# Program Looping

In this lesson we will study the constructs in C designed to perform repeated calculations, specifically:

* The **for** statement
* The **while** statement
* The **do** statement
* The **break** statement
* The **continue** statement

# The For Statement

The general format of the **for** construct is as follows:

for (**init\_expression**; **loop\_condition**; **loop\_expression**)

program statement (or statements)

For example:

for (n = 1; n <= 200; n = n + 1)

triangularNumber = triangularNumber + n;

Or another example involving multiple program statements:

for (n = 1; n <= 200; n = n + 1)

{

triangularNumber = triangularNumber + n;

triangularNumber = triangularNumber + 2;

}

The execution of the **for** statement proceeds as follows:

1. The **init\_expression** is evaluated first. This expression sets the initial values before the loop begins. It’s usually used to set the initial value of an index variable.
2. The **loop\_condition** is evaluated. If this condition is not met (FALSE), the loop is immediately terminated. Otherwise (TRUE), execution continues with the program statement contained in the body of the loop.
3. The **program statements** that constitute the body of the loop are evaluated.
4. The **loop\_expression** is evaluated. It is usually used to increment or decrement the index variable.
5. Return to step 2.

The **for** constructs can be nested up to any desired level, just keep in mind that you should indent the statements properly so they can be easily read.

# The for loop variants

The **for** construct admits some syntactic variations:

* **Multiple expressions:** You can include multiple expressions in any of the fields of the **for** loop, and these should be separated by commas.

This is an example with two initial expressions:

for (i = 0, j = 0; i < 10; ++i)

…

Following from the example above, we can add multiple expressions in the loop expression as:

for (i = 0, j = 0; i < 10; ++i, j = j + 1)

…

* **Omitting fields:** You can omit any or all of the fields in a **for** construct.

For example, this is a **for** construct with no initial expression:

for ( ; j != 100; ++j)

…

This is another example where the loop condition is omitted, making it an infinite loop unless certain statements (for example: break, return…) are reached:

for (j = 0; ; j++)

…

* **Declaring variables:** You can declare variables as part of the initial expressions, these variables can only be used within the scope of the **for** construct where they are declared.

An example of the above is:

for (int n = 1; j != 100; ++j)

…

# The while statement

The syntax of the **while** statement is as follows:

while (**expression**)

program statement (or statements)

For example:

while (j <= 5)

j++;

Or another example with multiple statements:

while (j <= 5)

{

i++;

i--;

}

The execution of the **while** loop is as follows:

1. The **expression** is evaluated. If the condition is satisfied (TRUE) the program statements that constitute the loop are executed. Otherwise (FALSE), the loop is immediately terminated.
2. The program statements are executed.
3. Repeat to step 1.

The function that a **while** loop performs can always be performed by an equivalent **for** loop

# The do statement

The two loop constructs explained before perform the test of the conditions before the program statements that constitute the loop are executed. Sometimes is desirable that the test is performed when at the end of the loop. You can achieve this with the **do** construct:

do

program statement (or statements);

while (loop\_expression);

The following steps explain in detail how the **do** construct is executed:

1. The program statements are executed.
2. The **loop\_expression** is evaluated. If the condition is satisfied (TRUE), the program statements continue to be executed. Otherwise (FALSE) the loop finishes.

This loop guarantees that the program statements are at least executed once.

# The break statement

The **break** statement causes the program to immediately exit from the loop it is executing, whether it’s a **for, while or do** loop. Execution continues with the program statements that follow the loop.

If a **break** is executed from within a block of nested loops, only the loop which is currently being executed is terminated. The syntax for this statement is as follows:

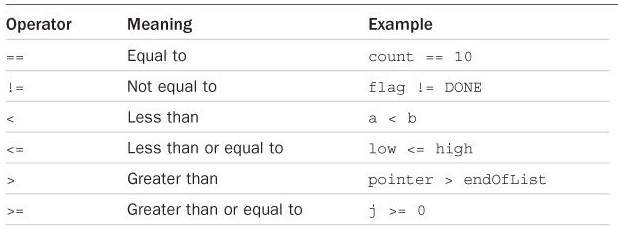
break;

# The continue statement

The **continue** statement causes the loop to which it is executed to be continued, this means, when the continue statement is executed, the statements in the loop that appear after the continue statement are skipped and execution of the loop continues as normal. The syntax is as follows:

continue;

# Relational operators

These are all the **relational operators** that are available in C.

It is important to note that **relational operators have lower precedence that all arithmetic operators.** This means, for example, that the following expression:

a < b + c

Will be evaluated as:

a < (b + c)

# Field width specifications in printf

Given the example below:

printf(“%2i\n”, n);

The number two in this statement is considered as the inclusion of a **field width specification**. This tells the printf routine that the integer to be displayed takes up to two columns. This example is **right-justified,** necessary spaces will be added to the left of the original number.

If the integer only takes one column, then a leading space will be displayed along with the integer in order to fill the two columns. This is similar for other field width specifications.

If the integer to be displayed takes more than two columns, the field width specification is ignored and the routine uses as many columns as needed.

The use of field width specifications is not limited to integers, it can be used with other types.

Another example of **left-justified**, where spaces are added to the right of the original number is as follows:

printf(“%-2i\n”, n);

# Requesting input data from the user

In order to let the user introduce his own information into the program, you can use the following function, very similar to the printf function:

scanf(“%i”, &n);

The scanf needs a format string, in this case “%i”, which tells the system what kind of values are to be read from the terminal, in this case an integer value.

The scanf function also needs the location where the read data will be stored, in this case n, which is provided to the function as a pointer.

Another example, in case you request two integers from the user could be:

scanf(“%i%i”, &n, &v);

# The increment and decrement operator

The **increment operator** adds 1 to its operand, similarly, the **decrement operator** subtracts 1 to its operand. The increment operator is as follows:

++n;

n++;

Similarly the decrement operator is as follows:

--n;

n--;

This is not all about increment and decrement operators, it is important to know when to put the operators before and after the variable name.

# Block of statements and curly braces

In general, any place in a C program where a single statement is permitted, a block of statements can be used, as long as they are enclosed within a pair of curly braces ({}). This makes the block of statements to be considered as a single entity.