8. Completing the control structures

Herman Kamper

This lecture finishes off program control. There is one more selection statement to meet, `switch`, and one small statement that goes with it, `break`. After that you will have seen every statement C gives you for directing the flow of a program: everything that comes later is about writing programs better and organising them more sensibly, not about new ways of deciding what happens next.

We start by tying off the logic from the previous lecture, including one confusion that catches almost everyone, and then spend most of the lecture on `switch` and `break`. At the end we put all the control structures side by side and look at what we have built.

Logic, one more time

C has no built-in words for true and false. It uses numbers instead, and the rule is short enough to memorise:

- `0` means false.
- `1` means true.
- In fact anything non-zero means true.

So a condition that evaluates to 0 fails and the block is skipped, and a condition that evaluates to anything else succeeds and the block runs. That is the whole story.

To build conditions out of smaller conditions there are three operators: `&&` for and, `||` for or, and `!` for not. There is also == for equality, which we return to in a moment. These can be combined as far as you like, as long as you say clearly with parentheses what belongs to what:

```
if ( ((x > 4) && (x <= 6)) || (x == 2) )
{
    // ...
}
```

You will probably never write anything quite this convoluted, but it is worth knowing that you can. The trick with reading it is not to read it left to right the way you read English. Read it the way you read mathematics, from the innermost parentheses outwards. First work out whether `x > 4`. Then whether `x <= 6`. Combine those two with `&&`. Separately, work out whether `x == 2`. Then combine those two results with `||`, and that answer is what the `if` acts on.

Drawing the values on a line makes it obvious what the condition means:

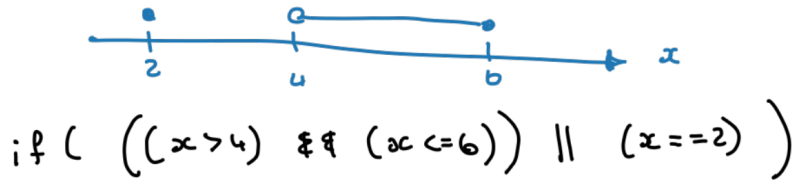

Figure 1: The values of x for which the combined condition is true: everything above 4 up to and including 6, together with the single value 2.

If you are comfortable with this, you are in good shape. Combining conditions like this is the basis of digital circuit design, which is the same idea implemented in hardware rather than in software.

Assignment is not comparison, one more time

At the end of the previous lecture we noted that = and == do completely different things. This is worth doing slowly, because it is the single easiest way to write a program that compiles perfectly, runs perfectly, and does the wrong thing.

The useful trick is that a condition is just an expression that evaluates to a number, and any number can be printed. So instead of arguing about what a condition means, print it:

```
int p = 7;

printf("Expression: %d\n", p == 7);
```

This displays:

```
Expression: 1
```

The expression `p == 7` asks whether the box called `p` contains a 7. It does, so the answer is true, and C says true by handing back a 1. Nothing in memory has changed.

Now try a few variations. The table below shows what each expression prints and what it leaves behind, all starting from `p` holding 7:

Expression	What is printed	What happens to `p`
`p == 7`	1	unchanged, still 7
`p == 0`	0	unchanged, still 7
`p == 2`	0	unchanged, still 7
`p = 2`	2	it becomes 2

The first three rows are comparisons. They look inside the box, report 1 or 0, and leave the contents alone. The last row is something else entirely. A single = is an instruction: open the box, throw away what is in there and put a 2 inside. The value of that whole expression is the value that was assigned, which is 2, so 2 is what gets printed. And `p` really has changed: print it again afterwards and you get 2.

Now put that last expression where a condition belongs:

```
if (p = 2)
{
    printf("p is equal to 2\n");
}
```

This is not asking anything. It stores 2 in `p`, and then the `if` is handed the value 2, which is non-zero, which is true. The message is displayed. It would also be displayed if you had written `p = 3`, or `p = -1`, or any other non-zero value, no matter what `p` contained before. The only way the block would be skipped is if you assigned 0.

Two equals signs check what is in the box. One equals sign puts something in the box. Get that clear now and you will avoid an entire category of bug.

