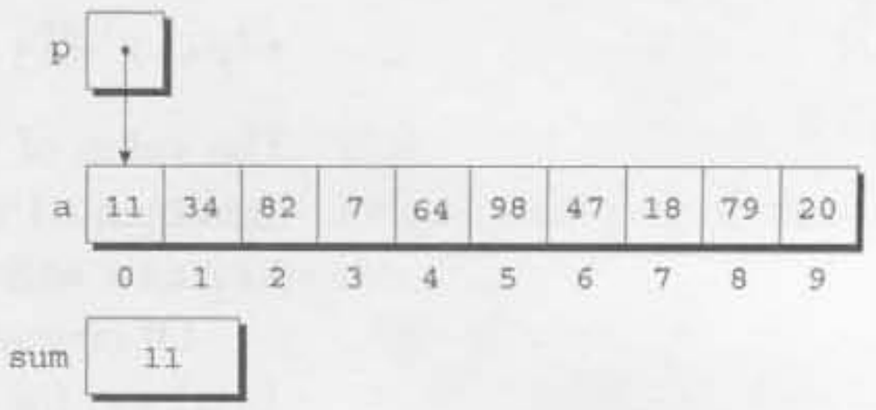
Pointer arithmetic allows us to visit the elements of an array by repeatedly incrementing a pointer variable. The following program fragment, which sums the elements of an array `a`, illustrates the technique. In this example, the pointer variable

p initially points to a[0]. Each time through the loop, p is incremented; as a result, it points to a[1], then a[2], and so forth. The loop terminates when p steps past the last element of a.

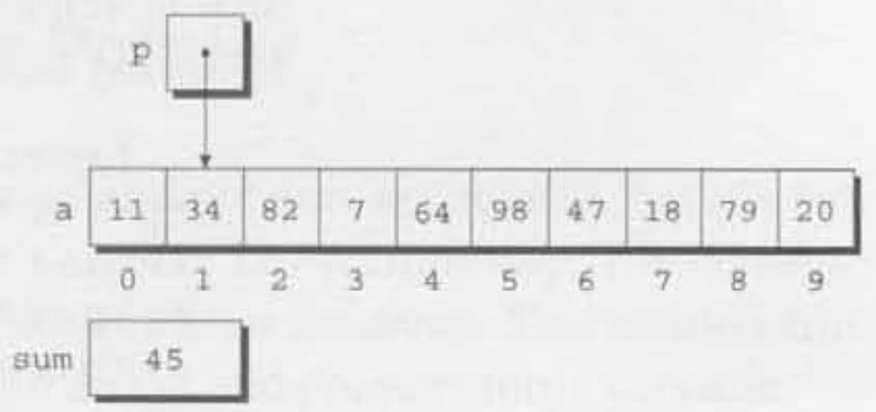
```
#define N 10
...
int a[N], sum, *p;
...
sum = 0;
for (p = &a[0]; p < &a[N]; p++)
  sum += *p;
```

The following figures show the contents of a, sum, and p at the end of the first three loop iterations (before p has been incremented).

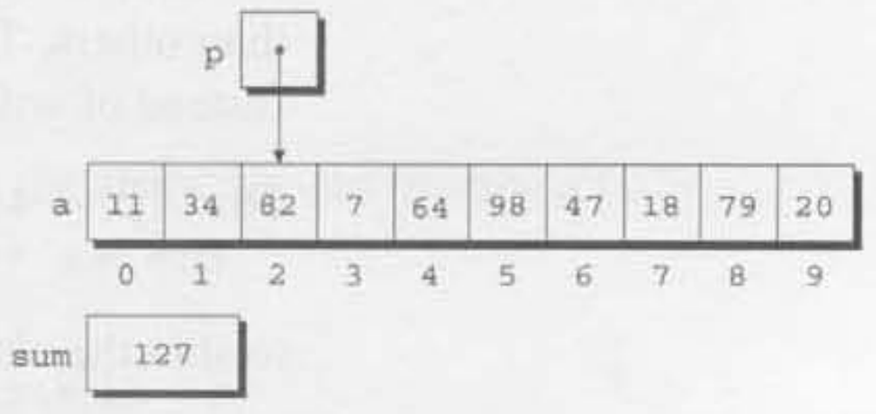
At the end of the first iteration:



At the end of the second iteration:



At the end of the third iteration:



The condition p < &a[N] in the for statement deserves special mention. Strange as it may seem, it's legal to apply the address operator to a[N], even though this element doesn't exist (a is indexed from 0 to N – 1). Using a[N] in this fashion is perfectly safe, since the loop doesn't attempt to examine its value. The body of the loop will be executed with p equal to &a[0], &a[1], ..., &a[N-1], but when p is equal to &a[N], the loop terminates.

We could just as easily have written the loop without pointers, of course, using subscripting instead. The argument most often cited in support of pointer arithmetic is that it can save execution time. However, that depends on the implementation—some C compilers actually produce better code for loops that rely on subscripting.

Q&A

Combining the * and ++ Operators

C programmers often combine the * (indirection) and ++ operators in statements that process array elements. Consider the simple case of storing a value into an array element and then advancing to the next element. Using array subscripting, we might write

```
a[i++] = j;
```

If p is pointing to an array element, the corresponding statement would be

```
*p++ = j;
```

Because the postfix version of ++ takes precedence over *, the compiler sees this as

```
*(p++) = j;
```

The value of p++ is p. (Since we're using the postfix version of ++, p won't be incremented until after the expression has been evaluated.) Thus, the value of *(p++) will be *p—the object to which p is pointing.

Of course, *p++ isn't the only legal combination of * and ++. We could write (*p)++, for example, which returns the value of the object that p points to, and then increments that object (p itself is unchanged). If you find this confusing, the following table may help:

Expression	*Meaning*
*p++ or *(p++)	Value of expression is *p before increment; increment p later
(*p)++	Value of expression is *p before increment; increment *p later
*++p or *(++p)	Increment p first; value of expression is *p after increment
++*p or ++(*p)	Increment *p first; value of expression is *p after increment

All four combinations appear in programs, although some are far more common than others. The one we'll see most frequently is *p++, which is handy in loops. Instead of writing

```
for (p = &a[0]; p < &a[N]; p++)
  sum += *p;
```

to sum the elements of the array a, we could write

```
p = &a[0];
while (p < &a[N])
  sum += *p++;
```

The * and -- operators mix in the same way as * and ++. For an application that combines * and --, let's return to the stack example of Section 10.2. The original version of the stack relied on an integer variable named top to keep track of the "top-of-stack" position in the contents array. Let's replace top by a pointer variable that points initially to element 0 of the contents array:

```
int *top_ptr = &contents[0];
```

Here are the new `push` and `pop` functions (updating the other stack functions is left as an exercise):

```
void push(int i)
{
  if (is_full())
    stack_overflow();
  else
    *top_ptr++ = i;
}

int pop(void)
{
  if (is_empty())
    stack_underflow();
  else
    return *--top_ptr;
}
```

Note that I've written `*--top_ptr`, not `*top_ptr--`, since I want `pop` to decrement `top_ptr` *before* fetching the value to which it points.

12.3 Using an Array Name as a Pointer

Pointer arithmetic is one way in which arrays and pointers are related, but it's not the only connection between the two. Here's another key relationship: *The name of an array can be used as a pointer to the first element in the array.* This relationship simplifies pointer arithmetic and makes both arrays and pointers more versatile.

For example, suppose that `a` is declared as follows:

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

Using `a` as a pointer to the first element in the array, we can modify `a[0]`:

```
*a = 7;   /* stores 7 in a[0] */
```

We can modify `a[1]` through the pointer `a + 1`:

```
*(a+1) = 12;   /* stores 12 in a[1] */
```

In general, `a + i` is the same as `&a[i]` (both represent a pointer to element `i` of `a`) and `*(a+i)` is equivalent to `a[i]` (both represent element `i` itself). In other words, array subscripting can be viewed as a form of pointer arithmetic.

The fact that an array name can serve as a pointer makes it easier to write loops that step through an array. Consider the following loop from Section 12.2:

```
for (p = &a[0]; p < &a[N]; p++)
  sum += *p;
```

To simplify the loop, we can replace `&a[0]` by `a` and `&a[N]` by `a + N`:

idiom
```
for (p = a; p < a + N; p++)
  sum += *p;
```

Although an array name can be used as a pointer, it's not possible to assign it a new value. Attempting to make it point elsewhere is an error:

```
while (*a != 0)
  a++;                  /*** WRONG ***/
```

This is no great loss; we can always copy `a` into a pointer variable, then change the pointer variable:

```
p = a;
while (*p != 0)
  p++;
```

PROGRAM

Reversing a Series of Numbers (Revisited)

The `reverse.c` program of Section 8.1 reads 10 numbers, then writes the numbers in reverse order. As the program reads the numbers, it stores them in an array. Once all the numbers are read, the program steps through the array backwards as it prints the numbers.

The original program used subscripting to access elements of the array. Here's a new version in which I've replaced subscripting with pointer arithmetic.

reverse3.c
```
/* Reverses a series of numbers (pointer version) */

#include <stdio.h>

#define N 10

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

  printf("Enter %d numbers: ", N);
  for (p = a; p < a + N; p++)
    scanf("%d", p);

  printf("In reverse order:");
  for (p = a + N - 1; p >= a; p--)
    printf(" %d", *p);
  printf("\n");

  return 0;
}
```

In the original program, an integer variable `i` kept track of the current position within the array. The new version replaces `i` with `p`, a pointer variable. The num-

bers are still stored in an array; we're simply using a different technique to keep track of where we are in the array.

Note that the second argument to `scanf` is `p`, not `&p`. Since `p` points to an array element, it's a satisfactory argument for `scanf`; `&p`, on the other hand, would be a pointer to a pointer to an array element.

Array Arguments (Revisited)

When passed to a function, an array name is always treated as a pointer. Consider the following function, which returns the largest element in an array of integers:

```
int find_largest(int a[], int n)
{
  int i, max;

  max = a[0];
  for (i = 1; i < n; i++)
    if (a[i] > max)
      max = a[i];
  return max;
}
```

Suppose that we call `find_largest` as follows:

```
largest = find_largest(b, N);
```

This call causes a pointer to the first element of `b` to be assigned to `a`; the array itself isn't copied.

The fact that an array argument is treated as a pointer has some important consequences:

- When an ordinary variable is passed to a function, its value is copied; any changes to the corresponding parameter don't affect the variable. In contrast, an array used as an argument isn't protected against change, since no copy is made of the array itself. For example, the following function (which we first saw in Section 9.3) modifies an array by storing zero into each of its elements:

```
void store_zeros(int a[], int n)
{
  int i;

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

To indicate that an array parameter won't be changed, we can include the word `const` in its declaration:

```
int find_largest(const int a[], int n)
{
  ...
}
```

If `const` is present, the compiler will check that no assignment to an element of `a` appears in the body of `find_largest`.

- The time required to pass an array to a function doesn't depend on the size of the array. There's no penalty for passing a large array, since no copy of the array is made.
- An array parameter can be declared as a pointer if desired. For example, `find_largest` could be defined as follows:

```
int find_largest(int *a, int n)
{
  ...
}
```

Declaring `a` to be a pointer is equivalent to declaring it to be an array; the compiler treats the declarations as though they were identical.

Q&A

Although declaring a *parameter* to be an array is the same as declaring it to be a pointer, the same isn't true for a *variable*. The declaration

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

causes the compiler to set aside space for 10 integers. In contrast, the declaration

```
int *a;
```

causes the compiler to allocate space for a pointer variable. In the latter case, `a` is not an array; attempting to use it as an array can have disastrous results. For example, the assignment

```
*a = 0;    /*** WRONG ***/
```

will store 0 where `a` is pointing. Since we don't know where `a` is pointing, the effect on the program is undefined.

- A function with an array parameter can be passed an array "slice"—a sequence of consecutive elements. Suppose that we want `find_largest` to locate the largest element in some portion of an array `b`, say elements `b[5]`, ..., `b[14]`. When we call `find_largest`, we'll pass it the address of `b[5]` and the number `10`, indicating that we want `find_largest` to examine 10 array elements, starting at `b[5]`:

```
largest = find_largest(&b[5], 10);
```

Using a Pointer as an Array Name

If we can use an array name as a pointer, will C allow us to subscript a pointer as though it were an array name? By now, you'd probably expect the answer to be yes, and you'd be right. Here's an example:

```
#define N 10
…
int a[N], i, sum = 0, *p = a;
…
for (i = 0; i < N; i++)
  sum += p[i];
```

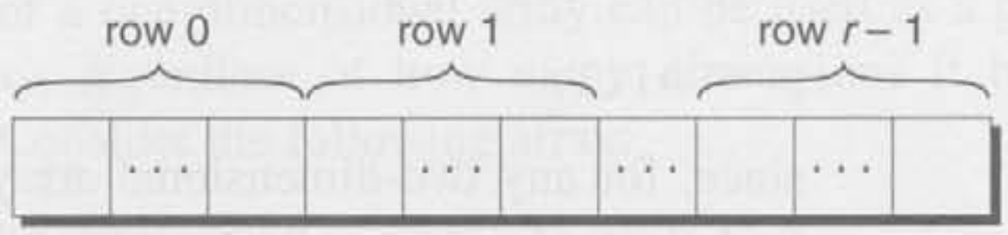
The compiler treats `p[i]` as `*(p+i)`, which is a perfectly legal use of pointer arithmetic. Although the ability to subscript a pointer may seem to be little more than a curiosity, we'll see in Section 17.3 that it's actually quite useful.

12.4 Pointers and Multidimensional Arrays

Just as pointers can point to elements of one-dimensional arrays, they can also point to elements of multidimensional arrays. In this section, we'll explore common techniques for using pointers to process the elements of multidimensional arrays. For simplicity, I'll stick to two-dimensional arrays, but everything we'll do applies equally to higher-dimensional arrays.

Processing the Elements of a Multidimensional Array

We saw in Section 8.2 that C stores two-dimensional arrays in row-major order; in other words, the elements of row 0 come first, followed by the elements of row 1, and so forth. An array with *r* rows would have the following appearance:



We can take advantage of this layout when working with pointers. If we make a pointer `p` point to the first element in a two-dimensional array (the element in row 0, column 0), we can visit every element in the array by incrementing `p` repeatedly.

As an example, let's look at the problem of initializing all elements of a two-dimensional array to zero. Suppose that the array has been declared as follows:

```
int a[NUM_ROWS][NUM_COLS];
```

The obvious technique would be to use nested `for` loops:

```
int row, col;
…
for (row = 0; row < NUM_ROWS; row++)
  for (col = 0; col < NUM_COLS; col++)
    a[row][col] = 0;
```

But if we view a as a one-dimensional array of integers (which is how it's stored), we can replace the pair of loops by a single loop:

```
int *p;
...
for (p = &a[0][0]; p <= &a[NUM_ROWS-1][NUM_COLS-1]; p++)
  *p = 0;
```

The loop begins with p pointing to a[0][0]. Successive increments of p make it point to a[0][1], a[0][2], a[0][3], and so on. When p reaches a[0][NUM_COLS-1] (the last element in row 0), incrementing it again makes p point to a[1][0], the first element in row 1. The process continues until p goes past a[NUM_ROWS-1][NUM_COLS-1], the last element in the array.

Q&A Although treating a two-dimensional array as one-dimensional may seem like cheating, it works with most C compilers. Whether it's a good idea to do so is another matter. Techniques like this one definitely hurt program readability, but—at least with some older compilers—produce a compensating increase in efficiency. With many modern compilers, though, there's often little or no speed advantage.

Processing the Rows of a Multidimensional Array

What about processing the elements in just one *row* of a two-dimensional array? Again, we have the option of using a pointer variable p. To visit the elements of row i, we'd initialize p to point to element 0 in row i in the array a:

```
p = &a[i][0];
```

Or we could simply write

```
p = a[i];
```

since, for any two-dimensional array a, the expression a[i] is a pointer to the first element in row i. To see why this works, recall the magic formula that relates array subscripting to pointer arithmetic: for any array a, the expression a[i] is equivalent to *(a + i). Thus, &a[i][0] is the same as &(*(a[i] + 0)), which is equivalent to &*a[i], which is the same as a[i], since the & and * operators cancel. We'll use this simplification in the following loop, which clears row i of the array a:

```
int a[NUM_ROWS][NUM_COLS], *p, i;
...
for (p = a[i]; p < a[i] + NUM_COLS; p++)
  *p = 0;
```

Since a[i] is a pointer to row i of the array a, we can pass a[i] to a function that's expecting a one-dimensional array as its argument. In other words, a function that's designed to work with one-dimensional arrays will also work with a row belonging to a two-dimensional array. As a result, functions such as

`find_largest` and `store_zeros` are more versatile than you might expect. Consider `find_largest`, which we originally designed to find the largest element of a one-dimensional array. We can just as easily use `find_largest` to determine the largest element in row `i` of the two-dimensional array `a`:

```
largest = find_largest(a[i], NUM_COLS);
```

Processing the Columns of a Multidimensional Array

Processing the elements in a *column* of a two-dimensional array isn't as easy, because arrays are stored by row, not by column. Here's a loop that clears column `i` of the array `a`:

```
int a[NUM_ROWS][NUM_COLS], (*p)[NUM_COLS], i;
...
for (p = &a[0]; p < &a[NUM_ROWS]; p++)
  (*p)[i] = 0;
```

I've declared `p` to be a pointer to an array of length `NUM_COLS` whose elements are integers. The parentheses around `*p` in `(*p)[NUM_COLS]` are required; without them, the compiler would treat `p` as an array of pointers instead of a pointer to an array. The expression `p++` advances `p` to the beginning of the next row. In the expression `(*p)[i]`, `*p` represents an entire row of `a`, so `(*p)[i]` selects the element in column `i` of that row. The parentheses in `(*p)[i]` are essential, because the compiler would interpret `*p[i]` as `*(p[i])`.

Using the Name of a Multidimensional Array as a Pointer

Just as the name of a one-dimensional array can be used as a pointer, so can the name of *any* array, regardless of how many dimensions it has. Some care is required, though. Consider the following array:

```
int a[NUM_ROWS][NUM_COLS];
```

`a` is *not* a pointer to `a[0][0]`; instead, it's a pointer to `a[0]`. This makes more sense if we look at it from the standpoint of C, which regards `a` not as a two-dimensional array but as a one-dimensional array whose elements are one-dimensional arrays. When used as a pointer, `a` has type `int (*)[NUM_COLS]` (pointer to an integer array of length `NUM_COLS`).

Knowing that `a` points to `a[0]` is useful for simplifying loops that process the elements of a two-dimensional array. For example, instead of writing

```
for (p = &a[0]; p < &a[NUM_ROWS]; p++)
  (*p)[i] = 0;
```

to clear column `i` of the array `a`, we can write

```
for (p = a; p < a + NUM_ROWS; p++)
  (*p)[i] = 0;
```

Another situation in which this knowledge comes in handy is when we want to "trick" a function into thinking that a multidimensional array is really one-dimensional. For example, consider how we might use `find_largest` to find the largest element in `a`. As the first argument to `find_largest`, let's try passing `a` (the address of the array); as the second, we'll pass `NUM_ROWS * NUM_COLS` (the total number of elements in `a`):

```
largest = find_largest(a, NUM_ROWS * NUM_COLS);   /* WRONG */
```

Unfortunately, the compiler will object to this statement, because the type of `a` is `int (*)[NUM_COLS]` but `find_largest` is expecting an argument of type `int *`. The correct call is

```
largest = find_largest(a[0], NUM_ROWS * NUM_COLS);
```

`a[0]` points to element 0 in row 0, and it has type `int *` (after conversion by the compiler), so the latter call will work correctly.

Q&A

12.5 Pointers and Variable-Length Arrays (C99)

Pointers are allowed to point to elements of variable-length arrays (VLAs), a feature of C99. An ordinary pointer variable would be used to point to an element of a one-dimensional VLA:

variable-length arrays ➤8.3

```
void f(int n)
{
  int a[n], *p;
  p = a;
  …
}
```

When the VLA has more than one dimension, the type of the pointer depends on the length of each dimension except for the first. Let's look at the two-dimensional case:

```
void f(int m, int n)
{
  int a[m][n], (*p)[n];
  p = a;
  …
}
```

Since the type of `p` depends on `n`, which isn't constant, `p` is said to have a ***variably modified type.*** Note that the validity of an assignment such as `p = a` can't always be determined by the compiler. For example, the following code will compile but is correct only if `m` and `n` are equal:

