4. Our first loops: the `while` loop

Herman Kamper

Every program we have written so far executes each of its statements exactly once. Sequential execution carries us from the top of `main` to the bottom, and the `if` statement lets us skip over parts of the way. What we cannot yet do is go backwards: repeat a piece of code over and over until something tells us to stop. That is what this lecture is about.

Repetition is the third and final kind of control structure. Once you have it, together with sequence and selection, you can write almost any program. (It might not look pretty, but you could get it to work.) This lecture introduces the simplest of C's loops, the `while` loop.

Why we need loops

Imagine a small robot car. It comes with a library, `robot.h`, that gives you exactly two commands: `turnFiveDeg()`, which turns the car five degrees, and `moveFwdTencm()`, which drives it forward ten centimetres. That is all you get.

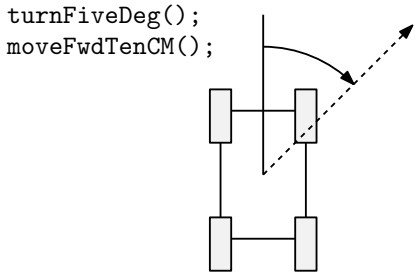

Figure 1: The robot car, with the two commands it understands. The task: turn 45 degrees, then drive 50 cm in the new direction.

Now suppose you want the car to turn 45 degrees and then drive forward 50 cm. With only those two commands, and only the tools we have covered so far, there is exactly one thing you can do:

```c
#include <robot.h>

int main()
{
    // Turn 45 degrees and move forward 50 cm
    turnFiveDeg();
    turnFiveDeg();
    turnFiveDeg();
    turnFiveDeg();
    turnFiveDeg();
    turnFiveDeg();
    turnFiveDeg();
    turnFiveDeg();
    turnFiveDeg();

    moveFwdTencm();
    moveFwdTencm();
    moveFwdTencm();
    moveFwdTencm();
    moveFwdTencm();

    return 0;
}
```

Nine turns and five forward steps. It works. And it is horrible. If you wanted a 90-degree turn you would have to count out eighteen calls; if you wanted the car to drive ten metres you would need a hundred. Copying a line a hundred times is tedious, easy to get wrong, and impossible to change later.

What we want to say instead is: "Do this thing, again and again, as long as some condition holds." That is a loop.

Where loops fit in

Recall the three kinds of control structure from which every program can be built:

- *Sequence*: statements executed one after another, in the order written. The nine `turnFiveDeg()` calls above are pure sequence.
- *Selection*: choosing whether or not to run a block of code, based on a condition. This is the `if` and `if...else` statements from the previous lecture.
- *Repetition*: running a block of code many times over. This is a loop.

C provides three different repetition statements. This lecture covers the first and most fundamental one, `while`; the others come later. Remember also that a loop, like everything else here, is a feature of the *program*, not the algorithm. When you plan a solution as pseudocode or as a flow diagram, you write down "repeat this until that". When you translate that step from pseudocode into actual C, then the `while` loop is the syntax you will use.

The `while` statement

The syntax is about as simple as it could be:

```
while (condition)
{
    // keep on doing this as long as the condition is true
}
```

The behaviour is as follows. When execution reaches the `while`, the condition inside the parentheses is evaluated. If it is true, the whole block inside the curly braces runs, top to bottom. When the block finishes, execution jumps back up to the condition and tests it again. If it is still true, the block runs again. This continues until, at some point, the test comes out false, at which point execution jumps past the closing brace and carries on with the rest of the program.

Two things follow from this that are worth stating explicitly. First, the condition is checked *before* each pass through the block, including the very first. If the condition is false at the outset, the block never runs at all. Second, nothing about the loop itself makes the condition eventually become false: that has to be arranged by the code inside the block. If nothing in the block ever changes the outcome of the test, the loop runs forever.

Here is a small example. Starting from `product` equal to 2, we keep doubling it for as long as it stays at or below 200:

```
int product = 2;

while (product <= 200)
{
    product = 2 * product;
}
```

The values taken by `product` are 2, 4, 8, 16, 32, 64, 128 and then 256. At that point `256 <= 200` is false and the loop stops. As a flow diagram, the loop looks like this:

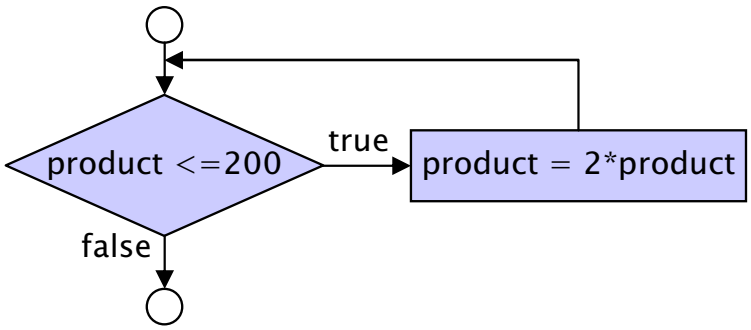

Figure 2: Flow diagram of a `while` loop. While the condition is true, the block runs and execution loops back to the test; when the condition is false, execution continues past the loop.

