6 Loops

A program without a loop and a structured variable isn't worth writing.

Chapter 5 covered C's selection statements, `if` and `switch`. This chapter introduces C's iteration statements, which allow us to set up loops.

A ***loop*** is a statement whose job is to repeatedly execute some other statement (the ***loop body***). In C, every loop has a ***controlling expression***. Each time the loop body is executed (an ***iteration*** of the loop), the controlling expression is evaluated; if the expression is true—has a value that's not zero—the loop continues to execute.

C provides three iteration statements: `while`, `do`, and `for`, which are covered in Sections 6.1, 6.2, and 6.3, respectively. The `while` statement is used for loops whose controlling expression is tested *before* the loop body is executed. The `do` statement is used if the expression is tested *after* the loop body is executed. The `for` statement is convenient for loops that increment or decrement a counting variable. Section 6.3 also introduces the comma operator, which is used primarily in `for` statements.

The last two sections of this chapter are devoted to C features that are used in conjunction with loops. Section 6.4 describes the `break`, `continue`, and `goto` statements. `break` jumps out of a loop and transfers control to the next statement after the loop, `continue` skips the rest of a loop iteration, and `goto` jumps to any statement within a function. Section 6.5 covers the null statement, which can be used to create loops with empty bodies.

6.1 The `while` Statement

Of all the ways to set up loops in C, the `while` statement is the simplest and most fundamental. The `while` statement has the form

while statement

```
while ( expression ) statement
```

The expression inside the parentheses is the controlling expression; the statement after the parentheses is the loop body. Here's an example:

```
while (i < n)    /* controlling expression */
  i = i * 2;     /* loop body */
```

Note that the parentheses are mandatory and that nothing goes between the right parenthesis and the loop body. (Some languages require the word `do`.)

When a `while` statement is executed, the controlling expression is evaluated first. If its value is nonzero (true), the loop body is executed and the expression is tested again. The process continues in this fashion—first testing the controlling expression, then executing the loop body—until the controlling expression eventually has the value zero.

The following example uses a `while` statement to compute the smallest power of 2 that is greater than or equal to a number `n`:

```
i = 1;
while (i < n)
  i = i * 2;
```

Suppose that `n` has the value 10. The following trace shows what happens when the `while` statement is executed:

`i = 1;`	`i` is now 1.
Is `i < n`?	Yes; continue.
`i = i * 2;`	`i` is now 2.
Is `i < n`?	Yes; continue.
`i = i * 2;`	`i` is now 4.
Is `i < n`?	Yes; continue.
`i = i * 2;`	`i` is now 8.
Is `i < n`?	Yes; continue.
`i = i * 2;`	`i` is now 16.
Is `i < n`?	No; exit from loop.

Notice how the loop keeps going as long as the controlling expression `(i < n)` is true. When the expression is false, the loop terminates, and `i` is greater than or equal to `n`, as desired.

Although the loop body must be a single statement, that's merely a technicality. If we want more than one statement, we can just use braces to create a single compound statement:

compound statements ➤5.2

```
while (i > 0) {
  printf("T minus %d and counting\n", i);
  i--;
}
```

Some programmers always use braces, even when they're not strictly necessary:

```
while (i < n) {   /* braces allowed, but not required */
  i = i * 2;
}
```

As a second example, let's trace the execution of the following statements, which display a series of "countdown" messages:

```
i = 10;
while (i > 0) {
  printf("T minus %d and counting\n", i);
  i--;
}
```

Before the `while` statement is executed, the variable `i` is assigned the value 10. Since 10 is greater than 0, the loop body is executed, causing the message `T minus 10 and counting` to be printed and `i` to be decremented. The condition `i > 0` is then tested again. Since 9 is greater than 0, the loop body is executed once more. This process continues until the message `T minus 1 and counting` is printed and `i` becomes 0. The test `i > 0` then fails, causing the loop to terminate.

The countdown example leads us to make several observations about the `while` statement:

- The controlling expression is false when a `while` loop terminates. Thus, when a loop controlled by the expression `i > 0` terminates, `i` must be less than or equal to 0. (Otherwise, we'd still be executing the loop!)
- The body of a `while` loop may not be executed at all. Since the controlling expression is tested *before* the loop body is executed, it's possible that the body isn't executed even once. If `i` has a negative or zero value when the countdown loop is first entered, the loop will do nothing.
- A `while` statement can often be written in a variety of ways. For example, we could make the countdown loop more concise by decrementing `i` inside the call of `printf`:

Q&A

```
while (i > 0)
  printf("T minus %d and counting\n", i--);
```

Infinite Loops

A `while` statement won't terminate if the controlling expression always has a nonzero value. In fact, C programmers sometimes deliberately create an ***infinite loop*** by using a nonzero constant as the controlling expression:

idiom

```
while (1) …
```

A `while` statement of this form will execute forever unless its body contains a statement that transfers control out of the loop (`break`, `goto`, `return`) or calls a function that causes the program to terminate.

