Flow of control in a C program

A program's control flow describes how sequences of statements are executed.

- As we've seen, C programs commence their execution at their main() function, execute their statements, and exit (return the flow of control) to the operating system.
- It's fairly obvious that statements need to be executed in a well-defined order, as we expect programs to always behave the same way (unless some randomness is introduced, as in computer games).
- Default flow of control executes each statement in order, top-to-bottom.
- Programs that only execute from top-to-bottom are pretty boring, and we need to control their flow with a variety of conditional statements and loops.
Conditional execution

Conditional statements first *evaluate* a Boolean condition and then, based on whether it's true or false, execute other statements.

The most common form is:

```c
if(condition) {
    // more statements;
    ....
} else {
    // more statements;
    ....
}
```

Sometimes, the *else* clause is omitted:

```c
if(condition) {
    // more statements;
    ....
}
```

Often, the *else* clause provides further *if* statements:

```c
if(condition1) {
    // more statements;
    ....
} else if(condition2) {
    // more statements;
    ....
} else {
    // more statements;
    ....
}
```

Note that in the examples, above, each *block of statements* to be executed has been written within curly-brackets.

The curly-brackets are *not required* (we could just write a single statement for either *if* or *else* clause). However, adding curly-brackets is considered a *good practice*. They provide a safeguard for when additional statements are added later.
Boolean values

Of significance, and a very common cause of errors in C programs, is that C standards, prior to ISO-C99, had no Boolean datatype.

Historically, an integer value of zero evaluated equivalent to a Boolean value of false; any non-zero integer value evaluated as true.

You may read some older C code:

```c
int initialised = 0; // evaluates to false
....
if(! initialised)
{
    // initialisation statements;
    initialised = 1;
}
```

which may be badly and accidently coded as:

```c
int initialised = 0; // evaluates to false
....
if(initialised = 0)
{
    // initialisation statements;
    initialised = 1;
}
```

In the second example, the conditional test always evaluates to false, as the single equals character requests an assignment, not a comparison.

It is possible (and occasionally reasonable) to perform an assignment as part of a Boolean condition - you'll often see: `while( (nextch = getc(fline)) != EOF ) {...`

Whenever requiring the `true` and `false` constants (introduced in C99), we need to provide the line:

```c
#include <stdbool.h>
```
## Switch statements

When the same (integer) expression is compared against a number of distinct values, it's preferred to evaluate the expression once, and compare it with possible values:

### Cascading if..else..if.. statements:

```c
if(expression == value1) {
    // more statements;
    ...;
} else if(expression == value2) {
    // more statements;
    ...;
} else {
    // more statements;
    ...;
}
```

### The equivalent switch statement:

```c
switch(expression) {
    case value1 :
        // more statements;
        break;
    case value2 :
        // more statements;
        break;
    case value3 :
        // more statements;
        break;
    default :
        // more statements;
        break;
}
```

### Less-common features of the switch statement:

```c
switch(expression) {
    case value1 :
        // handle either value1 or value2
        break;
    case value2 :
        // more statements;
        break;
    case value3 :
        // more statements;
        break;
    // no 'break' statement, drop through
    default :
        // more statements;
        break;
}
```

- Typically the 'expression' is simply an identifier, but it may be arbitrarily complex - such as an arithmetic expression, or a function call.
- The datatype of the 'expression' must be an integer (which includes characters, Booleans, and enumerated types), but it cannot be a real or floating-point datatype.
- The **break** statement at the end of each **case** indicates that we have finished with the 'current' value, and control-flow leaves the **switch** statement. Without a **break** statement, control-flow continues "downwards", flowing into the next **case** branch (even though the expression does not have that **case's** value!).

---

Flow of control in a C program - bounded loops

One of the most powerful features of computers, in general, is to perform thousands, millions, of repetitive tasks quickly. (in fact, one of the first uses of computers in the 1940s was to calculate trigonometric tables for the firing of artillery shells).

C provides its for control statement to loop through a sequence of statements, a block of statements, a known number of times:

The most common form appears below, in which we introduce a loop control variable, i, to count how many times we go through the loop:

```c
// here, variable i holds the values 1,2,....10
for(int i = 1 ; i <= 10 ; i = i+1)
// the above introduced a loop-control variable, i
   printf("loop number %%i\n", i);
// variable i is available down to here
// but variable i is not available from here
```

The loop control variable does not always have to be an integer:

```c
// here, variable ch holds each lowercase value
for(char ch = 'a' ; ch <= 'z' ; ch = ch+1)
   { ...... printf("loop using character '%c'\n", ch); ...... }
```

Notice that in both cases, above, we have introduced new variables, here i and ch, to specifically control the loop.

The variables may be used inside each loop, in the statement block, but then "disappear" once the block is finished (after its bottom curly bracket).

It's also possible to use any other variable as the loop control variable, even if defined outside of the for loop. In general, we'll try to avoid this practice - unless the value of the variable is required outside of the loop.

---

Flow of control in a C program - unbounded loops

The **for** loops that we've just seen should be used when we know, ahead of time, how many times we need to loop (i.e. 10 times, or over the range 'a'..'z'). Such loops are termed **bounded** loops and, unless we've made a silly coding error will always terminate after a fixed number of iterations.

There are also many occasions when we **don't** know, ahead of time, how many iterations may be required. Such occasions require **unbounded** loops.

