Loops

This lesson will teach you about the general concept of loops and will also cover two types of loops: the while loop and the do-while loop.

We'll cover the following

One of the most useful properties of programmable computers is that you can ask them to repeat a calculation or operation many many times, and they will not (usually) complain. The looping constructs in `C` allow us to repeatedly execute a block of code many times, without having to manually re-type or re-list the code by hand.

As a simple example, let’s say we want to compute the cumulative sum of integers between 1 and 100. We could write code like the following:

```
int cumsum = 0;
cumsum = cumsum + 1;
cumsum = cumsum + 2;
cumsum = cumsum + 3;
...
cumsum = cumsum + 100;
```

As you can see, the above code is not very efficient. As a general rule of programming, any time you find yourself copying and pasting code bits many times over, you should be thinking to yourself, there must be a better way, there must be a way to specify this code more abstractly so that it’s more compact and reusable. Here, we can use a for-loop to achieve what we need.

For Loop

Generic for-loop looks like this:

```
for (init_expression; loop_condition; loop_expression) {
    program_statements
}
```

The three expressions inside the round brackets set up the environment for the loop. The `init_expression` is executed before the loop starts and is typically where you define some initial value that will change each time through the loop. The `loop_condition` is an expression that determines whether the loop should continue, or stop. Of course if you don't specify a condition under which the loop should stop, it never will, and you will have an **endless loop** and your program will never terminate. The `loop_expression` specifies code that is executed each time through the loop, **after** the body of the loop is executed. This is all very abstract, so let's see a concrete example, by coding a loop to implement the cumulative sum as above.

```
#include <stdio.h>

int main() {
    int cumsum = 0;
    int i;
    for (i=1; i<=100; i++) {
        cumsum = cumsum + i;
    }
    printf("the cumulative sum up to 100 is %d\n", cumsum);
}
```

Let's go through the code for the for-loop and understand what is happening. On lines 4 and 5 we declare the integer variables `cumsum` and `i`, and we initialize the value of `cumsum` to `0`. On line 6 we declare the for-loop. The first expression in the round brackets is a statement which initializes the value of the `i` variable to be `1`. This is executed before the loop starts. The next expression is a conditional which is `TRUE` if `i` is less than or equal to `100` and is `FALSE` otherwise. This expression is evaluated before each iteration of the loop. When this expression is `FALSE` the loop terminates, otherwise the loop iterates again. Finally the third expression in the round brackets, `i++` is executed **after** each iteration of the loop. In this case, we increment the `i` variable by one.

On line 7 is the code block that forms the body of the loop. This is code that is executed on each iteration of the loop. In this case, we add the current value of `i` (which changes each time through the loop) to the `cumsum` variable. On line 8 the closing bracket `}` specifies the end of the body of the loop. On line 9 we print to the screen the value of the `cumsum` variable.

As you can see, the use of a for-loop enables us to write the code for incrementing the `cumsum` variable just **once**, and place it inside a loop, that is constructed so that it executes that code 100 times, and each time, uses a different value of `i`.

While Loop

The while-loop is another looping construct that you might find more appropriate than a for-loop under some circumstances. The while-loop looks like this:

```
while (conditional_expression) {
   program_statements;
}
```

The while-loop will first check the value of the `conditional_expression`, and if it is **not FALSE** (i.e. if the expression returns a **non-zero** (non-FALSE) result, the `program_statement` (or multiple statements) will be executed once. Afterward, the `conditional_expression` will be evaluated again, and if it returns a non-zero result, the `program_statment` (or multiple statements) will be executed again, and so on. The while-loop will stop **only** when the `conditional_expression` returns **zero**.

Here is a simple program that demonstrates the use of a while-loop:

```
#include <stdio.h>

int main() {
  int cumsum = 0;
  int i = 0;
  while(i <= 100) {
    cumsum = cumsum + i;
    i++;
  }
  printf("the cumulative sum up to 100 is %d\n", cumsum);
}
```

The program asks the user to enter an integer, and then the program prints to the screen the value of the integer multiplied by 10. If the user enters `999`, the program terminates. Enter it into your code editor and compile and run it, to see how it works.

Do-While Loop

There is another version of a while-loop that is essentially the same as a while-loop, but it reverses the order of the `program_statement` and `conditional_expression`:

```
#include <stdio.h>

int main() {
    int number = 1;
    printf("Multiplication Table for 5\n");
    do {
        printf("5 x %d = %d\n", number, 5*number);
        number++;
    }
    while (number <= 10);
}
```

The choice of a for-loop, a while-loop, or a do-while-loop, is up to you and the best one to use may differ depending on circumstance.

Now, we'll take a look at conditional statements, which allow code to be executed only when a certain condition is fulfilled.