```
int a[m][n], (*p)[m];
p = a;
```

If m ≠ n, any subsequent use of p will cause undefined behavior.

Variably modified types are subject to certain restrictions, just as variable-length arrays are. The most important restriction is that the declaration of a variably modified type must be inside the body of a function or in a function prototype.

Pointer arithmetic works with VLAs just as it does for ordinary arrays. Returning to the example of Section 12.4 that clears a single column of a two-dimensional array a, let's declare a as a VLA this time:

```
int a[m][n];
```

A pointer capable of pointing to a row of a would be declared as follows:

```
int (*p)[n];
```

The loop that clears column i is almost identical to the one we used in Section 12.4:

```
for (p = a; p < a + m; p++)
  (*p)[i] = 0;
```

Q & A

Q: I don't understand pointer arithmetic. If a pointer is an address, does that mean that an expression like p + j adds j to the address stored in p? [p. 258]

A: No. Integers used in pointer arithmetic are scaled depending on the type of the pointer. If p is of type int *, for example, then p + j typically adds 4 × j to p, assuming that int values are stored using 4 bytes. But if p has type double *, then p + j will probably add 8 × j to p, since double values are usually 8 bytes long.

Q: When writing a loop to process an array, is it better to use array subscripting or pointer arithmetic? [p. 261]

A: There's no easy answer to this question, since it depends on the machine you're using and the compiler itself. In the early days of C on the PDP-11, pointer arithmetic yielded a faster program. On today's machines, using today's compilers, array subscripting is often just as good, and sometimes even better. The bottom line: Learn both ways and then use whichever is more natural for the kind of program you're writing.

***Q: I read somewhere that i[a] is the same as a[i]. Is this true?**

A: Yes, it is, oddly enough. The compiler treats i[a] as *(i + a), which is the same as *(a + i). (Pointer addition, like ordinary addition, is commutative.) But *(a + i) is equivalent to a[i]. Q.E.D. But please don't use i[a] in programs unless you're planning to enter the next Obfuscated C contest.

Q: Why is `*a` the same as `a[]` in a parameter declaration? [p. 266]

A: Both indicate that the argument is expected to be a pointer. The same operations on `a` are possible in both cases (pointer arithmetic and array subscripting, in particular). And, in both cases, `a` itself can be assigned a new value within the function. (Although C allows us to use the name of an array *variable* only as a "constant pointer," there's no such restriction on the name of an array *parameter*.)

Q: Is it better style to declare an array parameter as `*a` or `a[]`?

A: That's a tough one. From one standpoint, `a[]` is the obvious choice, since `*a` is ambiguous (does the function want an array of objects or a pointer to a single object?). On the other hand, many programmers argue that declaring the parameter as `*a` is more accurate, since it reminds us that only a pointer is passed, not a copy of the array. Others switch between `*a` and `a[]`, depending on whether the function uses pointer arithmetic or subscripting to access the elements of the array. (That's the approach I'll use.) In practice, `*a` is more common than `a[]`, so you'd better get used to it. For what it's worth, Dennis Ritchie now refers to the `a[]` notation as "a living fossil" that "serves as much to confuse the learner as to alert the reader."

Q: We've seen that arrays and pointers are closely related in C. Would it be accurate to say that they're interchangeable?

A: No. It's true that array *parameters* are interchangeable with pointer parameters, but array *variables* aren't the same as pointer variables. Technically, the name of an array isn't a pointer; rather, the C compiler *converts* it to a pointer when necessary. To see this difference more clearly, consider what happens when we apply the `sizeof` operator to an array `a`. The value of `sizeof(a)` is the total number of bytes in the array—the size of each element multiplied by the number of elements. But if `p` is a pointer variable, `sizeof(p)` is the number of bytes required to store a pointer value.

Q: You said that treating a two-dimensional array as one-dimensional works with "most" C compilers. Doesn't it work with all compilers? [p. 268]

A: No. Some modern "bounds-checking" compilers track not only the type of a pointer, but—when it points to an array—also the length of the array. For example, suppose that `p` is assigned a pointer to `a[0][0]`. Technically, `p` points to the first element of `a[0]`, a one-dimensional array. If we increment `p` repeatedly in an effort to visit all the elements of `a`, we'll go out of bounds once `p` goes past the last element of `a[0]`. A compiler that performs bounds-checking may insert code to check that `p` is used only to access elements in the array pointed to by `a[0]`; an attempt to increment `p` past the end of this array would be detected as an error.

Q: If `a` is a two-dimensional array, why can we pass `a[0]`—but not `a` itself—to `find_largest`? Don't both `a` and `a[0]` point to the same place (the beginning of the array)? [p. 270]

A: They do, as a matter of fact—both point to element `a[0][0]`. The problem is that

a has the wrong type. When used as an argument, it's a pointer to an array, but `find_largest` is expecting a pointer to an integer. However, `a[0]` has type `int *`, so it's an acceptable argument for `find_largest`. This concern about types is actually good; if C weren't so picky, we could make all kinds of horrible pointer mistakes without the compiler noticing.

Exercises

Section 12.1

1. Suppose that the following declarations are in effect:

```
int a[] = {5, 15, 34, 54, 14, 2, 52, 72};
int *p = &a[1], *q = &a[5];
```

(a) What is the value of `*(p+3)`?
(b) What is the value of `*(q-3)`?
(c) What is the value of `q - p`?
(d) Is the condition `p < q` true or false?
(e) Is the condition `*p < *q` true or false?

W *2. Suppose that `high`, `low`, and `middle` are all pointer variables of the same type, and that `low` and `high` point to elements of an array. Why is the following statement illegal, and how could it be fixed?

```
middle = (low + high) / 2;
```

Section 12.2

3. What will be the contents of the a array after the following statements are executed?

```
#define N 10

int a[N] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int *p = &a[0], *q = &a[N-1], temp;

while (p < q) {
  temp = *p;
  *p++ = *q;
  *q-- = temp;
}
```

W 4. Rewrite the `make_empty`, `is_empty`, and `is_full` functions of Section 10.2 to use the pointer variable `top_ptr` instead of the integer variable `top`.

Section 12.3

5. Suppose that a is a one-dimensional array and p is a pointer variable. Assuming that the assignment `p = a` has just been performed, which of the following expressions are illegal because of mismatched types? Of the remaining expressions, which are true (have a nonzero value)?

(a) `p == a[0]`
(b) `p == &a[0]`
(c) `*p == a[0]`
(d) `p[0] == a[0]`

W 6. Rewrite the following function to use pointer arithmetic instead of array subscripting. (In other words, eliminate the variable `i` and all uses of the `[]` operator.) Make as few changes as possible.

```
int sum_array(const int a[], int n)
{
  int i, sum;

  sum = 0;
  for (i = 0; i < n; i++)
    sum += a[i];
  return sum;
}
```

7. Write the following function:

```
bool search(const int a[], int n, int key);
```

`a` is an array to be searched, `n` is the number of elements in the array, and `key` is the search key. `search` should return `true` if `key` matches some element of `a`, and `false` if it doesn't. Use pointer arithmetic—not subscripting—to visit array elements.

8. Rewrite the following function to use pointer arithmetic instead of array subscripting. (In other words, eliminate the variable `i` and all uses of the `[]` operator.) Make as few changes as possible.

```
void store_zeros(int a[], int n)
{
  int i;

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

9. Write the following function:

```
double inner_product(const double *a, const double *b,
                     int n);
```

`a` and `b` both point to arrays of length `n`. The function should return `a[0] * b[0] + a[1] * b[1] + ... + a[n-1] * b[n-1]`. Use pointer arithmetic—not subscripting—to visit array elements.

10. Modify the `find_middle` function of Section 11.5 so that it uses pointer arithmetic to calculate the return value.

11. Modify the `find_largest` function so that it uses pointer arithmetic—not subscripting—to visit array elements.

12. Write the following function:

```
void find_two_largest(const int *a, int n, int *largest,
                      int *second_largest);
```

`a` points to an array of length `n`. The function searches the array for its largest and second-largest elements, storing them in the variables pointed to by `largest` and `second_largest`, respectively. Use pointer arithmetic—not subscripting—to visit array elements.

Section 12.4

W 13. Section 8.2 had a program fragment in which two nested `for` loops initialized the array `ident` for use as an identity matrix. Rewrite this code, using a single pointer to step through the array one element at a time. *Hint:* Since we won't be using `row` and `col` index variables, it won't be easy to tell where to store 1. Instead, we can use the fact that the first element of the array should be 1, the next `N` elements should be 0, the next element should

be 1, and so forth. Use a variable to keep track of how many consecutive 0s have been stored; when the count reaches `N`, it's time to store 1.

14. Assume that the following array contains a week's worth of hourly temperature readings, with each row containing the readings for one day:

```
int temperatures[7][24];
```

Write a statement that uses the `search` function (see Exercise 7) to search the entire `temperatures` array for the value 32.

W 15. Write a loop that prints all temperature readings stored in row `i` of the `temperatures` array (see Exercise 14). Use a pointer to visit each element of the row.

16. Write a loop that prints the highest temperature in the `temperatures` array (see Exercise 14) for each day of the week. The loop body should call the `find_largest` function, passing it one row of the array at a time.

17. Rewrite the following function to use pointer arithmetic instead of array subscripting. (In other words, eliminate the variables `i` and `j` and all uses of the `[]` operator.) Use a single loop instead of nested loops.

```
int sum_two_dimensional_array(const int a[][LEN], int n)
{
  int i, j, sum = 0;

  for (i = 0; i < n; i++)
    for (j = 0; j < LEN; j++)
      sum += a[i][j];
  return sum;
}
```

18. Write the `evaluate_position` function described in Exercise 13 of Chapter 9. Use pointer arithmetic—not subscripting—to visit array elements. Use a single loop instead of nested loops.

Programming Projects

W 1. (a) Write a program that reads a message, then prints the reversal of the message:

```
Enter a message: Don't get mad, get even.
Reversal is: .neve teg ,dam teg t'noD
```

Hint: Read the message one character at a time (using `getchar`) and store the characters in an array. Stop reading when the array is full or the character read is `'\n'`.

(b) Revise the program to use a pointer instead of an integer to keep track of the current position in the array.

2. (a) Write a program that reads a message, then checks whether it's a palindrome (the letters in the message are the same from left to right as from right to left):

```
Enter a message: He lived as a devil, eh?
Palindrome

Enter a message: Madam, I am Adam.
Not a palindrome
```

Ignore all characters that aren't letters. Use integer variables to keep track of positions in the array.

(b) Revise the program to use pointers instead of integers to keep track of positions in the array.

W 3. Simplify Programming Project 1(b) by taking advantage of the fact that an array name can be used as a pointer.

4. Simplify Programming Project 2(b) by taking advantage of the fact that an array name can be used as a pointer.

5. Modify Programming Project 14 from Chapter 8 so that it uses a pointer instead of an integer to keep track of the current position in the array that contains the sentence.

6. Modify the `qsort.c` program of Section 9.6 so that `low`, `high`, and `middle` are pointers to array elements rather than integers. The `split` function will need to return a pointer, not an integer.

7. Modify the `maxmin.c` program of Section 11.4 so that the `max_min` function uses a pointer instead of an integer to keep track of the current position in the array.

13 Strings

It's difficult to extract sense from strings, but they're the only communication coin we can count on.

Although we've used `char` variables and arrays of `char` values in previous chapters, we still lack any convenient way to process a series of characters (a *string*, in C terminology). We'll remedy that defect in this chapter, which covers both string *constants* (or *literals*, as they're called in the C standard) and string *variables*, which can change during the execution of a program.

Section 13.1 explains the rules that govern string literals, including the rules for embedding escape sequences in string literals and for breaking long string literals. Section 13.2 then shows how to declare string variables, which are simply arrays of characters in which a special character—the null character—marks the end of a string. Section 13.3 describes ways to read and write strings. Section 13.4 shows how to write functions that process strings, and Section 13.5 covers some of the string-handling functions in the C library. Section 13.6 presents idioms that are often used when working with strings. Finally, Section 13.7 describes how to set up arrays whose elements are pointers to strings of different lengths. This section also explains how C uses such an array to supply command-line information to programs.

13.1 String Literals

A ***string literal*** is a sequence of characters enclosed within double quotes:

```
"When you come to a fork in the road, take it."
```

We first encountered string literals in Chapter 2; they often appear as format strings in calls of `printf` and `scanf`.

Escape Sequences in String Literals

escape sequences ➤7.3

String literals may contain the same escape sequences as character constants. We've used character escapes in `printf` and `scanf` format strings for some time. For example, we've seen that each `\n` character in the string

```
"Candy\nIs dandy\nBut liquor\nIs quicker.\n  --Ogden Nash\n"
```

causes the cursor to advance to the next line:

```
Candy
Is dandy
But liquor
Is quicker.
  --Ogden Nash
```

Although octal and hexadecimal escapes are also legal in string literals, they're not as common as character escapes.

Be careful when using octal and hexadecimal escape sequences in string literals. An octal escape ends after three digits or with the first non-octal character. For example, the string `"\1234"` contains two characters (`\123` and `4`), and the string `"\189"` contains three characters (`\1`, `8`, and `9`). A hexadecimal escape, on the other hand, isn't limited to three digits; it doesn't end until the first non-hex character. Consider what happens if a string contains the escape `\xfc`, which represents the character *ü* in the Latin1 character set, a common extension of ASCII. The string `"Z\xfcrich"` ("Zürich") has six characters (`Z`, `\xfc`, `r`, `i`, `c`, and `h`), but the string `"\xfcber"` (a failed attempt at "über") has only two (`\xfcbe` and `r`). Most compilers will object to the latter string, since hex escapes are usually limited to the range `\x0`–`\xff`.

Q&A

Continuing a String Literal

If we find that a string literal is too long to fit conveniently on a single line, C allows us to continue it on the next line, provided that we end the first line with a backslash character (\). No other characters may follow \ on the same line, other than the (invisible) new-line character at the end:

```
printf("When you come to a fork in the road, take it.  \
--Yogi Berra");
```

In general, the \ character can be used to join two or more lines of a program into a single line (a process that the C standard refers to as "splicing"). We'll see more examples of splicing in Section 14.3.

The \ technique has one drawback: the string must continue at the beginning of the next line, thereby wrecking the program's indented structure. There's a better way to deal with long string literals, thanks to the following rule: when two or more string literals are adjacent (separated only by white space), the compiler will

join them into a single string. This rule allows us to split a string literal over two or more lines:

```
printf("When you come to a fork in the road, take it.  "
       "--Yogi Berra");
```

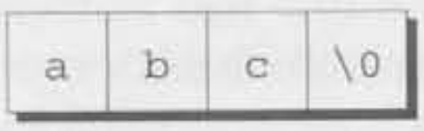
How String Literals Are Stored

We've used string literals often in calls of `printf` and `scanf`. But when we call `printf` and supply a string literal as an argument, what are we actually passing? To answer this question, we need to know how string literals are stored.

In essence, C treats string literals as character arrays. When a C compiler encounters a string literal of length n in a program, it sets aside $n + 1$ bytes of memory for the string. This area of memory will contain the characters in the string, plus one extra character—the ***null character***—to mark the end of the string. The null character is a byte whose bits are all zero, so it's represented by the `\0` escape sequence.

Don't confuse the null character (`'\0'`) with the zero character (`'0'`). The null character has the code 0; the zero character has a different code (48 in ASCII).

For example, the string literal `"abc"` is stored as an array of four characters (`a`, `b`, `c`, and `\0`):



String literals may be empty; the string `""` is stored as a single null character:

Since a string literal is stored as an array, the compiler treats it as a pointer of type `char *`. Both `printf` and `scanf`, for example, expect a value of type `char *` as their first argument. Consider the following example:

```
printf("abc");
```

When `printf` is called, it's passed the address of `"abc"` (a pointer to where the letter `a` is stored in memory).

Operations on String Literals

In general, we can use a string literal wherever C allows a `char *` pointer. For example, a string literal can appear on the right side of an assignment:

```
char *p;

p = "abc";
```

This assignment doesn't copy the characters in `"abc"`; it merely makes `p` point to the first character of the string.

C allows pointers to be subscripted, so we can subscript string literals:

```
char ch;

ch = "abc"[1];
```

The new value of `ch` will be the letter `b`. The other possible subscripts are 0 (which would select the letter `a`), 2 (the letter `c`), and 3 (the null character). This property of string literals isn't used that much, but occasionally it's handy. Consider the following function, which converts a number between 0 and 15 into a character that represents the equivalent hex digit:

```
char digit_to_hex_char(int digit)
{
  return "0123456789ABCDEF"[digit];
}
```

Attempting to modify a string literal causes undefined behavior:

```
char *p = "abc";

*p = 'd';    /*** WRONG ***/
```

Q&A A program that tries to change a string literal may crash or behave erratically.

String Literals versus Character Constants

A string literal containing a single character isn't the same as a character constant. The string literal `"a"` is represented by a *pointer* to a memory location that contains the character `a` (followed by a null character). The character constant `'a'` is represented by an *integer* (the numerical code for the character).

Don't ever use a character when a string is required (or vice versa). The call

```
printf("\n");
```

is legal, because `printf` expects a pointer as its first argument. The following call isn't legal, however:

```
printf('\n');    /*** WRONG ***/
```

13.2 String Variables

Some programming languages provide a special `string` type for declaring string variables. C takes a different tack: any one-dimensional array of characters can be used to store a string, with the understanding that the string is terminated by a null character. This approach is simple, but has significant difficulties. It's sometimes hard to tell whether an array of characters is being used as a string. If we write our own string-handling functions, we've got to be careful that they deal properly with the null character. Also, there's no faster way to determine the length of a string than a character-by-character search for the null character.

Let's say that we need a variable capable of storing a string of up to 80 characters. Since the string will need a null character at the end, we'll declare the variable to be an array of 81 characters:

idiom

```
#define STR_LEN 80
...
char str[STR_LEN+1];
```

We defined `STR_LEN` to be `80` rather than `81`, thus emphasizing the fact that `str` can store strings of no more than 80 characters, and then added 1 to `STR_LEN` in the declaration of `str`. This a common practice among C programmers.

When declaring an array of characters that will be used to hold a string, always make the array one character longer than the string, because of the C convention that every string is terminated by a null character. Failing to leave room for the null character may cause unpredictable results when the program is executed, since functions in the C library assume that strings are null-terminated.

Declaring a character array to have length `STR_LEN + 1` doesn't mean that it will always contain a string of `STR_LEN` characters. The length of a string depends on the position of the terminating null character, not on the length of the array in which the string is stored. An array of `STR_LEN + 1` characters can hold strings of various lengths, ranging from the empty string to strings of length `STR_LEN`.

Initializing a String Variable

A string variable can be initialized at the same time it's declared:

```
char date1[8] = "June 14";
```

The compiler will put the characters from `"June 14"` in the `date1` array, then add a null character so that `date1` can be used as a string. Here's what `date1` will look like:

Although `"June 14"` appears to be a string literal, it's not. Instead, C views it as an abbreviation for an array initializer. In fact, we could have written

```
char date1[8] = {'J', 'u', 'n', 'e', ' ', '1', '4', '\0'};
```

I think you'll agree that the original version is easier to read.

What if the initializer is too short to fill the string variable? In that case, the compiler adds extra null characters. Thus, after the declaration

```
char date2[9] = "June 14";
```

`date2` will have the following appearance:

array initializers ➤8.1

This behavior is consistent with C's treatment of array initializers in general. When an array initializer is shorter than the array itself, the remaining elements are initialized to zero. By initializing the leftover elements of a character array to `\0`, the compiler is following the same rule.

What if the initializer is longer than the string variable? That's illegal for strings, just as it's illegal for other arrays. However, C does allow the initializer (not counting the null character) to have exactly the same length as the variable:

```
char date3[7] = "June 14";
```

There's no room for the null character, so the compiler makes no attempt to store one:

If you're planning to initialize a character array to contain a string, be sure that the length of the array is longer than the length of the initializer. Otherwise, the compiler will quietly omit the null character, making the array unusable as a string.

The declaration of a string variable may omit its length, in which case the compiler computes it:

```
char date4[] = "June 14";
```

The compiler sets aside eight characters for `date4`, enough to store the characters in `"June 14"` plus a null character. (The fact that the length of `date4` isn't specified doesn't mean that the array's length can be changed later. Once the program is compiled, the length of `date4` is fixed at eight.) Omitting the length of a string variable is especially useful if the initializer is long, since computing the length by hand is error-prone.