PROGRAM Printing a Table of Squares

Let's write a program that prints a table of squares. The program will first prompt the user to enter a number *n*. It will then print *n* lines of output, with each line containing a number between 1 and *n* together with its square:

```
This program prints a table of squares.
Enter number of entries in table: 5
         1         1
         2         4
         3         9
         4        16
         5        25
```

Let's have the program store the desired number of squares in a variable named `n`. We'll need a loop that repeatedly prints a number `i` and its square, starting with `i` equal to 1. The loop will repeat as long as `i` is less than or equal to `n`. We'll have to make sure to add 1 to `i` each time through the loop.

We'll write the loop as a `while` statement. (Frankly, we haven't got much choice, since the `while` statement is the only kind of loop we've covered so far.) Here's the finished program:

square.c

```
/* Prints a table of squares using a while statement */

#include <stdio.h>

int main(void)
{
  int i, n;

  printf("This program prints a table of squares.\n");
  printf("Enter number of entries in table: ");
  scanf("%d", &n);

  i = 1;
  while (i <= n) {
    printf("%10d%10d\n", i, i * i);
    i++;
  }

  return 0;
}
```

Note how `square.c` displays numbers in neatly aligned columns. The trick is to use a conversion specification like `%10d` instead of just `%d`, taking advantage of the fact that `printf` right-justifies numbers when a field width is specified.

PROGRAM Summing a Series of Numbers

As a second example of the `while` statement, let's write a program that sums a series of integers entered by the user. Here's what the user will see:

```
This program sums a series of integers.
Enter integers (0 to terminate): 8 23 71 5 0
The sum is: 107
```

Clearly we'll need a loop that uses `scanf` to read a number and then adds the number to a running total.

Letting `n` represent the number just read and `sum` the total of all numbers previously read, we end up with the following program:

sum.c

```
/* Sums a series of numbers */

#include <stdio.h>

int main(void)
{
  int n, sum = 0;

  printf("This program sums a series of integers.\n");
  printf("Enter integers (0 to terminate): ");

  scanf("%d", &n);
  while (n != 0) {
    sum += n;
    scanf("%d", &n);
  }
  printf("The sum is: %d\n", sum);

  return 0;
}
```

Notice that the condition `n != 0` is tested just after a number is read, allowing the loop to terminate as soon as possible. Also note that there are two identical calls of `scanf`, which is often hard to avoid when using `while` loops.

6.2 The `do` Statement

The `do` statement is closely related to the `while` statement; in fact, the `do` statement is essentially just a `while` statement whose controlling expression is tested *after* each execution of the loop body. The `do` statement has the form

`do` statement

```
do statement while ( expression ) ;
```

As with the `while` statement, the body of a `do` statement must be one statement (possibly compound, of course) and the controlling expression must be enclosed within parentheses.

When a `do` statement is executed, the loop body is executed first, then the controlling expression is evaluated. If the value of the expression is nonzero, the loop

body is executed again and then the expression is evaluated once more. Execution of the `do` statement terminates when the controlling expression has the value 0 *after* the loop body has been executed.

Let's rewrite the countdown example of Section 6.1, using a `do` statement this time:

```
i = 10;
do {
  printf("T minus %d and counting\n", i);
  --i;
} while (i > 0);
```

When the `do` statement is executed, the loop body is first executed, causing the message `T minus 10 and counting` to be printed and `i` to be decremented. The condition `i > 0` is now tested. Since 9 is greater than 0, the loop body is executed a second time. This process continues until the message `T minus 1 and counting` is printed and `i` becomes 0. The test `i > 0` now fails, causing the loop to terminate. As this example shows, the `do` statement is often indistinguishable from the `while` statement. The difference between the two is that the body of a `do` statement is always executed at least once; the body of a `while` statement is skipped entirely if the controlling expression is 0 initially.

Incidentally, it's a good idea to use braces in *all* `do` statements, whether or not they're needed, because a `do` statement without braces can easily be mistaken for a `while` statement:

```
do
  printf("T minus %d and counting\n", i--);
while (i > 0);
```

A careless reader might think that the word `while` was the beginning of a `while` statement.

PROGRAM Calculating the Number of Digits in an Integer

Although the `while` statement appears in C programs much more often than the `do` statement, the latter is handy for loops that must execute at least once. To illustrate this point, let's write a program that calculates the number of digits in an integer entered by the user:

```
Enter a nonnegative integer: 60
The number has 2 digit(s).
```

Our strategy will be to divide the user's input by 10 repeatedly until it becomes 0; the number of divisions performed is the number of digits. Clearly we'll need some kind of loop, since we don't know how many divisions it will take to reach 0. But should we use a `while` statement or a `do` statement? The `do` statement turns out to be more attractive, because every integer—even 0—has at least *one* digit. Here's the program:

numdigits.c

```
/* Calculates the number of digits in an integer */

#include <stdio.h>

int main(void)
{
  int digits = 0, n;

  printf("Enter a nonnegative integer: ");
  scanf("%d", &n);

  do {
    n /= 10;
    digits++;
  } while (n > 0);

  printf("The number has %d digit(s).\n", digits);

  return 0;
}
```

To see why the `do` statement is the right choice, let's see what would happen if we were to replace the `do` loop by a similar `while` loop:

```
while (n > 0) {
  n /= 10;
  digits++;
}
```

If `n` is 0 initially, this loop won't execute at all, and the program would print

```
The number has 0 digit(s).
```

6.3 The `for` Statement

We now come to the last of C's loops: the `for` statement. Don't be discouraged by the `for` statement's apparent complexity; it's actually the best way to write many loops. The `for` statement is ideal for loops that have a "counting" variable, but it's versatile enough to be used for other kinds of loops as well.

