6. More repetition: the `for` loop

Herman Kamper

The previous two lectures gave us the `while` loop and put it to work. A `while` loop is completely general: it repeats a block for as long as some condition holds, and that condition can be anything at all. We used it to count to ten, and we also used it to keep reading grades until the user typed a sentinel value. Those are quite different jobs, and one statement handles both.

This lecture introduces a second kind of loop, the `for` loop. It does not let you do anything you could not already do. Every `for` loop can be rewritten as a `while` loop and every counting `while` loop can be rewritten as a `for` loop. What the `for` loop gives you is a way of writing counting loops that are shorter; it also puts everything about the counting in one place where you can see it together.

Why a second kind of loop

Look at what a counting `while` loop actually consists of:

```
counter = 1;                // where to start

while (counter <= 10)       // when to stop
{
    printf("%d\n", counter);

    counter = counter + 1;  // how to move on
}
```

There are three ingredients here, and they are the same three every single time you count something. You set the variable to a starting value. You test it against a stopping value. And somewhere inside the block, usually right at the end, you move it along.

The people who designed C noticed that they were writing those three lines over and over again. Engineers are productively lazy: when you find yourself typing the same shape of thing repeatedly, you invent a shorthand for it. So they added a piece of syntax that gathers the three ingredients into a single line.

Notice also that the three ingredients are scattered. The starting value is above the loop, the stopping test is in the loop header and the step is buried at the bottom of the block, possibly a long way down if the block is big. If you want to know how many times a loop runs, you have to look in three places. Forget the third step and the loop never ends.

The `for` statement

A `for` loop is written like this:

```
for (initialise; condition; step)
{
    // happens as long as
    // condition holds
}
```

You get the keyword `for`, then a pair of round brackets containing three things separated by semicolons, then a block in curly braces. The semicolons in the middle of a line look strange the first time you see them. They are there for the same reason semicolons are anywhere else in C: each of those three slots holds a statement, and this is what separates one statement from the next.

The figure below names the parts, using a loop that counts from 1 to 10.

