7. Logic and the `do...while` loop

Herman Kamper

This lecture does two things. It gives you the third and final loop in C, and then it gives you a way of combining conditions so that a single `if` can ask several questions at once. Neither of them lets you do anything you could not already do with what you have. Both of them make what you write shorter and clearer, and the second one in particular turns out to be where a surprising amount of engineering lives.

We start with the loop, since it is the smaller of the two ideas, and then spend most of the lecture on logic.

A loop that always runs once

Both of the loops you have so far test before they act. Look at the `while` loop on the left of the figure below. Control arrives at the top, goes to the condition, and only enters the block if the condition is true. Every time round, including the very first, the condition is checked before the actions happen. A `for` loop behaves the same way. If the condition happens to be false when we arrive, the block never runs at all.

That is usually what you want. But there is a family of problems where it is the wrong shape, and the clearest case is asking a user for input. Suppose you want a number between 1 and 10. You have to ask at least once, because until you have asked there is nothing to check: the question “is this valid?” only makes sense once the user has typed something. What you want to say is do the reading, and then keep going for as long as the answer is unacceptable.

So we want a loop that puts the actions first and the test second, as on the right of the figure:

Follow the right-hand diagram through. Control comes in at the top and goes straight into the actions, with nothing standing in the way. When the actions finish we reach the condition. If it is true we go back up and do the actions again. If it is false we drop out of the bottom and carry on with the program. The condition is at the bottom, which is the whole difference.

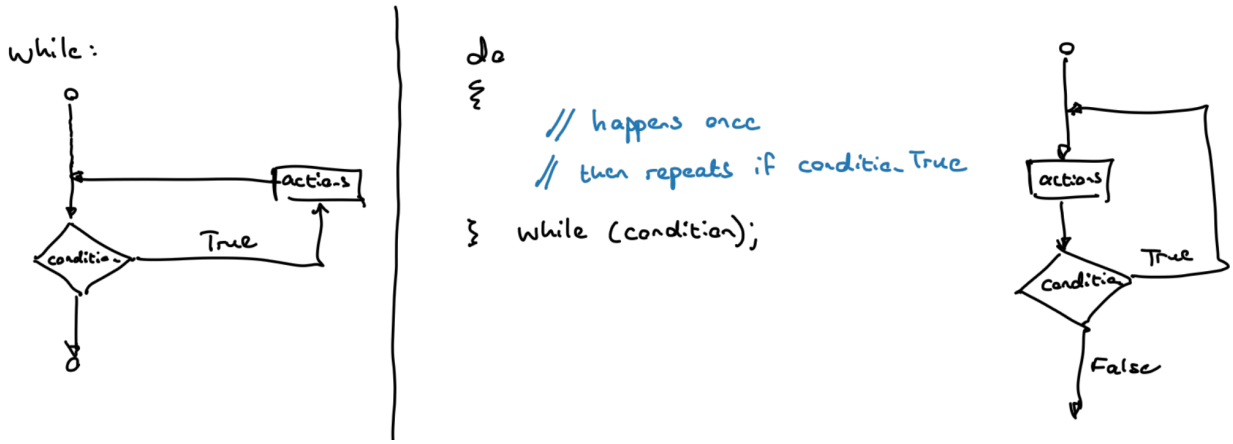

Figure 1: The two shapes side by side. A `while` loop tests before the block, so the block may run zero times. A `do...while` loop puts the block first and tests afterwards, so the block always runs at least once.

The `do...while` statement

The syntax is:

```
do {
    // happens once
    // then happens repeatedly if condition is true
} while (condition);
```

You get the keyword `do`, then a block in curly braces, then `while` with a condition in round brackets. Reading it out loud in the order it executes is a good habit: do this block, and then keep on doing it while the condition holds.

Two things to notice. The first is that the block always runs at least once. This is not a subtlety, it is the entire reason the statement exists. If your problem does not need that guarantee, use a `while` or a `for`.