The `for` statement has the form

`for` statement

`for ( `*expr1*` ; `*expr2*` ; `*expr3*` ) `*statement*

where *expr1*, *expr2*, and *expr3* are expressions. Here's an example:

```
for (i = 10; i > 0; i--)
  printf("T minus %d and counting\n", i);
```

When this `for` statement is executed, the variable `i` is initialized to 10, then `i` is tested to see if it's greater than 0. Since it is, the message `T minus 10 and`

counting is printed, then i is decremented. The condition i > 0 is then tested again. The loop body will be executed 10 times in all, with i varying from 10 down to 1.

The for statement is closely related to the while statement. In fact, except in a few rare cases, a for loop can always be replaced by an equivalent while loop:

Q&A

```
expr1;
while ( expr2 ) {
  statement
  expr3;
}
```

As this pattern shows, *expr1* is an initialization step that's performed only once, before the loop begins to execute, *expr2* controls loop termination (the loop continues executing as long as the value of *expr2* is nonzero), and *expr3* is an operation to be performed at the end of each loop iteration. Applying this pattern to our previous for loop example, we arrive at the following:

```
i = 10;
while (i > 0) {
  printf("T minus %d and counting\n", i);
  i--;
}
```

Studying the equivalent while statement can help us understand the fine points of a for statement. For example, suppose that we replace i-- by --i in our for loop example:

```
for (i = 10; i > 0; --i)
  printf("T minus %d and counting\n", i);
```

How does this change affect the loop? Looking at the equivalent while loop, we see that it has no effect:

```
i = 10;
while (i > 0) {
  printf("T minus %d and counting\n", i);
  --i;
}
```

Since the first and third expressions in a for statement are executed as statements, their values are irrelevant—they're useful only for their side effects. Consequently, these two expressions are usually assignments or increment/decrement expressions.

for Statement Idioms

The for statement is usually the best choice for loops that "count up" (increment a variable) or "count down" (decrement a variable). A for statement that counts up or down a total of n times will usually have one of the following forms:

- ***Counting up from* 0 *to* n–1*:***

idiom

```
for (i = 0; i < n; i++) …
```

- ***Counting up from* 1 *to* n*:***

idiom

```
for (i = 1; i <= n; i++) …
```

- ***Counting down from* n–1 *to* 0*:***

idiom

```
for (i = n - 1; i >= 0; i--) …
```

- ***Counting down from* n *to* 1*:***

idiom

```
for (i = n; i > 0; i--) …
```

Imitating these patterns will help you avoid some of the following errors, which beginning C programmers often make:

- Using < instead of > (or vice versa) in the controlling expression. Notice that "counting up" loops use the < or <= operator, while "counting down" loops rely on > or >=.
- Using == in the controlling expression instead of <, <=, >, or >=. A controlling expression needs to be true at the beginning of the loop, then later become false so that the loop can terminate. A test such as `i == n` doesn't make much sense, because it won't be true initially.
- "Off-by-one" errors such as writing the controlling expression as `i <= n` instead of `i < n`.

Omitting Expressions in a `for` Statement

The `for` statement is even more flexible than we've seen so far. Some `for` loops may not need all three of the expressions that normally control the loop, so C allows us to omit any or all of the expressions.

If the *first* expression is omitted, no initialization is performed before the loop is executed:

```
i = 10;
for (; i > 0; --i)
  printf("T minus %d and counting\n", i);
```

In this example, `i` has been initialized by a separate assignment, so we've omitted the first expression in the `for` statement. (Notice that the semicolon between the first and second expressions remains. The two semicolons must always be present, even when we've omitted some of the expressions.)

If we omit the *third* expression in a `for` statement, the loop body is responsible for ensuring that the value of the second expression eventually becomes false. Our `for` statement example could be written like this:

```
for (i = 10; i > 0;)
  printf("T minus %d and counting\n", i--);
```

To compensate for omitting the third expression, we've arranged for `i` to be decremented inside the loop body.

When the *first* and *third* expressions are both omitted, the resulting loop is nothing more than a `while` statement in disguise. For example, the loop

```
for (; i > 0;)
  printf("T minus %d and counting\n", i--);
```

is the same as

```
while (i > 0)
  printf("T minus %d and counting\n", i--);
```

The `while` version is clearer and therefore preferable.

If the *second* expression is missing, it defaults to a true value, so the `for` statement doesn't terminate (unless stopped in some other fashion). For example, some programmers use the following `for` statement to establish an infinite loop:

Q&A

idiom

```
for (;;) …
```

C99 `for` Statements in C99

In C99, the first expression in a `for` statement can be replaced by a declaration. This feature allows the programmer to declare a variable for use by the loop:

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

The variable `i` need not have been declared prior to this statement. (In fact, if a declaration of `i` already exists, this statement creates a *new* version of `i` that will be used solely within the loop.)