Many values, one variable

Here is a small problem. Ask the user how many wheels their vehicle has, and then say what kind of vehicle it probably is: one wheel is a unicycle, two a bicycle, three a tricycle, four a car, and anything else we cannot identify.

You can already write this. It is a chain of `if` and `else if`:

```
if (noWheels == 1)
{
    printf("It's a unicycle\n");
}
else if (noWheels == 2)
{
    printf("It is a bicycle\n");
}
else if (noWheels == 3)
{
    printf("Tricycle\n");
}
else if (noWheels == 4)
{
    printf("Car!\n");
}
else
{
    printf("I have no idea what that is\n");
}
```

There is nothing wrong with this. It works, and for two or three options it is perfectly readable. But look at what it is actually doing: it is testing one variable against a list of specific values, and the variable's name has to be repeated in every single test. As the list grows this becomes

long, repetitive and easy to get wrong. Mistype one of those names and the compiler will not necessarily complain.

This particular shape of problem comes up so often that C has a statement designed for it.

The `switch` statement

The `switch` statement selects between many alternatives based on the value of a single variable or expression. Its shape is:

```
switch (value)
{
case a:
    action(s);
    break;
case b:
    action(s);
    break;
default:
    action(s);
}
```

There are three new keywords here. `switch` introduces the whole structure, and the thing in the parentheses is the value being examined, usually just a variable. `case` labels the alternatives: each one names a specific value that the variable might have. `break` says jump out of here. The `a` and `b` above are placeholders, in the same way that `action(s)` is: in a real program you write actual values in their place, and we will see what those can be in a moment.

The execution works differently from anything we have seen so far. When control reaches the `switch`, the value is worked out once. C then scans down the list of `case` labels looking for one that matches. If it finds one, it jumps to that point and starts executing from there, carrying on until it meets a `break`, at which point it jumps out of the whole `switch` and the program continues below. If no case matches, it jumps to `default` instead.

The flow diagram shows this, with the arrows on the right being the jumps out:

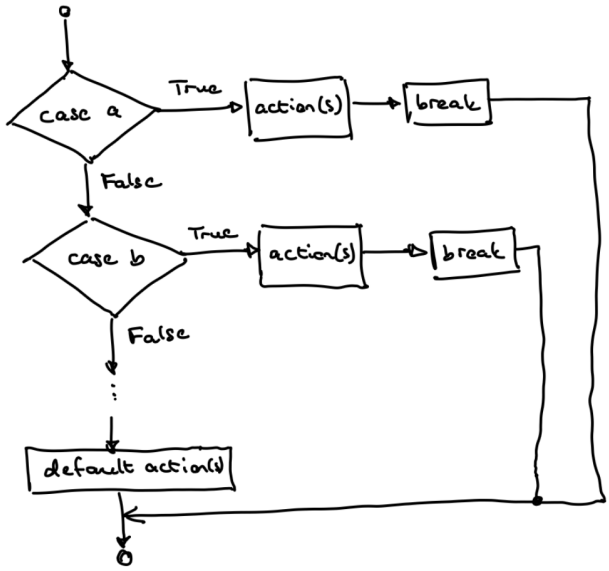

Figure 2: Flow of control through a `switch` statement. Each case is tested in turn; a match runs the actions for that case, and the `break` jumps out of the whole structure. If nothing matches, the default actions run.

Notice what the `break` arrows do. They all land at the same point, just past the bottom of the structure. That is the entire job of `break`: leave, immediately.

Here is the wheels program written with a `switch`:

```
#include <stdio.h>

int main()
{
    int noWheels;

    printf("How many wheels does your vehicle have? ");
    scanf("%d", &noWheels);

    switch (noWheels)
    {
    case 1:
        printf("It's a unicycle\n");
        break;
    case 2:
        printf("It is a bicycle\n");
        break;
    case 3:
        printf("Tricycle\n");
        break;
    case 4:
```

```
        printf("Car!\n");
        break;
    default:
        printf("I have no idea what that is\n");
    }

    return 0;
}
```

If you type 3, then 3 goes into the box called `noWheels`, the `switch` scans for a case matching 3, jumps to `case 3:`, prints, hits `break` and leaves:

```
How many wheels does your vehicle have? 3
Tricycle
```