The second is that semicolon at the very end, after the closing bracket. It is easy to forget, because nothing else in C that ends in a closing curly brace wants a semicolon after it. The way to remember it is that everything from `do` down to the end of `while (condition)` is one single statement, with the block sitting inside it, and statements in C end with a semicolon.

That is the last loop. Between `while`, `for` and `do...while` you now have every repetition structure the language offers. In practice `do...while` is the one you will reach for least often: it is worth using when you genuinely need the block to run before the test, and validating user input is the case where you genuinely do. We will build exactly that program later in this lecture. First we need something to put in the condition.

Logic

Here is a question that sounds easy. I want to check whether someone got a B for a course, and let us say a B is a mark between 65 and 75. In mathematics I would write $65 < x < 75$. How do I write that in C?

Not like that. C will let you type `65 < x < 75` and it will compile, but it does not mean what you want: C works out `65 < x` first, which gives 0 or 1, and then compares that answer with 75, which is always true. So the expression is true for every value of `x`, including -400. Chained comparisons are one of those things that look reasonable and quietly do the wrong thing.

With only the `if` statement you have one honest option, which is to put an `if` inside an `if`:

```
if (x > 65)
{
    if (x < 75)
    {
        printf("B");
    }
}
```

This works perfectly. It is also four lines of nesting and two sets of braces to express something you can say in six words in English, and if you wanted three conditions rather than two you would be three levels deep. There is a better way.

C has three logical operators:

Operator	Name	Meaning
`&&`	and	true when both sides are true
`||`	or	true when at least one side is true
`!`	not	flips true to false and false to true

Note that `&&` is two ampersands and `||` is two vertical bars. A single `&` and a single `|` are different operators in C which do something else entirely, so if your program compiles but behaves strangely, count the symbols.

With `&&` the B check collapses to one line:

```
if ((x > 65) && (x < 75))
{
    printf("B");
}
```

Read it as: if `x` is greater than 65, and `x` is less than 75, then print. Both comparisons are wrapped in their own brackets, and then the whole thing is wrapped in the brackets belonging to the `if`. You do not strictly need the inner brackets, but put them in anyway. They cost nothing and they make the two questions being asked visible at a glance.

Trying the operators out

The best way to get a feel for these is to write a small program that reads a mark and then asks it several different questions:

```
/*
 * Examples of using logical operators.
 */

#include <stdio.h>

int main()
{
    int grade;

    // Read the student's grade
    printf("Please enter your grade (0-100): ");
    scanf("%d", &grade);

    if ((grade > 65) && (grade < 75))
    {
        printf("Between 65 and 75: B!\n");
    }

    return 0;
}
```

Run this with 69 and it prints the message. Run it with 76 and nothing happens, because although the first comparison is true the second one is false, and `&&` needs both. Run it with 64 and nothing happens either, this time because the first comparison fails.

Now `||`. Suppose we want to catch the extreme cases, the marks above 90 and the marks below 10:

```
if ((grade >= 90) || (grade <= 10))
{
    printf("You are either up there, or too far gone to save\n");
}
```

Here only one of the two needs to hold. Enter 91 and the message appears. Enter 15 and nothing happens, because 15 is neither at least 90 nor at most 10. Enter 10 and the message appears again, because we wrote <= rather than < and so 10 is included. That is the kind of boundary you should check deliberately every time you write a condition, because off-by-one errors at the edges are the most common bugs in this sort of code.

Finally `!`, which takes one condition and reverses it. You already know how to test whether a mark is exactly 100:

```
if (grade == 100)
{
    printf("You win!\n");
}
```

The `!` operator lets you ask the opposite question:

```
if (!(grade == 100))
{
    printf("Second place is first loser\n");
}
```

The brackets after the exclamation mark matter. What is inside them, `grade == 100`, is worked out first and gives a true or a false; the `!` then flips it. So this block runs for every mark except 100. In this particular case you could equally have written `grade != 100` and saved yourself the trouble, and you should. Where `!` earns its keep is in front of a condition too complicated to negate by hand, and we will hit exactly such a case shortly.

Everything is a number

There is something going on underneath all of this that is worth making explicit, because it explains a lot of C's behaviour that otherwise looks arbitrary.