A variable declared by a `for` statement can't be accessed outside the body of the loop (we say that it's not ***visible*** outside the loop):

```
for (int i = 0; i < n; i++) {
  …
  printf("%d", i);   /* legal; i is visible inside loop */
  …
}
printf("%d", i);     /*** WRONG ***/
```

Having a `for` statement declare its own control variable is usually a good idea: it's convenient and it can make programs easier to understand. However, if the program needs to access the variable after loop termination, it's necessary to use the older form of the `for` statement.

Incidentally, a `for` statement may declare more than one variable, provided that all variables have the same type:

```
for (int i = 0, j = 0; i < n; i++)
  …
```

The Comma Operator

On occasion, we might like to write a `for` statement with two (or more) initialization expressions or one that increments several variables each time through the loop. We can do this by using a ***comma expression*** as the first or third expression in the `for` statement.

A comma expression has the form

comma expression

expr1 , *expr2*

where *expr1* and *expr2* are any two expressions. A comma expression is evaluated in two steps: First, *expr1* is evaluated and its value discarded. Second, *expr2* is evaluated; its value is the value of the entire expression. Evaluating *expr1* should always have a side effect; if it doesn't, then *expr1* serves no purpose.

For example, suppose that `i` and `j` have the values 1 and 5, respectively. When the comma expression `++i, i + j` is evaluated, `i` is first incremented, then `i + j` is evaluated, so the value of the expression is 7. (And, of course, `i` now has the value 2.) The precedence of the comma operator is less than that of all other operators, by the way, so there's no need to put parentheses around `++i` and `i + j`.

Occasionally, we'll need to chain together a series of comma expressions, just as we sometimes chain assignments together. The comma operator is left associative, so the compiler interprets

```
i = 1, j = 2, k = i + j
```

as

```
((i = 1), (j = 2)), (k = (i + j))
```

Since the left operand in a comma expression is evaluated before the right operand, the assignments `i = 1`, `j = 2`, and `k = i + j` will be performed from left to right.

The comma operator is provided for situations where C requires a single expression, but we'd like to have two or more expressions. In other words, the comma operator allows us to "glue" two expressions together to form a single expression. (Note the similarity to the compound statement, which allows us to treat a group of statements as a single statement.)

macro definitions ➤14.3

The need to glue expressions together doesn't arise that often. Certain macro definitions can benefit from the comma operator, as we'll see in a later chapter. The `for` statement is the only other place where the comma operator is likely to be found. For example, suppose that we want to initialize two variables when entering a `for` statement. Instead of writing

```
sum = 0;
for (i = 1; i <= N; i++)
  sum += i;
```

we can write

```
for (sum = 0, i = 1; i <= N; i++)
  sum += i;
```

The expression `sum = 0, i = 1` first assigns 0 to `sum`, then assigns 1 to `i`. With additional commas, the `for` statement could initialize more than two variables.

PROGRAM Printing a Table of Squares (Revisited)

The `square.c` program (Section 6.1) can be improved by converting its `while` loop to a `for` loop:

square2.c

```
/* Prints a table of squares using a for statement */

#include <stdio.h>

int main(void)
{
  int i, n;

  printf("This program prints a table of squares.\n");
  printf("Enter number of entries in table: ");
  scanf("%d", &n);

  for (i = 1; i <= n; i++)
    printf("%10d%10d\n", i, i * i);

  return 0;
}
```

We can use this program to illustrate an important point about the `for` statement: C places no restrictions on the three expressions that control its behavior. Although these expressions usually initialize, test, and update the same variable, there's no requirement that they be related in any way. Consider the following version of the same program:

square3.c

```
/* Prints a table of squares using an odd method */

#include <stdio.h>

int main(void)
{
  int i, n, odd, square;

  printf("This program prints a table of squares.\n");
  printf("Enter number of entries in table: ");
  scanf("%d", &n);

  i = 1;
  odd = 3;
  for (square = 1; i <= n; odd += 2) {
    printf("%10d%10d\n", i, square);
    ++i;
```

```
    square += odd;
  }

  return 0;
}
```

The `for` statement in this program initializes one variable (`square`), tests another (`i`), and increments a third (`odd`). `i` is the number to be squared, `square` is the square of `i`, and `odd` is the odd number that must be added to the current square to get the next square (allowing the program to compute consecutive squares without performing any multiplications).

The tremendous flexibility of the `for` statement can sometimes be useful; we'll find it to be a great help when working with linked lists. The `for` statement can easily be misused, though, so don't go overboard. The `for` loop in `square3.c` would be a lot clearer if we rearranged its pieces so that the loop is clearly controlled by `i`.

linked lists ➤17.5

6.4 Exiting from a Loop

We've seen how to write loops that have an exit point before the loop body (using `while` and `for` statements) or after it (using `do` statements). Occasionally, however, we'll need a loop with an exit point in the middle. We may even want a loop to have more than one exit point. The `break` statement makes it possible to write either kind of loop.

After we've examined the `break` statement, we'll look at a couple of related statements: `continue` and `goto`. The `continue` statement makes it possible to skip part of a loop iteration without jumping out of the loop. The `goto` statement allows a program to jump from one statement to another. Thanks to the availability of statements such as `break` and `continue`, the `goto` statement is rarely used.

The `break` Statement

We've already discussed how a `break` statement can transfer control out of a `switch` statement. The `break` statement can also be used to jump out of a `while`, `do`, or `for` loop.