Type something that is not in the list and the default runs instead:

```
How many wheels does your vehicle have? 7
I have no idea what that is
```

Two small points about the syntax. The `default` case is optional: leave it out and a value that matches nothing simply causes the whole `switch` to do nothing at all. Including it is usually the better choice, because unexpected input then gets noticed rather than silently ignored. The other point is that the values after `case` have to be constants that are known when the program is compiled. You cannot write `case x:` where `x` is a variable, and you cannot write a range or a comparison such as `case > 50:`. If your decision involves a range, you need an `if`.

Switching on characters

The cases do not have to be numbers. They can equally well be characters:

```
char noWheels;

printf("How many wheels does your vehicle have? ");
scanf(" %c", &noWheels);

switch (noWheels)
{
case 'a':
    printf("Unicycle\n");
    break;
case 'b':
    printf("Bicycle\n");
    break;
case 'c':
    printf("Tricycle\n");
    break;
}
```

Type a `b` and you get `Bicycle`. This is a nonsensical program, but it makes the point: the box is now a character box, the `%c` in the `scanf` reads a single character into it, and the `case` labels are characters written in single quotes. Nothing else about the `switch` changes.

The reason it works is something we saw when we first met characters. A `char` is stored as a number, so `'b'` is a constant in exactly the same way that 2 is a constant. The `switch` is still comparing numbers; we are just writing those numbers in a more readable form.

Reading characters with `scanf`. One detail is worth explaining, because it causes a lot of confusion the first time you read characters in a loop. Notice the space in `scanf(" %c", &noWheels)`. When you type a character and press enter, the enter key leaves a newline character sitting in the input waiting to be read. With `%d` this does not matter, because reading a number skips over whitespace automatically. With `%c` it matters a great deal: a newline is a perfectly good character, so the next `%c` will happily read it and you will appear to have skipped a turn. The space before the `%c` tells `scanf` to throw away any whitespace first. Put it in whenever you read single characters.

Cases that fall through

There is one edge case of the `switch` statement that looks like a mistake the first time you see it, but is not.

Suppose you are counting pass and fail results. The user types `P` or `F` for each student, but people are inconsistent about capitals, so you want lowercase `p` and `f` to work too. You could write four cases with duplicated code in each. Instead you can stack the labels:

```
switch (grade)
{
case 'F':
case 'f':
    failCount++;
    break;
case 'P':
case 'p':
    passCount++;
    break;
default:
    printf("Invalid character entered. ");
    printf("Enter a new result (P or F).\n");
}
```

Work through what happens when `grade` is a lowercase `f`. The `switch` scans, finds `case 'f':`, jumps there and runs from that point onwards until it sees a `break`. That means it runs `failCount++` and then leaves.

Now what happens when `grade` is a capital `F`? The `switch` finds `case 'F':` and jumps there, and then runs from that point onwards, exactly as before. There are no statements between `case 'F':` and `case 'f':`, so it simply carries on into the fail-counting code and then hits the

`break`. Both capital and lowercase end up doing the same thing, and we only had to write the code once.

This is the thing to understand about `switch`: a case label is a place to jump to, not a box that contains statements. Once control has jumped in, it keeps going until a `break` stops it, straight through any case labels it happens to meet along the way. Stacking labels the way we did above is a deliberate use of that. Forgetting a `break` at the end of a case is the accidental use of it, and it is a classic bug: your program runs the right case and then keeps running the next one as well.

Here is the whole grade-counting program, which also shows a `switch` nested inside a `do...while` loop:

```c
#include <stdio.h>

int main()
{
    int numStudents;
    char grade;
    int passCount = 0;
    int failCount = 0;

    printf("Enter the total number of students: ");
    scanf("%d", &numStudents);

    printf("Enter results (P for pass, F for fail):\n");

    do
    {
        scanf(" %c", &grade);

        switch (grade)
        {
        case 'F':
        case 'f':
            failCount++;
            break;
        case 'P':
        case 'p':
            passCount++;
            break;
        default:
            printf("Invalid character entered. ");
            printf("Enter a new result (P or F).\n");
        }
    } while (passCount + failCount < numStudents);
```

```
    printf("\nTest results:\n");
    printf("Passed: %d\n", passCount);
    printf("Failed: %d\n", failCount);

    return 0;
}
```