Character Arrays versus Character Pointers

Let's compare the declaration

```
char date[] = "June 14";
```

which declares `date` to be an *array*, with the similar-looking

```
char *date = "June 14";
```

which declares `date` to be a *pointer*. Thanks to the close relationship between arrays and pointers, we can use either version of `date` as a string. In particular, any function expecting to be passed a character array or character pointer will accept either version of `date` as an argument.

However, we must be careful not to make the mistake of thinking that the two versions of `date` are interchangeable. There are significant differences between the two:

- In the array version, the characters stored in `date` can be modified, like the elements of any array. In the pointer version, `date` points to a string literal, and we saw in Section 13.1 that string literals shouldn't be modified.
- In the array version, `date` is an array name. In the pointer version, `date` is a variable that can be made to point to other strings during program execution.

If we need a string that can be modified, it's our responsibility to set up an array of characters in which to store the string; declaring a pointer variable isn't enough. The declaration

```
char *p;
```

causes the compiler to set aside enough memory for a pointer variable; unfortunately, it doesn't allocate space for a string. (And how could it? We haven't indicated how long the string would be.) Before we can use `p` as a string, it must point to an array of characters. One possibility is to make `p` point to a string variable:

```
char str[STR_LEN+1], *p;

p = str;
```

`p` now points to the first character of `str`, so we can use `p` as a string. Another possibility is to make `p` point to a dynamically allocated string.

dynamically allocated strings ➤17.2

Using an uninitialized pointer variable as a string is a serious error. Consider the following example, which attempts to build the string `"abc"`:

```
char *p;

p[0] = 'a';     /*** WRONG ***/
p[1] = 'b';     /*** WRONG ***/
p[2] = 'c';     /*** WRONG ***/
p[3] = '\0';    /*** WRONG ***/
```

Since `p` hasn't been initialized, we don't know where it's pointing. Using the pointer to write the characters `a`, `b`, `c`, and `\0` into memory causes undefined behavior.

13.3 Reading and Writing Strings

Writing a string is easy using either the `printf` or `puts` functions. Reading a string is a bit harder, primarily because of the possibility that the input string may be longer than the string variable into which it's being stored. To read a string in a single step, we can use either `scanf` or `gets`. As an alternative, we can read strings one character at a time.

Writing Strings Using `printf` and `puts`

The `%s` conversion specification allows `printf` to write a string. Consider the following example:

```
char str[] = "Are we having fun yet?";

printf("%s\n", str);
```

The output will be

```
Are we having fun yet?
```

`printf` writes the characters in a string one by one until it encounters a null character. (If the null character is missing, `printf` continues past the end of the string until—eventually—it finds a null character somewhere in memory.)

To print just part of a string, we can use the conversion specification `%.`*p*`s`, where *p* is the number of characters to be displayed. The statement

```
printf("%.6s\n", str);
```

will print

```
Are we
```

A string, like a number, can be printed within a field. The `%ms` conversion will display a string in a field of size *m*. (A string with more than *m* characters will be printed in full, not truncated.) If the string has fewer than *m* characters, it will be right-justified within the field. To force left justification instead, we can put a minus sign in front of *m*. The *m* and *p* values can be used in combination: a conversion specification of the form `%m.ps` causes the first *p* characters of a string to be displayed in a field of size *m*.

`printf` isn't the only function that can write strings. The C library also provides `puts`, which is used in the following way:

```
puts(str);
```

`puts` has only one argument (the string to be printed). After writing the string, `puts` always writes an additional new-line character, thus advancing to the beginning of the next output line.

Reading Strings Using `scanf` and `gets`

The `%s` conversion specification allows `scanf` to read a string into a character array:

```
scanf("%s", str);
```

There's no need to put the `&` operator in front of `str` in the call of `scanf`; like any array name, `str` is treated as a pointer when passed to a function.

white-space characters ➤3.2

When `scanf` is called, it skips white space, then reads characters and stores them in `str` until it encounters a white-space character. `scanf` always stores a null character at the end of the string.

A string read using `scanf` will never contain white space. Consequently, `scanf` won't usually read a full line of input; a new-line character will cause `scanf` to stop reading, but so will a space or tab character. To read an entire line of input at a time, we can use `gets`. Like `scanf`, the `gets` function reads input characters into an array, then stores a null character. In other respects, however, `gets` is somewhat different from `scanf`:

- `gets` doesn't skip white space before starting to read the string (`scanf` does).
- `gets` reads until it finds a new-line character (`scanf` stops at any white-space character). Incidentally, `gets` discards the new-line character instead of storing it in the array; the null character takes its place.

To see the difference between `scanf` and `gets`, consider the following program fragment:

```
char sentence[SENT_LEN+1];

printf("Enter a sentence:\n");
scanf("%s", sentence);
```

Suppose that after the prompt

```
Enter a sentence:
```

the user enters the line

```
  To C, or not to C: that is the question.
```

`scanf` will store the string `"To"` in `sentence`. The next call of `scanf` will resume reading the line at the space after the word `To`.

Now suppose that we replace `scanf` by `gets`:

```
gets(sentence);
```

When the user enters the same input as before, `gets` will store the string

```
"  To C, or not to C: that is the question."
```

in `sentence`.

As they read characters into an array, `scanf` and `gets` have no way to detect when it's full. Consequently, they may store characters past the end of the array, causing undefined behavior. `scanf` can be made safer by using the conversion specification `%`*n*`s` instead of `%s`, where *n* is an integer indicating the maximum number of characters to be stored. `gets`, unfortunately, is inherently unsafe; `fgets` is a much better alternative.

`fgets` function ➤22.5

Reading Strings Character by Character

Since both `scanf` and `gets` are risky and insufficiently flexible for many applications, C programmers often write their own input functions. By reading strings one character at a time, these functions provide a greater degree of control than the standard input functions.

If we decide to design our own input function, we'll need to consider the following issues:

- Should the function skip white space before beginning to store the string?
- What character causes the function to stop reading: a new-line character, any white-space character, or some other character? Is this character stored in the string or discarded?
- What should the function do if the input string is too long to store: discard the extra characters or leave them for the next input operation?

Suppose we need a function that doesn't skip white-space characters, stops reading at the first new-line character (which isn't stored in the string), and discards extra characters. The function might have the following prototype:

```
int read_line(char str[], int n);
```

`str` represents the array into which we'll store the input, and `n` is the maximum number of characters to be read. If the input line contains more than `n` characters, `read_line` will discard the additional characters. We'll have `read_line` return the number of characters it actually stores in `str` (a number anywhere from 0 to `n`). We may not always need `read_line`'s return value, but it doesn't hurt to have it available.

getchar function ➤7.3

`read_line` consists primarily of a loop that calls `getchar` to read a character and then stores the character in `str`, provided that there's room left. The loop terminates when the new-line character is read. (Strictly speaking, we should also have the loop terminate if `getchar` should fail to read a character, but we'll ignore that complication for now.) Here's the complete definition of `read_line`:

Q&A

```
int read_line(char str[], int n)
{
  int ch, i = 0;

  while ((ch = getchar()) != '\n')
    if (i < n)
      str[i++] = ch;
  str[i] = '\0';        /* terminates string */
  return i;             /* number of characters stored */
}
```

Note that `ch` has `int` type rather than `char` type, because `getchar` returns the character that it reads as an `int` value.

Before returning, `read_line` puts a null character at the end of the string. Standard functions such as `scanf` and `gets` automatically put a null character at the end of an input string; if we're writing our own input function, however, we must take on that responsibility.

13.4 Accessing the Characters in a String

Since strings are stored as arrays, we can use subscripting to access the characters in a string. To process every character in a string `s`, for example, we can set up a loop that increments a counter `i` and selects characters via the expression `s[i]`.

Suppose that we need a function that counts the number of spaces in a string. Using array subscripting, we might write the function in the following way:

```
int count_spaces(const char s[])
{
  int count = 0, i;

  for (i = 0; s[i] != '\0'; i++)
    if (s[i] == ' ')
      count++;
  return count;
}
```

I've included `const` in the declaration of `s` to indicate that `count_spaces` doesn't change the array that `s` represents. If `s` were not a string, the function would need a second argument specifying the length of the array. Since `s` is a string, however, `count_spaces` can determine where it ends by testing for the null character.

Many C programmers wouldn't write `count_spaces` as we have. Instead, they'd use a pointer to keep track of the current position within the string. As we saw in Section 12.2, this technique is always available for processing arrays, but it proves to be especially convenient for working with strings.

Let's rewrite the `count_spaces` function using pointer arithmetic instead of array subscripting. We'll eliminate the variable `i` and use `s` itself to keep track of our position in the string. By incrementing `s` repeatedly, `count_spaces` can step through each character in the string. Here's our new version of the function:

```
int count_spaces(const char *s)
{
  int count = 0;

  for (; *s != '\0'; s++)
    if (*s == ' ')
      count++;
  return count;
}
```

Note that `const` doesn't prevent `count_spaces` from modifying `s`; it's there to prevent the function from modifying what `s` points to. And since `s` is a copy of the pointer that's passed to `count_spaces`, incrementing `s` doesn't affect the original pointer.

The `count_spaces` example raises some questions about how to write string functions:

- ***Is it better to use array operations or pointer operations to access the characters in a string?*** We're free to use whichever is more convenient; we can even mix the two. In the second version of `count_spaces`, treating `s` as a pointer simplifies the function slightly by removing the need for the variable `i`. Traditionally, C programmers lean toward using pointer operations for processing strings.
- ***Should a string parameter be declared as an array or as a pointer?*** The two versions of `count_spaces` illustrate the options: the first version declares `s` to be an array; the second declares `s` to be a pointer. Actually, there's no difference between the two declarations—recall from Section 12.3 that the compiler treats an array parameter as though it had been declared as a pointer.
- ***Does the form of the parameter (`s[]` or `*s`) affect what can be supplied as an argument?*** No. When `count_spaces` is called, the argument could be an array name, a pointer variable, or a string literal—`count_spaces` can't tell the difference.

13.5 Using the C String Library

Some programming languages provide operators that can copy strings, compare strings, concatenate strings, select substrings, and the like. C's operators, in contrast, are essentially useless for working with strings. Strings are treated as arrays in C, so they're restricted in the same ways as arrays—in particular, they can't be copied or compared using operators.

Direct attempts to copy or compare strings will fail. For example, suppose that `str1` and `str2` have been declared as follows:

```
char str1[10], str2[10];
```

Copying a string into a character array using the = operator is not possible:

```
str1 = "abc";   /*** WRONG ***/
str2 = str1;    /*** WRONG ***/
```

We saw in Section 12.3 that using an array name as the left operand of = is illegal. *Initializing* a character array using = is legal, though:

```
char str1[10] = "abc";
```

In the context of a declaration, = is not the assignment operator.

Attempting to compare strings using a relational or equality operator is legal but won't produce the desired result:

```
if (str1 == str2) ...   /*** WRONG ***/
```

This statement compares `str1` and `str2` as *pointers*; it doesn't compare the contents of the two arrays. Since `str1` and `str2` have different addresses, the expression `str1 == str2` must have the value 0.

Fortunately, all is not lost: the C library provides a rich set of functions for performing operations on strings. Prototypes for these functions reside in the `<string.h>` header, so programs that need string operations should contain the following line:

`<string.h>` header ➤23.6

```
#include <string.h>
```

Most of the functions declared in `<string.h>` require at least one string as an argument. String parameters are declared to have type `char *`, allowing the argument to be a character array, a variable of type `char *`, or a string literal—all are suitable as strings. Watch out for string parameters that aren't declared `const`, however. Such a parameter may be modified when the function is called, so the corresponding argument shouldn't be a string literal.

There are many functions in `<string.h>`; I'll cover a few of the most basic. In subsequent examples, assume that `str1` and `str2` are character arrays used as strings.

The `strcpy` (String Copy) Function

The `strcpy` function has the following prototype in `<string.h>`:

```
char *strcpy(char *s1, const char *s2);
```

`strcpy` copies the string `s2` into the string `s1`. (To be precise, we should say "`strcpy` copies the string pointed to by `s2` into the array pointed to by `s1`.") That is, `strcpy` copies characters from `s2` to `s1` up to (and including) the first null character in `s2`. `strcpy` returns `s1` (a pointer to the destination string). The string pointed to by `s2` isn't modified, so it's declared `const`.

The existence of `strcpy` compensates for the fact that we can't use the assignment operator to copy strings. For example, suppose that we want to store the string `"abcd"` in `str2`. We can't use the assignment

```
str2 = "abcd";              /*** WRONG ***/
```

because `str2` is an array name and can't appear on the left side of an assignment. Instead, we can call `strcpy`:

```
strcpy(str2, "abcd");   /* str2 now contains "abcd" */
```

Similarly, we can't assign `str2` to `str1` directly, but we can call `strcpy`:

```
strcpy(str1, str2);     /* str1 now contains "abcd" */
```

Most of the time, we'll discard the value that `strcpy` returns. On occasion, though, it can be useful to call `strcpy` as part of a larger expression in order to use its return value. For example, we could chain together a series of `strcpy` calls:

```
strcpy(str1, strcpy(str2, "abcd"));
  /* both str1 and str2 now contain "abcd" */
```

In the call `strcpy(str1, str2)`, `strcpy` has no way to check that the string pointed to by `str2` will actually fit in the array pointed to by `str1`. Suppose that `str1` points to an array of length n. If the string that `str2` points to has no more than $n - 1$ characters, then the copy will succeed. But if `str2` points to a longer string, undefined behavior occurs. (Since `strcpy` always copies up to the first null character, it will continue copying past the end of the array that `str1` points to.)

`strncpy` function ➤23.6

Calling the `strncpy` function is a safer, albeit slower, way to copy a string. `strncpy` is similar to `strcpy` but has a third argument that limits the number of characters that will be copied. To copy `str2` into `str1`, we could use the following call of `strncpy`:

```
strncpy(str1, str2, sizeof(str1));
```

As long as `str1` is large enough to hold the string stored in `str2` (including the null character), the copy will be done correctly. `strncpy` itself isn't without danger, though. For one thing, it will leave the string in `str1` without a terminating null character if the length of the string stored in `str2` is greater than or equal to the size of the `str1` array. Here's a safer way to use `strncpy`:

```
strncpy(str1, str2, sizeof(str1) - 1);
str1[sizeof(str1)-1] = '\0';
```

The second statement guarantees that `str1` is always null-terminated, even if `strncpy` fails to copy a null character from `str2`.

The `strlen` (String Length) Function

The `strlen` function has the following prototype:

```
size_t strlen(const char *s);
```

size_t type ➤7.6

`size_t`, which is defined in the C library, is a `typedef` name that represents one of C's unsigned integer types. Unless we're dealing with extremely long strings, this technicality need not concern us—we can simply treat the return value of `strlen` as an integer.

`strlen` returns the length of a string `s`: the number of characters in `s` up to, but not including, the first null character. Here are a few examples:

```
int len;

len = strlen("abc");    /* len is now 3 */
len = strlen("");       /* len is now 0 */
strcpy(str1, "abc");
len = strlen(str1);     /* len is now 3 */
```

The last example illustrates an important point. When given an array as its argument, `strlen` doesn't measure the length of the array itself; instead, it returns the length of the string stored in the array.

The `strcat` (String Concatenation) Function

The `strcat` function has the following prototype:

```
char *strcat(char *s1, const char *s2);
```

`strcat` appends the contents of the string `s2` to the end of the string `s1`; it returns `s1` (a pointer to the resulting string).

Here are some examples of `strcat` in action:

```
strcpy(str1, "abc");
strcat(str1, "def");   /* str1 now contains "abcdef" */
```

```
strcpy(str1, "abc");
strcpy(str2, "def");
strcat(str1, str2);     /* str1 now contains "abcdef" */
```

As with `strcpy`, the value returned by `strcat` is normally discarded. The following example shows how the return value might be used:

```
strcpy(str1, "abc");
strcpy(str2, "def");
strcat(str1, strcat(str2, "ghi"));
  /* str1 now contains "abcdefghi"; str2 contains "defghi" */
```

The effect of the call `strcat(str1, str2)` is undefined if the array pointed to by `str1` isn't long enough to accommodate the additional characters from `str2`. Consider the following example:

```
char str1[6] = "abc";

strcat(str1, "def");    /*** WRONG ***/
```

`strcat` will attempt to add the characters `d`, `e`, `f`, and `\0` to the end of the string already stored in `str1`. Unfortunately, `str1` is limited to six characters, causing `strcat` to write past the end of the array.

strncat function ➤23.6

The `strncat` function is a safer but slower version of `strcat`. Like `strncpy`, it has a third argument that limits the number of characters it will copy. Here's what a call might look like:

```
strncat(str1, str2, sizeof(str1) - strlen(str1) - 1);
```

`strncat` will terminate `str1` with a null character, which isn't included in the third argument (the number of characters to be copied). In the example, the third argument calculates the amount of space remaining in `str1` (given by the expression `sizeof(str1) - strlen(str1)`) and then subtracts 1 to ensure that there will be room for the null character.

The `strcmp` (String Comparison) Function

The `strcmp` function has the following prototype:

```
int strcmp(const char *s1, const char *s2);
```

`strcmp` compares the strings `s1` and `s2`, returning a value less than, equal to, or greater than 0, depending on whether `s1` is less than, equal to, or greater than `s2`. For example, to see if `str1` is less than `str2`, we'd write

```
if (strcmp(str1, str2) < 0)     /* is str1 < str2? */
  ...
```

To test whether `str1` is less than or equal to `str2`, we'd write

```
if (strcmp(str1, str2) <= 0)    /* is str1 <= str2? */
  ...
```

By choosing the proper relational operator (<, <=, >, >=) or equality operator (==, !=), we can test any possible relationship between `str1` and `str2`.

`strcmp` compares strings based on their lexicographic ordering, which resembles the way words are arranged in a dictionary. More precisely, `strcmp` considers `s1` to be less than `s2` if either one of the following conditions is satisfied:

- The first *i* characters of `s1` and `s2` match, but the (*i*+1)st character of `s1` is less than the (*i*+1)st character of `s2`. For example, `"abc"` is less than `"bcd"`, and `"abd"` is less than `"abe"`.
- All characters of `s1` match `s2`, but `s1` is shorter than `s2`. For example, `"abc"` is less than `"abcd"`.

As it compares characters from two strings, `strcmp` looks at the numerical codes that represent the characters. Some knowledge of the underlying character set is helpful in order to predict what `strcmp` will do. For example, here are a few important properties of the ASCII character set:

ASCII character set ➤ *Appendix E*

- The characters in each of the sequences A–Z, a–z, and 0–9 have consecutive codes.
- All upper-case letters are less than all lower-case letters. (In ASCII, codes between 65 and 90 represent upper-case letters; codes between 97 and 122 represent lower-case letters.)
- Digits are less than letters. (Codes between 48 and 57 represent digits.)
- Spaces are less than all printing characters. (The space character has the value 32 in ASCII.)

PROGRAM Printing a One-Month Reminder List

To illustrate the use of the C string library, we'll now develop a program that prints a one-month list of daily reminders. The user will enter a series of reminders, with each prefixed by a day of the month. When the user enters 0 instead of a valid day, the program will print a list of all reminders entered, sorted by day. Here's what a session with the program will look like:

```
Enter day and reminder: 24 Susan's birthday
Enter day and reminder: 5 6:00 - Dinner with Marge and Russ
Enter day and reminder: 26 Movie - "Chinatown"
Enter day and reminder: 7 10:30 - Dental appointment
Enter day and reminder: 12 Movie - "Dazed and Confused"
Enter day and reminder: 5 Saturday class
Enter day and reminder: 12 Saturday class
Enter day and reminder: 0
```

```
Day Reminder
  5 Saturday class
  5 6:00 - Dinner with Marge and Russ
  7 10:30 - Dental appointment
 12 Saturday class
 12 Movie - "Dazed and Confused"
 24 Susan's birthday
 26 Movie - "Chinatown"
```

The overall strategy isn't very complicated: we'll have the program read a series of day-and-reminder combinations, storing them in order (sorted by day), and then display them. To read the days, we'll use `scanf`; to read the reminders, we'll use the `read_line` function of Section 13.3.