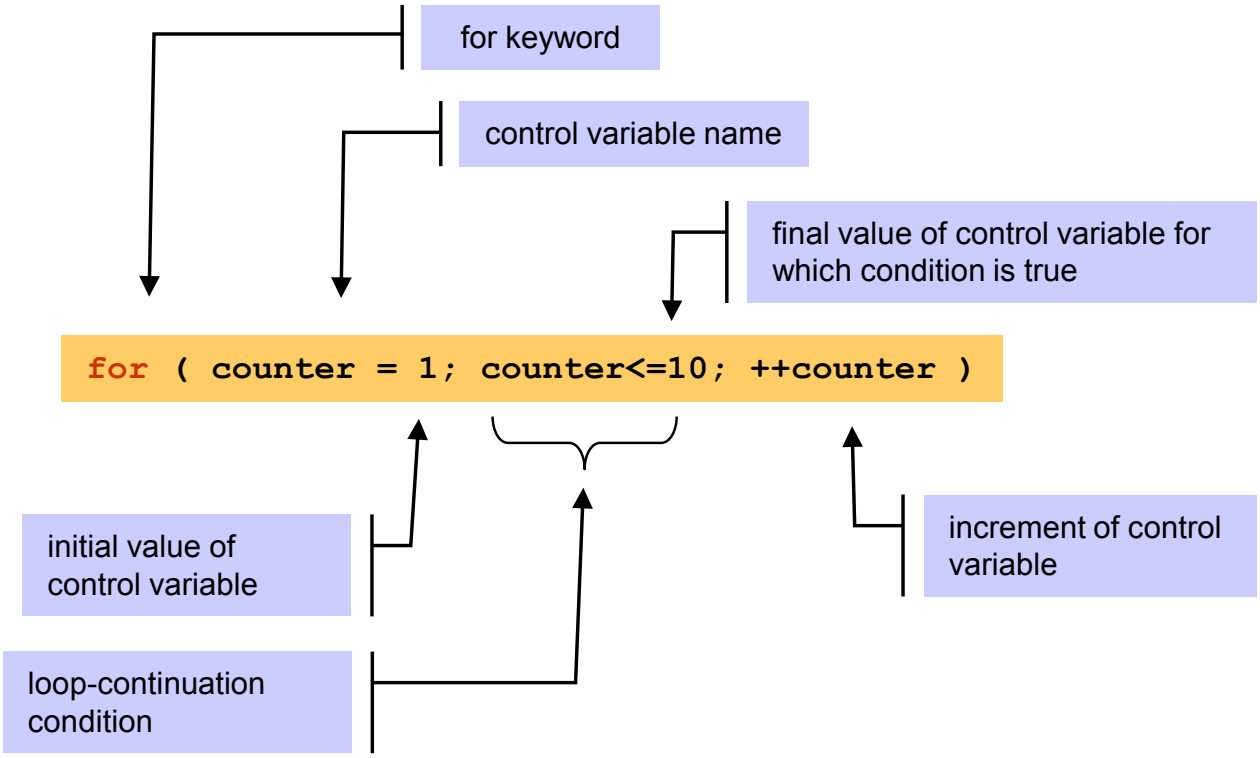

Figure 1: The parts of a `for` loop header. The three slots are the initialisation, the loop-continuation condition and the increment, separated by semicolons.

The order things happen in

This is the part that takes a moment to sink in, so it is worth going through slowly. The three slots do not run in the order they are written. They run in this order:

1. The initialisation runs, once, before anything else. This is the only time it ever runs.

2. The condition is tested.
3. If the condition is true, the block runs.
4. When the block finishes, the step runs.
5. Back to the condition, and round again from step 3.

If at any point the condition is false, we jump straight out past the closing brace and carry on with the rest of the program. Note that this can happen on the very first test: if the condition is false to begin with, the block never runs at all.

Drawing the jumps onto the code makes the shape clearer:

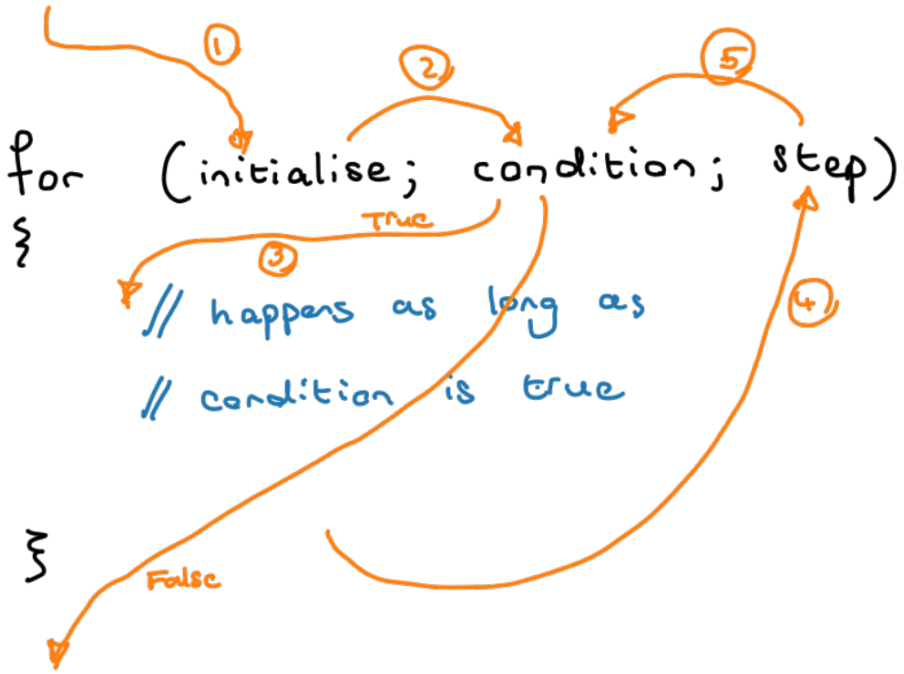

Figure 2: The order of execution of a `for` loop. The initialisation happens once, then we cycle between the condition, the block and the step until the condition is false.

The two arrows that surprise people are number 4 and number 5. After the block finishes, control does not go back to the condition directly: it goes to the step first, and only then to the condition. And the step never runs before the first pass through the block, because it comes after the block, not before it.

