5. Putting it together: nested control structures

Herman Kamper

This lecture is a bit of a mix. Almost everything in it builds on something we have already seen, and the main idea is to combine those pieces rather than to introduce something entirely new. The main highlight is that we are going to put an `if` statement inside a `while` loop.

That sounds like a small step, and syntactically it is. But it is worth pausing on what it buys us. Once you can execute statements in sequence, make a decision and repeat a block of code, you can write a program for any algorithm. It might not be an elegant program, but it will work. Everything else that follows in the course, and in C generally, is about writing programs that are shorter, clearer and easier to maintain – not about making new things possible.

Alongside that main idea, this lecture picks up a few pieces of C syntax that will make your programs shorter: the assignment operators, the increment and decrement operators, and a closer look at how `printf` formats what it prints.

What we have so far

Two control structures have carried us this far. The first is the `if...else` statement, which chooses between two blocks of code:

```
if (condition)
{
    // this happens when the condition is true
}
else
{
    // this happens when the condition is false
}
```

The second is the `while` loop, which repeats a block of code:

```
while (condition)
{
    // this repeats as long as the condition is true
}
```

Remember how the loop runs: we arrive at the `while`, check the condition, and if it is true we execute the block once and pop back up to the condition. Check again, run again, check again. As soon as the condition is false we skip past the block and carry on with the rest of the program.

There is nothing stopping us from putting one of these inside the other. The block of a `while` loop can contain any statements at all, and an `if` statement is just a statement. When one control structure sits inside another like this, we call them *nested control structures*, and they are the subject of this lecture.

Counting passes and fails

Here is the problem we are going to solve:

> Develop a program that will count and display the number of students that have passed and the number of students that have failed from a list of exam results for 10 students. If more than 8 students have passed, display "Bonus to instructor!"

Before doing anything else, be sure you understand what is being asked. There is a class of ten students. We loop through the ten students, and for each one we ask a question: what is the student's grade? Based on this, whe ask: did this student pass or did this student fail? The loop controls how many students we process; the decision controls what we do with each one.

Now, what do you do first? You do not open your editor and start typing. You take a piece of paper or a tablet and you scribble down the steps you are going to take – you write the algorithm as pseudocode, or you draw a flow diagram. Once that is done, translating it into C is the easy part.

Refining the pseudocode

Start at the top. The whole program in one line is:

```
Analyse exam results and decide if the instructor should receive a bonus
```

That is pseudocode, but it is useless pseudocode – it just restates the problem. So we break it down into slightly finer steps. This is called *refinement*, and you keep doing it until each line is small enough to write directly in C:

```
Initialise variables
Input the 10 exam grades and count passes and failures
Display a summary of the exam results
Decide if the instructor should receive a bonus
```

Better, but the second line is still doing far too much. Let us refine that one line further. We know we need to loop through ten students, so a counter is involved:

```
While the counter is less than 10
    Ask for the student's grade
    If the grade is greater than or equal to 50
        Add one to the passes
    Otherwise
        Add one to the fails
    Add one to the counter
```

Read that again and check it against the problem. While the counter is less than ten, ask for a grade; if the grade is at least 50, add one to the passes, otherwise add one to the fails; and add one to the counter regardless of what happened. That is the whole algorithm, and it took no C at all to work out. Now you just have to be clever enough to type it in using correct C syntax.

A habit worth adopting: write the refined pseudocode straight into your source file as comments, using `//`. The structure is then already there, and you fill in the C underneath each line.

Building the program one piece at a time

We know from the pseudocode which variables we need. There are the two things we are counting, the counter that controls the loop, and somewhere to put whatever the user types:

```
int passes = 0;
int fails = 0;
int counter = 0;
int grade;
```

Both `passes` and `fails` start at zero, because at the start nobody has passed and nobody has failed. Forgetting to initialise a counter is one of the most common bugs in this kind of program: if you leave out the `= 0`, the variable starts with whatever rubbish happens to be in that box in memory, and your final counts will be nonsense.

Next comes the loop and the input inside it:

```
while (counter < 10)
{
    printf("Student's grade: ");
    scanf("%d", &grade);
}
```

At this point, before adding anything else, compile and run it. Not because you expect it to be finished, but to check that you have not missed a semicolon or misspelled something. And in fact, running it now teaches us something: it asks for a grade over and over and never stops. Of course it does. Nothing in the loop ever changes `counter`, so `counter` is zero forever, and zero is always less than ten. That is an infinite loop, and we will fix it in a moment.