We'll store the strings in a two-dimensional array of characters, with each row of the array containing one string. After the program reads a day and its associated reminder, it will search the array to determine where the day belongs, using `strcmp` to do comparisons. It will then use `strcpy` to move all strings *below* that point down one position. Finally, the program will copy the day into the array and call `strcat` to append the reminder to the day. (The day and the reminder have been kept separate up to this point.).

Of course, there are always a few minor complications. For example, we want the days to be right-justified in a two-character field, so that their ones digits will line up. There are many ways to handle the problem. I've chosen to have the program use `scanf` to read the day into an integer variable, then call `sprintf` to convert the day back into string form. `sprintf` is a library function that's similar to `printf`, except that it writes output into a string. The call

sprintf function ➤22.8

```
sprintf(day_str, "%2d", day);
```

writes the value of `day` into `day_str`. Since `sprintf` automatically adds a null character when it's through writing, `day_str` will contain a properly null-terminated string.

Another complication is making sure that the user doesn't enter more than two digits. We'll use the following call of `scanf` for this purpose:

```
scanf("%2d", &day);
```

The number 2 between `%` and `d` tells `scanf` to stop reading after two digits, even if the input has more digits.

With those details out of the way, here's the program:

remind.c

```
/* Prints a one-month reminder list */

#include <stdio.h>
#include <string.h>

#define MAX_REMIND 50   /* maximum number of reminders */
#define MSG_LEN 60      /* max length of reminder message */
```

```
int read_line(char str[], int n);

int main(void)
{
  char reminders[MAX_REMIND][MSG_LEN+3];
  char day_str[3], msg_str[MSG_LEN+1];
  int day, i, j, num_remind = 0;

  for (;;) {
    if (num_remind == MAX_REMIND) {
      printf("-- No space left --\n");
      break;
    }

    printf("Enter day and reminder: ");
    scanf("%2d", &day);
    if (day == 0)
      break;
    sprintf(day_str, "%2d", day);
    read_line(msg_str, MSG_LEN);

    for (i = 0; i < num_remind; i++)
      if (strcmp(day_str, reminders[i]) < 0)
        break;
    for (j = num_remind; j > i; j--)
      strcpy(reminders[j], reminders[j-1]);

    strcpy(reminders[i], day_str);
    strcat(reminders[i], msg_str);

    num_remind++;
  }

  printf("\nDay Reminder\n");
  for (i = 0; i < num_remind; i++)
    printf(" %s\n", reminders[i]);

  return 0;
}

int read_line(char str[], int n)
{
  int ch, i = 0;

  while ((ch = getchar()) != '\n')
    if (i < n)
      str[i++] = ch;
  str[i] = '\0';
  return i;
}
```

Although `remind.c` is useful for demonstrating the `strcpy`, `strcat`, and `strcmp` functions, it lacks something as a practical reminder program. There are

obviously a number of improvements needed, ranging from minor tweaks to major enhancements (such as saving the reminders in a file when the program terminates). We'll discuss several improvements in the programming projects at the end of this chapter and in later chapters.

13.6 String Idioms

Functions that manipulate strings are a particularly rich source of idioms. In this section, we'll explore some of the most famous idioms by using them to write the `strlen` and `strcat` functions. You'll never have to write these functions, of course, since they're part of the standard library, but you may have to write functions that are similar.

The concise style I'll use in this section is popular with many C programmers. You should master this style even if you don't plan to use it in your own programs, since you're likely to encounter it in code written by others.

One last note before we get started. If you want to try out any of the versions of `strlen` and `strcat` in this section, be sure to alter the name of the function (changing `strlen` to `my_strlen`, for example). As Section 21.1 explains, we're not allowed to write a function that has the same name as a standard library function, even when we don't include the header to which the function belongs. In fact, all names that begin with `str` and a lower-case letter are reserved (to allow functions to be added to the `<string.h>` header in future versions of the C standard).

Searching for the End of a String

Many string operations require searching for the end of a string. The `strlen` function is a prime example. The following version of `strlen` searches its string argument to find the end, using a variable to keep track of the string's length:

```
size_t strlen(const char *s)
{
  size_t n;

  for (n = 0; *s != '\0'; s++)
    n++;
  return n;
}
```

As the pointer `s` moves across the string from left to right, the variable `n` keeps track of how many characters have been seen so far. When `s` finally points to a null character, `n` contains the length of the string.

Let's see if we can condense the function. First, we'll move the initialization of `n` to its declaration:

```
size_t strlen(const char *s)
{
  size_t n = 0;

  for (; *s != '\0'; s++)
    n++;
  return n;
}
```

Next, we notice that the condition `*s != '\0'` is the same as `*s != 0`, because the integer value of the null character is 0. But testing `*s != 0` is the same as testing `*s`; both are true if `*s` isn't equal to 0. These observations lead to our next version of `strlen`:

```
size_t strlen(const char *s)
{
  size_t n = 0;

  for (; *s; s++)
    n++;
  return n;
}
```

But, as we saw in Section 12.2, it's possible to increment `s` and test `*s` in the same expression:

```
size_t strlen(const char *s)
{
  size_t n = 0;

  for (; *s++;)
    n++;
  return n;
}
```

Replacing the `for` statement with a `while` statement, we arrive at the following version of `strlen`:

```
size_t strlen(const char *s)
{
  size_t n = 0;

  while (*s++)
    n++;
  return n;
}
```

Although we've condensed `strlen` quite a bit, it's likely that we haven't increased its speed. Here's a version that *does* run faster, at least with some compilers:

```
size_t strlen(const char *s)
{
  const char *p = s;
```

```
  while (*s)
    s++;
  return s - p;
}
```

This version of `strlen` computes the length of the string by locating the position of the null character, then subtracting from it the position of the first character in the string. The improvement in speed comes from not having to increment `n` inside the `while` loop. Note the appearance of the word `const` in the declaration of `p`, by the way; without it, the compiler would notice that assigning `s` to `p` places the string that `s` points to at risk.

The statement

idiom
```
while (*s)
  s++;
```

and the related

idiom
```
while (*s++)
  ;
```

are idioms meaning "search for the null character at the end of a string." The first version leaves `s` pointing to the null character. The second version is more concise, but leaves `s` pointing just past the null character.

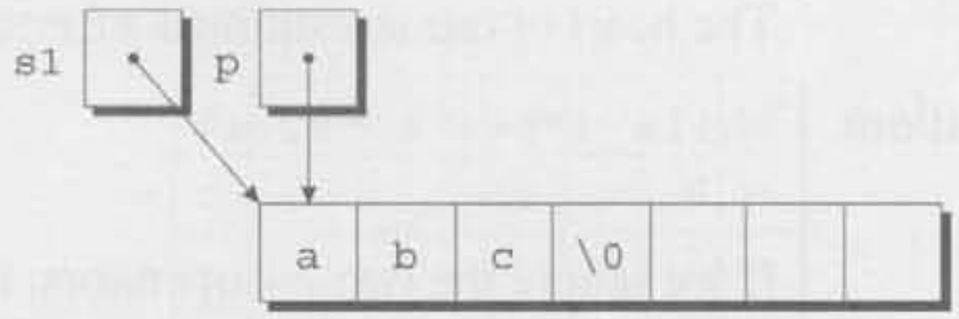
Copying a String

Copying a string is another common operation. To introduce C's "string copy" idiom, we'll develop two versions of the `strcat` function. Let's start with a straightforward but somewhat lengthy version:

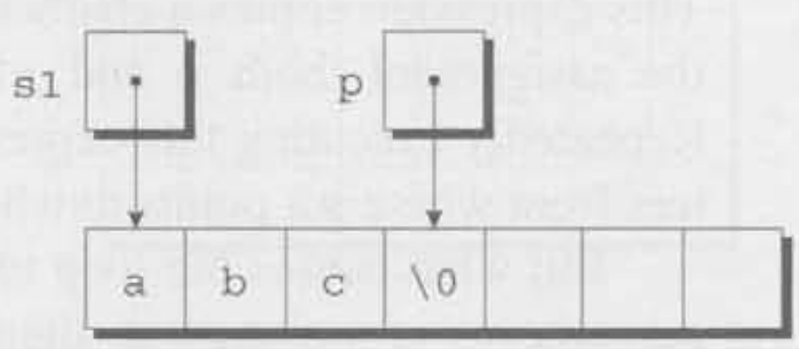
```
char *strcat(char *s1, const char *s2)
{
  char *p = s1;

  while (*p != '\0')
    p++;
  while (*s2 != '\0') {
    *p = *s2;
    p++;
    s2++;
  }
  *p = '\0';
  return s1;
}
```

This version of `strcat` uses a two-step algorithm: (1) Locate the null character at the end of the string `s1` and make `p` point to it. (2) Copy characters one by one from `s2` to where `p` is pointing.

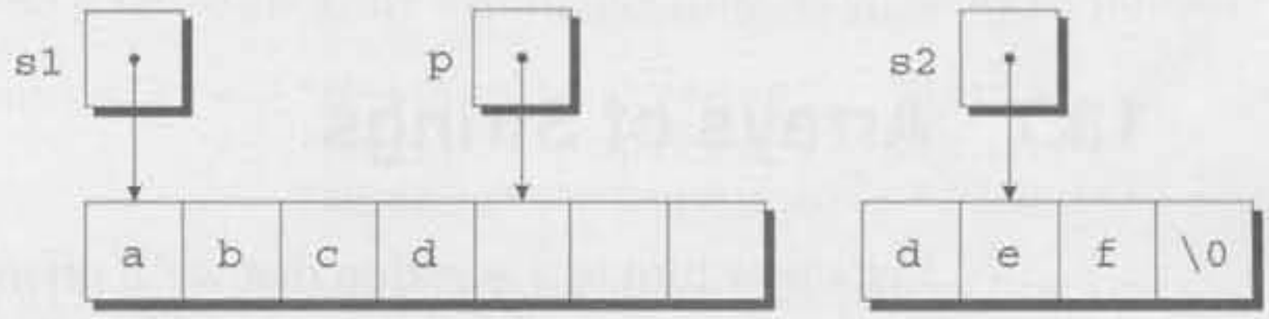
The first `while` statement in the function implements step (1). `p` is set to point to the first character in the `s1` string. Assuming that `s1` points to the string `"abc"`, we have the following picture:



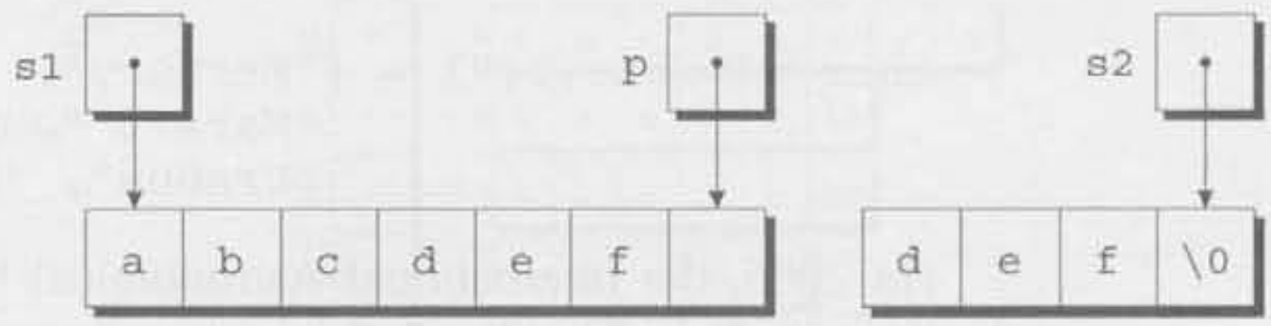
p is then incremented as long as it doesn't point to a null character. When the loop terminates, p must be pointing to the null character:



The second `while` statement implements step (2). The loop body copies one character from where `s2` points to where `p` points, then increments both `p` and `s2`. If `s2` originally points to the string `"def"`, here's what the strings will look like after the first loop iteration:



The loop terminates when `s2` points to the null character:



After putting a null character where `p` is pointing, `strcat` returns.

By a process similar to the one we used for `strlen`, we can condense the definition of `strcat`, arriving at the following version:

```
char *strcat(char *s1, const char *s2)
{
  char *p = s1;

  while (*p)
    p++;
  while (*p++ = *s2++)
    ;
  return s1;
}
```

The heart of our streamlined `strcat` function is the "string copy" idiom:

idiom
```
while (*p++ = *s2++)
  ;
```

If we ignore the two ++ operators, the expression inside the parentheses simplifies to an ordinary assignment:

```
*p = *s2
```

This expression copies a character from where `s2` points to where `p` points. After the assignment, both `p` and `s2` are incremented, thanks to the ++ operators. Repeatedly executing this expression has the effect of copying a series of characters from where `s2` points to where `p` points.

But what causes the loop to terminate? Since the primary operator inside the parentheses is assignment, the `while` statement tests the value of the assignment—the character that was copied. All characters except the null character test true, so the loop won't terminate until the null character has been copied. And since the loop terminates *after* the assignment, we don't need a separate statement to put a null character at the end of the new string.

13.7 Arrays of Strings

Let's now turn to a question that we'll often encounter: what's the best way to store an array of strings? The obvious solution is to create a two-dimensional array of characters, then store the strings in the array, one per row. Consider the following example:

```
char planets[][8] = {"Mercury", "Venus", "Earth",
                     "Mars", "Jupiter", "Saturn",
                     "Uranus", "Neptune", "Pluto"};
```

(In 2006, the International Astronomical Union demoted Pluto from "planet" to "dwarf planet," but I've left it in the `planets` array for old times' sake.) Note that we're allowed to omit the number of rows in the `planets` array—since that's obvious from the number of elements in the initializer—but C requires that we specify the number of columns.

The figure at the top of the next page shows what the `planets` array will look like. Not all our strings were long enough to fill an entire row of the array, so C padded them with null characters. There's a bit of wasted space in this array, since only three planets have names long enough to require eight characters (including the terminating null character). The `remind.c` program (Section 13.5) is a glaring example of this kind of waste. It stores reminders in rows of a two-dimensional character array, with 60 characters set aside for each reminder. In our example, the reminders ranged from 18 to 37 characters in length, so the amount of wasted space was considerable.

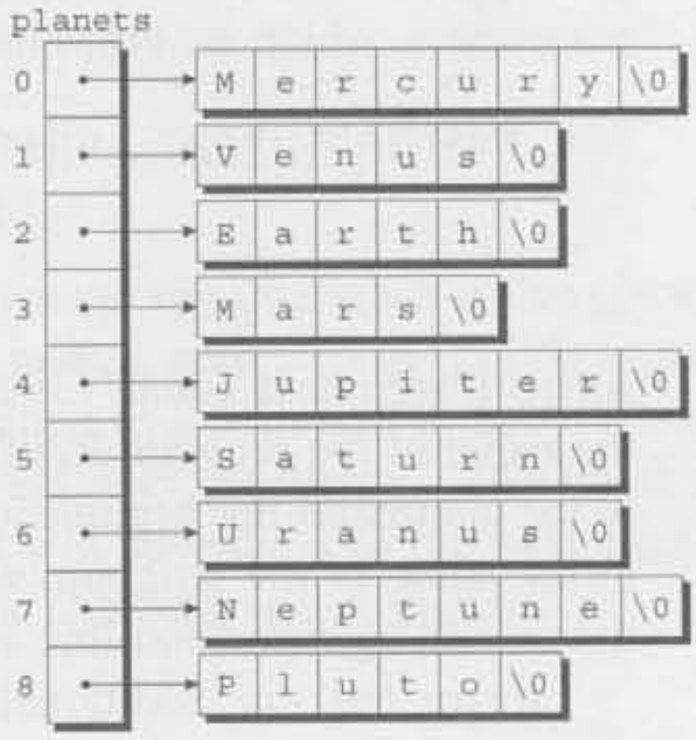
	0	1	2	3	4	5	6	7
0	M	e	r	c	u	r	y	\0
1	V	e	n	u	s	\0	\0	\0
2	E	a	r	t	h	\0	\0	\0
3	M	a	r	s	\0	\0	\0	\0
4	J	u	p	i	t	e	r	\0
5	S	a	t	u	r	n	\0	\0
6	U	r	a	n	u	s	\0	\0
7	N	e	p	t	u	n	e	\0
8	P	l	u	t	o	\0	\0	\0

The inefficiency that's apparent in these examples is common when working with strings, since most collections of strings will have a mixture of long strings and short strings. What we need is a ***ragged array:*** a two-dimensional array whose rows can have different lengths. C doesn't provide a "ragged array type," but it does give us the tools to simulate one. The secret is to create an array whose elements are *pointers* to strings.

Here's the `planets` array again, this time as an array of pointers to strings:

```
char *planets[] = {"Mercury", "Venus", "Earth",
                   "Mars", "Jupiter", "Saturn",
                   "Uranus", "Neptune", "Pluto"};
```

Not much of a change, eh? We simply removed one pair of brackets and put an asterisk in front of `planets`. The effect on how `planets` is stored is dramatic, though:



Each element of `planets` is a pointer to a null-terminated string. There are no longer any wasted characters in the strings, although we've had to allocate space for the pointers in the `planets` array.

To access one of the planet names, all we need do is subscript the `planets` array. Because of the relationship between pointers and arrays, accessing a character in a planet name is done in the same way as accessing an element of a two-

dimensional array. To search the `planets` array for strings beginning with the letter M, for example, we could use the following loop:

```
for (i = 0; i < 9; i++)
  if (planets[i][0] == 'M')
    printf("%s begins with M\n", planets[i]);
```

Command-Line Arguments

When we run a program, we'll often need to supply it with information—a file name, perhaps, or a switch that modifies the program's behavior. Consider the UNIX `ls` command. If we run `ls` by typing

```
ls
```

at the command line, it will display the names of the files in the current directory. But if we instead type

```
ls -l
```

then `ls` will display a "long" (detailed) listing of files, showing the size of each file, the file's owner, the date and time the file was last modified, and so forth. To modify the behavior of `ls` further, we can specify that it show details for just one file:

```
ls -l remind.c
```

`ls` will display detailed information about the file named `remind.c`.

Command-line information is available to all programs, not just operating system commands. To obtain access to these ***command-line arguments*** (called ***program parameters*** in the C standard), we must define `main` as a function with two parameters, which are customarily named `argc` and `argv`:

Q&A

Q&A

```
int main(int argc, char *argv[])
{
  ...
}
```

`argc` ("argument count") is the number of command-line arguments (including the name of the program itself). `argv` ("argument vector") is an array of pointers to the command-line arguments, which are stored in string form. `argv[0]` points to the name of the program, while `argv[1]` through `argv[argc-1]` point to the remaining command-line arguments.

`argv` has one additional element, `argv[argc]`, which is always a ***null pointer***—a special pointer that points to nothing. We'll discuss null pointers in a later chapter; for now, all we need to know is that the macro `NULL` represents a null pointer.

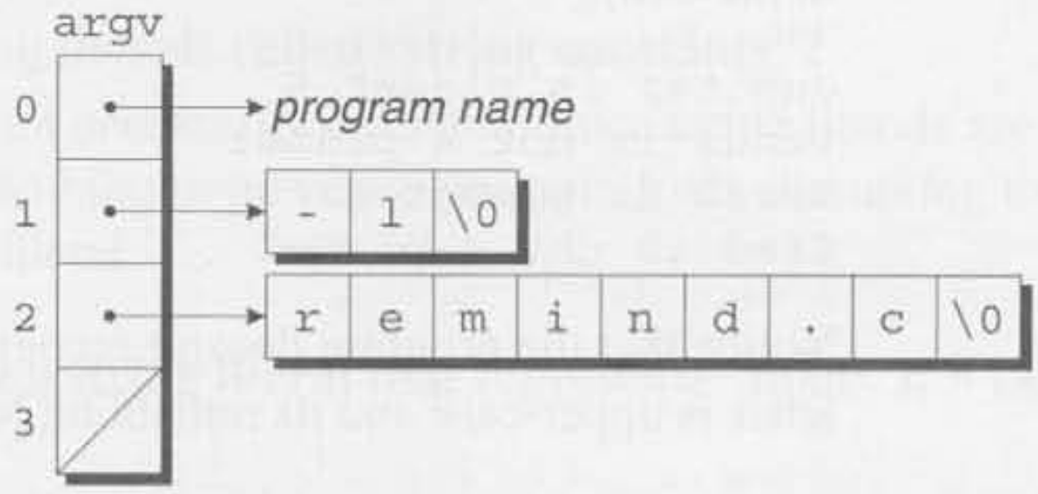
null pointers ➤ 17.1

If the user enters the command line

```
ls -l remind.c
```

then `argc` will be 3, `argv[0]` will point to a string containing the program

name, `argv[1]` will point to the string `"-l"`, `argv[2]` will point to the string `"remind.c"`, and `argv[3]` will be a null pointer:



This figure doesn't show the program name in detail, since it may include a path or other information that depends on the operating system. If the program name isn't available, `argv[0]` points to an empty string.