The loop condition counts how many valid results we have collected so far, so an invalid entry does not use up one of the students. Entering a `k` produces a complaint and then asks again:

```
Enter the total number of students: 5
Enter results (P for pass, F for fail):
p
P
f
F
k
Invalid character entered. Enter a new result (P or F).
P

Test results:
Passed: 3
Failed: 2
```

Escaping a loop with `break`

So far `break` has only appeared inside a `switch`, but it is a statement in its own right and it works in loops too. When C meets a `break` it looks for the nearest enclosing loop or `switch` and jumps out of it, continuing with the first statement after that structure. It does not matter how much of the loop body is left, and it does not matter whether the loop condition is still true.

That gives you a way of stopping a loop early when you have found what you were looking for. Here is an example.

The Square Kilometre Array is a radio telescope being built across sites in South Africa and Australia. Rather than one dish, it is a very large collection of antennas staring at the sky together, gathering data used to answer questions about the early universe:

These telescopes do not take pictures in the way a camera does. They observe radio frequencies, and they produce a staggering amount of data: rough estimates put the finished array at something like an exabyte per day, which is 10^{18} bytes. For comparison, roughly $4.35 \cdot 10^{17}$ seconds have passed since the big bang. Every day the array would generate more bytes of data than there have been seconds in the history of the universe. Processing that data efficiently is not a nice-to-have.

One thing astronomers look for in this data is pulsars. A pulsar is a rapidly spinning neutron star that sweeps a beam of radio waves past us as it turns, like a lighthouse. Their rotation is so

Figure 3: An artist's impression of the Square Kilometre Array. Image from Wikipedia.

regular that they rival atomic clocks, which makes them useful for testing general relativity and for detecting gravitational waves.

Here is our massively oversimplified version of the problem. We have a collection of images from the telescope. We want to work through them one at a time, and the moment we find a pulsar we want to stop looking and report where we found it. There is no point in scanning the remaining images once we have what we came for.

In pseudocode:

```
loop through all the images
    determine whether the image contains a pulsar
    if it is a pulsar: stop the loop
    otherwise: carry on
```

Building it up

We know how to loop through a fixed number of things: that is a counting loop, so it is a `for` loop. Start there and nothing else:

```c
int noOfImages = 5;
int i;

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

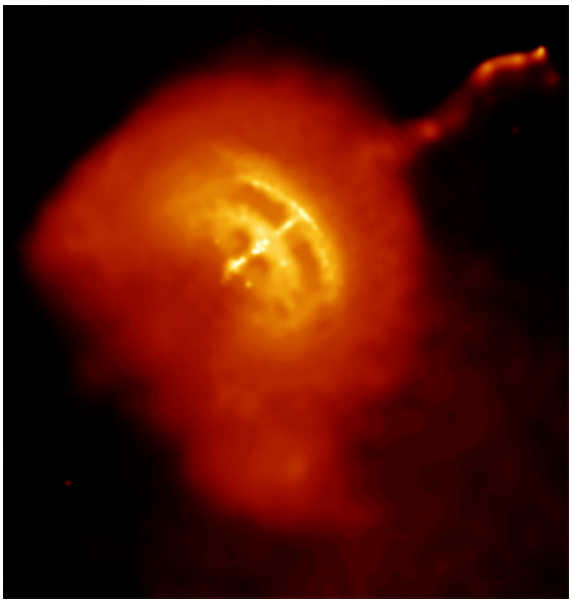

Figure 4: The Vela pulsar and the jet of particles it emits. Image from Wikipedia.

That `printf` is not part of the program we are writing. It is there so that we can run what we have and see the numbers 1 to 5 come out, which tells us the loop is right before we build anything on top of it. This is worth making a habit. When a program misbehaves, the one tool you always have is `printf`: put it wherever you are unsure, print the values you think you know, and let the program tell you where your idea of what is happening stops matching what is actually happening.

With the loop working, we can replace the placeholder with the real work: ask about the current image, read the answer, and stop if it is a pulsar.

The finished program

