Chapter 15: Iteration Statements/Loops: for, while, do-while

Section 15.1: For loop

In order to execute a block of code over an over again, loops comes into the picture. The `for` loop is to be used when a block of code is to executed a fixed number of times. For example, in order to fill an array of size `n` with the user inputs, we need to execute `scanf()` for `n` times.

Version ≥ C99

```
#include <stddef.h>          // for size_t

int array[10];               // array of 10 int

for (size_t i = 0; i < 10; i++)    // i starts at 0 and finishes with 9
{
   scanf("%d", &array[i]);
}
```

In this way the `scanf()` function call is executed `n` times (10 times in our example), but is written only once.

Here, the variable `i` is the loop index, and it is best declared as presented. The type `size_t` (*size type*) should be used for everything that counts or loops through data objects.

This way of declaring variables inside the `for` is only available for compilers that have been updated to the C99 standard. If for some reason you are still stuck with an older compiler you can declare the loop index before the `for` loop:

Version < C99

```
#include <stddef.h>        /* for size_t */
size_t i;
int array[10];             /* array of 10 int */

for (i = 0; i < 10; i++)       /* i starts at 0 and finishes at 9 */
{
   scanf("%d", &array[i]);
}
```

Section 15.2: Loop Unrolling and Duff's Device

Sometimes, the straight forward loop cannot be entirely contained within the loop body. This is because, the loop needs to be primed by some statements ***B***. Then, the iteration begins with some statements ***A***, which are then followed by ***B*** again before looping.

```
do_B();
while (condition) {
    do_A();
    do_B();
}
```

To avoid potential cut/paste problems with repeating ***B*** twice in the code, Duff's Device could be applied to start the loop from the middle of the `while` body, using a switch statement and fall through behavior.

```
switch (true) while (condition) {
case false: do_A(); /* FALL THROUGH */
```

```
default:    do_B(); /* FALL THROUGH */
}
```

Duff's Device was actually invented to implement loop unrolling. Imagine applying a mask to a block of memory, where `n` is a signed integral type with a positive value.

```
do {
    *ptr++ ^= mask;
} while (--n > 0);
```

If `n` were always divisible by 4, you could unroll this easily as:

```
do {
    *ptr++ ^= mask;
    *ptr++ ^= mask;
    *ptr++ ^= mask;
    *ptr++ ^= mask;
} while ((n -= 4) > 0);
```

But, with Duff's Device, the code can follow this unrolling idiom that jumps into the right place in the middle of the loop if `n` is not divisible by 4.

```
switch (n % 4) do {
case 0: *ptr++ ^= mask; /* FALL THROUGH */
case 3: *ptr++ ^= mask; /* FALL THROUGH */
case 2: *ptr++ ^= mask; /* FALL THROUGH */
case 1: *ptr++ ^= mask; /* FALL THROUGH */
} while ((n -= 4) > 0);
```

This kind of manual unrolling is rarely required with modern compilers, since the compiler's optimization engine can unroll loops on the programmer's behalf.

Section 15.3: While loop

A `while` loop is used to execute a piece of code while a condition is true. The `while` loop is to be used when a block of code is to be executed a variable number of times. For example the code shown gets the user input, as long as the user inserts numbers which are not `0`. If the user inserts `0`, the while condition is not true anymore so execution will exit the loop and continue on to any subsequent code:

```
int num = 1;

while (num != 0)
{
    scanf("%d", &num);
}
```

Section 15.4: Do-While loop

Unlike `for` and `while` loops, `do-while` loops check the truth of the condition at the end of the loop, which means the `do` block will execute once, and then check the condition of the `while` at the bottom of the block. Meaning that a `do-while` loop will *always* run at least once.

For example this `do-while` loop will get numbers from user, until the sum of these values is greater than or equal to `50`:

```c
int num, sum;
num = sum = 0;

do
{
  scanf("%d", &num);
  sum += num;

} while (sum < 50);
```

`do-while` loops are relatively rare in most programming styles.

Section 15.5: Structure and flow of control in a for loop

```c
for ([declaration-or-expression]; [expression2]; [expression3])
{
    /* body of the loop */
}
```

In a `for` loop, the loop condition has three expressions, all optional.

- The first expression, `declaration-or-expression`, *initializes* the loop. It is executed exactly once at the beginning of the loop.

`Version ≥ C99`

It can be either a declaration and initialization of a loop variable, or a general expression. If it is a declaration, the scope of the declared variable is restricted by the `for` statement.

`Version < C99`

Historical versions of C only allowed an expression, here, and the declaration of a loop variable had to be placed before the `for`.

- The second expression, `expression2`, is the *test condition*. It is first executed after the initialization. If the condition is **`true`**, then the control enters the body of the loop. If not, it shifts to outside the body of the loop at the end of the loop. Subsequently, this conditon is checked after each execution of the body as well as the update statement. When **`true`**, the control moves back to the beginning of the body of the loop. The condition is usually intended to be a check on the number of times the body of the loop executes. This is the primary way of exiting a loop, the other way being using jump statements.
- The third expression, `expression3`, is the *update statement*. It is executed after each execution of the body of the loop. It is often used to increment a variable keeping count of the number of times the loop body has executed, and this variable is called an *iterator*.

Each instance of execution of the loop body is called an *iteration*.

Example:

`Version ≥ C99`

```c
for(int i = 0; i < 10 ; i++)
{
    printf("%d", i);
}
```

The output is:

```
0123456789
```

In the above example, first `i = 0` is executed, initializing `i`. Then, the condition `i < 10` is checked, which evaluates to be **`true`**. The control enters the body of the loop and the value of `i` is printed. Then, the control shifts to `i++`, updating the value of `i` from 0 to 1. Then, the condition is again checked, and the process continues. This goes on till the value of `i` becomes 10. Then, the condition `i < 10` evaluates **`false`**, after which the control moves out of the loop.

Section 15.6: Infinite Loops

A loop is said to be an *infinite loop* if the control enters but never leaves the body of the loop. This happens when the test condition of the loop never evaluates to **`false`**.

Example:

Version ≥ C99

```
for (int i = 0; i >= 0; )
{
    /* body of the loop where i is not changed*/
}
```

In the above example, the variable `i`, the iterator, is initialized to 0. The test condition is initially **`true`**. However, `i` is not modified anywhere in the body and the update expression is empty. Hence, `i` will remain 0, and the test condition will never evaluate to **`false`**, leading to an infinite loop.

Assuming that there are no jump statements, another way an infinite loop might be formed is by explicitly keeping the condition true:

```
while (true)
{
    /* body of the loop */
}
```

In a `for` loop, the condition statement optional. In this case, the condition is always **`true`** vacuously, leading to an infinite loop.

```
for (;;)
{
    /* body of the loop */
}
```

However, in certain cases, the condition might be kept **`true`** intentionally, with the intention of exiting the loop using a jump statement such as **`break`**.

```
while (true)
{
    /* statements */
    if (condition)
    {
        /* more statements */
        break;
    }
}
```