Suppose that we're writing a program that checks whether a number `n` is prime. Our plan is to write a `for` statement that divides `n` by the numbers between 2 and `n` – 1. We should break out of the loop as soon as any divisor is found; there's no need to try the remaining possibilities. After the loop has terminated, we can use an `if` statement to determine whether termination was premature (hence `n` isn't prime) or normal (`n` is prime):

```
for (d = 2; d < n; d++)
  if (n % d == 0)
    break;
```

```
if (d < n)
  printf("%d is divisible by %d\n", n, d);
else
  printf("%d is prime\n", n);
```

The `break` statement is particularly useful for writing loops in which the exit point is in the middle of the body rather than at the beginning or end. Loops that read user input, terminating when a particular value is entered, often fall into this category:

```
for (;;) {
  printf("Enter a number (enter 0 to stop): ");
  scanf("%d", &n);
  if (n == 0)
    break;
  printf("%d cubed is %d\n", n, n * n * n);
}
```

A `break` statement transfers control out of the *innermost* enclosing `while`, `do`, `for`, or `switch` statement. Thus, when these statements are nested, the `break` statement can escape only one level of nesting. Consider the case of a `switch` statement nested inside a `while` statement:

```
while (…) {
  switch (…) {
    …
    break;
    …
  }
}
```

The `break` statement transfers control out of the `switch` statement, but not out of the `while` loop. I'll return to this point later.

The `continue` Statement

The `continue` statement doesn't really belong here, because it doesn't exit from a loop. It's similar to `break`, though, so its inclusion in this section isn't completely arbitrary. `break` transfers control just *past* the end of a loop, while `continue` transfers control to a point just *before* the end of the loop body. With `break`, control leaves the loop; with `continue`, control remains inside the loop. There's another difference between `break` and `continue`: `break` can be used in `switch` statements and loops (`while`, `do`, and `for`), whereas `continue` is limited to loops.

The following example, which reads a series of numbers and computes their sum, illustrates a simple use of `continue`. The loop terminates when 10 nonzero numbers have been read. Whenever the number 0 is read, the `continue` statement is executed, skipping the rest of the loop body (the statements `sum += i;` and `n++;`) but remaining inside the loop.

```
n = 0;
sum = 0;
while (n < 10) {
  scanf("%d", &i);
  if (i == 0)
    continue;
  sum += i;
  n++;
  /* continue jumps to here */
}
```

If `continue` were not available, we could have written the example as follows:

```
n = 0;
sum = 0;
while (n < 10) {
  scanf("%d", &i);
  if (i != 0) {
    sum += i;
    n++;
  }
}
```

The `goto` Statement

`break` and `continue` are jump statements that transfer control from one point in the program to another. Both are restricted: the target of a `break` is a point just *beyond* the end of the enclosing loop, while the target of a `continue` is a point just *before* the end of the loop. The `goto` statement, on the other hand, is capable of jumping to *any* statement in a function, provided that the statement has a ***label***. (C99 places an additional restriction on the `goto` statement: it can't be used to bypass the declaration of a variable-length array.)

C99

variable-length arrays ➤8.3

A label is just an identifier placed at the beginning of a statement:

labeled statement

identifier `:` *statement*

A statement may have more than one label. The `goto` statement itself has the form

`goto` statement

`goto` *identifier* `;`

Executing the statement `goto` *L*`;` transfers control to the statement that follows the label *L*, which must be in the same function as the `goto` statement itself.

If C didn't have a `break` statement, here's how we might use a `goto` statement to exit prematurely from a loop:

```
for (d = 2; d < n; d++)
  if (n % d == 0)
    goto done;
```

```
done:
if (d < n)
  printf("%d is divisible by %d\n", n, d);
else
  printf("%d is prime\n", n);
```

Q&A

exit function ➤9.5

The `goto` statement, a staple of older programming languages, is rarely needed in everyday C programming. The `break`, `continue`, and `return` statements—which are essentially restricted `goto` statements—and the `exit` function are sufficient to handle most situations that might require a `goto` in other languages.

Nonetheless, the `goto` statement can be helpful once in a while. Consider the problem of exiting a loop from within a `switch` statement. As we saw earlier, the `break` statement doesn't quite have the desired effect: it exits from the `switch`, but not from the loop. A `goto` statement solves the problem:

```
while (…) {
  switch (…) {
    …
    goto loop_done;   /* break won't work here */
    …
  }
}
loop_done: …
```

The `goto` statement is also useful for exiting from nested loops.

PROGRAM Balancing a Checkbook

Many simple interactive programs are menu-based: they present the user with a list of commands to choose from. Once the user has selected a command, the program performs the desired action, then prompts the user for another command. This process continues until the user selects an "exit" or "quit" command.