```c
#include <stdio.h>

int main()
{
    int noOfImages = 5;
    int pulsarOrNot;  // 1 for pulsar, 2 for not a pulsar
    int i;

    // Loop through all the images
    for (i = 1; i <= noOfImages; i++)
    {
        // Determine whether pulsar or not
        printf("Does image %d contain a pulsar (1: yes, 2: no)? ", i);
        scanf("%d", &pulsarOrNot);

        // If it is a pulsar: stop the loop
        if (pulsarOrNot == 1)
        {
            break;
        }

        // otherwise carry on
    }

    printf("Last image you looked at was image %d\n", i);

    return 0;
}
```

If we never find a pulsar, the loop runs to completion:

```
Does image 1 contain a pulsar (1: yes, 2: no)? 2
Does image 2 contain a pulsar (1: yes, 2: no)? 2
Does image 3 contain a pulsar (1: yes, 2: no)? 2
Does image 4 contain a pulsar (1: yes, 2: no)? 2
Does image 5 contain a pulsar (1: yes, 2: no)? 2
Last image you looked at was image 6
```

If we find one at image 3, the `break` fires and the remaining images are never examined:

```
Does image 1 contain a pulsar (1: yes, 2: no)? 2
Does image 2 contain a pulsar (1: yes, 2: no)? 2
Does image 3 contain a pulsar (1: yes, 2: no)? 1
Last image you looked at was image 3
```

Look carefully at the two final lines. In the second run `i` is 3, which is where we stopped.

In the first run `i` is 6, not 5. That is not a bug: the loop only ends when the condition `i <= noOfImages` fails, and for that to happen `i` has to have been incremented past 5. A counting variable that survives its loop always ends up one step beyond the last value it was used for, and forgetting that is a common source of off-by-one errors.

With five images this whole exercise is pointless: you could look at all of them. With a million it is not, and `break` is the difference between stopping at image 150 and grinding through the other 999,850.

All the control structures in one place

That is the last control statement. It is worth stopping to see what we have, because the collection is small and it is complete.

There are three kinds of control structure. The first is sequence, which is so obvious that it barely feels like a structure: one statement after another, in the order written.

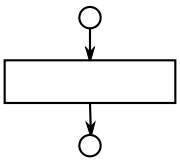

Figure 5: The sequence structure: do this, then do the next thing.

The second is selection, where the program chooses between alternatives. C gives you three selection statements: `if` on its own, `if...else`, and now `switch`.

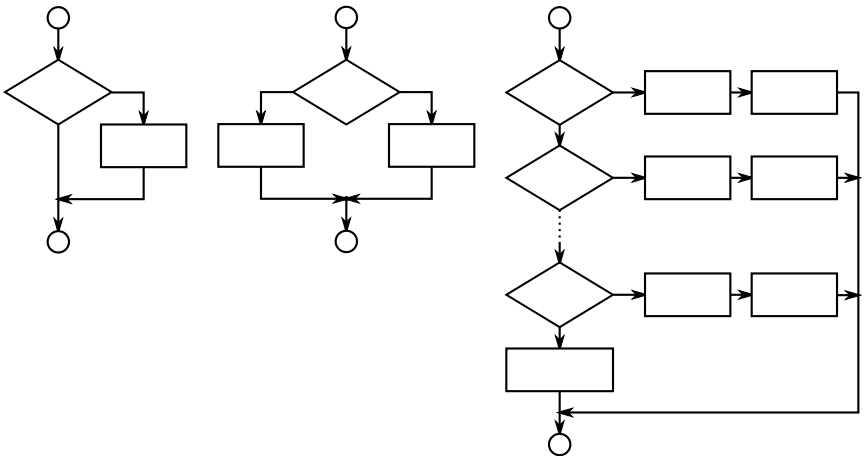

Figure 6: The three selection structures: `if`, `if...else` and `switch`.

The third is repetition, where a block runs more than once. C gives you three repetition statements as well: `while`, `do...while` and `for`. The middle diagram is the `do...while`, and you can see the difference at a glance: its test is at the bottom, so the block always runs at least once.