Now the decision. Inside the loop, after reading the grade:

```
if (grade >= 50)
{
    passes = passes + 1;
}
else
{
    fails = fails + 1;
}
```

Note the >= rather than >. A student who gets exactly 50 also passes, and writing `grade > 50` would quietly fail them. Boundary conditions like this are worth checking twice.

Also note what `passes = passes + 1;` actually does. It opens the box in memory called `passes`, takes out what is currently inside, adds one, and puts the result back in the box. If `passes` held 0, it now holds 1.

Run it again. It compiles, it asks for grades, and it does not appear to do anything at all – because the counting is happening silently inside variables we never print. So how do you know the decision is even working? A trick that is worth using constantly: temporarily print something.

```
if (grade >= 50)
{
    printf(":)\n");         // temporary, just to check
    passes = passes + 1;
}
else
{
    printf(":(\n");         // temporary, just to check
    fails = fails + 1;
}
```

Now type in 101, then 51, then 49, and watch the faces come out: smile, smile, frown. The decision works. Delete the two temporary lines and move on. This is not cheating – it is a good habit. Do not trust yourself to have written correct code; check it as you go, in small pieces.

The infinite loop is fixed by the last line of our pseudocode, which we have not yet written:

```
counter = counter + 1;
```

That line goes inside the loop, after the `if...else`, so it runs once per student no matter which branch was taken. Run it again and count ten grades in: this time the program stops.

Finally, the summary and the bonus:

```
printf("No. of passes: %d\n", passes);
printf("No. of fails: %d\n", fails);

if (passes > 8)
{
    printf("Bonus to instructor!\n");
}
```

Putting all of it together:

```c
/*
 * Count passes and fails in a class of ten students.
 */

#include <stdio.h>

int main()
{
    // Initialise variables
    int passes = 0;
    int fails = 0;
    int counter = 0;
    int grade;

    // While the counter is less than 10
    while (counter < 10)
    {
        // Ask for the student's grade
        printf("Student's grade: ");
        scanf("%d", &grade);

        // If the grade is greater than or equal to 50
        if (grade >= 50)
        {
            // Add one to the passes
            passes = passes + 1;
        }
        else
        {
            // Add one to the fails
            fails = fails + 1;
        }

        // Add one to the counter
        counter = counter + 1;
    }

    // Display a summary of the exam results
    printf("No. of passes: %d\n", passes);
    printf("No. of fails: %d\n", fails);

    // Decide if the instructor should receive a bonus
    if (passes > 8)
```

```
    {
        printf("Bonus to instructor!\n");
    }

    return 0;
}
```

One small warning from experience: it is easy to declare a variable under one name and then use another. If you declare `int input;` but write `scanf("%d", &grade);`, the compiler will stop with an error about `grade` being undeclared. That is the compiler doing you a favour, so read the error message rather than panicking at it.

Two `if` statements in two different places

The program has two `if` statements, and where each one sits is not an accident. To see what is happening as the program runs, suppose the first four grades entered are 65, 48, 71 and 30:

Student	Grade	`passes`	`fails`
1	65	1	0
2	48	1	1
3	71	2	1
4	30	2	2

Each pass through the loop does the same four things: read a grade, decide whether it is a pass, update the right counter, move on to the next student. The first `if` is inside the loop because it applies to every single student. The second `if`, the one about the bonus, is after the loop because it depends on the final result, once all ten grades are in. Putting that second test inside the loop would test a half-finished count.

This is a fairly long program by our standards, and it is still a simple one. But look at what it does. It reads input, makes a decision for each item, keeps running counts, and then makes a further decision based on those counts. That is the shape of an enormous number of real programs.

Assignment operators

The line `passes = passes + 1;` takes a variable, does something to it, and then stores the result back in the same variable. This type of instruction turns up so often that the designers of C got lazy and invented a shorthand for it. Instead of writing:

```
passes = passes + 1;
```

you can write:

```
passes += 1;
```

These two lines are exactly the same. The += operator takes what is in the box, adds whatever is on the right-hand side, and stores the result back in the box.

This works for any of the arithmetic operators, not just +. Writing `[]` for any one of `+`, `-`, `*`, `/` or `%`, the general rule is that:

```
var = var [] statement;
```

can always be rewritten as:

```
var []= statement;
```

The two mean exactly the same thing. Note where the operator moves to: it comes immediately before the =, with no space between them. So all of these pairs are equivalent:

Long form	Short form
`c = c + 3;`	`c += 3;`
`d = d - 4;`	`d -= 4;`
`e = e * 5;`	`e *= 5;`
`f = f / 3;`	`f /= 3;`
`g = g % 9;`	`g %= 9;`

The right-hand side does not have to be a single number. It can be any expression. So:

```
total = total + grade*100;
```

can be written as:

```
total += grade*100;
```

In our pass-and-fail program, `passes = passes + 1;` can be written as `passes += 1;`.