Since `argv` is an array of pointers, accessing command-line arguments is easy. Typically, a program that expects command-line arguments will set up a loop that examines each argument in turn. One way to write such a loop is to use an integer variable as an index into the `argv` array. For example, the following loop prints the command-line arguments, one per line:

```
int i;

for (i = 1; i < argc; i++)
  printf("%s\n", argv[i]);
```

Another technique is to set up a pointer to `argv[1]`, then increment the pointer repeatedly to step through the rest of the array. Since the last element of `argv` is always a null pointer, the loop can terminate when it finds a null pointer in the array:

```
char **p;

for (p = &argv[1]; *p != NULL; p++)
  printf("%s\n", *p);
```

Since `p` is a *pointer* to a *pointer* to a character, we've got to use it carefully. Setting `p` equal to `&argv[1]` makes sense; `argv[1]` is a pointer to a character, so `&argv[1]` will be a pointer to a pointer. The test `*p != NULL` is OK, since `*p` and `NULL` are both pointers. Incrementing `p` looks good; `p` points to an array element, so incrementing it will advance it to the next element. Printing `*p` is fine, since `*p` points to the first character in a string.

PROGRAM

Checking Planet Names

Our next program, `planet.c`, illustrates how to access command-line arguments. The program is designed to check a series of strings to see which ones are names of planets. When the program is run, the user will put the strings to be tested on the command line:

```
planet Jupiter venus Earth fred
```

The program will indicate whether or not each string is a planet name; if it is, the program will also display the planet's number (with planet 1 being the one closest to the Sun):

```
Jupiter is planet 5
venus is not a planet
Earth is planet 3
fred is not a planet
```

Notice that the program doesn't recognize a string as a planet name unless its first letter is upper-case and its remaining letters are lower-case.

planet.c

```
/* Checks planet names */

#include <stdio.h>
#include <string.h>

#define NUM_PLANETS 9

int main(int argc, char *argv[])
{
  char *planets[] = {"Mercury", "Venus", "Earth",
                     "Mars", "Jupiter", "Saturn",
                     "Uranus", "Neptune", "Pluto"};
  int i, j;

  for (i = 1; i < argc; i++) {
    for (j = 0; j < NUM_PLANETS; j++)
      if (strcmp(argv[i], planets[j]) == 0) {
        printf("%s is planet %d\n", argv[i], j + 1);
        break;
      }
    if (j == NUM_PLANETS)
      printf("%s is not a planet\n", argv[i]);
  }

  return 0;
}
```

The program visits each command-line argument in turn, comparing it with the strings in the `planets` array until it finds a match or reaches the end of the array. The most interesting part of the program is the call of `strcmp`, in which the arguments are `argv[i]` (a pointer to a command-line argument) and `planets[j]` (a pointer to a planet name).

Q & A

Q: How long can a string literal be?

A: According to the C89 standard, compilers must allow string literals to be at least

509 characters long. (Yes, you read that right—509. Don't ask.) C99 increases the
C99 minimum to 4095 characters.

Q: Why aren't string literals called "string constants"?

A: Because they're not necessarily constant. Since string literals are accessed through pointers, there's nothing to prevent a program from attempting to modify the characters in a string literal.

Q: How do we write a string literal that represents "über" if `"\xfcber"` doesn't work? [p. 278]

A: The secret is to write two adjacent string literals and let the compiler join them into one. In this example, writing `"\xfc" "ber"` will give us a string literal that represents the word "über."

Q: Modifying a string literal seems harmless enough. Why does it cause undefined behavior? [p. 280]

A: Some compilers try to reduce memory requirements by storing single copies of identical string literals. Consider the following example:

```
char *p = "abc", *q = "abc";
```

A compiler might choose to store `"abc"` just once, making both `p` and `q` point to it. If we were to change `"abc"` through the pointer `p`, the string that `q` points to would also be affected. Needless to say, this could lead to some annoying bugs. Another potential problem is that string literals might be stored in a "read-only" area of memory; a program that attempts to modify such a literal will simply crash.

Q: Should every array of characters include room for a null character?

A: Not necessarily, since not every array of characters is used as a string. Including room for the null character (and actually putting one into the array) is necessary only if you're planning to pass it to a function that requires a null-terminated string.

You do *not* need a null character if you'll only be performing operations on individual characters. For example, a program might have an array of characters that it will use to translate from one character set to another:

```
char translation_table[128];
```

The only operation that the program will perform on this array is subscripting. (The value of `translation_table[ch]` will be the translated version of the character `ch`.) We would not consider `translation_table` to be a string: it need not contain a null character, and no string operations will be performed on it.

Q: If `printf` and `scanf` expect their first argument to have type `char *`, does that mean that the argument can be a string *variable* instead of a string *literal*?

A: Yes, as the following example shows:

```
char fmt[] = "%d\n";
int i;
...
printf(fmt, i);
```

This ability opens the door to some intriguing possibilities—reading a format string as input, for example.

Q: If I want `printf` to write a string `str`, can't I just supply `str` as the format string, as in the following example?

```
printf(str);
```

A: Yes, but it's risky. If `str` contains the `%` character, you won't get the desired result, since `printf` will assume it's the beginning of a conversion specification.

***Q: How can `read_line` detect whether `getchar` has failed to read a character? [p. 287]**

A: If it can't read a character, either because of an error or because of end-of-file, `getchar` returns the value `EOF`, which has type `int`. Here's a revised version of `read_line` that tests whether the return value of `getchar` is `EOF`. Changes are marked in **bold:**

EOF macro ➤22.4

```
int read_line(char str[], int n)
{
  int ch, i = 0;

  while ((ch = getchar()) != '\n' && ch != EOF)
    if (i < n)
      str[i++] = ch;
  str[i] = '\0';
  return i;
}
```

Q: Why does `strcmp` return a number that's less than, equal to, or greater than zero? Also, does the exact return value have any significance? [p. 292]

A: `strcmp`'s return value probably stems from the way the function is traditionally written. Consider the version in Kernighan and Ritchie's *The C Programming Language:*

```
int strcmp(char *s, char *t)
{
  int i;

  for (i = 0; s[i] == t[i]; i++)
    if (s[i] == '\0')
      return 0;
  return s[i] - t[i];
}
```

The return value is the difference between the first "mismatched" characters in the `s` and `t` strings, which will be negative if `s` points to a "smaller" string than `t` and positive if `s` points to a "larger" string. There's no guarantee that `strcmp` is actually written this way, though, so it's best not to assume that the magnitude of its return value has any particular meaning.

Q: My compiler issues a warning when I try to compile the `while` statement in the `strcat` function:

```
while (*p++ = *s2++)
  ;
```

What am I doing wrong?

A: Nothing. Many compilers—but not all, by any means—issue a warning if you use = where == is normally expected. This warning is valid at least 95% of the time, and it will save you a lot of debugging if you heed it. Unfortunately, the warning isn't relevant in this particular example; we actually *do* mean to use =, not ==. To get rid of the warning, rewrite the `while` loop as follows:

```
while ((*p++ = *s2++) != 0)
  ;
```

Since the `while` statement normally tests whether `*p++ = *s2++` is not 0, we haven't changed the meaning of the statement. The warning goes away, however, because the statement now tests a condition, not an assignment. With the GCC compiler, putting a pair of parentheses around the assignment is another way to avoid a warning:

```
while ((*p++ = *s2++))
  ;
```

Q: Are the `strlen` and `strcat` functions actually written as shown in Section 13.6?

A: Possibly, although it's common practice for compiler vendors to write these functions—and many other string functions—in assembly language instead of C. The string functions need to be as fast as possible, since they're used often and have to deal with strings of arbitrary length. Writing these functions in assembly language makes it possible to achieve great efficiency by taking advantage of any special string-handling instructions that the CPU may provide.

Q: Why does the C standard use the term "program parameters" instead of "command-line arguments"? [p. 302]

A: Programs aren't always run from a command line. In a typical graphical user interface, for example, programs are launched with a mouse click. In such an environment, there's no traditional command line, although there may be other ways of passing information to a program; the term "program parameters" leaves the door open for these alternatives.

Q: Do I have to use the names `argc` and `argv` for `main`'s parameters? [p. 302]

A: No. Using the names `argc` and `argv` is merely a convention, not a language requirement.

Q: I've seen `argv` declared as `**argv` instead of `*argv[]`. Is this legal?

A: Certainly. When declaring a parameter, writing `*a` is always the same as writing `a[]`, regardless of the type of `a`'s elements.

Q: We've seen how to set up an array whose elements are pointers to string literals. Are there any other applications for arrays of pointers?

A: Yes. Although we've focused on arrays of pointers to character strings, that's not the only application of arrays of pointers. We could just as easily have an array whose elements point to any type of data, whether in array form or not. Arrays of pointers are particularly useful in conjunction with dynamic storage allocation.

dynamic storage allocation ➤17.1

Exercises

Section 13.3

1. The following function calls supposedly write a single new-line character, but some are incorrect. Identify which calls don't work and explain why.

```
(a) printf("%c", '\n');      (g) putchar('\n');
(b) printf("%c", "\n");      (h) putchar("\n");
(c) printf("%s", '\n');      (i) puts('\n');
(d) printf("%s", "\n");      (j) puts("\n");
(e) printf('\n');            (k) puts("");
(f) printf("\n");
```

W 2. Suppose that `p` has been declared as follows:

```
char *p = "abc";
```

Which of the following function calls are legal? Show the output produced by each legal call, and explain why the others are illegal.

```
(a) putchar(p);
(b) putchar(*p);
(c) puts(p);
(d) puts(*p);
```

*3. Suppose that we call `scanf` as follows:

```
scanf("%d%s%d", &i, s, &j);
```

If the user enters `12abc34 56def78`, what will be the values of `i`, `s`, and `j` after the call? (Assume that `i` and `j` are `int` variables and `s` is an array of characters.)

W 4. Modify the `read_line` function in each of the following ways:

(a) Have it skip white space before beginning to store input characters.

(b) Have it stop reading at the first white-space character. *Hint:* To determine whether or not a character is white space, call the `isspace` function.

isspace function ➤23.5

(c) Have it stop reading at the first new-line character, then store the new-line character in the string.
(d) Have it leave behind characters that it doesn't have room to store.

Section 13.4

5. (a) Write a function named `capitalize` that capitalizes all letters in its argument. The argument will be a null-terminated string containing arbitrary characters, not just letters. Use array subscripting to access the characters in the string. *Hint:* Use the `toupper` function to convert each character to upper-case.

`toupper` function ➤23.5

(b) Rewrite the `capitalize` function, this time using pointer arithmetic to access the characters in the string.

W 6. Write a function named `censor` that modifies a string by replacing every occurrence of `foo` by `xxx`. For example, the string `"food fool"` would become `"xxxd xxxl"`. Make the function as short as possible without sacrificing clarity.

Section 13.5

7. Suppose that `str` is an array of characters. Which one of the following statements is not equivalent to the other three?

(a) `*str = 0;`
(b) `str[0] = '\0';`
(c) `strcpy(str, "");`
(d) `strcat(str, "");`

W *8. What will be the value of the string `str` after the following statements have been executed?

```
strcpy(str, "tire-bouchon");
strcpy(&str[4], "d-or-wi");
strcat(str, "red?");
```

9. What will be the value of the string `s1` after the following statements have been executed?

```
strcpy(s1, "computer");
strcpy(s2, "science");
if (strcmp(s1, s2) < 0)
  strcat(s1, s2);
else
  strcat(s2, s1);
s1[strlen(s1)-6] = '\0';
```

W 10. The following function supposedly creates an identical copy of a string. What's wrong with the function?

```
char *duplicate(const char *p)
{
  char *q;

  strcpy(q, p);
  return q;
}
```

11. The Q&A section at the end of this chapter shows how the `strcmp` function might be written using array subscripting. Modify the function to use pointer arithmetic instead.

12. Write the following function:

```
void get_extension(const char *file_name, char *extension);
```

`file_name` points to a string containing a file name. The function should store the extension on the file name in the string pointed to by `extension`. For example, if the file name is `"memo.txt"`, the function will store `"txt"` in the string pointed to by `extension`. If the file name doesn't have an extension, the function should store an empty string (a single null character) in the string pointed to by `extension`. Keep the function as simple as possible by having it use the `strlen` and `strcpy` functions.

13. Write the following function:

```
void build_index_url(const char *domain, char *index_url);
```

`domain` points to a string containing an Internet domain, such as `"knking.com"`. The function should add `"http://www."` to the beginning of this string and `"/index.html"` to the end of the string, storing the result in the string pointed to by `index_url`. (In this example, the result will be `"http://www.knking.com/index.html"`.) You may assume that `index_url` points to a variable that is long enought to hold the resulting string. Keep the function as simple as possible by having it use the `strcat` and `strcpy` functions.

Section 13.6

*14. What does the following program print?

```
#include <stdio.h>

int main(void)
{
  char s[] = "Hsjodi", *p;

  for (p = s; *p; p++)
    --*p;
  puts(s);
  return 0;
}
```

W *15. Let `f` be the following function:

```
int f(char *s, char *t)
{
  char *p1, *p2;

  for (p1 = s; *p1; p1++) {
    for (p2 = t; *p2; p2++)
      if (*p1 == *p2) break;
    if (*p2 == '\0') break;
  }
  return p1 - s;
}
```

(a) What is the value of `f("abcd", "babc")`?
(b) What is the value of `f("abcd", "bcd")`?
(c) In general, what value does `f` return when passed two strings `s` and `t`?

W 16. Use the techniques of Section 13.6 to condense the `count_spaces` function of Section 13.4. In particular, replace the `for` statement by a `while` loop.

17. Write the following function:

```
bool test_extension(const char *file_name,
                    const char *extension);
```

`file_name` points to a string containing a file name. The function should return `true` if the file's extension matches the string pointed to by `extension`, ignoring the case of letters. For example, the call `test_extension("memo.txt", "TXT")` would return `true`. Incorporate the "search for the end of a string" idiom into your function. *Hint:* Use the `toupper` function to convert characters to upper-case before comparing them.

`toupper` function ➤23.5

18. Write the following function:

```
void remove_filename(char *url);
```

`url` points to a string containing a URL (Uniform Resource Locator) that ends with a file name (such as `"http://www.knking.com/index.html"`). The function should modify the string by removing the file name and the preceding slash. (In this example, the result will be `"http://www.knking.com"`.) Incorporate the "search for the end of a string" idiom into your function. *Hint:* Have the function replace the last slash in the string by a null character.

Programming Projects

W 1. Write a program that finds the "smallest" and "largest" in a series of words. After the user enters the words, the program will determine which words would come first and last if the words were listed in dictionary order. The program must stop accepting input when the user enters a four-letter word. Assume that no word is more than 20 letters long. An interactive session with the program might look like this:

```
Enter word: dog
Enter word: zebra
Enter word: rabbit
Enter word: catfish
Enter word: walrus
Enter word: cat
Enter word: fish

Smallest word: cat
Largest word: zebra
```

Hint: Use two strings named `smallest_word` and `largest_word` to keep track of the "smallest" and "largest" words entered so far. Each time the user enters a new word, use `strcmp` to compare it with `smallest_word`; if the new word is "smaller," use `strcpy` to save it in `smallest_word`. Do a similar comparison with `largest_word`. Use `strlen` to determine when the user has entered a four-letter word.

2. Improve the `remind.c` program of Section 13.5 in the following ways:
   (a) Have the program print an error message and ignore a reminder if the corresponding day is negative or larger than 31. *Hint:* Use the `continue` statement.
   (b) Allow the user to enter a day, a 24-hour time, and a reminder. The printed reminder list should be sorted first by day, then by time. (The original program allows the user to enter a time, but it's treated as part of the reminder.)
   (c) Have the program print a one-*year* reminder list. Require the user to enter days in the form *month/day*.

3. Modify the `deal.c` program of Section 8.2 so that it prints the full names of the cards it deals:

```
Enter number of cards in hand: 5
Your hand:
Seven of clubs
Two of spades
Five of diamonds
Ace of spades
Two of hearts
```

Hint: Replace `rank_code` and `suit_code` by arrays containing pointers to strings.

W 4. Write a program named `reverse.c` that echoes its command-line arguments in reverse order. Running the program by typing

```
reverse void and null
```

should produce the following output:

```
null and void
```

5. Write a program named `sum.c` that adds up its command-line arguments, which are assumed to be integers. Running the program by typing

```
sum 8 24 62
```

should produce the following output:

```
Total: 94
```

atoi function ➤26.2

Hint: Use the `atoi` function to convert each command-line argument from string form to integer form.

W 6. Improve the `planet.c` program of Section 13.7 by having it ignore case when comparing command-line arguments with strings in the `planets` array.

7. Modify Programming Project 11 from Chapter 5 so that it uses arrays containing pointers to strings instead of `switch` statements. For example, instead of using a `switch` statement to print the word for the first digit, use the digit as an index into an array that contains the strings `"twenty"`, `"thirty"`, and so forth.

8. Modify Programming Project 5 from Chapter 7 so that it includes the following function:

```
int compute_scrabble_value(const char *word);
```

The function returns the SCRABBLE value of the string pointed to by `word`.

9. Modify Programming Project 10 from Chapter 7 so that it includes the following function:

```
int compute_vowel_count(const char *sentence);
```

The function returns the number of vowels in the string pointed to by the `sentence` parameter.

10. Modify Programming Project 11 from Chapter 7 so that it includes the following function:

```
void reverse_name(char *name);
```

The function expects `name` to point to a string containing a first name followed by a last name. It modifies the string so that the last name comes first, followed by a comma, a space, the first initial, and a period. The original string may contain extra spaces before the first name, between the first and last names, and after the last name.

11. Modify Programming Project 13 from Chapter 7 so that it includes the following function:

```
double compute_average_word_length(const char *sentence);
```

The function returns the average length of the words in the string pointed to by `sentence`.

12. Modify Programming Project 14 from Chapter 8 so that it stores the words in a two-dimensional `char` array as it reads the sentence, with each row of the array storing a single word. Assume that the sentence contains no more than 30 words and no word is more than 20 characters long. Be sure to store a null character at the end of each word so that it can be treated as a string.

13. Modify Programming Project 15 from Chapter 8 so that it includes the following function:

    ```
    void encrypt(char *message, int shift);
    ```

    The function expects `message` to point to a string containing the message to be encrypted; `shift` represents the amount by which each letter in the message is to be shifted.

14. Modify Programming Project 16 from Chapter 8 so that it includes the following function:

    ```
    bool are_anagrams(const char *word1, const char *word2);
    ```

    The function returns `true` if the strings pointed to by `word1` and `word2` are anagrams.

15. Modify Programming Project 6 from Chapter 10 so that it includes the following function:

    ```
    int evaluate_RPN_expression(const char *expression);
    ```

    The function returns the value of the RPN expression pointed to by `expression`.

16. Modify Programming Project 1 from Chapter 12 so that it includes the following function:

    ```
    void reverse(char *message);
    ```

    The function reverses the string pointed to by `message`. *Hint:* Use two pointers, one initially pointing to the first character of the string and the other initially pointing to the last character. Have the function reverse these characters and then move the pointers toward each other, repeating the process until the pointers meet.

17. Modify Programming Project 2 from Chapter 12 so that it includes the following function:

    ```
    bool is_palindrome(const char *message);
    ```

    The function returns `true` if the string pointed to by `message` is a palindrome.

18. Write a program that accepts a date from the user in the form *mm/dd/yyyy* and then displays it in the form *month dd, yyyy*, where *month* is the name of the month:

    ```
    Enter a date (mm/dd/yyyy): 2/17/2011
    You entered the date February 17, 2011
    ```

    Store the month names in an array that contains pointers to strings.

14 The Preprocessor

There will always be things we wish to say in our programs that in all known languages can only be said poorly.

In previous chapters, I've used the `#define` and `#include` directives without going into detail about what they do. These directives—and others that we haven't yet covered—are handled by the ***preprocessor***, a piece of software that edits C programs just prior to compilation. Its reliance on a preprocessor makes C (along with C++) unique among major programming languages.