Notice the arrow that goes back up. That is the whole point of a loop, and it is the one thing that no diagram we have drawn before has had.

Counting with a loop

Very often the thing being tested is a variable that we are using to count passes through the loop. Such a variable is called a *counter*, and the pattern looks like this:

```
int counter = 1;

while (counter <= 5)
{
    counter = counter + 1;
}
```

Its flow diagram has exactly the same shape as before, with the counter test in the diamond and the counter update in the box:

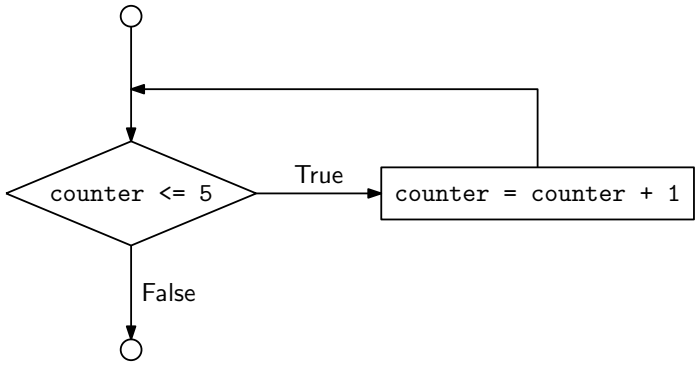

Figure 3: A counting loop as a flow diagram. The counter is tested, incremented, and tested again, until it exceeds 5.

Trace it. The counter starts at 1. The test `1 <= 5` is true, so we go into the block and the counter becomes 2. Back to the test: `2 <= 5` is true, so the counter becomes 3. And so on. Eventually the counter reaches 5; `5 <= 5` is still true, so the block runs once more and the counter becomes 6. Now `6 <= 5` is false, and the loop ends. The block ran five times, once for each of the counter values 1 through 5.

Three separate things had to happen for that to work, and they are worth naming because they appear in almost every loop you will write:

- *Initialisation*: the counter is given a starting value before the loop begins.
- *Condition*: the test that decides whether another pass happens.
- *Update*: something inside the block that changes the counter, so that the condition eventually becomes false.

Leave out the update and you get the classic beginner's bug:

```
int counter = 1;

while (counter <= 5)
{
    printf("%d\n", counter);    // nothing ever changes counter
}
```

Since `counter` stays at 1 forever, the condition is always true and the program prints 1 until you kill it. This is an *infinite loop*. Whenever you write a loop, ask yourself the question: what is it that will eventually make this condition false? If you cannot answer, you have a bug.

Counter-controlled repetition

Let us use this on a real problem.

> A class of ten students took a test. The grades, integers in the range 0 to 100, are available to you. Determine the class average for the test.

Ten students, so the program should ask for ten grades, add them up, divide by ten and print the answer. The number of repetitions is known in advance, before the loop ever starts, which is what makes this *counter-controlled repetition*.

As pseudocode:

```
set total to 0
set counter to 1

while counter is less than or equal to 10:
    ask the user for a grade
    read in the grade
    add the grade to the total
    add 1 to the counter

set the average to the total divided by 10
print the average
```

Two variables are doing the accounting here. The counter tracks how many grades we have dealt with, and controls the loop. The `total` accumulates the grades as they come in: it starts at zero, and each pass through the loop adds one more grade to it. A variable used this way is called an *accumulator*, and setting it to zero before the loop is essential. Leave that out and you will be adding grades to whatever rubbish happened to be sitting in that piece of memory.

Rather than writing the whole thing at once, it is worth building it up in pieces and running it after each change. Start with just the reading:

```
printf("Please enter the grade: ");
scanf("%d", &grade);
```

Compile and run that, check it does what you expect, and only then wrap it in a loop. Then add the counter. Then add the total. Programmers who write forty lines and only then hit compile spend a long time working out which of the forty is wrong.

Here is the finished program:

```
/*
 * Compute the average grade for a class of ten students.
 */
#include <stdio.h>

int main()
{
    int grade;
    int average;
    int total;
    int counter;

    total = 0;      // no grades added yet
    counter = 1;    // about to deal with the first student

    while (counter <= 10)
    {
        printf("Please enter the grade: ");
        scanf("%d", &grade);

        total = total + grade;
        counter = counter + 1;
    }

    average = total / (counter - 1);

    printf("Class average: %d\n", average);

    return 0;
}
```

The division at the end deserves a comment. We could simply have written `total / 10`, since we know there are ten students. Dividing by `counter - 1` says the same thing but says it in terms of how many grades were actually read, which means that if you later change the 10 in the condition to 15, the average still comes out right. The `- 1` is there because of where the counter finishes: after the last pass it is incremented one final time before the test fails, so it ends at 11, not 10.