C provides two types of unbounded loop:

The most common is the **while** loop, where zero or more iterations are made through the loop:

```
#define NLOOPS 20
int i = 1;
int n = 0;
......
while(i <= NLOOPS)
{
    printf("iteration number %i\n", i);
    ......
    i = some_calculation_setting_i;
    n = n + 1;
}
printf("loop was traversed %i times\n", n);
```

Less common is the **do....while** loop, where at least one iteration is made through the loop:

```
#define NLOOPS 20
int i = 1;
int n = 0;
......
do
{
    printf("iteration number %i\n", i);
    ......
    i = some_calculation_setting_i;
    n = n + 1;
} while(i <= NLOOPS);
printf("loop was traversed %i times\n", n);
```

Notice that in both cases we still use a variable, *i*, to control the number of iterations of each loop, and that the changing value of the variable is used to determine if the loop should "keep going".

However, the statements used to modify the control variable may appear almost anywhere in the loops. They provide flexibility, but can also be confusing when loops become several tens or hundreds of lines long.

Notice also that **while**, and **do....while** loops cannot introduce new variables to control their iterations, and so we have to use "more global" variables.
Writing loops within loops

There's a number of occasions when we wish to loop a number of times (and so we use a for loop) and within that loop we wish to perform another loop. While a little confusing, this construct is often quite common. It is termed a nested loop.

```c
#define NRROWS 6
#define NCOLS 4

for(int row = 1 ; row <= NRROWS ; row = row+1) // the 'outer' loop
{   for(int col = 1 ; col <= NCOLS ; col = col+1) // the 'inner' loop
    {printf("%d,%d\n", row, col);  // print row and col as if "coordinates"
     printf("\n"); // finish printing on this line
    }
}
```

The resulting output will be:

```
(1,1) (1,2) (1,3) (1,4)
(2,1) (2,2) (2,3) (2,4)
(3,1) (3,2) (3,3) (3,4)
(4,1) (4,2) (4,3) (4,4)
(5,1) (5,2) (5,3) (5,4)
(6,1) (6,2) (6,3) (6,4)
```

Notice that we have two distinct loop-control variables, row and col.

Each time that the inner loop (col’s loop) starts, col’s value is initialized to 1, and advances to 4 (NCOLS).

As programs become more complex, we will see the need for, and write, all combinations of:

- for loops within for loops,
- while loops within while loops,
- for loops within while loops,
- and so on....
Changing the regular flow of control within loops

There are many occasions when the default flow of control in loops needs to be modified.

Sometimes we need to leave a loop early, using the **break** statement, possibly skipping some iterations and some statements:

```c
for(int i = 1 ; i <= 10 ; i += 1)
{
    // Read an input character from the keyboard
    ....
    if(input_char == 'Q') // Should we quit?
        break;
    ....
} // Come here after the 'break'. i is unavailable
```

Sometimes we need to start the next iteration of a loop, even before executing all statements in the loop:

```c
for(char ch = 'a' ; ch <= 'z' ; ch = ch+1)
{
    if(ch == 'm') // skip over the character 'm'
        continue;
    ....
    statements that will never see ch == 'm'
    ....
} // Break out of the bounded loop.
```

In the first example, we iterate through the loop at most 10 times, each time reading a line of input from the keyboard. If the user indicates they wish to quit, we **break** out of the bounded loop.

In the second example, we wish to perform some work for all lowercase characters, except 'm'. We use **continue** to ignore the following statements, and to start the next loop (with ch == 'y').
The equivalence of bounded and unbounded loops

We should now be able to see that the for, while, and do ... while control flow statements are each closely related.

To fully understand this, however, we need to accept (for now), that the three "pieces" of the for construct, are not always initialization, condition, modification.

More generally, the three pieces may be C expressions - for the moment we'll consider these as C statements which, if they produce a value, the value is often ignored.

The following loops are actually equivalent:

```
for(expression1; expression2; expression3)
{
    statement1;
    ....
}
```

```
expression1;
while(expression2)
{
    statement1;
    ....
    expression3;
}
```

In both cases, we're expecting expression2 to produce a Boolean value, either true or false, as we need that truth value to determine if our loops should "keep going".

You should think about these carefully, perhaps perform some experiments, to determine where control flow really goes when we introduce break and continue statements.
Some unusual loops you will encounter

As you read more C programs, you'll see some statements that look like `for` or `while` loops, but appear to have something missing. In fact, any (or all!) of the 3 “parts” of a `for` loop may be omitted.

For example, the following loop initially sets `i` to 1, and increments it each iteration, but it doesn't have a "middle" conditional test to see if the loop has finished. The missing condition constantly evaluates to true:

```c
for(int i = 1; /* condition is missing */ i = i+1)
{
    ..... 
}
```

Some loops don't even have a loop-control variable, and don't test for their termination. This loop will run forever, until we interrupt or terminate the operating system process running the C program.

We term these infinite loops:

```c
// cryptic - avoid this mechanism
for( ; ; )
{
    ..... 
}
```

```c
#include <stdbool.h>

// clearer - use this mechanism
while( true )
{
    ..... 
}
```

While we often see and write such loops, we don't usually want them to run forever!

We will typically use an enclosed condition and a `break` statement to terminate the loop, either based on some user input, or the state of some calculation.