The preprocessor is a powerful tool, but it also can be a source of hard-to-find bugs. Moreover, the preprocessor can easily be misused to create programs that are almost impossible to understand. Although some C programmers depend heavily on the preprocessor, I recommend that it—like so many other things in life—be used in moderation.

This chapter begins by describing how the preprocessor works (Section 14.1) and giving some general rules that affect all preprocessing directives (Section 14.2). Sections 14.3 and 14.4 cover two of the preprocessor's major capabilities: macro definition and conditional compilation. (I'll defer detailed coverage of file inclusion, the other major capability, until Chapter 15.) Section 14.5 discusses the preprocessor's lesser-used directives: `#error`, `#line`, and `#pragma`.

14.1 How the Preprocessor Works

The behavior of the preprocessor is controlled by ***preprocessing directives:*** commands that begin with a # character. We've encountered two of these directives, `#define` and `#include`, in previous chapters.

The `#define` directive defines a ***macro***—a name that represents something else, such as a constant or frequently used expression. The preprocessor responds to a `#define` directive by storing the name of the macro together with its definition.

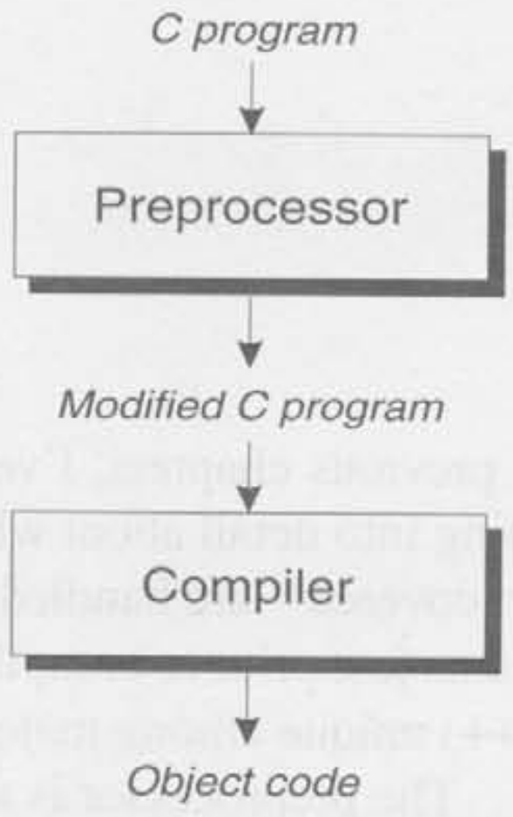
When the macro is used later in the program, the preprocessor "expands" the macro, replacing it by its defined value.

The `#include` directive tells the preprocessor to open a particular file and "include" its contents as part of the file being compiled. For example, the line

```
#include <stdio.h>
```

instructs the preprocessor to open the file named `stdio.h` and bring its contents into the program. (Among other things, `stdio.h` contains prototypes for C's standard input/output functions.)

The following diagram shows the preprocessor's role in the compilation process:



The input to the preprocessor is a C program, possibly containing directives. The preprocessor executes these directives, removing them in the process. The output of the preprocessor is another C program: an edited version of the original program, containing no directives. The preprocessor's output goes directly into the compiler, which checks the program for errors and translates it to object code (machine instructions).

To see what the preprocessor does, let's apply it to the `celsius.c` program of Section 2.6. Here's the original program:

```
/* Converts a Fahrenheit temperature to Celsius */

#include <stdio.h>

#define FREEZING_PT 32.0f
#define SCALE_FACTOR (5.0f / 9.0f)

int main(void)
{
  float fahrenheit, celsius;

  printf("Enter Fahrenheit temperature: ");
  scanf("%f", &fahrenheit);

  celsius = (fahrenheit - FREEZING_PT) * SCALE_FACTOR;
```

```
  printf("Celsius equivalent is: %.1f\n", celsius);

  return 0;
}
```

After preprocessing, the program will have the following appearance:

```
Blank line
Blank line
Lines brought in from stdio.h
Blank line
Blank line
Blank line
Blank line
int main(void)
{
  float fahrenheit, celsius;

  printf("Enter Fahrenheit temperature: ");
  scanf("%f", &fahrenheit);

  celsius = (fahrenheit - 32.0f) * (5.0f / 9.0f);

  printf("Celsius equivalent is: %.1f\n", celsius);

  return 0;
}
```

The preprocessor responded to the `#include` directive by bringing in the contents of `stdio.h`. The preprocessor also removed the `#define` directives and replaced `FREEZING_PT` and `SCALE_FACTOR` wherever they appeared later in the file. Notice that the preprocessor doesn't remove lines containing directives; instead, it simply makes them empty.

As this example shows, the preprocessor does a bit more than just execute directives. In particular, it replaces each comment with a single space character. Some preprocessors go further and remove unnecessary white-space characters, including spaces and tabs at the beginning of indented lines.

In the early days of C, the preprocessor was a separate program that fed its output into the compiler. Nowadays, the preprocessor is often part of the compiler, and some of its output may not necessarily be C code. (For example, including a standard header such as `<stdio.h>` may have the effect of making its functions available to the program without necessarily copying the contents of the header into the program's source code.) Still, it's useful to think of the preprocessor as separate from the compiler. In fact, most C compilers provide a way to view the output of the preprocessor. Some compilers generate preprocessor output when a certain option is specified (GCC will do so when the `-E` option is used). Others come with a separate program that behaves like the integrated preprocessor. Check your compiler's documentation for more information.

A word of caution: The preprocessor has only a limited knowledge of C. As a result, it's quite capable of creating illegal programs as it executes directives. Often the original program looks fine, making errors harder to find. In complicated

programs, examining the output of the preprocessor may prove useful for locating this kind of error.

14.2 Preprocessing Directives

Most preprocessing directives fall into one of three categories:

- ***Macro definition.*** The `#define` directive defines a macro; the `#undef` directive removes a macro definition.
- ***File inclusion.*** The `#include` directive causes the contents of a specified file to be included in a program.
- ***Conditional compilation.*** The `#if`, `#ifdef`, `#ifndef`, `#elif`, `#else`, and `#endif` directives allow blocks of text to be either included in or excluded from a program, depending on conditions that can be tested by the preprocessor.

The remaining directives—`#error`, `#line`, and `#pragma`—are more specialized and therefore used less often. We'll devote the rest of this chapter to an in-depth examination of preprocessing directives. The only directive we won't discuss in detail is `#include`, since it's covered in Section 15.2.

Before we go further, let's look at a few rules that apply to all directives:

- ***Directives always begin with the # symbol.*** The # symbol need not be at the beginning of a line, as long as only white space precedes it. After the # comes the name of the directive, followed by any other information the directive requires.
- ***Any number of spaces and horizontal tab characters may separate the tokens in a directive.*** For example, the following directive is legal:

```
#    define    N    100
```

- ***Directives always end at the first new-line character, unless explicitly continued.*** To continue a directive to the next line, we must end the current line with a \ character. For example, the following directive defines a macro that represents the capacity of a hard disk, measured in bytes:

```
#define DISK_CAPACITY (SIDES *                  \
                       TRACKS_PER_SIDE *        \
                       SECTORS_PER_TRACK *      \
                       BYTES_PER_SECTOR)
```

- ***Directives can appear anywhere in a program.*** Although we usually put `#define` and `#include` directives at the beginning of a file, other directives are more likely to show up later, even in the middle of function definitions.
- ***Comments may appear on the same line as a directive.*** In fact, it's good practice to put a comment at the end of a macro definition to explain the meaning of the macro:

```
#define FREEZING_PT 32.0f    /* freezing point of water */
```

14.3 Macro Definitions

The macros that we've been using since Chapter 2 are known as *simple* macros, because they have no parameters. The preprocessor also supports *parameterized* macros. We'll look first at simple macros, then at parameterized macros. After covering them separately, we'll examine properties shared by both.

Simple Macros

The definition of a ***simple macro*** (or ***object-like macro,*** as it's called in the C standard) has the form

#define directive (simple macro)

```
#define identifier replacement-list
```

replacement-list is any sequence of ***preprocessing tokens,*** which are similar to the tokens discussed in Section 2.8. Whenever we use the term "token" in this chapter, it means "preprocessing token."

A macro's replacement list may include identifiers, keywords, numeric constants, character constants, string literals, operators, and punctuation. When it encounters a macro definition, the preprocessor makes a note that *identifier* represents *replacement-list*; wherever *identifier* appears later in the file, the preprocessor substitutes *replacement-list*.

Don't put any extra symbols in a macro definition—they'll become part of the replacement list. Putting the = symbol in a macro definition is a common error:

```
#define N = 100    /*** WRONG ***/
...
int a[N];          /* becomes int a[= 100]; */
```

In this example, we've (incorrectly) defined `N` to be a pair of tokens (`=` and `100`). Ending a macro definition with a semicolon is another popular mistake:

```
#define N 100;     /*** WRONG ***/
...
int a[N];          /* becomes int a[100;]; */
```

Here `N` is defined to be the tokens `100` and `;`.

The compiler will detect most errors caused by extra symbols in a macro definition. Unfortunately, the compiler will flag each use of the macro as incorrect, rather than identifying the actual culprit—the macro's definition—which will have been removed by the preprocessor.

Simple macros are primarily used for defining what Kernighan and Ritchie call "manifest constants." Using macros, we can give names to numeric, character, and string values:

```
#define STR_LEN 80
#define TRUE    1
#define FALSE   0
#define PI      3.14159
#define CR      '\r'
#define EOS     '\0'
#define MEM_ERR "Error: not enough memory"
```

Using `#define` to create names for constants has several significant advantages:

- ***It makes programs easier to read.*** The name of the macro—if well-chosen—helps the reader understand the meaning of the constant. The alternative is a program full of "magic numbers" that can easily mystify the reader.
- ***It makes programs easier to modify.*** We can change the value of a constant throughout a program by modifying a single macro definition. "Hard-coded" constants are more difficult to change, especially since they sometimes appear in a slightly altered form. (For example, a program with an array of length 100 may have a loop that goes from 0 to 99. If we merely try to locate occurrences of `100` in the program, we'll miss the `99`.)
- ***It helps avoid inconsistencies and typographical errors.*** If a numerical constant like `3.14159` appears many times in a program, chances are it will occasionally be written `3.1416` or `3.14195` by accident.

Although simple macros are most often used to define names for constants, they do have other applications:

- ***Making minor changes to the syntax of C.*** We can—in effect—alter the syntax of C by defining macros that serve as alternate names for C symbols. For example, programmers who prefer Pascal's `begin` and `end` to C's { and } can define the following macros:

  ```
  #define BEGIN {
  #define END   }
  ```

  We could go so far as to invent our own language. For example, we might create a `LOOP` "statement" that establishes an infinite loop:

  ```
  #define LOOP for (;;)
  ```

  Changing the syntax of C usually isn't a good idea, though, since it can make programs harder for others to understand.
- ***Renaming types.*** In Section 5.2, we created a Boolean type by renaming `int`:

  ```
  #define BOOL int
  ```

  type definitions ➤7.5

  Although some programmers use macros for this purpose, type definitions are a superior way to define type names.
- ***Controlling conditional compilation.*** Macros play an important role in controlling conditional compilation, as we'll see in Section 14.4. For example, the presence of the following line in a program might indicate that it's to be com-

piled in "debugging mode," with extra statements included to produce debugging output:

```
#define DEBUG
```

Incidentally, it's legal for a macro's replacement list to be empty, as this example shows.

When macros are used as constants, C programmers customarily capitalize all letters in their names. However, there's no consensus as to how to capitalize macros used for other purposes. Since macros (especially parameterized macros) can be a source of bugs, some programmers like to draw attention to them by using all upper-case letters in their names. Others prefer lower-case names, following the style of Kernighan and Ritchie's *The C Programming Language*.

Parameterized Macros

The definition of a ***parameterized macro*** (also known as a ***function-like macro***) has the form

#define directive (parameterized macro)

`#define` *identifier* (x_1 , x_2 , ... , x_n) *replacement-list*

where x_1, x_2, ..., x_n are identifiers (the macro's ***parameters***). The parameters may appear as many times as desired in the replacement list.

There must be *no space* between the macro name and the left parenthesis. If space is left, the preprocessor will assume that we're defining a simple macro; it will treat (x_1, x_2, ..., x_n) as part of the replacement list.

When the preprocessor encounters the definition of a parameterized macro, it stores the definition away for later use. Wherever a macro ***invocation*** of the form *identifier* (y_1, y_2, ..., y_n) appears later in the program (where y_1, y_2, ..., y_n are sequences of tokens), the preprocessor replaces it with *replacement-list*, substituting y_1 for x_1, y_2 for x_2, and so forth.

For example, suppose that we've defined the following macros:

```
#define MAX(x,y)    ((x)>(y)?(x):(y))
#define IS_EVEN(n)  ((n)%2==0)
```

(The number of parentheses in these macros may seem excessive, but there's a reason, as we'll see later in this section.) Now suppose that we invoke the two macros in the following way:

```
i = MAX(j+k, m-n);
if (IS_EVEN(i)) i++;
```

The preprocessor will replace these lines by

```
i = ((j+k)>(m-n)?(j+k):(m-n));
if (((i)%2==0)) i++;
```

As this example shows, parameterized macros often serve as simple functions. `MAX` behaves like a function that computes the larger of two values. `IS_EVEN` behaves like a function that returns 1 if its argument is an even number and 0 otherwise.

Here's a more complicated macro that behaves like a function:

```
#define TOUPPER(c) ('a'<=(c)&&(c)<='z'?(c)-'a'+'A':(c))
```

This macro tests whether the character `c` is between `'a'` and `'z'`. If so, it produces the upper-case version of `c` by subtracting `'a'` and adding `'A'`. If not, it leaves `c` unchanged. (The `<ctype.h>` header provides a similar function named `toupper` that's more portable.)

<ctype.h> header ➤23.5

A parameterized macro may have an empty parameter list. Here's an example:

```
#define getchar() getc(stdin)
```

The empty parameter list isn't really needed, but it makes `getchar` resemble a function. (Yes, this is the same `getchar` that belongs to `<stdio.h>`. We'll see in Section 22.4 that `getchar` is usually implemented as a macro as well as a function.)

Using a parameterized macro instead of a true function has a couple of advantages:

- ***The program may be slightly faster.*** A function call usually requires some overhead during program execution—context information must be saved, arguments copied, and so forth. A macro invocation, on the other hand, requires no run-time overhead. (Note, however, that C99's inline functions provide a way to avoid this overhead without the use of macros.)

C99

inline functions ➤18.6

- ***Macros are "generic."*** Macro parameters, unlike function parameters, have no particular type. As a result, a macro can accept arguments of any type, provided that the resulting program—after preprocessing—is valid. For example, we could use the `MAX` macro to find the larger of two values of type `int`, `long`, `float`, `double`, and so forth.

But parameterized macros also have disadvantages:

- ***The compiled code will often be larger.*** Each macro invocation causes the insertion of the macro's replacement list, thereby increasing the size of the source program (and hence the compiled code). The more often the macro is used, the more pronounced this effect is. The problem is compounded when macro invocations are nested. Consider what happens when we use `MAX` to find the largest of three numbers:

  ```
  n = MAX(i, MAX(j, k));
  ```

  Here's the same statement after preprocessing:

  ```
  n = ((i)>(((j)>(k)?(j):(k)))?(i):(((j)>(k)?(j):(k))));
  ```

- ***Arguments aren't type-checked.*** When a function is called, the compiler checks each argument to see if it has the appropriate type. If not, either the argument is converted to the proper type or the compiler produces an error message. Macro arguments aren't checked by the preprocessor, nor are they converted.
- ***It's not possible to have a pointer to a macro.*** As we'll see in Section 17.7, C allows pointers to functions, a concept that's quite useful in certain programming situations. Macros are removed during preprocessing, so there's no corresponding notion of "pointer to a macro"; as a result, macros can't be used in these situations.
- ***A macro may evaluate its arguments more than once.*** A function evaluates its arguments only once; a macro may evaluate its arguments two or more times. Evaluating an argument more than once can cause unexpected behavior if the argument has side effects. Consider what happens if one of `MAX`'s arguments has a side effect:

```
n = MAX(i++, j);
```

Here's the same line after preprocessing:

```
n = ((i++)>(j)?(i++):(j));
```

If `i` is larger than `j`, then `i` will be (incorrectly) incremented twice and `n` will be assigned an unexpected value.

Errors caused by evaluating a macro argument more than once can be difficult to find, because a macro invocation looks the same as a function call. To make matters worse, a macro may work properly most of the time, failing only for certain arguments that have side effects. For self-protection, it's a good idea to avoid side effects in arguments.

Parameterized macros are good for more than just simulating functions. In particular, they're often used as patterns for segments of code that we find ourselves repeating. Suppose that we grow tired of writing

```
printf("%d\n", i);
```

every time we need to print an integer `i`. We might define the following macro, which makes it easier to display integers:

```
#define PRINT_INT(n) printf("%d\n", n)
```

Once `PRINT_INT` has been defined, the preprocessor will turn the line

```
PRINT_INT(i/j);
```

into

```
printf("%d\n", i/j);
```

The # Operator

Macro definitions may contain two special operators, # and ##. Neither operator is recognized by the compiler; instead, they're executed during preprocessing.

Q&A The # operator converts a macro argument into a string literal; it can appear only in the replacement list of a parameterized macro. (The operation performed by # is known as "stringization," a term that I'm sure you won't find in the dictionary.)

There are a number of uses for #; let's consider just one. Suppose that we decide to use the `PRINT_INT` macro during debugging as a convenient way to print the values of integer variables and expressions. The # operator makes it possible for `PRINT_INT` to label each value that it prints. Here's our new version of `PRINT_INT`:

```
#define PRINT_INT(n) printf(#n " = %d\n", n)
```

The # operator in front of `n` instructs the preprocessor to create a string literal from `PRINT_INT`'s argument. Thus, the invocation

```
PRINT_INT(i/j);
```

will become

```
printf("i/j" " = %d\n", i/j);
```

We saw in Section 13.1 that the compiler automatically joins adjacent string literals, so this statement is equivalent to

```
printf("i/j = %d\n", i/j);
```

When the program is executed, `printf` will display both the expression `i/j` and its value. If `i` is 11 and `j` is 2, for example, the output will be

```
i/j = 5
```

The ## Operator

The ## operator can "paste" two tokens (identifiers, for example) together to form a single token. (Not surprisingly, the ## operation is known as "token-pasting.") If one of the operands is a macro parameter, pasting occurs after the parameter has been replaced by the corresponding argument. Consider the following macro:

```
#define MK_ID(n) i##n
```

When `MK_ID` is invoked (as `MK_ID(1)`, say), the preprocessor first replaces the parameter `n` by the argument (`1` in this case). Next, the preprocessor joins `i` and `1` to make a single token (`i1`). The following declaration uses `MK_ID` to create three identifiers:

```
int MK_ID(1), MK_ID(2), MK_ID(3);
```

After preprocessing, this declaration becomes

```
int i1, i2, i3;
```

The ## operator isn't one of the most frequently used features of the preprocessor; in fact, it's hard to think of many situations that require it. To find a realistic application of ##, let's reconsider the MAX macro described earlier in this section. As we observed then, MAX doesn't behave properly if its arguments have side effects. The alternative to using the MAX macro is to write a max function. Unfortunately, one max function usually isn't enough; we may need a max function whose arguments are int values, one whose arguments are float values, and so on. All these versions of max would be identical except for the types of the arguments and the return type, so it seems a shame to define each one from scratch.

The solution is to write a macro that expands into the definition of a max function. The macro will have a single parameter, type, which represents the type of the arguments and the return value. There's just one snag: if we use the macro to create more than one max function, the program won't compile. (C doesn't allow two functions to have the same name if both are defined in the same file.) To solve this problem, we'll use the ## operator to create a different name for each version of max. Here's what the macro will look like:

```
#define GENERIC_MAX(type)       \
type type##_max(type x, type y) \
{                               \
  return x > y ? x : y;         \
}
```

Notice how type is joined with _max to form the name of the function.