The same thing as a flow diagram, for the loop that counts from 1 to 10:

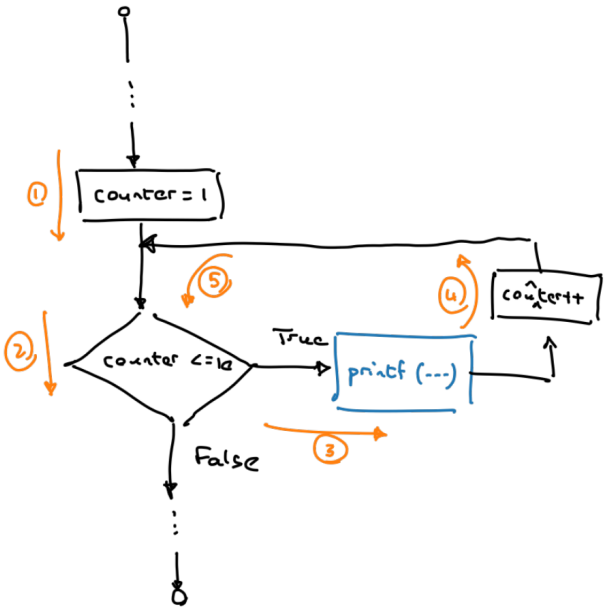

Figure 3: A `for` loop drawn as a flow diagram. Compare this with the `while` loop diagram from the previous lecture: the initialisation and the increment are now part of the loop statement itself. The numbers match the steps in Figure 2.

Rewriting a `while` loop as a `for` loop

Let us take the counting `while` loop from earlier and write it the new way. We want to count from 1 to 10, so 1 is where we start, `counter <= 10` is the condition, and adding one is the step:

```
for (counter = 1; counter <= 10; counter++)
{
    printf("%d\n", counter);
}
```

Trace it once by hand. The initialisation puts 1 in the `counter` box. The condition asks whether 1 is less than or equal to 10; it is, so we go into the block and print 1. The block ends, so we do the step: `counter++` opens the box, takes out the 1, adds one and puts 2 back. Now back to the condition: is 2 less than or equal to 10? Yes, so into the block and print 2. And so on.

The interesting pass is the last one. When `counter` holds 10, the condition asks whether 10 is less than or equal to 10. It is, so we print 10. Then the step makes `counter` 11, the condition asks whether 11 is less than or equal to 10, and this time the answer is no. We jump out. The output is:

```
1
2
3
4
5
```

```
6
7
8
9
10
```

Compare the two versions side by side. The `while` version:

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

and the `for` version:

```
for (counter = 1; counter <= 10; counter++)
{
    printf("%d\n", counter);
}
```

These do exactly the same thing. The `for` version is one line shorter, but the real gain is that all three ingredients now sit next to each other in the header. You can read off "starts at 1, runs while it is at most 10, goes up by one" without looking anywhere else.

One thing has not changed: you still have to declare the variable: `counter` is an ordinary `int` and needs an `int counter;` somewhere above, exactly as before.

Changing where the count starts is now a one-character edit. This:

```
for (counter = 5; counter <= 10; counter++)
{
    printf("%d\n", counter);
}
```

prints 5, 6, 7, 8, 9, 10.

More than one thing in each slot

Each of the three slots in the `for` header can hold more than one statement, separated by commas. Here two variables are initialised and two are stepped:

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

This is worth tracing carefully, because it exercises everything in the order-of-execution list. Assume `i` and `j` have been declared as `int` above. Somewhere in memory there are now two

boxes, `i` and `j`, each holding whatever junk was there before. The initialisation runs once and puts 0 in both boxes. The condition asks whether `i + j` is at most 10; it is 0, so we go in and print 0. The block ends and the step runs: `j++` makes `j` go to 1, and `i++` makes `i` go to 1. Back to the condition: `i + j` is now 2, still at less than 10, so we print 2.

The full trace looks like this:

`i`	`j`	`i + j`	Condition	Printed
0	0	0	true	0
1	1	2	true	2
2	2	4	true	4
3	3	6	true	6
4	4	8	true	8
5	5	10	true	10
6	6	12	false	–

So the output is 0, 2, 4, 6, 8, 10, and the loop stops when both boxes hold 6. If you can follow that trace, you understand `for` loops. If you cannot, go back to the order-of-execution list and walk through it again with a pen: the trace above is not something to memorise, it is something you should be able to produce yourself.

Any statement will do

The three slots are not restricted to simple assignments and comparisons. Any expression is allowed, including arithmetic on other variables. Suppose `x` holds 2 and `y` holds 10:

```
x = 2;
y = 10;

for (j = x; j <= 4*x*y; j += y/x)
```

Work out the three slots: `j` starts at 2, the condition is `j <= 80` because `4*2*10` is 80, and the step adds `10/2`, which is 5. So this header is exactly equivalent to:

```
for (j = 2; j <= 80; j += 5)
```

That is a slightly silly example, but the underlying point is useful: the loop bounds can be computed from variables rather than written as literal numbers. A loop that runs over however many readings the user asked for is a `for` loop whose condition mentions a variable.

