

The if statement

Comments and questions to [John Rowe](#).

Like everything else in C, `if` statements operate in a simple and consistent way, the key is in how we use them to create more-complicated programs.

Recap

As we mentioned last week we can preface any statement by:

```
if (expression)
```

to obtain a new statement where the (original) sub-statement is only executed if the controlling expression is true:

```
if ( x > 0 )
    printf("x is greater than zero\n");
```

Optionally, we may also have a statement to be executed if the condition is false:

```
if ( x > 0 )
    printf("x is greater than zero\n");
else
    printf("x is not greater than zero\n");
```

Either or both of the statements may be replaced by a `{ ... }` block. Finally we noted that:

- There is no semi-colon after the `if (...)`. This would end the `if` statement at that point.
- There is no semi-colon after the `{ ... }` block.

What is truth?

C does not use a separate "logical" or Boolean (true/false) class for its conditional expressions. Instead, the expression in an `if` statement is simply an arithmetic expression which is taken to be "true" if it is non-zero. Thus, in principle we can write: `if (-0.001)`, which would always be considered to be "true" as `-0.001` is not equal to zero. Needless to say, this wouldn't be a very good idea!

Relational operators

The following relational operators all return the `int` values **one** ("true") or **zero** ("false").

Operator	One if
<code>expr1 > expr2</code>	<code>expr1</code> is greater than <code>expr2</code>
<code>expr1 < expr2</code>	<code>expr1</code> is less than <code>expr2</code>
<code>expr1 >= expr2</code>	<code>expr1</code> is greater than or equal to <code>expr2</code>
<code>expr1 <= expr2</code>	<code>expr1</code> is less than or equal to <code>expr2</code>
<code>expr1 == expr2</code>	<code>expr1</code> is equal to <code>expr2</code>
<code>expr1 != expr2</code>	<code>expr1</code> is not equal to <code>expr2</code>

As with the arithmetic operators, C "does the right thing" if the operands are of different type, e.g. `if (0.001 > 0)` behaves as you would expect ("true"), it doesn't get confused by the fact that one side is a `double` and the other an `integer`.

Note that the equality operator is a **double** equals sign, a single equals sign is the assignment operator (see below).

Other "logical" operators

C also defines the following operators for the logical operations "AND", "OR", etc.

These operators all have `int` operands, which are nearly always individual relational expressions such as `x > 0`, etc.

In the following examples we use constants to make clear what the results should be although in practice some or all of

An *operand* is a fancy name for the things operators operate on, e.g. in the expression "`x * y`" the operator is `*` and the operands `x` and `y`.

them would be variables, for obvious reasons.

Operator	Meaning	Example & result		Example & result	
&&	AND	8 > 1 && 7 < 4	0	8 > 1 && 4 < 7	1
	OR	8 > 1 7 < 4	1	8 > 1 4 < 7	1
()	Parentheses	(6 > 4 4 < 6) && 6 == 3	0	6 > 4 (4 < 6 && 6 == 3)	1
!	Negation	! 1	0	! (8 > 1 && 4 < 7)	0

Note that the OR operator "||" evaluates to one ("true") if either or both of its operands are non-zero.

Example: Game of Life

Conway's classic Game of Life considers a rectangular grid of cells, each of which is either alive or dead. Each cell has eight neighbours; the diagram below shows a live central cell with four live neighbours and four dead ones:

*		*
*	*	
	*	

The game progresses in generations, in each generation a live cell with either two or three live neighbours stays alive, otherwise it dies. A dead cell with exactly three live neighbours becomes alive, otherwise it stays dead. In the above example the central cell would die.

Put another way: a cell with exactly three live neighbours will be alive in the next generation, as will a cell that is alive in this generation with exactly two live neighbours. All other cells will be dead.

If we denote a live cell by the value one and a dead cell by zero, then an `if` statement to calculate the next-generation state of a cell would be:

```
if ( neighbours == 3 || ( state == 1 && neighbours == 2 ) )
    nextstate = 1;
else
    nextstate = 0;
```

We don't actually need the parentheses after the "||" but there's less chance of us making an error if we put them in.

Three common mistakes to look out for

The compiler on the Mac will warn you about all the errors below with the Yellow Triangle of Peril™, although the warning may be a little opaque, for example words to the effect of "use parentheses if you really want to do this".

1. The grandmother of all bugs - missing out an = sign

The equality operator is the **double** equals sign `==`, and the assignment operator **single** equals sign `=`.

