

# Control Loops

Objectives:

Having read this section you should have an idea about C's:

1. Conditional, or Logical, Expressions as used in program control
2. the do while loop
3. the while loop
4. the for loop

Go With The Flow:

Our programs are getting a bit more sophisticated, but they still lack that essential something that makes a computer so necessary. Exactly what they lack is the most difficult part to describe to a beginner. There are only two great ideas in computing. The first is the variable and you've already met that. The second is flow of control.

When you write a list of instructions for someone to perform you usually expect them to follow the list from the top to the bottom, one at a time. This is the simple default flow of control through a program. The C programs we have written so far use this one-after-another default flow of control.

This is fine and simple, but it limits the running time of any program we can write. Why? Simply because there is a limit to the number of instructions you can write and it doesn't take long for a computer to read though and obey your list. So how is it that we have programs that run for hours on end if need be? The answer is statements that alter the one-after-another order of obeying instructions. Perhaps the most useful is the loop.

Suppose we ask you to display "Hello World!" five times on the screen. Easy! you'd say:

```
#include <stdio.h>
main()
{
    printf("Hello World!\n");
    printf("Hello World!\n");
    printf("Hello World!\n");
    printf("Hello World!\n");
    printf("Hello World!\n");
}
```

Indeed, this does exactly what was asked. But now we up the bet and ask you to do the same job for 100 hellos or, if you're still willing to type that much code, maybe 1,000 Hello World's, 10,000 Hello World's, or whatever it takes you to realise this isn't a sensible method!

What you really need is some way of repeating the printf statements without having to write it out each time. The solution to this problem is the while loop or the do while loop.

The while and do while Loops:

You can repeat any statement using either the while loop:

```
while(condition) compound statement;
```

or the do while loop:

```
do compound statement while(condition);
```

The condition is just a test to control how long you want the compound statement to carry on repeating.

Each line of a C program up to the semicolon is called a statement. The semicolon is the statement's terminator. The braces { and } which have appeared at the beginning and end of our program unit can also be used to group together related declarations and statements into a compound statement or a block.

In the case of the while loop before the compound statement is carried out the condition is checked, and if it is true the statement is obeyed one more time. If the condition turns

out to be false, the looping isn't obeyed and the program moves on to the next statement. So you can see that the instruction really means while something or other is true keep on doing the statement.

In the case of the do while loop it will always execute the code within the loop at least once, since the condition controlling the loop is tested at the bottom of the loop. The do while loop repeats the instruction while the condition is true. If the condition turns out to be false, the looping isn't obeyed and the program moves on to the next statement.

Conditions or Logical Expressions:

The only detail we need to clear up is what the condition (or Logical Expression) can be. How, for example, do we display 100 or 10,000 "Hello World!" messages? The condition can be any test of one value against another. For example:

`a>0`

is true if a contains a value greater than zero;

`b<0`

is true if b contains a value less than zero.

The only complication is that the test for 'something equals something else' uses the character sequence `==` and not `=`. That's right: a test for equality uses two equal-signs, as in `a==0`, while an assignment, as in `a=0`, uses one. This use of the double or single equal sign to mean slightly different things is a cause of many a program bug for beginner and expert alike!

So what about answering the question? What about the 100 "Hello World"s? Well, for the moment we know easily how to produce an infinite number of Hello World!"s using while loop:

```
#include <stdio.h>
main()
{
    while (1 == 1) printf("Hello World!\n");
}
```

and using the do while loop:

```
#include <stdio.h>
main()
{
    do
        printf("Hello World!\n");
    while (1 == 1)
}
```

If you type either of these programs in and run it you will find that your screen fills with a never ending list of "Hello World!"s. Why? Because the condition to keep the repeat going is `( 1 == 1 )`, one equals one in plain English, which is always true! So how do we stop the loop? In some cases it could be by pulling the plug out - but usually you can stop an infinite loop by pressing Ctrl-Break or Ctrl-C.

An infinite loop is sometimes useful - I certainly hope the program controlling the nearest nuclear power station is an infinite loop that never receives a Ctrl-Break signal! Most loops, however, have to stop some time.

To solve our problem of printing 100 "Hello World!"s we need a counter and a test for when that counter reaches 100. A counter is a simple variable that has one added to it each time through the loop, using an instruction like this:

`a=a+1;`

This always confuses beginners, because they aren't used to seeing the variable on both sides of the equal-sign. All this means is that a has one added to it to produce a new value, and this value is stored back in the location called `<B>a`. If you're worried, try thinking about it as:

```
temp = a+1;
a = temp;
```

The two approaches are more or less the same. C is a language where anything that's used often can be said concisely, so it lets you say "add one to a variable" using the shorter notation:

```
++a;
```

The double plus is read "increment a by one". Make sure you know that ++a; and a=a+1; are the same thing because you will often see both in typical C programs.