Three types, seven statements. That is everything C has for controlling the flow of a program, and with them you can implement any algorithm you can describe.

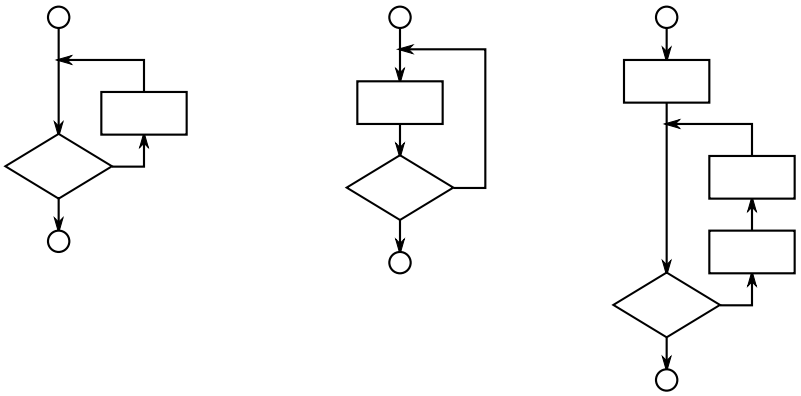

Figure 7: The three repetition structures: `while`, `do...while` and `for`.

The reason so few pieces go so far is that they combine in two ways. You can stack them, running one structure after another. And you can nest them, putting a whole structure inside the action block of another. We have been doing this for several lectures already: an `if` inside a `for`, a `for` inside a `for`, and just now a `switch` inside a `do...while`. The figure below shows the nesting idea in the abstract, with the second structure dropped into the action block of the first:

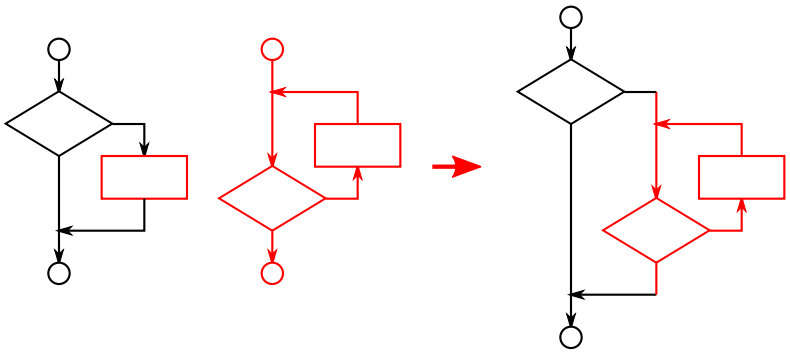

Figure 8: Nesting: the block of one structure is replaced by an entire structure. Here a loop, in red, is placed inside the action block of an `if` statement.

Any program you write from here on, however large, is these seven statements stacked and nested inside each other.

Looking forward

You now have the complete set of C control structures. Everything that follows in the course is about writing programs better rather than about new ways of directing the flow.

The next lecture starts on functions, which are the main tool for that. Up to now everything has gone inside `main`, and you have probably already noticed that programs get unwieldy that way. Functions let you take a piece of a program, give it a name, and use it without having to think again about how it works internally, which is exactly what we have been doing all along with `printf` and `scanf`.

Exercises

1. Write a program that reads a number from 1 to 7 and displays the corresponding day of the week using a `switch` statement. Handle invalid input with a `default` case.
2. Write a simple calculator. The user enters two numbers and an operator character (`+`, `-`, `*` or `/`), and a `switch` statement performs the right operation. Take care with division.
3. Take the wheels program and remove all of the `break` statements. Predict what it will print when the user enters 2, then run it and check.
4. Write a program that reads a single character and reports whether it is a vowel, using a `switch` statement with stacked case labels so that both uppercase and lowercase letters work.
5. In the pulsar program, what is the value of `i` after the loop if there are 5 images and the pulsar is in the last one? Work it out on paper first, then check.
6. Predict the output of the following, then check:

```c
int p = 3;
printf("%d\n", p == 3);
printf("%d\n", p = 5);
printf("%d\n", p == 3);
```