Suppose that we happen to need a max function that works with float values. Here's how we'd use GENERIC_MAX to define the function:

```
GENERIC_MAX(float)
```

The preprocessor expands this line into the following code:

```
float float_max(float x, float y) { return x > y ? x : y; }
```

General Properties of Macros

Now that we've discussed both simple and parameterized macros, let's look at some rules that apply to both:

- ***A macro's replacement list may contain invocations of other macros.*** For example, we could define the macro TWO_PI in terms of the macro PI:

  ```
  #define PI      3.14159
  #define TWO_PI (2*PI)
  ```

  When it encounters TWO_PI later in the program, the preprocessor replaces it by (2*PI). The preprocessor then ***rescans*** the replacement list to see if it

contains invocations of other macros (`PI`, in this case). The preprocessor will rescan the replacement list as many times as necessary to eliminate all macro names.

Q&A

- ***The preprocessor replaces only entire tokens, not portions of tokens.*** As a result, the preprocessor ignores macro names that are embedded in identifiers, character constants, and string literals. For example, suppose that a program contains the following lines:

```
#define SIZE 256

int BUFFER_SIZE;

if (BUFFER_SIZE > SIZE)
  puts("Error: SIZE exceeded");
```

After preprocessing, these lines will have the following appearance:

```
int BUFFER_SIZE;

if (BUFFER_SIZE > 256)
  puts("Error: SIZE exceeded");
```

The identifier `BUFFER_SIZE` and the string `"Error: SIZE exceeded"` weren't affected by preprocessing, even though both contain the word `SIZE`.

- ***A macro definition normally remains in effect until the end of the file in which it appears.*** Since macros are handled by the preprocessor, they don't obey normal scope rules. A macro defined inside the body of a function isn't local to that function; it remains defined until the end of the file.
- ***A macro may not be defined twice unless the new definition is identical to the old one.*** Differences in spacing are allowed, but the tokens in the macro's replacement list (and the parameters, if any) must be the same.
- ***Macros may be "undefined" by the `#undef` directive.*** The `#undef` directive has the form

`#undef` directive

```
#undef identifier
```

where *identifier* is a macro name. For example, the directive

```
#undef N
```

removes the current definition of the macro `N`. (If `N` hasn't been defined as a macro, the `#undef` directive has no effect.) One use of `#undef` is to remove the existing definition of a macro so that it can be given a new definition.

Parentheses in Macro Definitions

The replacement lists in our macro definitions have been full of parentheses. Is it really necessary to have so many? The answer is an emphatic yes; if we use fewer

parentheses, the macros will sometimes give unexpected—and undesirable—results.

There are two rules to follow when deciding where to put parentheses in a macro definition. First, if the macro's replacement list contains an operator, always enclose the replacement list in parentheses:

```
#define TWO_PI (2*3.14159)
```

Second, if the macro has parameters, put parentheses around each parameter every time it appears in the replacement list:

```
#define SCALE(x) ((x)*10)
```

Without the parentheses, we can't guarantee that the compiler will treat replacement lists and arguments as whole expressions. The compiler may apply the rules of operator precedence and associativity in ways that we didn't anticipate.

To illustrate the importance of putting parentheses around a macro's replacement list, consider the following macro definition, in which the parentheses are missing:

```
#define TWO_PI 2*3.14159
  /* needs parentheses around replacement list */
```

During preprocessing, the statement

```
conversion_factor = 360/TWO_PI;
```

becomes

```
conversion_factor = 360/2*3.14159;
```

The division will be performed before the multiplication, yielding a result different from the one intended.

Putting parentheses around the replacement list isn't enough if the macro has parameters—each occurrence of a parameter needs parentheses as well. For example, suppose that `SCALE` is defined as follows:

```
#define SCALE(x) (x*10)    /* needs parentheses around x */
```

During preprocessing, the statement

```
j = SCALE(i+1);
```

becomes

```
j = (i+1*10);
```

Since multiplication takes precedence over addition, this statement is equivalent to

```
j = i+10;
```

Of course, what we wanted was

```
j = (i+1)*10;
```

A shortage of parentheses in a macro definition can cause some of C's most frustrating errors. The program will usually compile and the macro will appear to work, failing only at the least convenient times.

Creating Longer Macros

The comma operator can be useful for creating more sophisticated macros by allowing us to make the replacement list a series of expressions. For example, the following macro will read a string and then print it:

```
#define ECHO(s) (gets(s), puts(s))
```

Calls of `gets` and `puts` are expressions, so it's perfectly legal to combine them using the comma operator. We can invoke `ECHO` as though it were a function:

```
ECHO(str);    /* becomes (gets(str), puts(str)); */
```

Instead of using the comma operator in the definition of `ECHO`, we could have enclosed the calls of `gets` and `puts` in braces to form a compound statement:

```
#define ECHO(s) { gets(s); puts(s); }
```

Unfortunately, this method doesn't work as well. Suppose that we use `ECHO` in an `if` statement:

```
if (echo_flag)
  ECHO(str);
else
  gets(str);
```

Replacing `ECHO` gives the following result:

```
if (echo_flag)
  { gets(str); puts(str); };
else
  gets(str);
```

The compiler treats the first two lines as a complete `if` statement:

```
if (echo_flag)
  { gets(str); puts(str); }
```

It treats the semicolon that follows as a null statement and produces an error message for the `else` clause, since it doesn't belong to any `if`. We could solve the problem by remembering not to put a semicolon after each invocation of `ECHO`, but then the program would look odd.

The comma operator solves this problem for `ECHO`, but not for all macros. Suppose that a macro needs to contain a series of *statements*, not just a series of *expressions*. The comma operator is of no help; it can glue together expressions,

but not statements. The solution is to wrap the statements in a do loop whose condition is false (and which therefore will be executed just once):

```
do { … } while (0)
```

Notice that the do statement isn't complete—it needs a semicolon at the end. To see this trick (ahem, technique) in action, let's incorporate it into our ECHO macro:

```
#define ECHO(s)     \
        do {        \
          gets(s);  \
          puts(s);  \
        } while (0)
```

When ECHO is used, it must be followed by a semicolon, which completes the do statement:

```
ECHO(str);
  /* becomes do { gets(str); puts(str); } while (0); */
```

Predefined Macros

C has several predefined macros. Each macro represents an integer constant or string literal. As Table 14.1 shows, these macros provide information about the current compilation or about the compiler itself.

Table 14.1 Predefined Macros

Name	*Description*
__LINE__	Line number of file being compiled
__FILE__	Name of file being compiled
__DATE__	Date of compilation (in the form "Mmm dd yyyy")
__TIME__	Time of compilation (in the form "hh:mm:ss")
__STDC__	1 if the compiler conforms to the C standard (C89 or C99)

The __DATE__ and __TIME__ macros identify when a program was compiled. For example, suppose that a program begins with the following statements:

```
printf("Wacky Windows (c) 2010 Wacky Software, Inc.\n");
printf("Compiled on %s at %s\n", __DATE__, __TIME__);
```

Each time it begins to execute, the program will print two lines of the form

```
Wacky Windows (c) 2010 Wacky Software, Inc.
Compiled on Dec 23 2010 at 22:18:48
```

This information can be helpful for distinguishing among different versions of the same program.

We can use the __LINE__ and __FILE__ macros to help locate errors. Consider the problem of detecting the location of a division by zero. When a C program terminates prematurely because it divided by zero, there's usually no indication of which division caused the problem. The following macro can help us pinpoint the source of the error:

```
#define CHECK_ZERO(divisor) \
  if (divisor == 0) \
    printf("*** Attempt to divide by zero on line %d " \
           "of file %s ***\n", __LINE__, __FILE__)
```

The CHECK_ZERO macro would be invoked prior to a division:

```
CHECK_ZERO(j);
k = i / j;
```

If `j` happens to be zero, a message of the following form will be printed:

```
*** Attempt to divide by zero on line 9 of file foo.c ***
```

Error-detecting macros like this one are quite useful. In fact, the C library has a general-purpose error-detecting macro named `assert`.

assert macro ➤24.1

The `__STDC__` macro exists and has the value 1 if the compiler conforms to the C standard (either C89 or C99). By having the preprocessor test this macro, a program can adapt to a compiler that predates the C89 standard (see Section 14.4 for an example).

C99 Additional Predefined Macros in C99

C99 provides a few additional predefined macros (Table 14.2).

Table 14.2 Additional Predefined Macros in C99

Name	*Description*
`__STDC_HOSTED__`	1 if this is a hosted implementation; 0 if it is freestanding
`__STDC_VERSION__`	Version of C standard supported
`__STDC_IEC_559__`†	1 if IEC 60559 floating-point arithmetic is supported
`__STDC_IEC_559_COMPLEX__`†	1 if IEC 60559 complex arithmetic is supported
`__STDC_ISO_10646__`†	*yyyymm*`L` if `wchar_t` values match the ISO 10646 standard of the specified year and month

†Conditionally defined

To understand the meaning of `__STDC_HOSTED__`, we need some new vocabulary. An ***implementation*** of C consists of the compiler plus other software necessary to execute C programs. C99 divides implementations into two categories: hosted and freestanding. A ***hosted implementation*** must accept any program that conforms to the C99 standard, whereas a ***freestanding implementation*** doesn't have to compile programs that use complex types or standard headers beyond a few of the most basic. (In particular, a freestanding implementation doesn't have to support the `<stdio.h>` header.) The `__STDC_HOSTED__` macro represents the constant 1 if the compiler is a hosted implementation; otherwise, the macro has the value 0.

complex types ➤27.3

Q&A

The `__STDC_VERSION__` macro provides a way to check which version of the C standard is recognized by the compiler. This macro first appeared in Amendment 1 to the C89 standard, where its value was specified to be the long

integer constant `199409L` (representing the year and month of the amendment). If a compiler conforms to the C99 standard, the value is `199901L`. For each subsequent version of the standard (and each amendment to the standard), this macro will have a different value.

A C99 compiler may (or may not) define three additional macros. Each macro is defined only if the compiler meets a certain requirement:

- `__STDC_IEC_559__` is defined (and has the value 1) if the compiler performs floating-point arithmetic according to the IEC 60559 standard (another name for the IEEE 754 standard).

IEEE floating-point standard ➤7.2

- `__STDC_IEC_559_COMPLEX__` is defined (and has the value 1) if the compiler performs complex arithmetic according to the IEC 60559 standard.
- `__STDC_ISO_10646__` is defined as an integer constant of the form *yyyymm*`L` (for example, `199712L`) if values of type `wchar_t` are represented by the codes in the ISO/IEC 10646 standard (with revisions as of the specified year and month).

wchar_t type ➤25.2

ISO/IEC 10646 standard ➤25.2

C99 Empty Macro Arguments

C99 allows any or all of the arguments in a macro call to be empty. Such a call will contain the same number of commas as a normal call, however. (That way, it's easy to see which arguments have been omitted.)

In most cases, the effect of an empty argument is clear. Wherever the corresponding parameter name appears in the replacement list, it's replaced by nothing—it simply disappears from the replacement list. Here's an example:

```
#define ADD(x,y) (x+y)
```

After preprocessing, the statement

```
i = ADD(j,k);
```

becomes

```
i = (j+k);
```

whereas the statement

```
i = ADD(,k);
```

becomes

```
i = (+k);
```

When an empty argument is an operand of the # or ## operators, special rules apply. If an empty argument is "stringized" by the # operator, the result is `""` (the empty string):

```
#define MK_STR(x) #x
...
char empty_string[] = MK_STR();
```

After preprocessing, the declaration will have the following appearance:

```
char empty_string[] = "";
```

If one of the arguments of the ## operator is empty, it's replaced by an invisible "placemarker" token. Concatenating an ordinary token with a placemarker token yields the original token (the placemarker disappears). If two placemarker tokens are concatenated, the result is a single placemarker. Once macro expansion has been completed, placemarker tokens disappear from the program. Consider the following example:

```
#define JOIN(x,y,z) x##y##z
...
int JOIN(a,b,c), JOIN(a,b,), JOIN(a,,c), JOIN(,,c);
```

After preprocessing, the declaration will have the following appearance:

```
int abc, ab, ac, c;
```

The missing arguments were replaced by placemarker tokens, which then disappeared when concatenated with any nonempty arguments. All three arguments to the JOIN macro could even be missing, which would yield an empty result.

C99 Macros with a Variable Number of Arguments

variable-length argument lists
➤26.1

In C89, a macro must have a fixed number of arguments, if it has any at all. C99 loosens things up a bit, allowing macros that take an unlimited number of arguments. This feature has long been available for functions, so it's not surprising that macros were finally put on an equal footing.

The primary reason for having a macro with a variable number of arguments is that it can pass these arguments to a function that accepts a variable number of arguments, such as `printf` or `scanf`. Here's an example:

```
#define TEST(condition, ...) ((condition)? \
  printf("Passed test: %s\n", #condition): \
  printf(__VA_ARGS__))
```

The . . . token, known as ***ellipsis,*** goes at the end of a macro's parameter list, preceded by ordinary parameters, if there are any. `__VA_ARGS__` is a special identifier that can appear only in the replacement list of a macro with a variable number of arguments; it represents all the arguments that correspond to the ellipsis. (There must be at least one argument that corresponds to the ellipsis, although that argument may be empty.) The `TEST` macro requires at least two arguments. The first argument matches the `condition` parameter; the remaining arguments match the ellipsis.

Here's an example that shows how the `TEST` macro might be used:

```
TEST(voltage <= max_voltage,
     "Voltage %d exceeds %d\n", voltage, max_voltage);
```

The preprocessor will produce the following output (reformatted for readability):

```
((voltage <= max_voltage)?
  printf("Passed test: %s\n", "voltage <= max_voltage"):
  printf("Voltage %d exceeds %d\n", voltage, max_voltage));
```

When the program is executed, the program will display the message

```
Passed test: voltage <= max_voltage
```

if `voltage` is no more than `max_voltage`. Otherwise, it will display the values of `voltage` and `max_voltage`:

```
Voltage 125 exceeds 120
```

C99 The `__func__` Identifier

Another new feature of C99 is the `__func__` identifier. `__func__` has nothing to do with the preprocessor, so it actually doesn't belong in this chapter. However, like many preprocessor features, it's useful for debugging, so I've chosen to discuss it here.

Every function has access to the `__func__` identifier, which behaves like a string variable that stores the name of the currently executing function. The effect is the same as if each function contains the following declaration at the beginning of its body:

```
static const char __func__[] = "function-name";
```

where *function-name* is the name of the function. The existence of this identifier makes it possible to write debugging macros such as the following:

```
#define FUNCTION_CALLED() printf("%s called\n", __func__);
#define FUNCTION_RETURNS() printf("%s returns\n", __func__);
```

Calls of these macros can then be placed inside functions to trace their calls:

```
void f(void)
{
  FUNCTION_CALLED();   /* displays "f called" */
  ...
  FUNCTION_RETURNS();  /* displays "f returns" */
}
```

Another use of `__func__`: it can be passed to a function to let it know the name of the function that called it.

14.4 Conditional Compilation

The C preprocessor recognizes a number of directives that support ***conditional compilation***—the inclusion or exclusion of a section of program text depending on the outcome of a test performed by the preprocessor.

The #if and #endif Directives

Suppose we're in the process of debugging a program. We'd like the program to print the values of certain variables, so we put calls of `printf` in critical parts of the program. Once we've located the bugs, it's often a good idea to let the `printf` calls remain, just in case we need them later. Conditional compilation allows us to leave the calls in place, but have the compiler ignore them.

Here's how we'll proceed. We'll first define a macro and give it a nonzero value:

```
#define DEBUG 1
```

The name of the macro doesn't matter. Next, we'll surround each group of `printf` calls by an `#if`-`#endif` pair:

```
#if DEBUG
printf("Value of i: %d\n", i);
printf("Value of j: %d\n", j);
#endif
```

During preprocessing, the `#if` directive will test the value of `DEBUG`. Since its value isn't zero, the preprocessor will leave the two calls of `printf` in the program (the `#if` and `#endif` lines will disappear, though). If we change the value of `DEBUG` to zero and recompile the program, the preprocessor will remove all four lines from the program. The compiler won't see the calls of `printf`, so they won't occupy any space in the object code and won't cost any time when the program is run. We can leave the `#if`-`#endif` blocks in the final program, allowing diagnostic information to be produced later (by recompiling with `DEBUG` set to 1) if any problems turn up.

In general, the `#if` directive has the form

`#if` directive

```
#if constant-expression
```

The `#endif` directive is even simpler:

`#endif` directive

```
#endif
```

Q&A

When the preprocessor encounters the `#if` directive, it evaluates the constant expression. If the value of the expression is zero, the lines between `#if` and `#endif` will be removed from the program during preprocessing. Otherwise, the lines between `#if` and `#endif` will remain in the program to be processed by the compiler—the `#if` and `#endif` will have had no effect on the program.

It's worth noting that the `#if` directive treats undefined identifiers as macros that have the value 0. Thus, if we neglect to define `DEBUG`, the test

```
#if DEBUG
```

will fail (but not generate an error message), while the test

```
#if !DEBUG
```

will succeed.

The `defined` Operator

We encountered the # and ## operators in Section 14.3. There's just one other operator, `defined`, that's specific to the preprocessor. When applied to an identifier, `defined` produces the value 1 if the identifier is a currently defined macro; it produces 0 otherwise. The `defined` operator is normally used in conjunction with the `#if` directive; it allows us to write

```
#if defined(DEBUG)
...
#endif
```

The lines between the `#if` and `#endif` directives will be included in the program only if `DEBUG` is defined as a macro. The parentheses around `DEBUG` aren't required; we could simply write

```
#if defined DEBUG
```

Since `defined` tests only whether `DEBUG` is defined or not, it's not necessary to give `DEBUG` a value:

```
#define DEBUG
```

The `#ifdef` and `#ifndef` Directives

The `#ifdef` directive tests whether an identifier is currently defined as a macro:

`#ifdef` directive

```
#ifdef identifier
```

Using `#ifdef` is similar to using `#if`:

```
#ifdef identifier
Lines to be included if identifier is defined as a macro
#endif
```

Q&A Strictly speaking, there's no need for `#ifdef`, since we can combine the `#if` directive with the `defined` operator to get the same effect. In other words, the directive

```
#ifdef identifier
```

is equivalent to

```
#if defined(identifier)
```

The `#ifndef` directive is similar to `#ifdef`, but tests whether an identifier is *not* defined as a macro:

`#ifndef` directive

```
#ifndef identifier
```

Writing

```
#ifndef identifier
```

is the same as writing

```
#if !defined(identifier)
```

The `#elif` and `#else` Directives

`#if`, `#ifdef`, and `#ifndef` blocks can be nested just like ordinary `if` statements. When nesting occurs, it's a good idea to use an increasing amount of indentation as the level of nesting grows. Some programmers put a comment on each closing `#endif` to indicate what condition the matching `#if` tests:

```
#if DEBUG
...
#endif /* DEBUG */
```

This technique makes it easier for the reader to find the beginning of the `#if` block.

For additional convenience, the preprocessor supports the `#elif` and `#else` directives:

`#elif` directive

```
#elif constant-expression
```

`#else` directive

```
#else
```

`#elif` and `#else` can be used in conjunction with `#if`, `#ifdef`, or `#ifndef` to test a series of conditions:

```
#if expr1
Lines to be included if expr1 is nonzero
#elif expr2
Lines to be included if expr1 is zero but expr2 is nonzero
#else
Lines to be included otherwise
#endif
```

Although the `#if` directive is shown above, an `#ifdef` or `#ifndef` directive can be used instead. Any number of `#elif` directives—but at most one `#else`—may appear between `#if` and `#endif`.

Uses of Conditional Compilation

Conditional compilation is certainly handy for debugging, but its uses don't stop there. Here are a few other common applications:

- ***Writing programs that are portable to several machines or operating systems.*** The following example includes one of three groups of lines depending on whether `WIN32`, `MAC_OS`, or `LINUX` is defined as a macro:

  ```
  #if defined(WIN32)
  ...
  #elif defined(MAC_OS)
  ...
  #elif defined(LINUX)
  ...
  #endif
  ```

  A program might contain many of these `#if` blocks. At the beginning of the program, one (and only one) of the macros will be defined, thereby selecting a particular operating system. For example, defining the `LINUX` macro might indicate that the program is to run under the Linux operating system.