If you are not yet convinced of that, trace the program by hand. Take a piece of paper, draw a box for each variable, and step through the statements one at a time, rubbing out and rewriting the value in a box whenever an assignment changes it. Suppose the first two students scored 55

and 34:

Step	`counter`	`grade`	`total`
after initialisation	1	–	0
test `1 <= 10`: true, enter block	1	–	0
read first grade	1	55	0
add to total, increment counter	2	55	55
test `2 <= 10`: true, enter block	2	55	55
read second grade	2	34	55
add to total, increment counter	3	34	89
…	…	…	…
after tenth pass	11	–	sum of all ten
test `11 <= 10`: false, leave loop	11	–	sum of all ten

The one row that surprises people is the last: the counter really does reach 11. The loop stops only *after* the test has failed, and the test can only fail once the counter has gone past 10.[1]

One more habit worth picking up. If a program is misbehaving and you cannot see why, put a temporary `printf` inside the loop that prints the values you care about:

```
printf("Total so far: %d\n", total);
```

Run it, watch the numbers, delete the line once you have found the problem. This is how a great deal of real debugging gets done.

There is one flaw in this program that we have quietly ignored. Both `total` and `average` are integers, so `total / (counter - 1)` is an integer division and the fractional part is simply thrown away: a true average of 82.7 is reported as 82. We will fix that in the next program.

Sentinel-controlled repetition

Now change the problem slightly:

> Develop a class averaging program that will process an arbitrary number of grades each time the program is run.

The difference is small to state, but large in consequence. We no longer know how many grades there will be. It might be 15 students, it might be 300. The condition `counter <= 10` is useless to us, because there is no number to put in place of the 10.

One reasonable fix is to ask the user upfront how many grades they are going to enter, and then use that number in the condition. That works. But there is a neater solution: let the user signal the end of the input by entering a special value.

[1] Stepping through code like this by hand is a skill worth practising until it feels mechanical. If you would rather watch it happen, tools such as Python Tutor will execute a short C program one line at a time and draw the variables for you.

Such a value is called a *sentinel*: it stands guard at the end of the data and stops the loop when it is reached. The sentinel has to be a value that could never be a genuine grade, otherwise the program could not tell the difference between real data and the end marker. Zero would be a poor choice, since in a class of 300 somebody will certainly score zero. A grade of -1, on the other hand, is impossible, which makes it a safe sentinel.

The algorithm now becomes:

```
set total to 0
set counter to 0

ask for a grade and read it in

while the grade is not -1:
    add the grade to the total
    add 1 to the counter
    ask for the next grade and read it in

if the counter is not 0:
    set the average to the total divided by the counter
    print the average
otherwise:
    report that no grades were entered
```

Look carefully at the order of the statements, because it is the one genuinely tricky part of this program. There are two `scanf` calls, not one. The first sits *before* the loop, and its job is to get a value into `grade` so that the condition has something to test the first time round. The second sits at the *end* of the block, so that the next grade is read just before the condition is checked again.

To see why this arrangement matters, imagine putting the `scanf` at the top of the block instead, with nothing before the loop. Then the sequence of events on the final pass is: read -1, add -1 to the total, increment the counter, and only then jump back and discover that the loop should have stopped. The sentinel gets counted as if it were a real grade, and the total is off by one and the count is off by one. Reading ahead of the test avoids this entirely: the value that ends the loop is never added to anything.

Here is the program:

```c
/*
 * Compute the average of an arbitrary number of grades,
 * terminated by a sentinel value of -1.
 */
#include <stdio.h>

int main()
{
    int grade;
    int counter;
    float total;
    float average;

    total = 0;      // no grades added yet
    counter = 0;    // no grades counted yet

    // Read the first grade, which may already be the sentinel
    printf("Enter grade, -1 to end: ");
    scanf("%d", &grade);

    while (grade != -1)
    {
        total = total + grade;
        counter = counter + 1;

        // Read the next grade, which may be the sentinel
        printf("Enter grade, -1 to end: ");
        scanf("%d", &grade);
    }

    if (counter == 0)
    {
        printf("No grades were entered.\n");
    }
    else
    {
        average = total / counter;
        printf("Average: %.2f\n", average);
    }

    return 0;
}
```

Two details in this program are worth drawing out.

The first is the counter, which now starts at 0 rather than 1 and is incremented only once a genuine grade has been accepted. When the loop ends it therefore holds exactly the number of grades entered, with no adjustment needed at the end.

