3. Breaking the flow: the `if` statement

Herman Kamper

Every program so far has run its statements one after another, top to bottom: declare a variable, read a value, do some arithmetic, print a result, stop. This is called *sequential execution*, and it is fine for simple tasks. But most interesting problems require the program to make a decision: to do one thing in some situations and something different in others. This lecture introduces the tool that makes decisions possible in C, the `if` statement, which lets a program break out of its strict top-to-bottom flow and choose what to do based on the values it is working with.

The running example for this lecture is a small program that reads a student's grade and reports whether they passed or failed. It is a deliberately simple problem, but it is enough to introduce everything we need.

Controlling the flow

It is a remarkable fact that every program ever written, in any language, can be built from just three kinds of *control structures*:

- *Sequence*: statements executed one after another, in the order written. This is what we have used up to now.
- *Selection*: choosing whether or not to execute a block of code, depending on some condition. This is the subject of this lecture.
- *Repetition*: repeating a block of code, often until some condition is met. These are loops, covered in the next lecture.

C provides seven different statements for controlling the flow of a program,[1] but they all fall into these three categories. The selection statements are `if`, `if...else` and `switch`. This lecture covers the first two; `switch` comes later.

Algorithms and programs

Before writing any C, it helps to separate two ideas that are easy to confuse: the algorithm and the program.

[1] Only three control structures are needed to express any computation: sequence, selection and repetition. This was proved by Böhm and Jacopini in 1966. The idea traces back further: Alan Turing's 1936 model of computation, the Turing machine, uses conditional branching at its core, deciding what to do next based on what it reads from a tape. Böhm and Jacopini showed that this is all you ever need.

An *algorithm* is the recipe: the sequence of steps that solves the problem, written in whatever form is convenient. A *program* is the actual code that carries out those steps, written in a real programming language and following its exact syntax. The algorithm is the thinking; the program is the typing.

This distinction matters because working out the algorithm is usually the hard part. Once you have a correct sequence of steps, translating it into C is comparatively mechanical. Most of the difficulty in programming is in the problem-solving, not the syntax.

For the pass/fail problem, the algorithm is something like this:

```
read in the student's grade
if the grade is greater than or equal to 50:
    print "passed"
otherwise:
    print "failed"
    print some encouragement
```

Notice that only one of the two branches runs: a student who scores 60 sees only “passed”, while a student who scores 40 sees “failed” followed by the encouragement. That branching is exactly what the `if` statement will give us.

Pseudocode

What we wrote above is a form of *pseudocode*: an informal, English-like description of an algorithm. Pseudocode is deliberately sloppy. It is not written in any real language, cannot be run on a computer, and has no strict rules. Its whole purpose is to let you think through the logic of a program before worrying about the precise syntax of C.

Writing pseudocode first is a good habit. It lets you solve the problem in plain language, then translate that solution into code once you are confident it is correct.

Flow diagrams

Pseudocode is one way to describe an algorithm. The other common way is a *flow diagram*, which represents the algorithm graphically. A flow diagram is built from a few standard symbols connected by arrows that show the order of execution:

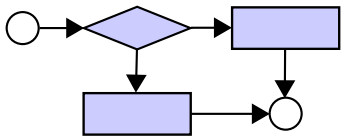

Figure 1: The three basic flow-diagram symbols: an oval marks the start or end, a rectangle is any action or step, and a diamond marks a decision.

The three symbols are:

- An oval for the beginning or end of the program

- A rectangle for any action, such as a calculation or a print statement
- A diamond for a decision, where the path splits depending on a condition

The programs we have written so far, being purely sequential, produce the simplest possible flow diagrams: a straight line of actions from start to end. The diagram below shows a short sequence that adds a grade to a running total and then increments a counter.

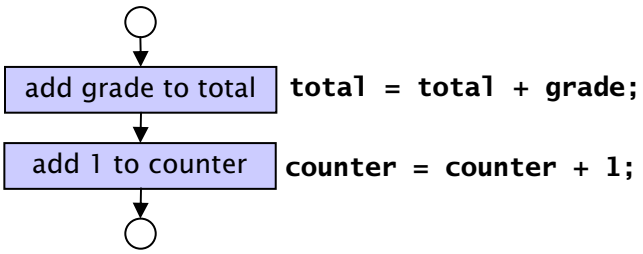

Figure 2: A flow diagram for a purely sequential program: each action follows the previous one with no branching.