The heart of such a program will obviously be a loop. Inside the loop will be statements that prompt the user for a command, read the command, then decide what action to take:

```
for (;;) {
  prompt user to enter command;
  read command;
  execute command;
}
```

Executing the command will require a `switch` statement (or cascaded `if` statement):

```
for (;;) {
  prompt user to enter command;
  read command;
  switch (command) {
    case command1: perform operation1; break;
```

```
    case command2: perform operation2; break;
    .
    .
    .
    case commandn: perform operationn; break;
    default: print error message; break;
  }
}
```

To illustrate this arrangement, let's develop a program that maintains a checkbook balance. The program will offer the user a menu of choices: clear the account balance, credit money to the account, debit money from the account, display the current balance, and exit the program. The choices are represented by the integers 0, 1, 2, 3, and 4, respectively. Here's what a session with the program will look like:

```
*** ACME checkbook-balancing program ***
Commands: 0=clear, 1=credit, 2=debit, 3=balance, 4=exit

Enter command: 1
Enter amount of credit: 1042.56
Enter command: 2
Enter amount of debit: 133.79
Enter command: 1
Enter amount of credit: 1754.32
Enter command: 2
Enter amount of debit: 1400
Enter command: 2
Enter amount of debit: 68
Enter command: 2
Enter amount of debit: 50
Enter command: 3
Current balance: $1145.09
Enter command: 4
```

When the user enters the command 4 (exit), the program needs to exit from the `switch` statement *and* the surrounding loop. The `break` statement won't help, and we'd prefer not to use a `goto` statement. Instead, we'll have the program execute a `return` statement, which will cause the `main` function to return to the operating system.

checking.c

```
/* Balances a checkbook */

#include <stdio.h>

int main(void)
{
  int cmd;
  float balance = 0.0f, credit, debit;

  printf("*** ACME checkbook-balancing program ***\n");
  printf("Commands: 0=clear, 1=credit, 2=debit, ");
  printf("3=balance, 4=exit\n\n");
```

```
  for (;;) {
    printf("Enter command: ");
    scanf("%d", &cmd);
    switch (cmd) {
      case 0:
        balance = 0.0f;
        break;
      case 1:
        printf("Enter amount of credit: ");
        scanf("%f", &credit);
        balance += credit;
        break;
      case 2:
        printf("Enter amount of debit: ");
        scanf("%f", &debit);
        balance -= debit;
        break;
      case 3:
        printf("Current balance: $%.2f\n", balance);
        break;
      case 4:
        return 0;
      default:
        printf("Commands: 0=clear, 1=credit, 2=debit, ");
        printf("3=balance, 4=exit\n\n");
        break;
    }
  }
}
```

Note that the `return` statement is not followed by a `break` statement. A `break` immediately following a `return` can never be executed, and many compilers will issue a warning message.

6.5 The Null Statement

A statement can be ***null***—devoid of symbols except for the semicolon at the end. Here's an example:

```
i = 0; ; j = 1;
```

This line contains three statements: an assignment to `i`, a null statement, and an assignment to `j`.

Q&A The null statement is primarily good for one thing: writing loops whose bodies are empty. As an example, recall the prime-finding loop of Section 6.4:

```
for (d = 2; d < n; d++)
  if (n % d == 0)
    break;
```

If we move the `n % d == 0` condition into the loop's controlling expression, the body of the loop becomes empty:

```
for (d = 2; d < n && n % d != 0; d++)
  /* empty loop body */ ;
```

Each time through the loop, the condition `d < n` is tested first; if it's false, the loop terminates. Otherwise, the condition `n % d != 0` is tested, and if that's false, the loop terminates. (In the latter case, `n % d == 0` must be true; in other words, we've found a divisor of `n`.)

Note how we've put the null statement on a line by itself, instead of writing

```
for (d = 2; d < n && n % d != 0; d++);
```

Q&A C programmers customarily put the null statement on a line by itself. Otherwise, someone reading the program might get confused about whether the statement after the `for` was actually its body:

```
for (d = 2; d < n && n % d != 0; d++);
if (d < n)
  printf("%d is divisible by %d\n", n, d);
```

Converting an ordinary loop into one with an empty body doesn't buy much: the new loop is often more concise but usually no more efficient. In a few cases, though, a loop with an empty body is clearly superior to the alternatives. For example, we'll find these loops to be handy for reading character data.

reading characters ➤7.3

Accidentally putting a semicolon after the parentheses in an `if`, `while`, or `for` statement creates a null statement, thus ending the `if`, `while`, or `for` prematurely.

- In an `if` statement, putting a semicolon after the parentheses creates an `if` statement that apparently performs the same action regardless of the value of its controlling expression:

  ```
  if (d == 0);                                /*** WRONG ***/
    printf("Error: Division by zero\n");
  ```

  The call of `printf` isn't inside the `if` statement, so it's performed regardless of whether `d` is equal to 0.

- In a `while` statement, putting a semicolon after the parentheses may create an infinite loop:

  ```
  i = 10;
  while (i > 0);                              /*** WRONG ***/
  {
    printf("T minus %d and counting\n", i);
    --i;
  }
  ```

Another possibility is that the loop terminates, but the statement that should be the loop body is executed only once, after the loop has terminated:

```
i = 11;
while (--i > 0);                          /*** WRONG ***/
  printf("T minus %d and counting\n", i);
```

This example prints the message

```
T minus 0 and counting
```

- In a `for` statement, putting a semicolon after the parentheses causes the statement that should be the loop body to be executed only once:

```
for (i = 10; i > 0; i--);                 /*** WRONG ***/
  printf("T minus %d and counting\n", i);
```

This example also prints the message

```
T minus 0 and counting
```

Q & A

Q: The following loop appears in Section 6.1:

```
while (i > 0)
  printf("T minus %d and counting\n", i--);
```

Why not shorten the loop even more by removing the "`> 0`" test?

```
while (i)
  printf("T minus %d and counting\n", i--);
```

This version will stop when `i` reaches 0, so it should be just as good as the original. [p. 101]

A: The new version is certainly more concise, and many C programmers would write the loop in just this way. It does have drawbacks, though.