The second is the type of `total`, which is now a `float` rather than an `int`. This is what fixes the truncation problem from the previous program. If `total` and `counter` are both integers, then `total / counter` is an integer division: entering grades of 15 and 20 gives a total of 35 and a count of 2, and $35/2$ comes out as 17, not 17.5. Declaring `total` as a `float` makes the division a floating-point one and the answer comes out right. The format specifier `%.2f` then prints the result to two decimal places.

Finally, notice the `if...else` around the calculation. It guards against a case that is easy to overlook: the user enters -1 immediately, without entering any grades at all. The loop body then never runs, the counter stays at 0, and computing `total / counter` would ask the machine to divide zero by zero. C does not stop you from doing this. What you get for such a floating-point division is a special value that prints as `nan`, short for "not a number", and it will propagate silently through the rest of your calculations. Checking for the empty case before dividing costs three lines but could save you a lot of pain down the line.

Common misconception

One common misunderstanding is worth pointing out now, because it is easy to pick up from the examples in this lecture and it can survive for a surprisingly long time. Many of the loops we have written tests a variable called `counter`, and it is tempting to conclude that `counter` is part of the C language: a special word that, placed in a `while` condition, makes the block repeat the right number of times. **It is not.** `counter` is a name we chose, and C has never heard of it.

You can prove this to yourself in a few seconds. Here is the counting loop from earlier, with a line added so that we can watch it run, and with every occurrence of `counter` replaced by a name that means nothing at all:

```
int hamburgers = 1;

while (hamburgers <= 5)
{
    printf("%d\n", hamburgers);
    hamburgers = hamburgers + 1;
}
```

This prints 1 to 5, exactly as the original did. The program does not run differently, or more slowly, or with a warning. Now try the same trick on `while`: rename it to `whilst` and the program will not compile at all. That is the difference. `while` and `int` are C's words, with fixed meanings that you cannot change. `counter` and `hamburgers` are yours, and they mean nothing to the compiler beyond "the box I named". Your editor will tell you as much if you look at it: it colours `while` and `int`, because it recognises them, and leaves `counter` plain, because

it does not. (If you look at this document in colour, you will also see this in the syntax printed highlighting.)

Related to this is the misconception that the loop itself is doing the counting. It is not. The `while` statement has exactly one job, the one described at the start of this lecture. Evaluate the condition, run the block if it is true, come back and test again. It has no notion of counting, no notion of ten, and no idea that there are a counter or students involved.

The same goes for every other variable in this lecture. `total`, `grade` and `average` describe our intentions, not the machine's. C sees three boxes, and the meaning lives entirely in your head and in the arithmetic you wrote around them. Choose names that say what you mean, because the next person to read the program will probably be you. Just never expect the name itself to do any of the work.

Looking forward

You now have all three control structures. You can execute statements in sequence, you can choose between alternatives with `if` and `if...else`, and you can repeat a block with `while`. That is not a small collection of tools: it is, in a precise sense, all of them. Any computation that can be described as an algorithm can be written using just these three things.

That does not mean you have nothing left to learn. Loops in particular come in more than one flavour, and the counting pattern we wrote out by hand – initialise, test, update – is so common that C provides a dedicated statement for it. A future lecture introduces the `for` loop, which packs all three parts into a single line, and looks at what happens when you put one loop inside another. Before we get there, though, we will look a little more closely at how decisions (`if`) and loops (`while`) can be nested.

Exercises

1. Explain in your own words why the condition of a `while` loop is tested before the block runs rather than after. Give an example of a loop whose block never runs at all.

2. What does each of the following loops print? Trace them by hand before running them.

   (a) `int i = 1; while (i < 4) { printf("%d ", i); i = i + 1; }`
   (b) `int i = 1; while (i <= 4) { printf("%d ", i); i = i + 1; }`
   (c) `int i = 0; while (i < 10) { printf("%d ", i); i = i + 3; }`

3. Write a program that displays the even numbers from 2 to 20. Do this in two different ways: once by counting in steps of two, and once by counting in steps of one and using an `if` statement inside the loop.

4. Write a program that calculates the sum of the integers from 1 to 100 and prints the result. Check your answer: it should be 5050.

5. Write a program that reads numbers from the user until the user enters 0, and then reports how many numbers were entered and what the largest one was.

6. Take the counter-controlled averaging program and rewrite it so that it first asks the user how many students are in the class, and then reads that many grades. What has to change in the loop condition, and what has to change at the end when the average is computed?

7. Modify the sentinel-controlled averaging program so that it rejects invalid grades: if the user enters a number that is neither in the range 0 to 100 nor equal to -1, the program should print a warning and ask again, without counting the bad value.