Increment and decrement operators

Adding exactly one to a variable is even more common than the general case, so C has a shorthand for that too. All three of these lines do the same thing:

```
j = j + 1;
j += 1;
j++;
```

The ++ operator opens the box, adds one, and stores the result back. Its partner `--` subtracts one:

```
j = j - 1;
j -= 1;
j--;
```

This is where the name of the language C++ comes from: it is C, one better.

Pre- and post-increment

Now we get to something genuinely awkward. The ++ can be written either before or after the variable, and the two are not always the same. Consider:

```
int j;

j = 7;

printf("current value of j: %d\n", j++);
printf("after statement: %d\n", j);
```

What does the first line print? Not 8. It prints 7, and then the second line prints 8. Here is what C does: it works through the whole `printf` statement using the current value of `j`, ignoring the ++ entirely, and only once the statement is finished does it go back and perform the increment. So `j` is still 7 when it gets printed, and it has become 8 by the time we reach the next line.

Writing the operator in front changes this. With `++j` the increment happens first, before the rest of the statement is evaluated:

```
int j;

j = 7;

printf("current value of j: %d\n", ++j);
printf("after statement: %d\n", j);
```

This prints 8 and then 8. The same distinction applies to `--`: with `j--` the statement runs first and then `j` drops by one, while with `--j` it drops by one first.

The technical terms are *post-increment* for `j++` and *pre-increment* for `++j`, with *post-decrement* and *pre-decrement* for the two forms of `--`. Summarised:

- `++a` increments `a` by 1, then uses the new value in the surrounding expression
- `a++` uses the current value of `a` in the surrounding expression, then increments it by 1
- `--b` decrements `b` by 1, then uses the new value in the surrounding expression
- `b--` uses the current value of `b` in the surrounding expression, then decrements it by 1

There is one important simplification. All of this only matters when the variable appears inside a larger expression. If the increment stands alone on its own line, as in `counter++;`, then `counter++;` and `++counter;` do exactly the same thing, and you can use whichever you prefer.

Honestly, mixing an increment into the middle of another statement is a habit worth avoiding in your own code – it makes programs harder to read for very little gain. But other people do write code like this, so you need to be able to read it and work out what it does.

Formatted output

The last topic for this lecture is a closer look at `printf`. You have met most of this already, in pieces, but it is worth gathering in one place.

The character after the % tells `printf` how to interpret the value you hand it. The table below shows the common ones: each row declares a variable, prints it with `printf` using the format shown, and gives the output.

Declaration	`printf` format	Output
`int myint = 45;`	`"%d"`	`45`
`float myfloat = 79.54;`	`"%f"`	`79.540001`
`float myfloat = 79.54;`	`"%e"`	`7.954000e+001`
`char mychar = 'a';`	`"%c"`	`a`
`char mychar = 'a';`	`"%d"`	`97`
`char mychar = 98;`	`"%c"`	`b`
`char mystring[] = "Hello";`	`"%s"`	`Hello`

Two of these rows deserve comment. The `%e` specifier prints in scientific notation, which is exactly what you want when your numbers are enormous or tiny. And notice that `%f` on the value 79.54 prints `79.540001` rather than `79.540000`: a `float` cannot represent 79.54 exactly, so what comes out is the closest value it can store. Single characters are written with single quotes, `'a'`, while strings use double quotes, `"Hello"`. Strings are a topic we will come back to properly later.

Why a character can print as a number

The strangest row in that table is the one where a `char` is printed with `%d` and comes out as 97. To understand it, think about what is actually stored in the box.

When you write `char mychar = 'a';`, the computer creates a box in memory called `mychar`. It does not put a letter in there – it cannot. All it can store is ones and zeros. The pattern it stores for `'a'` is `0110 0001`, and the pattern for `'b'` is `0110 0010`.

Now the two `printf` calls do different things with the same box. When you ask for `%c`, it takes those ones and zeros and interprets them as a character, so `a` appears on the screen. When you ask for `%d`, it takes the same ones and zeros and interprets them as an ordinary integer, which gives 97. There is nothing mysterious going on: 97 is simply the decimal value of the bit pattern `0110 0001`.

Character	Binary	Decimal
`a`	`0110 0001`	97
`b`	`0110 0010`	98

The mapping from characters to numbers is a standard called the American standard code for information interchange (ASCII), which we will look at later. You do not need to memorise the

codes. What you do need is the idea that everything in memory is ones and zeros, and that the format specifier decides how those ones and zeros are interpreted.