- ***Writing programs that can be compiled with different compilers.*** Different compilers often recognize somewhat different versions of C. Some accept a standard version of C, some don't. Some provide machine-specific language extensions; some don't, or provide a different set of extensions. Conditional compilation can allow a program to adjust to different compilers. Consider the problem of writing a program that might have to be compiled using an older, nonstandard compiler. The `__STDC__` macro allows the preprocessor to detect whether a compiler conforms to the standard (either C89 or C99); if it doesn't, we may need to change certain aspects of the program. In particular, we may have to use old-style function declarations (discussed in the Q&A at the end of Chapter 9) instead of function prototypes. At each point where functions are declared, we can put the following lines:

  ```
  #if __STDC__
  Function prototypes
  #else
  Old-style function declarations
  #endif
  ```

- ***Providing a default definition for a macro.*** Conditional compilation allows us to check whether a macro is currently defined and, if not, give it a default definition. For example, the following lines will define the macro `BUFFER_SIZE` if it wasn't previously defined:

  ```
  #ifndef BUFFER_SIZE
  #define BUFFER_SIZE 256
  #endif
  ```

- ***Temporarily disabling code that contains comments.*** We can't use a /*...*/ comment to "comment out" code that already contains /*...*/ comments. Instead, we can use an `#if` directive:

```
#if 0
Lines containing comments
#endif
```

Q&A Disabling code in this way is often called "conditioning out."

Section 15.2 discusses another common use of conditional compilation: protecting header files against multiple inclusion.

14.5 Miscellaneous Directives

To end the chapter, we'll take a brief look at the `#error`, `#line`, and `#pragma` directives. These directives are more specialized than the ones we've already examined, and they're used much less frequently.

The `#error` Directive

`#error` directive

```
#error message
```

where *message* is any sequence of tokens. If the preprocessor encounters an `#error` directive, it prints an error message which must include *message*. The exact form of the error message can vary from one compiler to another; it might be something like

```
Error directive: message
```

or perhaps just

```
#error message
```

Encountering an `#error` directive indicates a serious flaw in the program; some compilers immediately terminate compilation without attempting to find other errors.

`#error` directives are frequently used in conjunction with conditional compilation to check for situations that shouldn't arise during a normal compilation. For example, suppose that we want to ensure that a program can't be compiled on a machine whose `int` type isn't capable of storing numbers up to 100,000. The largest possible `int` value is represented by the `INT_MAX` macro, so all we need do is invoke an `#error` directive if `INT_MAX` isn't at least 100,000:

INT_MAX macro ➤23.2

```
#if INT_MAX < 100000
#error int type is too small
#endif
```

Attempting to compile the program on a machine whose integers are stored in 16 bits will produce a message such as

```
Error directive: int type is too small
```

The `#error` directive is often found in the `#else` part of an `#if`-`#elif`-`#else` series:

```
#if defined(WIN32)
...
#elif defined(MAC_OS)
...
#elif defined(LINUX)
...
#else
#error No operating system specified
#endif
```

The `#line` Directive

The `#line` directive is used to alter the way program lines are numbered. (Lines are usually numbered 1, 2, 3, as you'd expect.) We can also use this directive to make the compiler think that it's reading the program from a file with a different name.

The `#line` directive has two forms. In one form, we specify a line number:

`#line` directive (form 1)

```
#line n
```

n must be a sequence of digits representing an integer between 1 and 32767 (2147483647 in C99). This directive causes subsequent lines in the program to be numbered *n*, *n* + 1, *n* + 2, and so forth.

C99

In the second form of the `#line` directive, both a line number and a file name are specified:

`#line` directive (form 2)

```
#line n "file"
```

The lines that follow this directive are assumed to come from *file*, with line numbers starting at *n*. The values of *n* and/or the *file* string can be specified using macros.

One effect of the `#line` directive is to change the value of the `__LINE__` macro (and possibly the `__FILE__` macro). More importantly, most compilers will use the information from the `#line` directive when generating error messages.

For example, suppose that the following directive appears at the beginning of the file `foo.c`:

```
#line 10 "bar.c"
```

Let's say that the compiler detects an error on line 5 of `foo.c`. The error message will refer to line 13 of file `bar.c`, not line 5 of file `foo.c`. (Why line 13? The directive occupies line 1 of `foo.c`, so the renumbering of `foo.c` begins at line 2, which is treated as line 10 of `bar.c`.)

At first glance, the `#line` directive is mystifying. Why would we want error messages to refer to a different line and possibly a different file? Wouldn't this make programs harder to debug?

In fact, the `#line` directive isn't used very often by programmers. Instead, it's used primarily by programs that generate C code as output. The most famous example of such a program is `yacc` (Yet Another Compiler-Compiler), a UNIX utility that automatically generates part of a compiler. (The GNU version of `yacc` is named `bison`.) Before using `yacc`, the programmer prepares a file that contains information for `yacc` as well as fragments of C code. From this file, `yacc` generates a C program, `y.tab.c`, that incorporates the code supplied by the programmer. The programmer then compiles `y.tab.c` in the usual way. By inserting `#line` directives in `y.tab.c`, `yacc` tricks the compiler into believing that the code comes from the original file—the one written by the programmer. As a result, any error messages produced during the compilation of `y.tab.c` will refer to lines in the original file, not lines in `y.tab.c`. This makes debugging easier, because error messages refer to the file written by the programmer, not the (more complicated) file generated by `yacc`.

The #pragma Directive

The `#pragma` directive provides a way to request special behavior from the compiler. This directive is most useful for programs that are unusually large or that need to take advantage of the capabilities of a particular compiler.

The `#pragma` directive has the form

#pragma directive

```
#pragma tokens
```

where *tokens* are arbitrary tokens. `#pragma` directives can be very simple (a single token) or they can be much more elaborate:

```
#pragma data(heap_size => 1000, stack_size => 2000)
```

Not surprisingly, the set of commands that can appear in `#pragma` directives is different for each compiler; you'll have to consult the documentation for your compiler to see which commands it allows and what those commands do. Incidentally, the preprocessor must ignore any `#pragma` directive that contains an unrecognized command; it's not permitted to give an error message.

In C89, there are no standard pragmas—they're all implementation-defined.

C99 C99 has three standard pragmas, all of which use `STDC` as the first token following `#pragma`. These pragmas are `FP_CONTRACT` (covered in Section 23.4), `CX_LIMITED_RANGE` (Section 27.4), and `FENV_ACCESS` (Section 27.6).

C99 The `_Pragma` Operator

C99 introduces the `_Pragma` operator, which is used in conjunction with the `#pragma` directive. A `_Pragma` expression has the form

`_Pragma` expression

`_Pragma` (*string-literal*)

When it encounters such an expression, the preprocessor "destringizes" the string literal (yes, that's the term used in the C99 standard!) by removing the double quotes around the string and replacing the escape sequences `\"` and `\\` by the characters `"` and `\`, respectively. The result is a series of tokens, which are then treated as though they appear in a `#pragma` directive. For example, writing

```
_Pragma("data(heap_size => 1000, stack_size => 2000)")
```

is the same as writing

```
#pragma data(heap_size => 1000, stack_size => 2000)
```

The `_Pragma` operator lets us work around a limitation of the preprocessor: the fact that a preprocessing directive can't generate another directive. `_Pragma`, however, is an operator, not a directive, and can therefore appear in a macro definition. This makes it possible for a macro expansion to leave behind a `#pragma` directive.

Let's look at an example from the GCC manual. The following macro uses the `_Pragma` operator:

```
#define DO_PRAGMA(x) _Pragma(#x)
```

The macro would be invoked as follows:

```
DO_PRAGMA(GCC dependency "parse.y")
```

After expansion, the result will be

```
#pragma GCC dependency "parse.y"
```

which is one of the pragmas supported by GCC. (It issues a warning if the date of the specified file—`parse.y` in this example—is more recent than the date of the current file—the one being compiled.) Note that the argument to the call of `DO_PRAGMA` is a series of tokens. The # operator in the definition of `DO_PRAGMA` causes the tokens to be stringized into `"GCC dependency \"parse.y\""`; this string is then passed to the `_Pragma` operator, which destringizes it, producing a `#pragma` directive containing the original tokens.

Q & A

Q: I've seen programs that contain a # on a line by itself. Is this legal?

A: Yes. This is the ***null directive;*** it has no effect. Some programmers use null directives for spacing within conditional compilation blocks:

```
#if INT_MAX < 100000
#
#error int type is too small
#
#endif
```

Blank lines would also work, of course, but the # helps the reader see the extent of the block.

Q: I'm not sure which constants in a program need to be defined as macros. Are there any guidelines to follow? [p. 319]

A: One rule of thumb says that every numeric constant, other than 0 or 1, should be a macro. Character and string constants are problematic, since replacing a character or string constant by a macro doesn't always improve readability. I recommend using a macro instead of a character constant or string literal provided that (1) the constant is used more than once and (2) the possibility exists that the constant might someday be modified. Because of rule (2), I don't use macros such as

```
#define NUL '\0'
```

although some programmers do.

Q: What does the # operator do if the argument that it's supposed to "stringize" contains a " or \ character? [p. 324]

A: It converts " to \" and \ to \\. Consider the following macro:

```
#define STRINGIZE(x) #x
```

The preprocessor will replace `STRINGIZE("foo")` by `"\"foo\""`.

***Q: I can't get the following macro to work properly:**

```
#define CONCAT(x,y) x##y
```

`CONCAT(a,b)` gives `ab`, as expected, but `CONCAT(a,CONCAT(b,c))` gives an odd result. What's going on?

A: Thanks to rules that Kernighan and Ritchie call "bizarre," macros whose replacement lists depend on ## usually can't be called in a nested fashion. The problem is that `CONCAT(a,CONCAT(b,c))` isn't expanded in a "normal" fashion, with `CONCAT(b,c)` yielding `bc`, then `CONCAT(a,bc)` giving `abc`. Macro parameters that are preceded or followed by ## in a replacement list aren't expanded at

the time of substitution. As a result, `CONCAT(a,CONCAT(b,c))` expands to `aCONCAT(b,c)`, which can't be expanded further, since there's no macro named `aCONCAT`.

There's a way to solve the problem, but it's not pretty. The trick is to define a second macro that simply calls the first one:

```
#define CONCAT2(x,y) CONCAT(x,y)
```

Writing `CONCAT2(a,CONCAT2(b,c))` now yields the desired result. As the preprocessor expands the outer call of `CONCAT2`, it will expand `CONCAT2(b,c)` as well; the difference is that `CONCAT2`'s replacement list doesn't contain ##. If none of this makes any sense, don't worry; it's not a problem that arises often.

The # operator has a similar difficulty, by the way. If `#x` appears in a replacement list, where `x` is a macro parameter, the corresponding argument is not expanded. Thus, if `N` is a macro representing `10`, and `STR(x)` has the replacement list `#x`, expanding `STR(N)` yields `"N"`, not `"10"`. The solution is similar to the one we used with `CONCAT`: defining a second macro whose job is to call `STR`.

***Q: Suppose that the preprocessor encounters the original macro name during rescanning, as in the following example:**

```
#define N (2*M)
#define M (N+1)

i = N;   /* infinite loop? */
```

The preprocessor will replace `N` by `(2*M)`, then replace `M` by `(N+1)`. Will the preprocessor replace `N` again, thus going into an infinite loop? [p. 326]

A: Some old preprocessors will indeed go into an infinite loop, but newer ones shouldn't. According to the C standard, if the original macro name reappears during the expansion of a macro, the name is not replaced again. Here's how the assignment to `i` will look after preprocessing:

```
i = (2*(N+1));
```

Some enterprising programmers take advantage of this behavior by writing macros whose names match reserved words or functions in the standard library. Consider the `sqrt` library function. `sqrt` computes the square root of its argument, returning an implementation-defined value if the argument is negative. Perhaps we would prefer that `sqrt` return 0 if its argument is negative. Since `sqrt` is part of the standard library, we can't easily change it. We can, however, define a `sqrt` *macro* that evaluates to 0 when given a negative argument:

`sqrt` function ➤23.3

```
#undef sqrt
#define sqrt(x) ((x)>=0?sqrt(x):0)
```

A later call of `sqrt` will be intercepted by the preprocessor, which expands it into the conditional expression shown here. The call of `sqrt` inside the conditional expression won't be replaced during rescanning, so it will remain for the compiler

to handle. (Note the use of `#undef` to undefine `sqrt` before defining the `sqrt` macro. As we'll see in Section 21.1, the standard library is allowed to have both a macro and a function with the same name. Undefining `sqrt` before defining our own `sqrt` macro is a defensive measure, in case the library has already defined `sqrt` as a macro.)

Q: I get an error when I try to use predefined macros such as `__LINE__` and `__FILE__`. Is there a special header that I need to include?

A: No. These macros are recognized automatically by the preprocessor. Make sure that you have *two* underscores at the beginning and end of each macro name, not one.

Q: What's the purpose of distinguishing between a "hosted implementation" and a "freestanding implementation"? If a freestanding implementation doesn't even support the `<stdio.h>` header, what use is it? [p. 330]

A: A hosted implementation is needed for most programs (including the ones in this book), which rely on the underlying operating system for input/output and other essential services. A freestanding implementation of C would be used for programs that require no operating system (or only a minimal operating system). For example, a freestanding implementation would be needed for writing the kernel of an operating system (which requires no traditional input/output and therefore doesn't need `<stdio.h>` anyway). Freestanding implementations are also useful for writing software for embedded systems.

Q: I thought the preprocessor was just an editor. How can it evaluate constant expressions? [p. 334]

A: The preprocessor is more sophisticated than you might expect; it knows enough about C to be able to evaluate constant expressions, although it doesn't do so in quite the same way as the compiler. (For one thing, the preprocessor treats any undefined name as having the value 0. The other differences are too esoteric to go into here.) In practice, the operands in a preprocessor constant expression are usually constants, macros that represent constants, and applications of the `defined` operator.

Q: Why does C provide the `#ifdef` and `#ifndef` directives, since we can get the same effect using the `#if` directive and the `defined` operator? [p. 335]

A: The `#ifdef` and `#ifndef` directives have been a part of C since the 1970s. The `defined` operator, on the other hand, was added to C in the 1980s during standardization. So the real question is: Why was `defined` added to the language? The answer is that `defined` adds flexibility. Instead of just being able to test the existence of a single macro using `#ifdef` or `#ifndef`, we can now test any number of macros using `#if` together with `defined`. For example, the following directive checks whether `FOO` and `BAR` are defined but `BAZ` is not defined:

```
#if defined(FOO) && defined(BAR) && !defined(BAZ)
```

Q: I wanted to compile a program that I hadn't finished writing, so I "conditioned out" the unfinished part:

```
#if 0
...
#endif
```

When I compiled the program, I got an error message referring to one of the lines between `#if` and `#endif`. Doesn't the preprocessor just ignore these lines? [p. 338]

A: No, the lines aren't completely ignored. Comments are processed before preprocessing directives are executed, and the source code is divided into preprocessing tokens. Thus, an unterminated comment between `#if` and `#endif` may cause an error message. Also, an unpaired single quote or double quote character may cause undefined behavior.

Exercises

Section 14.3

1. Write parameterized macros that compute the following values.
   (a) The cube of `x`.
   (b) The remainder when `n` is divided by 4.
   (c) 1 if the product of `x` and `y` is less than 100, 0 otherwise.

   Do your macros always work? If not, describe what arguments would make them fail.

W 2. Write a macro `NELEMS(a)` that computes the number of elements in a one-dimensional array `a`. *Hint:* See the discussion of the `sizeof` operator in Section 8.1.

3. Let `DOUBLE` be the following macro:

   ```
   #define DOUBLE(x) 2*x
   ```

   (a) What is the value of `DOUBLE(1+2)`?
   (b) What is the value of `4/DOUBLE(2)`?
   (c) Fix the definition of `DOUBLE`.

W 4. For each of the following macros, give an example that illustrates a problem with the macro and show how to fix it.

   (a) `#define AVG(x,y)  (x+y)/2`

   (b) `#define AREA(x,y)  (x)*(y)`

W *5. Let `TOUPPER` be the following macro:

   ```
   #define TOUPPER(c) ('a'<=(c)&&(c)<='z'?(c)-'a'+'A':(c))
   ```

   Let `s` be a string and let `i` be an `int` variable. Show the output produced by each of the following program fragments.

   (a)
   ```
   strcpy(s, "abcd");
   i = 0;
   putchar(TOUPPER(s[++i]));
   ```

(b)
```
strcpy(s, "0123");
i = 0;
putchar(TOUPPER(s[++i]));
```

6. (a) Write a macro `DISP(f,x)` that expands into a call of `printf` that displays the value of the function `f` when called with argument `x`. For example,

```
DISP(sqrt, 3.0);
```

should expand into

```
printf("sqrt(%g) = %g\n", 3.0, sqrt(3.0));
```

(b) Write a macro `DISP2(f,x,y)` that's similar to `DISP` but works for functions with two arguments.

W *7. Let `GENERIC_MAX` be the following macro:

```
#define GENERIC_MAX(type)          \
type type##_max(type x, type y)    \
{                                  \
  return x > y ? x : y;            \
}
```

(a) Show the preprocessor's expansion of `GENERIC_MAX(long)`.
(b) Explain why `GENERIC_MAX` doesn't work for basic types such as `unsigned long`.
(c) Describe a technique that would allow us to use `GENERIC_MAX` with basic types such as `unsigned long`. *Hint:* Don't change the definition of `GENERIC_MAX`.

*8. Suppose we want a macro that expands into a string containing the current line number and file name. In other words, we'd like to write

```
const char *str = LINE_FILE;
```

and have it expand into

```
const char *str = "Line 10 of file foo.c";
```

where `foo.c` is the file containing the program and 10 is the line on which the invocation of `LINE_FILE` appears. *Warning:* This exercise is for experts only. Be sure to read the Q&A section carefully before attempting!

9. Write the following parameterized macros.

(a) `CHECK(x,y,n)` – Has the value 1 if both `x` and `y` fall between 0 and `n` – 1, inclusive.
(b) `MEDIAN(x,y,z)` – Finds the median of `x`, `y`, and `z`.
(c) `POLYNOMIAL(x)` – Computes the polynomial $3x^5 + 2x^4 - 5x^3 - x^2 + 7x - 6$.

10. Functions can often—but not always—be written as parameterized macros. Discuss what characteristics of a function would make it unsuitable as a macro.

11. (C99) C programmers often use the `fprintf` function to write error messages:

`fprintf` function ➤22.3

```
fprintf(stderr, "Range error: index = %d\n", index);
```

`stderr` stream ➤22.1

`stderr` is C's "standard error" stream; the remaining arguments are the same as those for `printf`, starting with the format string. Write a macro named `ERROR` that generates the call of `fprintf` shown above when given a format string and the items to be displayed:

```
ERROR("Range error: index = %d\n", index);
```

Section 14.4

W 12. Suppose that the macro `M` has been defined as follows:

```
#define M 10
```

Which of the following tests will fail?

(a) `#if M`
(b) `#ifdef M`
(c) `#ifndef M`
(d) `#if defined(M)`
(e) `#if !defined(M)`

13. (a) Show what the following program will look like after preprocessing. You may ignore any lines added to the program as a result of including the `<stdio.h>` header.

```
#include <stdio.h>

#define N 100

void f(void);

int main(void)
{
  f();
#ifdef N
#undef N
#endif
  return 0;
}

void f(void)
{
#if defined(N)
  printf("N is %d\n", N);
#else
  printf("N is undefined\n");
#endif
}
```

(b) What will be the output of this program?

W *14. Show what the following program will look like after preprocessing. Some lines of the program may cause compilation errors; find all such errors.

```
#define N = 10
#define INC(x) x+1
#define SUB (x,y) x-y
#define SQR(x) ((x)*(x))
#define CUBE(x) (SQR(x)*(x))
#define M1(x,y) x##y
#define M2(x,y) #x #y

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

#ifdef N
  i = j;
#else
  j = i;
#endif

  i = 10 * INC(j);
```

```
    i = SUB(j, k);
    i = SQR(SQR(j));
    i = CUBE(j);
    i = M1(j, k);
    puts(M2(i, j));

#undef SQR
    i = SQR(j);
#define SQR
    i = SQR(j);

    return 0;
}
```

15. Suppose that a program needs to display messages in either English, French, or Spanish. Using conditional compilation, write a program fragment that displays one of the following three messages, depending on whether or not the specified macro is defined:

```
Insert Disk 1          (if ENGLISH is defined)
Inserez Le Disque 1    (if FRENCH is defined)
Inserte El Disco 1     (if SPANISH is defined)
```

Section 14.5

*16. (C99) Assume that the following macro definitions are in effect:

```
#define IDENT(x) PRAGMA(ident #x)
#define PRAGMA(x) _Pragma(#x)
```

What will the following line look like after macro expansion?

```
IDENT(foo)
```