This leads to the classic bug:

```
if ( mass = 0 ) /* Wrong! */
    printf("The mass should not be zero!\n");
```

This has two consequences:

1. The variable `mass` is assigned the value zero.
2. This value of the variable `mass` is used as the condition of the `if()` which is of course false, as `mass` has just been set to zero.

2. && and || are double too

Less common is to have a single `&` or `|` when you meant to have two:

```
if ( j > 0 & j % 2 == 0 ) /* Wrong */
    printf("j is positive and even\n");
```

This is also legal (but almost certainly wrong) as a single `&` is the "bitwise and" operator which we won't be using until a later lecture.

3. a == b == c doesn't do what you expect

Use:

```
a == b && b == c
```

instead.

This because C, which places great value on consistency, treats the expression `a == b == c` in the same way as it treats `a * b * c`. First it evaluates `a == b` to obtain either zero or one, then it checks that `c` is equal to that value (zero or one). The expression is therefore equivalent to:

```
( a == b && c == 1 ) || ( a != b && c == 0 )
```

which is probably not what you want!

Similarly, use

```
a > b && b > c
```

instead of:

```
a > b > c      /* Wrong */
```

Introducing the else if ()

if()s can be nested

It's possible to put an `if ()` inside another `if ()` (called **nesting**) as in this example:

```
if ( j < 0 ) {
    printf("j is negative\n");
}
else {
    /* j >= 0 */
    if ( j % 2 == 0 )
        printf("j is even and positive\n");
    else
        printf("j is odd and negative\n");
}
```

The `{ }`s are unnecessary here but I think they make the code look much clearer.

Because it is inside the `else`, the second `if` statement, `if (j % 2 == 0)`, doesn't even get looked at if the first `if` statement is not true. Furthermore the second `else` only gets considered if neither of the two `if` statements is true.

Losing the curly braces

As we mentioned above, the curly braces are in fact unnecessary and it's useful see the above code slightly reformatted. (Remember that C ignores line breaks and indentation.)

We'll show it twice once with braces and once with them removed:

With one pair of braces removed

```
if ( j < 0 ) {
    printf("j is negative\n");
}
else if ( j % 2 == 0 ) {
    printf("j is even and positive\n");
}
else {
    printf("j is odd and positive\n");
}
```

With all the braces removed

```
if ( j < 0 )
    printf("j is negative\n");
else if ( j % 2 == 0 )
    printf("j is even and positive\n");
else
    printf("j is odd and positive\n");
```

Although this is technically two `if`, statements, one inside the other, it looks and behaves just like one large `if` statement with a new statement called `else if`. Furthermore, we can repeat this trick as often as we like.

statement with a new statement called `else`. Furthermore, we can repeat this trick as often as we like. Technically, this is the origin of the next section.

else if ()

The expression `else if ()` is used to introduce an alternative that is only examined if the first `if ()` was false. There can be as many `else if ()`s as we like, they are examined in order but once any one of them evaluates to "true" (non-zero) the rest are ignored.

```
if ( x > 100 )
    printf("x is greater than 100\n");
else if ( x > 10 )
    printf("x is greater than 10 but less then or equal to 100\n");
else if ( x > 0 )
    printf("x is greater than 0 but less then or equal to 10\n");
else
    printf("x is negative\n");
```

There are two things to notice:

1. There is a space in "else if".
2. Had we missed out the `else`s then all four `printf()`s would have executed, which obviously would have been an error.

An extended example

The previous example only distinguishes odd from even for positive numbers, not negative. We can fix this for the "nested" example as follows:

```
if ( j > 0 ) {
    if ( j % 2 == 0 )
        printf("j is even and greater than zero\n");
    else
        printf("j is odd and greater than zero\n");
}
else {
    /* j <= 0 */
    if ( j % 2 == 0 )
        printf("j is even and negative\n");
    else
        printf("j is odd and negative\n");
}
```

The `{}`s are again unnecessary here but again I think they make the code look much clearer.

Armed with "else if()" we could rewrite this as follows:

```
if ( j > 0 && j % 2 == 0 )
    printf("j is even and greater than zero\n");
else if ( j > 0 && j % 2 != 0 )
    printf("j is odd and greater than zero\n");
else if ( j <= 0 && j % 2 == 0 )
    printf("j is even and negative\n");
else if ( j <= 0 && j % 2 != 0 )
    printf("j is odd and negative\n");
else
    printf("I seem to have made a mistake!\n");
```

One minute discussion: turn to your neighbour and discuss which version of the even/odd, positive/negative code is clearer.