First, the new loop is not as easy to read as the original. It's clear that the loop will terminate when `i` reaches 0, but it's not obvious whether we're counting up or down. In the original loop, that information can be deduced from the controlling expression, `i > 0`.

Second, the new loop behaves differently than the original if `i` should happen to have a negative value when the loop begins to execute. The original loop terminates immediately, but the new loop doesn't.

Q: Section 6.3 says that, except in rare cases, `for` loops can be converted to `while` loops using a standard pattern. Can you give an example of such a case? [p. 106]

A: When the body of a `for` loop contains a `continue` statement, the `while` pattern shown in Section 6.3 is no longer valid. Consider the following example from Section 6.4:

```
n = 0;
sum = 0;
while (n < 10) {
  scanf("%d", &i);
  if (i == 0)
    continue;
  sum += i;
  n++;
}
```

At first glance, it looks as though we could convert the `while` loop into a `for` loop:

```
sum = 0;
for (n = 0; n < 10; n++) {
  scanf("%d", &i);
  if (i == 0)
    continue;
  sum += i;
}
```

Unfortunately, this loop isn't equivalent to the original. When `i` is equal to 0, the original loop doesn't increment `n`, but the new loop does.

Q: Which form of infinite loop is preferable, `while (1)` or `for (;;)`? [p. 108]

A: C programmers have traditionally preferred `for (;;)` for reasons of efficiency; older compilers would often force programs to test the `1` condition each time through the `while` loop. With modern compilers, however, there should be no difference in performance.

Q: I've heard that programmers should never use the `continue` statement. Is this true?

A: It's true that `continue` statements are rare. Still, `continue` is handy once in a while. Suppose we're writing a loop that reads some input data, checks that it's valid, and, if so, processes the input in some way. If there are a number of validity tests, or if they're complex, `continue` can be helpful. The loop would look something like this:

```
for (;;) {
  read data;
  if (data fails first test)
    continue;
  if (data fails second test)
    continue;
  .
  .
  .
```

```
    if (data fails last test)
      continue;
    process data;
}
```

Q: What's so bad about the `goto` statement? [p. 114]

A: The `goto` statement isn't inherently evil; it's just that we usually have better alternatives. Programs that use more than a few `goto` statements can quickly degenerate into "spaghetti code," with control blithely jumping from here to there. Spaghetti code is hard to understand and hard to modify.

`goto` statements make programs hard to read because they can jump either forward or backward. (In contrast, `break` and `continue` only jump forward.) A program that contains `goto` statements often requires the reader to jump back and forth in an attempt to follow the flow of control.

`goto` statements can make programs hard to modify, since they make it possible for a section of code to serve more than one purpose. For example, a statement that is preceded by a label might be reachable either by "falling through" from the previous statement or by executing one of several `goto` statements.

Q: Does the null statement have any uses besides indicating that the body of a loop is empty? [p. 116]

A: Very few. Since the null statement can appear wherever a statement is allowed, there are many *potential* uses for the null statement. In practice, however, there's only one other use of the null statement, and it's rare.

Suppose that we need to put a label at the end of a compound statement. A label can't stand alone; it must always be followed by a statement. Putting a null statement after the label solves the problem:

```
{
  …
  goto end_of_stmt;
  …
  end_of_stmt: ;
}
```

Q: Are there any other ways to make an empty loop body stand out besides putting the null statement on a line by itself? [p. 117]

A: Some programmers use a dummy `continue` statement:

```
for (d = 2; d < n && n % d != 0; d++)
  continue;
```

Others use an empty compound statement:

```
for (d = 2; d < n && n % d != 0; d++)
  {}
```

Exercises

Section 6.1

1. What output does the following program fragment produce?

```
i = 1;
while (i <= 128) {
  printf("%d ", i);
  i *= 2;
}
```

Section 6.2

2. What output does the following program fragment produce?

```
i = 9384;
do {
  printf("%d ", i);
  i /= 10;
} while (i > 0);
```

Section 6.3

*3. What output does the following `for` statement produce?

```
for (i = 5, j = i - 1; i > 0, j > 0; --i, j = i - 1)
  printf("%d ", i);
```

W 4. Which one of the following statements is not equivalent to the other two (assuming that the loop bodies are the same)?

(a) `for (i = 0; i < 10; i++) …`
(b) `for (i = 0; i < 10; ++i) …`
(c) `for (i = 0; i++ < 10; ) …`

5. Which one of the following statements is not equivalent to the other two (assuming that the loop bodies are the same)?

(a) `while (i < 10) {…}`
(b) `for (; i < 10;) {…}`
(c) `do {…} while (i < 10);`

6. Translate the program fragment of Exercise 1 into a single `for` statement.

7. Translate the program fragment of Exercise 2 into a single `for` statement.

*8. What output does the following `for` statement produce?

```
for (i = 10; i >= 1; i /= 2)
  printf("%d ", i++);
```

9. Translate the `for` statement of Exercise 8 into an equivalent `while` statement. You will need one statement in addition to the `while` loop itself.

