is a perfectly valid `for` loop, according to the rules of C. It does nothing before it starts, then it performs no test (and in doing so, arrives at the value "true"), then it does nothing and does the non-existent test again, and so on. Endlessly. Which brings us back to the distinction between treadmills and spirals.

Loops should spiral, and move at each iteration towards a goal. When a controlled variable is initialized, tested in the guard, and then updated, as has been the case in all of the examples shown so far, positive progress along the path of the spiral is pretty much guaranteed. But in the loop "`for(i=0;i<10;)`" there is no progress made, and no spiral. It is a treadmill – which in computing is called an *infinite loop* – and is an unwelcome addition to a program. In this simple case the error is reasonably obvious, and almost certainly there is an `i++` or similar statement missing. In more complex settings, writing a loop that doesn't make progress is an easy mistake to make, and all programmers have agonized over the "why" of an endless loop at some stage of their career.

So if you are faced with a program that appears to be "stuck" somewhere, the very first thing to check is the loops – ensure that every loop makes progress, in that whatever variable is being controlled in the loop moves towards the stopping condition, as expressed in the guard.

In most `for` loops the controlled variable will appear in all of the initialize, the guard, and the update parts.

The other reason why a program might get "stuck" is that it might be at a `scanf` waiting for you to enter some data, without you realizing. That is why you should always use `printf` to generate a prompt immediately prior to every `scanf` – to minimize this possible confusion.

Sometimes a loop appears to unexpectedly execute fewer times that you think it should. Look at the following code, and see if you can work out what it generates:

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

It prints the numbers 0 to 9, right? Wrong. In fact, the third semi-colon in the first line of the fragment means that the `for` executes the *empty statement* ten times, and then goes on to do the `printf` once and write the message `i=10`.

Beware of loops that execute the empty statement.

4.2 Case study: Calculating compound interest

You are now ready to try an exercise that involves loops. Have a go at the following task before reading on:

Write a program that shows how, for interest rates of 2%, 3%, 4%, 5%, 6%, and 7%, a regular savings amount of $100 per month grows over periods of 1 to 7 years. Figure 4.5 shows the desired output.

```
Monthly savings of $100, with monthly compounded interest
Annual Rate   |   2%     3%     4%     5%     6%     7%
After 1 years |  1211   1217   1222   1228   1234   1239
After 2 years |  2447   2470   2494   2519   2543   2568
After 3 years |  3707   3762   3818   3875   3934   3993
After 4 years |  4993   5093   5196   5301   5410   5521
After 5 years |  6305   6465   6630   6801   6977   7159
After 6 years |  7643   7878   8122   8376   8641   8916
After 7 years |  9008   9334   9675  10033  10407  10800
```

Figure 4.5: A two-dimensional table.

Figure 4.6 shows a complete program that makes use of nested loops to create a two-dimensional table. Each iteration of the outer loop – the one in which the control variable alters more slowly – generates one row of the table. Each iteration of the second loop generates a single number for the table. And to generate that value, a third loop is required, which calculates the number to be printed into that cell of the table. (The innermost loop could be replaced by a direct evaluation of a formula if we were prepared to investigate the mathematics involved, but for our purposes an iterative computation is perfectly valid.) After each row of values is generated, a newline character is required. The `printf` that generates that newline is the last statement in each iteration of the outermost loop. The two outer loops directly reflect the structure of the table – it is a two-dimensional report where each entry is a function of two parameters. A two-dimensional loop structure is thus the appropriate way of generating it.

A lot of the actual program (Figure 4.6) is concerned with output formatting. This is not uncommon – to make the output of a program look neat and tidy can be quite hard work, but as has already been noted, is well worth doing.

4.3 Program layout and style

A point to note in connection with programs in general is that tidy program layout is essential if human readers are to be able to access the structure of your program. Believe it or not, the jumble in the top box of Figure 4.7 has exactly the same functionality as the fragment in Figure 4.2 on page 47 – the C compiler doesn't care whether spaces or newlines or tabs are used to delimit the various components. But can you read it? And can you be sure that it does what it claims to? In this example, at least the variable names have been retained. Imagine if they were called `qxfgc` and `fczxp`, and so on. Indeed, there is a regular "obfuscated C" programming competition, see `http://www.ioccc.org/`, in which the objective is to write the most creative, but non-obvious, C program. When you are an expert programmer you can enter this competition if you wish; while you are learning, you should strive for simplicity and transparency.

The lower box in Figure 4.7 shows a third layout style that is preferred by some programmers. Supporters of this style argue that hiding the opening brace of a compound statement at the end of a line is misleading, and that both opening and closing