Loops

As mentioned in the preface, most programs have some sort of *loop* to enable things to be done multiple times. C has three such loops, today we will look at the simplest, the `while()` statement (or the `while()` loop).

The while() loop

The **while()** loop has the form:

```
while ( expression ) {
    /* Do something ... */
}
```

The contents of the loop, which as always may be a single statement or a { ... } block, are executed for as long as the condition is true. **If the condition is false the body of the loop never executes at all.**

For example:

```
while (x > 1)
    x = x / 2.0;
```

In this rather contrived example, *x* is halved every time the loop is executed until at last the test condition is false and the loop quits. If *x* had started with a value of less than or equal to one the loop never not have executed even once.

Thus the `while()` operates in a similar way to the `if()` except there is no `else` and it repeats as long as the condition is true (non-zero).

Example: power function

We can use the `while` loop to write a "poor man's power function" which calculates *value* to the power of *exponent* where *exponent* is a positive integer. We show the core loop of program first, with the complete program following below:

```
result = 1.0;
while ( exponent > 0 ) {
    result = result * value;
    exponent = exponent - 1;
}
```

When examining such a loop it's good to ask "what would happen if the condition were false the very first time it was evaluated?". In this case we are talking about `exponent == 0` in which case the loop would never execute. Thus `result` would be 1.0, the correct answer.

If `exponent == 1` the loop will execute once, making `result` equal to `value`, again the correct answer. If `exponent == 2` the loop will execute twice, making `result` equal to `value` squared, etc.

The complete program

```
/*
 * Poor man's pow() calculation with only positive integer exponents
 * Calculates "value" to the power of "exponent"
 */
#include <stdio.h>

int main() {
    double value = 1.414;
    int exponent = 4;
    double result;

    printf("%f to the power %d is ", value, exponent);

    result = 1.0;
    while ( exponent > 0 ) {
        result = result * value;
        exponent = exponent - 1;
    }

    printf("%f\n", result);
    return 0;
}
```

The result is:

```
1.414000 to the power 4 is 3.997584
```

Second example: prime number test

Here we show the complete program, which uses the logical `&&` in its while condition:

```

/*
 * Print the lowest factor of an integer or say that it is prime.
 * Written to illustrate the while() loop
 * The basic algorithm is to try every possible factor up to the square
 * root of the number being looked at (the target)
 */
#include <stdio.h>
#include <math.h>

int main() {
    int target, factor, maxfactor;

    target = 55;

    maxfactor = sqrt(target); /* Largest factor we need to try */

    factor = 2;
    while ( target % factor != 0 && factor <= maxfactor )
        factor = factor + 1;

    if ( factor > maxfactor )
        printf("%d is prime\n", target);
    else
        printf("\t%d is not prime, its lowest factor is %d\n",
            target, factor);

    return 0;
}

```

Any { ... } block can have its own variables

We have already seen that the `main() { ... }` block can have variables declared inside of it. C being consistent, so can **any** `{ ... }` block.

We illustrate this by taking the guts of the previous "prime number" example and enclosing it in a while loop to find all the primes up to 1000:

```

/*
 * Find the lowest factor of the integers 2 to 1000, or say it is prime.
 * The basic algorithm is to try every possible factor up to the square
 * root of the number being looked at (the target)
 */
#include <stdio.h>
#include <math.h>

int main() {
    int target = 2;

    while (target <= 1000 ) {
        int factor, maxfactor;

        maxfactor = sqrt(target); /* Largest factor we need to try */

        factor = 2;
        while ( target % factor != 0 && factor <= maxfactor )
            factor = factor + 1;

        if ( factor > maxfactor )
            printf("%d is prime\n", target);
        else
            printf("\t%d is not prime, its lowest factor is %d\n",
                target, factor);
        target = target + 1;
    }
    return 0;
}

```

In this example the variables `factor` and `maxfactor` can only be used inside the `{ ... }` block they are declared inside of. Also, **their values are not preserved between iterations of the `while` loop**, they are "new" variables each time.

This example also illustrates that there are no restrictions on how `while` and `if` statements can be nested.

We shall meet C's two other loops, "`for`" and "`do ... while()`", in later lectures, they are essentially variations of the `while()` loop;