The increment operator ++ and the equivalent decrement operator --, can be used as either prefix (before the variable) or postfix (after the variable). Note: ++a increments a before using its value; whereas a++ which means use the value in a then increment the value stored in a.

Now it is easy to print "Hello World!" 100 times using the while loop:

```
#include <stdio.h>
main()
{
    int count;
    count=0;
    while (count < 100)
    {
        ++count;
        printf("Hello World!\n");
    }
}
```

[program]

or the do while loop:

```
#include <stdio.h>
main()
{
    int count;
    count=0;
    do
    {
        ++count;
        printf("Hello, World!\n");
    } while (count < 100)
}
```

Note: the use of the { and } to form a compound statement; all statements between the braces will be executed before the loop check is made.

The integer variable count is declared and then set to zero, ready to count the number of times we have gone round the loop. Each time round the loop the value of count is checked against 100. As long as it is less, the loop carries on. Each time the loop carries on, count is incremented and "Hello World!" is printed - so eventually count does reach 100 and the loop stops.

These little programs are just a bit more subtle than you might think. Ask yourself, do they really print exactly 100 times? Ask yourself: what is the final value of count? If you want to make sure you are right change the printf to:

```
printf("count is %d",count);
```

and add a printf after the loop:

```
printf("final value is %d",count);
```

Make sure you understand why you get the results that you do. What would happen if you changed the initial value of count to be one rather than zero?

Looping the Loop:

We have seen that any list of statements enclosed in curly brackets is treated as a single statement, a compound statement. So to repeat a list of statements all you have to do is put them inside a pair of curly brackets as in:

```
while (condition)
{
    statement1;
    statement2;
    statement3;
}
```

which repeats the list while the condition is true. Notice that the statements within the curly brackets have to be terminated by semicolons as usual. Notice also that as the while statement is a complete statement it too has to be terminated by a semi-colon - except for the influence of one other punctuation rule. You never have to follow a right curly bracket with a semi-colon. This rule was introduced to make C look tidier by avoiding things like

```
};};}
```

at the end of a complicated program. You can write the semi-colon after the right bracket if you want to, but most C programmers don't. You can use a compound statement anywhere you can use a single statement.

The for Loop:

The while, and do-while, loop is a completely general way of repeating a section of program over and over again - and you don't really need anything else but... The while loop repeats a list of instructions while some condition or other is true and often you want to repeat something a given number of times.

The traditional solution to this problem is to introduce a variable that is used to count the number of times that a loop has been repeated and use its value in the condition to end the loop. For example, the loop:

```
i=1;
while (i<10)
{
    printf("%d \n",i);
    ++i;
}
```

repeats while i is less than 10. As the ++ operator is used to add one to i each time through the loop you can see that i is a loop counter and eventually it will get bigger than 10, i.e. the loop will end.

The question is how many times does the loop go round? More specifically what values of i is the loop carried out for? If you run this program snippet you will find that it prints 1,2,3... and finishes at 10. That is, the loop repeats 10 times for values of i from 1 to 10. This sort of loop - one that runs from a starting value to a finishing value going up by one each time - is so common that nearly all programming languages provide special commands to implement it. In C this special type of loop can be implemented as a for loop.

```
for ( counter=start_value; counter <= finish_value; ++counter )
    compound statement
```

which is entirely equivalent to:

```
counter=start;
while (couner <= finish)
{
    statements;
    ++counter;
}
```

```
}
```

The condition operator `<=` should be interpreted as less than or equal too. We will be covering all of C's conditions , or logical expressions, in the next section.

For example to print the numbers 1 to 100 you could use:

```
for ( i=1; i <= 100; ++i ) printf("%d \n",i);
```

You can, of course repeat a longer list of instructions simply by using a compound statement.

The C for loop is much more flexible than this simple description. Indeed, many would be horrified at the way we have described the for loop without displaying its true generality, but keep in mind that there is more to come.

In the meantime consider the following program, it does a temperature conversion, but it also introduces one or two new concepts:

- 1.our counter does not have to be incremented (deremented) by 1; we can use any value.
- 2.we can do calculations within the printf statement.

```
#include <stdio.h>

main()
{
    int fahr;

    for ( fahr = 0 ; fahr <= 300 ; fahr = fahr + 20)
        printf("%4d %6.1f\n" , fahr , (5.0/9.0)*(fahr-32));
}
```

and here's another one for you to look at:

```
#include <stdio.h>

main()
{
    int lower , upper , step;
    float fahr , celsius;

    lower = 0 ;
    upper = 300;
    step = 20 ;

    fahr = lower;

    while ( fahr <= upper ) {
        celsius = (5.0 / 9.0) * (fahr - 32.0);
        printf("%4.0f %6.1f\n" , fahr , celsius);
        fahr = fahr + step;
    }
}
```