C has no true and false. Some languages have keywords for them; C does not. Instead, every logical expression you write evaluates to an ordinary integer:

- 0 means false
- 1 means true

That is not a metaphor. `grade == 100` is an expression with a numerical value, in the same way that `3 + 4` is an expression with a numerical value, and you can print it:

```
printf("Expression evaluates to: %d\n", grade == 100);
```

Enter 100 and this prints `1`. Enter 99 and it prints `0`. Nothing is being hidden from you: the comparison is a calculation whose answer happens to be a 0 or a 1, and the `if` statement is nothing more grand than a construct that runs its block when it is handed a non-zero number.

The logical operators work the same way, and you can print them directly without any variables involved at all:

```
printf("Expression evaluates to: %d\n", 1 && 1);   // prints 1
printf("Expression evaluates to: %d\n", 1 && 0);   // prints 0
printf("Expression evaluates to: %d\n", 0 && 1);   // prints 0
printf("Expression evaluates to: %d\n", 0 && 0);   // prints 0
printf("Expression evaluates to: %d\n", 0 || 1);   // prints 1
```

```
printf("Expression evaluates to: %d\n", 0 || 0);   // prints 0
printf("Expression evaluates to: %d\n", !(0 || 0)); // prints 1
```

The last line is worth pausing on. Inside the brackets, false or false is false, which is 0. The `!` then flips that 0 to a 1. Work through it from the inside out and there is nothing to guess at.

The tables below collect the whole story. They are the definitions of the three operators, with the and column giving `a && b` and the or column giving `a || b`:

a	b	and	or
0	0	0	0
non-zero	0	0	1
0	non-zero	0	1
non-zero	non-zero	1	1

a	!a
0	1
non-zero	0

Notice the word “non-zero” rather than 1. This is the last piece of the rule: when C is deciding whether something counts as true, zero is false and anything else at all is true. So `1999 || 0` gives 1, and `10 && 3` gives 1, and an `if` handed the value -7 will run its block. The operators produce 0 or 1, but they accept anything.

Where all of this comes from

It is worth knowing where this idea came from, because it is one of the few places in a first-year course where you can point at a single document and say: everything after this is different.

George Boole worked out an algebra of true and false in the 1850s, as pure mathematics, with no application in mind whatsoever. Eighty years later Claude Shannon was a 21-year-old master’s student at the Massachusetts Institute of Technology (MIT), earning his keep by maintaining the relay circuits on Vannevar Bush’s differential analyser, a room-sized mechanical computer. Shannon noticed that Boole’s algebra described the relays.

The observation is easier to see than to state. Consider a battery, a lamp, and two switches wired one after the other, as in the top circuit below. To light the lamp, both switches must be closed. That is `&&`. Now wire the two switches side by side instead, as in the bottom circuit. Now the lamp lights if either switch is closed, and it is dark only when both are open. That is `||`:

Shannon’s 1937 master’s thesis, *A Symbolic Analysis of Relay and Switching Circuits*, made this systematic. Once you accept the correspondence, designing a circuit becomes simplifying an equation, and simplifying an equation is something mathematics already knew how to do. The thesis is routinely called the most important master’s thesis of the century, and essentially every digital system built since rests on it: the relays became vacuum tubes and then transistors,

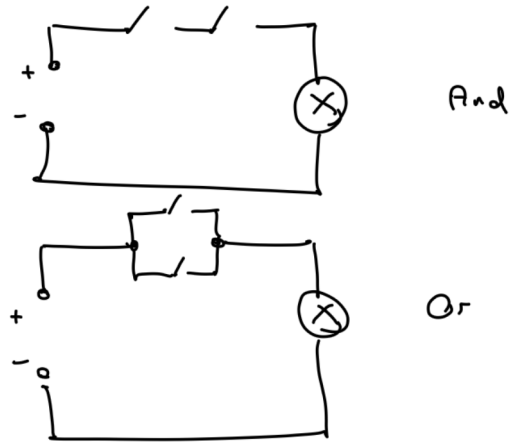

Figure 2: Two switches in series and two switches in parallel. Series gives you and: the lamp lights only if both switches are closed. Parallel gives you or: the lamp lights if either is closed.

but the algebra never changed. The `&&` you typed a page ago is Boole's, wired up by Shannon, running on a few hundred million transistors.