The step slot is where you will actually use this freedom. Recall the assignment operators from the previous lecture: `j += 5` is shorthand for `j = j + 5`, and `j -= 2` is shorthand for `j = j - 2`. Combined with `++` and `--`, these give you all the counting patterns you need.

The table below shows some more examples, illustrating common counting patterns. Each row gives what you want the control variable to do and the header that does it:

What you want	Header
1 to 100, up by 1	`for (i = 1; i <= 100; i++)`
100 down to 1, by 1	`for (i = 100; i >= 1; i--)`
7 to 77, up by 7	`for (i = 7; i <= 77; i += 7)`
20 down to 2, by 2	`for (i = 20; i >= 2; i -= 2)`
2, 5, 8, 11, 14, 17	`for (i = 2; i <= 17; i += 3)`
44, 33, 22, 11, 0	`for (i = 44; i >= 0; i -= 11)`

A negative step is perfectly ordinary. All that changes is that the condition has to point the other way: with `i--` you need `i >= 1` rather than `i <= 100`, otherwise the condition is false on the very first test and the block never runs.

Design example: compound interest

Here is a problem worth solving properly.

> A person invests R1000 in a savings account yielding 10% interest annually. Assuming that all interest is left on deposit in the account, calculate and display the amount of money in the account at the end of each year for ten years.

The reason this is a good example is that each year's amount depends on the previous year's. You earn interest on your original deposit, and then next year you earn interest on the deposit plus last year's interest, and so on. That compounding is the whole point of investing early, and it is exactly the kind of thing a loop is good for.

There is a closed-form formula for this:

$$a = p(1 + r)^n$$

where p is the original amount invested, r is the annual interest rate, n is the number of years and a is the amount at the end of the nth year. We are not going to use it. With a loop we can just carry the running balance forward year by year, which is closer to what is actually happening in the account.

Work out the answer before you touch the keyboard

The temptation at this point is to start typing. Resist it. Take a pen and write down the first three lines that the program should print, using $p = 1000$ and $r = 0.1$.

At the end of year one you have your original R1000 plus 10% of R1000, so R1100. At the end of year two you have that R1100 plus 10% of R1100, so R1210. At the end of year three you have R1210 plus 10% of R1210, so R1331:

```
Year 1: 1000 + 1000 x 0.1 = 1100
Year 2: 1100 + 1100 x 0.1 = 1210
Year 3: 1210 + 1210 x 0.1 = 1331
...
```

Doing this by hand takes about a minute and it is a minute well spent, for two reasons. First, you now know what correct output looks like, so you will notice immediately if the program is wrong. Second, you have discovered the algorithm as a side effect: every line is the previous amount plus that amount times the interest rate. That is the calculation the loop has to do.

Notice, too, why we want a program at all. Ten years is tedious on a calculator. Thirty years is unbearable.

The pseudocode

Writing down what the algorithm that we just worked out a bit more explicitly:

```
Initialise the investment to 1000
Initialise the interest rate to 0.1
Set the current amount to the initial investment

For years 1 to 10 in increments of 1:
    Calculate the amount at the end of the year
    Display the year and the amount
```

Building it up

Start with the loop skeleton and nothing else. Get the counting right before you worry about the arithmetic:

```c
int year;

for (year = 1; year <= 10; year++)
{
    printf("%d\n", year);
}
```

Run it, confirm that it prints 1 to 10, and now you know the loop is working. The `printf` here is temporary scaffolding: it exists so you can see the loop working, and it will be replaced in a moment.

Next, the money. We need a variable holding the current balance, set to the initial investment before the loop starts, because year one's calculation needs last year's amount and before the loop there is no last year:

```c
double initialInvestment = 1000.0;
double interest = 0.1;
double amount;

amount = initialInvestment;
```

Then, inside the loop, each pass updates the balance:

```
amount = amount + amount*interest;
```

Read that from the right. Open the `amount` box and look at what is inside. Add to it that same value times the interest rate. Store the result back in the `amount` box. Next time round the loop, the box already holds the new amount, which is precisely how compounding interest works.

