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++)
  ...
```