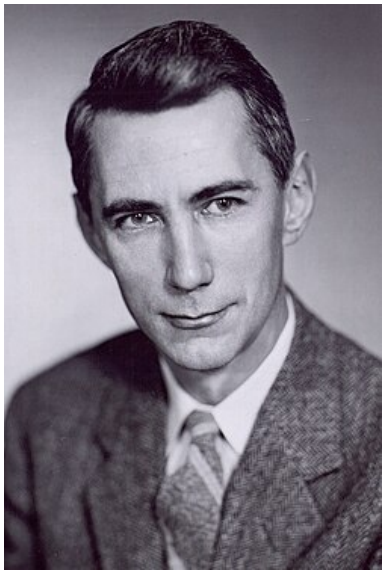

Figure 3: Claude Shannon. Image from Wikimedia Commons.

The same person then did it again. In 1948 he published *A Mathematical Theory of Communication*, which invented information theory more or less complete: it defined the field and solved most of its problems in one paper. That is what tells you how much data you can push through a noisy channel, and it is why your Wi-Fi works and why every compression format exists. Two ideas, both foundational to electrical engineering, one person.

Incidentally, the artificial intelligence assistant called Claude is named after him.

Design example: a happiness meter

Now let us put the loop and the logic together in one program.

> As part of a survey, read the user's level of happiness on a scale of 1 to 10 and tell the user whether that is a normal level (3 to 8) or an abnormal one (1, 2, 9 or 10). Make sure the user enters a valid level.

That last sentence is the interesting one. If someone assumes the scale runs to 100 and types 95, we do not want to store 95 and report on it. We want to ask again, and to keep asking until we get something sensible. That is a loop whose body must run at least once, which is our new statement.

The pseudocode:

```
Do
    Read the user's level of happiness
Until the user has entered a valid level

If the user has a normal level of happiness
    Tell the user the level is normal
Else
    Tell the user the level is abnormal
```

Getting the loop working first

Start with the reading, and put a condition in that you know keeps the loop going, just so that you can watch it go round:

```c
int happiness;

do
{
    printf("Happiness between 1 and 10: ");
    scanf("%d", &happiness);
} while (1);
```

You now know why `while (1)` loops forever: 1 is non-zero, non-zero is true, and the condition is never anything else. Running this asks over and over regardless of what you type, which is not the program we want but does confirm that the shape is right.

Getting the condition right

Now for the real condition. Take a moment and work out for yourself what should go in those brackets before reading on, because this is the step where nearly everyone slips.

The natural first attempt is:

```
do
{
    // ...
} while ((happiness >= 1) && (happiness <= 10));    // wrong
```

Read it: happiness is at least 1 and at most 10. That is a perfectly correct description of a valid input, which is why it feels right. Now read it again as a loop condition. The condition is what keeps you in the loop. So this says: keep asking while the answer is valid, and stop as soon as the user types something ridiculous. It does exactly the opposite of what we want.

This is worth dwelling on, because the mistake is not really about `&&`. It comes from writing the condition that describes success, when a `do...while` for input validation needs the condition that describes failure. You stay in the loop while things are still wrong.

There are two clean ways to fix it. The first keeps the expression you already wrote and negates the whole thing:

```
do
{
    // ...
} while (!((happiness >= 1) && (happiness <= 10)));
```

This is where `!` earns its keep. You do not have to think about how to invert the condition; you write down the thing you understand, which is what valid means, and then flip it.

The second states the failure directly:

```
do
{
    // ...
} while ((happiness < 1) || (happiness > 10));
```

Read it: keep going while the answer is too small, or too big. Notice that the `&&` has become an `||` and both comparisons have flipped. That is not a coincidence, it is a general rule about negating conditions, and if you carry on in engineering you will meet it again under the name De Morgan's laws. For now, satisfy yourself that the two versions really are the same by trying both with -1, with 11, and with 8.