Putting it together:

```
/*
 * Calculate annual investments.
 */

#include <stdio.h>

int main()
{
    // Variables
    double initialInvestment = 1000.0;  // p
    double interest = 0.1;              // r
    int year;                           // n
    double amount;                      // a

    amount = initialInvestment;

    for (year = 1; year <= 10; year++)
    {
        // Calculate the amount at the end of the year
        amount = amount + amount*interest;

        // Display the year and the amount
        printf("Year %d: %f\n", year, amount);
    }

    return 0;
}
```

The first three lines it prints are 1100.000000, 1210.000000 and 1331.000000, which is what we worked out by hand. The program is correct.

We used `double` rather than `float` here. A `double` holds more digits than a `float`, and for money that is the safer default, though for an amount this size a `float` would have been fine too.

Making the output presentable

Six decimal places on a bank balance is silly, and the columns do not line up. Both are fixed with the formatting from the previous lecture. Adding a precision of two and a field width of 21:

```
printf("%4s%21s\n", "Year", "Amount of deposit");

for (year = 1; year <= 10; year++)
{
    amount = amount + amount*interest;
    printf("%4d%21.2f\n", year, amount);
}
```

The `%4d` prints the year right-aligned in a field four characters wide, and `%21.2f` prints the amount to two decimal places, right-aligned in a field twenty-one characters wide. Because every number is padded to the same width, the decimal points line up automatically. The header uses `%4s` and `%21s` with the same widths so that the column titles sit above their columns:

```
Year    Amount of deposit
   1              1100.00
   2              1210.00
   3              1331.00
   4              1464.10
   5              1610.51
   6              1771.56
   7              1948.72
   8              2143.59
   9              2357.95
  10              2593.74
```

R1000 becomes R2593.74 in ten years without you doing anything at all. Change the 10 in the condition to 30 and see what happens over a career.

Which loop should you use?

Both loops can solve both kinds of problem, so this is a question of style rather than of capability. The guideline is simple:

- Use a `for` loop when you know before you start how many times you want to go round. Counting from 1 to 10, working through ten years of interest, processing a fixed number of readings.
- Use a `while` loop when you do not. Reading grades until the user enters the sentinel, waiting for a measurement to settle, retrying until something succeeds.

The distinction has names: counter-controlled repetition when the count is known in advance, and sentinel-controlled repetition when it is not. Writing `for (year = 1; year <= 10; year++)` tells a reader immediately that this runs ten times, and writing `while (grade != -1)` tells

them the user decides. Choosing the loop that matches your intent is a message to whoever reads the code next.

Looking forward

The `for` loop completes the basic toolkit. You now have sequence, selection with `if` and `else`, and repetition in two forms. With these you can express any algorithm.

The next lecture adds one more loop, `do...while`, which differs from the two we have in a small but important way: it tests its condition at the bottom rather than the top, so the block always runs at least once. That turns out to be exactly what you want when you are asking a user for input and refusing to accept nonsense.

Exercises

1. Write a `for` loop that displays the multiples of 5 from 5 to 100, and another that counts backwards from 20 to 1.
2. Without running it, work out what each of the following prints, then check by running it:
   (a) `for (i = 1; i < 10; i++)`
   (b) `for (i = 0; i <= 10; i += 3)`
   (c) `for (i = 10; i > 0; i -= 4)` with `printf("%d\n", i);` as the block in each case.
3. How many times does the block run in `for (i = 0; i < 10; i++)`, and how many times in `for (i = 1; i < 10; i++)`? Explain the difference in one sentence.
4. What does `for (i = 10; i <= 1; i++)` print? Explain why.
5. Rewrite the following `while` loop as a `for` loop, and confirm that both produce the same output:

```
int n = 100;
while (n >= 0)
{
    printf("%d\n", n);
    n = n - 7;
}
```

6. Write a program that calculates the sum of the integers from 1 to 100 using a `for` loop.
7. Trace the following loop by hand, filling in a table like the one earlier in this lecture, then check your answer by running it:

```
for (i = 1, j = 10; i < j; i++, j--)
{
    printf("%d %d\n", i, j);
}
```

8. The compound interest program calculates the balance by carrying it forward year by year. Rewrite it to use the formula $a = p(1 + r)^n$ instead, with the `pow` function. You will need `#include <math.h>` at the top, and `pow(1.0 + interest, year)` computes $(1 + r)^n$. Do the two versions agree to two decimal places? Should they?

9. What happens if you leave the step slot empty, as in `for (i = 1; i <= 10; )`? What about leaving the condition empty, as in `for (i = 1; ; i++)`? Try both to see what happens.