Field width

You can put a number between the `%` and the conversion character. For integers this is the *field width*: the minimum number of character positions the value should occupy.

```
printf("%4d\n", 1);
printf("%4d\n", 1234);
printf("%4d\n", 12345);
```

This prints:

```
   1
1234
12345
```

The first line pads the value out to four positions with three spaces in front. The second already fills four positions exactly. The third needs five, and since the field width is a minimum rather than a maximum, the value simply overruns it.

The point of this is alignment. Numbers of different lengths lined up in a column are far easier to read than ragged ones, and this is how you produce a neat table of output.

Precision

For floating-point values you can also specify the *precision*: the number of digits after the decimal point, written as a full stop followed by a number.

```
printf("%.3f\n", 3.8663);
printf("%9f\n", 3.8663);
printf("%9.3f\n", 3.8663);
printf("%9.3lf\n", 3.8663);
```

This prints:

```
3.866
 3.866300
    3.866
    3.866
```

The first line asks for three decimal places and gets them, rounded. The second asks for a field width of nine but no precision, so it falls back on the default of six decimal places, padded out to nine positions. The third combines the two: nine positions in total, three of them after the point. Count the characters and you will find exactly nine.

The last line uses `%lf`, which stands for long float, otherwise known as a `double`. For printing, `%lf` behaves exactly the same as `%f`. It exists mainly because `scanf` does need the distinction, and it is convenient to be able to write the same specifier in both `printf` and `scanf`.

One restriction: the field width and precision have to be written into the format string as literal numbers. You cannot put a variable there. Some other languages allow it, but C does not.

How far this can take you

It is worth ending with a demonstration of the claim made at the start. Below is the opening of a text adventure game – a small choose-your-own-adventure story set at Hogwarts, where you type a number at each step to pick what happens next, and you gain or lose house points along the way:

```
====================================
   A NIGHT AT HOGWARTS
====================================

It is well past midnight. The castle is asleep, the fire in
the Gryffindor common room has burned down to embers, and you
cannot sleep a wink.

At each step, type the number of your choice and press Enter.

------------------------------------
You are curled in an armchair by the dying fire.
The portrait hole is just there, in the shadows.

  1) Sneak out through the portrait hole
  2) Be sensible and go up to bed
```

The whole game is one big `while` loop. A variable called `scene` remembers where you are in the story. Each time round the loop, the program looks at `scene`, prints what is happening, reads the player's choice and works out the next scene. A few other variables remember things about you, such as how many house points you have and whether you have found the invisibility cloak. When the story reaches an ending, a variable called `playing` is set to zero and the loop stops.

There are no functions in it, no arrays, no `switch`, not even a logical "and" or "or". It is nothing but sequence, `if` and `while` statements – which is to say, nothing but what is in this lecture and the two before it.

Try and build your own adventure game like this!

Looking forward

You now have all three control structures. Sequence executes statements one after the other, selection chooses between blocks with `if` and `if...else`, and repetition runs a block many times with `while`. Nested inside one another, these are enough to express any algorithm you will ever need.

What comes next is not more power, but more convenience. The next lecture introduces other kinds of loop, starting with the `for` loop, which packages the initialisation, condition and update of a counter-controlled loop into a single line. After that come further selection statements. None of them let you do anything you cannot already do – they just let you say it more clearly.

If you ever lose track of what a program is doing to its variables, it is worth stepping through it one line at a time and watching the boxes in memory change.

Exercises

1. Write a program that reads ten integers and counts how many of them are positive.
2. Modify the pass-and-fail program from this lecture so that it also keeps a running total of the grades and prints the class average at the end. Be careful about integer division.
3. A factory inspects 20 products. Each one is either acceptable or defective, entered as 1 or 2. Write a program that counts the number of defective products.
4. What does each of the following print? Work it out on paper before running it.

   ```
   int k = 4;
   printf("%d\n", k--);
   printf("%d\n", k);
   printf("%d\n", --k);
   printf("%d\n", k);
   ```

5. Explain in your own words why `counter++;` and `++counter;` are interchangeable when they stand on a line of their own, but `printf("%d\n", counter++);` and `printf("%d\n", ++counter);` are not.
6. What does the following print, and why?

   ```
   char c = 'z';
   printf("%c\n", c);
   printf("%d\n", c);
   printf("%c\n", c + 1);
   ```

7. Write the `printf` statement that would print the value 12.3456 as `12.35`, i.e. rounded to two decimal places and padded to a total width of seven characters.

8. Take the following program and predict its output before running it.

```c
int n = 0;
int total = 0;

while (n < 5)
{
    if (n % 2 == 0)
    {
        total += n;
    }
    n++;
}

printf("%d\n", total);
```