Finishing the program

The rest is an `if` with an `&&` in it, exactly like the B check from earlier:

```c
/*
 * Analyse a user's happiness.
 */

#include <stdio.h>

int main()
{
    int happiness;

    // Keep asking until the user enters a valid level
    do
    {
        printf("Happiness between 1 and 10: ");
        scanf("%d", &happiness);
    } while ((happiness < 1) || (happiness > 10));

    // If happiness is 3 to 8
    if ((happiness >= 3) && (happiness <= 8))
    {
        printf("Normal levels\n");
    }
    else
    {
        printf("Possible abnormal levels\n");
    }
    return 0;
}
```

Running it and deliberately misbehaving:

```
Happiness between 1 and 10: -1
Happiness between 1 and 10: 11
Happiness between 1 and 10: 7
Normal levels
```

and with a valid but extreme answer:

```
Happiness between 1 and 10: 2
Possible abnormal levels
```

If you understand this program, you understand logic in C. It is only twenty lines, but there is a loop that is guaranteed to run once, a condition combining two comparisons, and a decision combining two more.

An aside: thresholds in the real world

The happiness meter is a joke, but only partly.

The first conference I ever went to as a master's student was a speech processing conference, and the first talk I sat in on was a paper about detecting depression from a person's speech signal: you record someone talking, extract some numbers from the audio, and try to decide whether the speaker is depressed:

An Investigation of Depressed Speech Detection: Features and Normalization

Nicholas Cummins[1], *Julien Epps*[1,2], *Michael Breakspear*[3,4], *and Roland Goecke*[5]

[1] School of Elec. Eng. and Telecomm., The University of New South Wales, Sydney, Australia
[2] ATP Research Laboratory, National ICT Australia (NICTA), Australia
[3] Black Dog Institute and School of Psychiatry, The University of New South Wales, Australia
[4] Division of Mental Health Research, Queensland Institute of Medical Research, Australia
[5] Faculty of Information Sciences and Engineering, University of Canberra, Australia
nicholas.cummins@student.unsw.edu.au, j.epps@unsw.edu.au

Figure 4: The paper that happened to be the first conference talk I ever attended. Its system takes in a speech signal and produces a decision.

People are still working on this problem today and it is genuinely hard. But somewhere at the end of any such system there is a decision, and a decision means a threshold: this quantity is above that value, or between these two values, and therefore we report one thing rather than another. The numbers going into the comparison come from a machine learning model rather than from a `scanf`, and there are usually rather more than two of them, but the statement that finally makes the call looks a lot like the `if` you just wrote.

Decisions inside loops inside loops

One last example, and it is a hard one. A colleague of mine in applied mathematics says that a loop inside a loop is where you lose students. So this is a test of whether you can actually follow what a program does, rather than recognise what it looks like.

Work out what this prints, without running it:

```
#include <stdio.h>

int main()
{
    int i, j;

    for (i = 1; i <= 7; i++)
    {
        for (j = 1; j <= 7; j++)
        {
            if (!(i == 4 || j == 4))
            {
                printf("* ");
            }
            else
            {
                printf("  ");
            }
        }
        printf("\n");
    }

    return 0;
}
```

I find this genuinely difficult and I have to go slowly, so here is how I do it: step through the code one line at a time, keeping a note of what is in each box.

The outer loop initialises `i` to 1. The condition asks whether 1 is at most 7; it is, so we go into the block. Now we are in the inner loop, which initialises `j` to 1. Is 1 at most 7? Yes, so into the inner block, where we meet the `if`.

Read the condition from the inside out, exactly as you would in mathematics. Innermost first: `i == 4` is false, because `i` is 1. Then `j == 4` is false, because `j` is 1. False or false is false. Then the `!` flips that to true. So we print `*`.

The inner block is done, so we do the inner step: `j++` makes `j` 2. Is 2 at most 7? Yes, back into the block. `i` is still 1 and `j` is 2, so the condition is true again and we print another `*`. Same for `j` equal to 3.