Section 6.4

W 10. Show how to replace a `continue` statement by an equivalent `goto` statement.

11. What output does the following program fragment produce?

```
sum = 0;
for (i = 0; i < 10; i++) {
  if (i % 2)
    continue;
  sum += i;
}
printf("%d\n", sum);
```

W 12. The following "prime-testing" loop appeared in Section 6.4 as an example:

```
for (d = 2; d < n; d++)
  if (n % d == 0)
    break;
```

This loop isn't very efficient. It's not necessary to divide `n` by all numbers between 2 and `n` – 1 to determine whether it's prime. In fact, we need only check divisors up to the square root of `n`. Modify the loop to take advantage of this fact. *Hint:* Don't try to compute the square root of `n`; instead, compare `d * d` with `n`.

Section 6.5

*13. Rewrite the following loop so that its body is empty:

```
for (n = 0; m > 0; n++)
  m /= 2;
```

W*14. Find the error in the following program fragment and fix it.

```
if (n % 2 == 0);
  printf("n is even\n");
```

Programming Projects

1. Write a program that finds the largest in a series of numbers entered by the user. The program must prompt the user to enter numbers one by one. When the user enters 0 or a negative number, the program must display the largest nonnegative number entered:

```
Enter a number: 60
Enter a number: 38.3
Enter a number: 4.89
Enter a number: 100.62
Enter a number: 75.2295
Enter a number: 0

The largest number entered was 100.62
```

Notice that the numbers aren't necessarily integers.

W 2. Write a program that asks the user to enter two integers, then calculates and displays their greatest common divisor (GCD):

```
Enter two integers: 12 28
Greatest common divisor: 4
```

Hint: The classic algorithm for computing the GCD, known as Euclid's algorithm, goes as follows: Let `m` and `n` be variables containing the two numbers. If `n` is 0, then stop: `m` contains the GCD. Otherwise, compute the remainder when `m` is divided by `n`. Copy `n` into `m` and copy the remainder into `n`. Then repeat the process, starting with testing whether `n` is 0.

3. Write a program that asks the user to enter a fraction, then reduces the fraction to lowest terms:

```
Enter a fraction: 6/12
In lowest terms: 1/2
```

Hint: To reduce a fraction to lowest terms, first compute the GCD of the numerator and denominator. Then divide both the numerator and denominator by the GCD.

W 4. Add a loop to the `broker.c` program of Section 5.2 so that the user can enter more than one trade and the program will calculate the commission on each. The program should terminate when the user enters `0` as the trade value:

```
Enter value of trade: 30000
Commission: $166.00

Enter value of trade: 20000
Commission: $144.00

Enter value of trade: 0
```

5. Programming Project 1 in Chapter 4 asked you to write a program that displays a two-digit number with its digits reversed. Generalize the program so that the number can have one, two, three, or more digits. *Hint:* Use a `do` loop that repeatedly divides the number by 10, stopping when it reaches 0.

W 6. Write a program that prompts the user to enter a number *n*, then prints all even squares between 1 and *n*. For example, if the user enters 100, the program should print the following:

```
4
16
36
64
100
```

7. Rearrange the `square3.c` program so that the `for` loop initializes `i`, tests `i`, and increments `i`. Don't rewrite the program; in particular, don't use any multiplications.

W 8. Write a program that prints a one-month calendar. The user specifies the number of days in the month and the day of the week on which the month begins:

```
Enter number of days in month: 31
Enter starting day of the week (1=Sun, 7=Sat): 3

       1  2  3  4  5
 6  7  8  9 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28 29 30 31
```

Hint: This program isn't as hard as it looks. The most important part is a `for` statement that uses a variable `i` to count from 1 to `n`, where `n` is the number of days in the month, printing each value of `i`. Inside the loop, an `if` statement tests whether `i` is the last day in a week; if so, it prints a new-line character.

9. Programming Project 8 in Chapter 2 asked you to write a program that calculates the remaining balance on a loan after the first, second, and third monthly payments. Modify the program so that it also asks the user to enter the number of payments and then displays the balance remaining after each of these payments.

10. Programming Project 9 in Chapter 5 asked you to write a program that determines which of two dates comes earlier on the calendar. Generalize the program so that the user may enter any number of dates. The user will enter 0/0/0 to indicate that no more dates will be entered:

```
Enter a date (mm/dd/yy): 3/6/08
Enter a date (mm/dd/yy): 5/17/07
Enter a date (mm/dd/yy): 6/3/07
Enter a date (mm/dd/yy): 0/0/0
5/17/07 is the earliest date
```

11. The value of the mathematical constant *e* can be expressed as an infinite series:

    $e = 1 + 1/1! + 1/2! + 1/3! + \ldots$

    Write a program that approximates *e* by computing the value of

    $1 + 1/1! + 1/2! + 1/3! + \ldots + 1/n!$

    where *n* is an integer entered by the user.

12. Modify Programming Project 11 so that the program continues adding terms until the current term becomes less than ε, where ε is a small (floating-point) number entered by the user.