Decisions are where flow diagrams become more interesting, and that is what we turn to now.

The `if` statement

The `if` statement executes a block of code only when a condition is true. Its form is:

```
if (condition)
{
    // these statements run only if the condition is true
}
```

The computer evaluates the condition inside the parentheses. If it is true, the statements inside the curly braces run. If it is false, they are skipped entirely, and execution jumps straight to the code after the closing brace. Either way, the program then carries on.

As a flow diagram, the `if` statement looks like this:

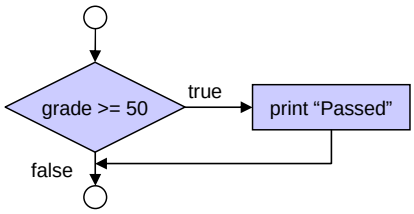

Figure 3: Flow diagram of an `if` statement. When the condition is true the action runs; when it is false it is skipped, and both paths continue from the same point.

The `if...else` statement

Often we want to do one thing when a condition is true and a different thing when it is false. For this we add an `else` block:

```c
if (condition)
{
    // runs if the condition is true
}
else
{
    // runs if the condition is false
}
```

Exactly one of the two blocks runs, never both and never neither. The flow diagram makes the split clear:

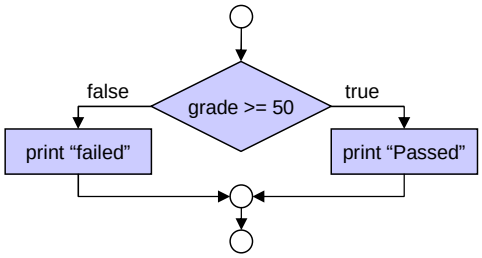

Figure 4: Flow diagram of an `if...else` statement. The condition selects exactly one of the two branches; both then rejoin and continue.

Pass or fail

We now have everything we need to write the pass/fail program. The two commands we already know, `scanf` for reading input and `printf` for output, handle the first and last steps; the `if...else` statement handles the decision in the middle. Before you look at the answer below, you can give it an attempt yourself (even just with pen on paper).

```c
/*
 * A program to tell a student whether they pass or fail.
 */
#include <stdio.h>

int main()
{
    int grade;

    // Read in the student's grade
    printf("Enter student's grade: ");
    scanf("%d", &grade);

    if (grade >= 50)
    {
        printf("You pass!\n");
    }
    else
    {
        printf("You fail.\n");
        printf("See you again next year!\n");
    }

    return 0;
}
```

Trace it with an example. Suppose the user enters 67. The condition `grade >= 50` is true, so the first block runs and the program prints:

```
You pass!
```

Now suppose the user enters 42. The condition is false, so the first block is skipped and the `else` block runs instead:

```
You fail.
See you again next year!
```

The boundary case is worth checking. If the user enters exactly 50, the condition `grade >= 50` is still true, because >= means "greater than or equal to". Had we written `grade > 50` instead, a mark of exactly 50 would fail, which is not what we want. Getting this boundary right is a common source of bugs, so always read the condition carefully against the requirement.

One small point about braces. When a block contains only a single statement, C lets you omit the curly braces:

```c
if (grade >= 50)
    printf("You pass!\n");
```

This works, but it is a trap. If you later add a second statement to the block and forget to add

braces, only the first statement will be governed by the `if`, and the second will run unconditionally. To avoid this whole class of bug, the advice in this course is simple: always use braces, even for a single statement.

Comparing values

The condition in an `if` statement is usually a comparison. C provides six operators for comparing values:

Operator	Meaning
`==`	equal to
`!=`	not equal to
`<`	less than
`>`	greater than
`<=`	less than or equal to
`>=`	greater than or equal to

A couple of these trip up beginners. Note that "equal to" is written with two equals signs, `==`, not one. And "not equal to" is `!=`, with the exclamation mark standing in for "not".