Now `j` becomes 4, and this is the interesting pass. `i == 4` is still false, but `j == 4` is now true.

False or true is true. The `!` flips it to false, so we take the `else` branch and print two spaces. Then `j` goes to 5, 6, 7, and the stars come back.

When `j` becomes 8 the inner condition fails and we drop out of the inner loop entirely. Only then do we reach the `printf("\n")`, which is inside the outer block but outside the inner one, so it runs once per row. This is worth being careful about: control does not go back to `i++` until everything between the outer braces has finished. Follow the braces and you will not get lost.

Then `i` becomes 2 and the whole inner loop runs again from scratch, `j` starting at 1 once more. The one row that comes out differently is `i` equal to 4, where `i == 4` is true for every single value of `j`, so the entire row is spaces.

Putting it together, we get a seven by seven grid of stars with row four and column four knocked out:

```
* * *   * * *
* * *   * * *
* * *   * * *

* * *   * * *
* * *   * * *
* * *   * * *
```

If you found that hard, that is the correct reaction, and there is a tool that helps enormously while you are learning. Paste a program into https://pythontutor.com/visualize.html, choose C, and it will step through it line by line, showing you an arrow at the current statement, the current contents of every variable, and the output as it accumulates. It is free and open source. Use it on this program and watch `i` and `j` move.

Confusing == and =

Before leaving conditions, a warning about a bug that will bite you at least once.

The equality operator is == and the assignment operator is =. They are different, and typing one where you meant the other is legal C. This is what you meant to write:

```c
if (payCode == 4)
{
    printf("You get a bonus!\n");
}
```

and this is the typing mistake:

```c
if (payCode = 4)
{
    printf("You get a bonus!\n");
}
```

The second version does not compare anything. It stores 4 in `payCode`, destroying whatever was there. The value of an assignment expression is the value assigned, so the `if` is handed 4, and 4

is non-zero, and non-zero is true. Every employee gets a bonus, and `payCode` has been quietly overwritten as well. The compiler will usually warn you about this, and you should read your warnings.

The same confusion runs the other way. Writing:

```
x = 1;
```

puts 1 into `x`, while:

```
x == 1;
```

tests whether `x` is 1, throws the answer away and changes nothing at all. It is a complete, legal statement that does nothing.

Looking forward

You now have all three loops and the full set of logical operators, and between them they cover essentially every control structure you need. Sequence, selection, repetition, and a way of asking compound questions: any algorithm can be expressed with these.

The next lecture adds one more selection statement, `switch`, which handles the case where you want to compare one variable against a list of specific values. It does nothing that a chain of `if` and `else if` cannot do, but for that particular shape of problem it reads much better.

Exercises

1. Write a program that uses a `do...while` loop to ask the user for a number between 1 and 100, refusing to accept anything outside that range.
2. Write a program that displays a menu with three options – add, delete and quit – reads the user's choice, and repeats until the user chooses quit. Explain why a `do...while` loop fits this better than a `while` loop.
3. Without running them, work out what each of the following prints:

   (a) `printf("%d\n", 5 && 0);`
   (b) `printf("%d\n", 5 || 0);`
   (c) `printf("%d\n", !5);`
   (d) `printf("%d\n", !(3 > 2));`
   (e) `printf("%d\n", (2 < 3) && (3 < 2));`
4. Explain in one sentence why `if (65 < x < 75)` compiles but is always true.
5. Write a program that reads a year and reports whether it is a leap year. A year is a leap year if it is divisible by 4, except that years divisible by 100 are not leap years unless they are also divisible by 400. The `%` operator gives you the remainder, so `year % 4 == 0` tests divisibility by 4.

6. What does the following loop print, and how many times does the block run?

```
int n = 20;
do
{
    printf("%d\n", n);
    n++;
} while (n < 10);
```

7. Modify the nested loop example (the one with the stars *) so that it prints a seven by seven grid with the diagonal blank rather than row and column four. You only need to change the condition.

8. Take the nested loop example (the one with the stars *) and work out, without running it, what happens if the `!` is removed. Then check.