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;
}
```

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.

Q&A

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 arguments must appear when the function is called. However, there are legitimate reasons for omitting parameter names, and some programmers prefer to do so.

Q&A

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));
```

C99

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.

Array Arguments

Q&A

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:

```
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.

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}.

lvalues ➤4.2

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.

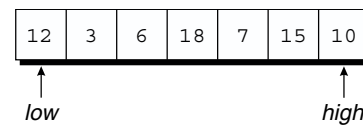
After step 1, the element e is in its proper location. Since the elements to the left of e are all less than or equal to it, they'll be in their proper places once they've been sorted in step 2; similar reasoning applies to the elements to the right of e.

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

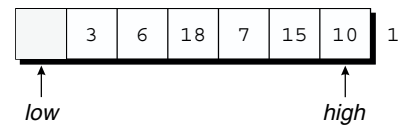
that's easy to understand but not particularly efficient. I'll first describe the partitioning algorithm informally; later, we'll translate it into C code.

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:

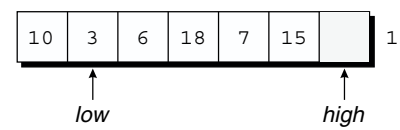
Let's start with an array containing seven elements. *low* points to the first element; *high* points to the last one.



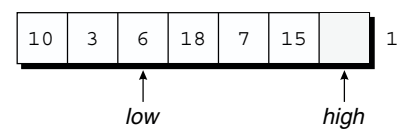
The first element, 12, is the partitioning element. Copying it somewhere else leaves a hole at the beginning of the array.



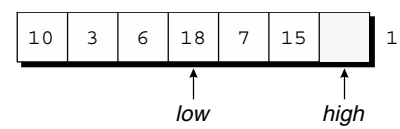
We now compare the element pointed to by *high* with 12. Since 10 is smaller than 12, it's on the wrong side of the array, so we move it to the hole and shift *low* to the right.



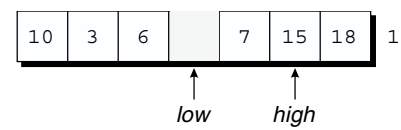
low points to the number 3, which is less than 12 and therefore doesn't need to be moved. We shift *low* to the right instead.



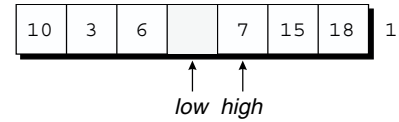
Since 6 is also less than 12, we shift *low* again.



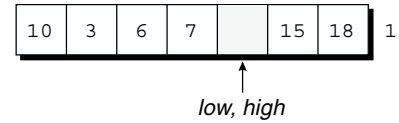
low now points to 18, which is larger than 12 and therefore out of position. After moving 18 to the hole, we shift *high* to the left.



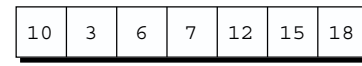
high points to 15, which is greater than 12 and thus doesn't need to be moved. We shift *high* to the left and continue.



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.

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 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.