There is also a restriction that catches people coming from mathematics. In maths you might write $0 < x < 5$ to mean "x is between 0 and 5". In C this does not work: you cannot chain comparisons like `0 < x < 5`. You have to test the two conditions separately, which for now we can do by nesting one `if` inside another:[2]

```
if (x > 0)
{
    if (x < 5)
    {
        printf("x lies between 0 and 5\n");
    }
}
```

The most common mistake: = versus ==

Because a single = and a double == look so similar, mixing them up is probably the single most common error in C, and one that even experienced programmers make. The two are completely different:

- `grade = 50` is *assignment*: it stores the value 50 in `grade`.
- `grade == 50` is a *comparison*: it asks whether `grade` currently holds the value 50, and gives a true or false answer.

[2] A cleaner way, using logical operators, comes later.

The danger is that writing = where you meant == is not a syntax error; the program still compiles and runs, but does the wrong thing. To see why, we need to know how C actually treats a condition.

Internally, C has no separate notion of "true" and "false". It simply checks whether the value in the condition is zero or non-zero: a value of zero counts as false, and any non-zero value counts as true. A comparison like `grade == 50` produces 1 when true and 0 when false, which is why it works as a condition.

Now consider the bug. Suppose you write:

```
if (grade = 50)   // WRONG: single '=' instead of '=='
{
    printf("Exactly fifty\n");
}
```

Instead of comparing, this *assigns* 50 to `grade`. The value of the assignment is 50, which is non-zero, so the condition is always true, no matter what the user actually entered. The message prints every time, and the original grade is silently overwritten. Some languages reject this outright, but C accepts it, so the responsibility for catching it is yours.

Nested `if...else` statements

Because the block inside an `if` or `else` can contain any statements at all, it can contain further `if` statements. This lets us test several cases in turn. Suppose we want to convert a grade into a symbol: A for 80 and above, B for 70 to 79, C for 60 to 69, and D for 50 to 59. The algorithm nests each test inside the `else` of the previous one:

```
if (grade >= 80)
{
    printf("A\n");
}
else
{
    if (grade >= 70)
    {
        printf("B\n");
    }
    else
    {
        if (grade >= 60)
        {
            printf("C\n");
        }
        else
        {
            if (grade >= 50)
            {
                printf("D\n");
            }
        }
    }
}
```

Read it from the top. If the grade is at least 80, we print A and skip everything else. Otherwise we are inside the first `else`, where we already know the grade is below 80, and we ask whether it is at least 70; if so, it must be between 70 and 79, so we print B. Each `else` block narrows the range further. The indentation is what makes this structure readable, which is exactly why consistent indentation matters.

Looking forward

With the `if` and `if...else` statements, our programs can now choose between different actions, which is a large step up from a fixed sequence of instructions. The `if` statement is the first of the three control structures beyond plain sequence, and it alone already lets us express a surprising range of decisions.

The next lecture introduces the third kind of control structure: *repetition*, or loops. Where

selection lets a program choose whether to run a block of code, repetition lets it run a block many times over. Together, sequence, selection and repetition are enough to build any program.

Exercises

1. Explain, in your own words, the difference between an algorithm and a program. Why is it often useful to write pseudocode before writing C?
2. Write the condition for each of the following:
   (a) `x` is negative
   (b) `y` is exactly zero
   (c) `age` is not equal to 18
   (d) `temperature` is at least 100
3. The following code is meant to print "Winner" only when `score` equals 100, but it prints "Winner" every time. What is the bug, and how do you fix it?

```c
if (score = 100)
{
    printf("Winner\n");
}
```

4. Write a complete C program that asks the user for an integer and prints whether it is positive or negative.
5. Write a complete C program that asks the user for the current temperature as an integer. If it is below 0, print "Water freezes."; otherwise print "Water does not freeze."
6. Draw a complete flow diagram for the pass/fail program from this lecture.
7. Write a program that reads two numbers from the user and reports whether they are the same, or if they are different, which one is larger. First write the algorithm as pseudocode, then translate it into a C program. For example, given the input 12 and 7, the program should report that 12 is larger.