To print the numbers from 0 to 9, using what we have learnt so far, we can write the program like this:

```
1  int main() {
2      int i = 0;
3      printf("%d\n", i);
4      i++;
5      printf("%d\n", i);
6      i++;
7      printf("%d\n", i);
8      i++;
9      printf("%d\n", i);
10     i++;
11     printf("%d\n", i);
12     i++;
13     printf("%d\n", i);
14     i++;
15     printf("%d\n", i);
16     i++;
17     printf("%d\n", i);
18     i++;
19     printf("%d\n", i);
20     i++;
21     printf("%d\n", i);
22 }
```

Need for loops

Of course no one will write code this way; nor should they. Still let us look at the problems that arise due to this sort of code:

- Increased code size
- which leads to poor readability
- makes changes difficult.

Of course we can imagine more issues of serious and not so serious impact, but the point is that we need a way to repeatedly execute the same code, rather than write the same code multiple times.

In other words, we need mechanism(s) to execute a block of code several number of times. A loop is precisely such a mechanism.

A loop uses a condition to control the repetition. The repetition is done as long as the condition is `true`.

We can describe the general form of a loop as:

- Declare and initialize a variable.
- Check condition to start the loop
- Executing statements inside loop.
- Modify the value of the variable.

Types of Looping Statements

C provides three looping constructs:

while The most general construct. Should be used when you want to repeatedly execute a set of statements based on the condition.

for The counting loop. In C there is no effective difference between **for** and **while**. A **for** loop can be rewritten as a **while** and vice versa. But it is strongly recommended that you use the for only in situatons you want to loop a (known) number of times.

do . . . while The least used construct. Is useful only in very special situation where you want to run the body of the loop at least once.

While Loop

A **while** loop repeatedly executes target statement(s) *as long as* a given condition is true.

Syntax


```
while (boolean_expression) {
  statement(s);
}
```

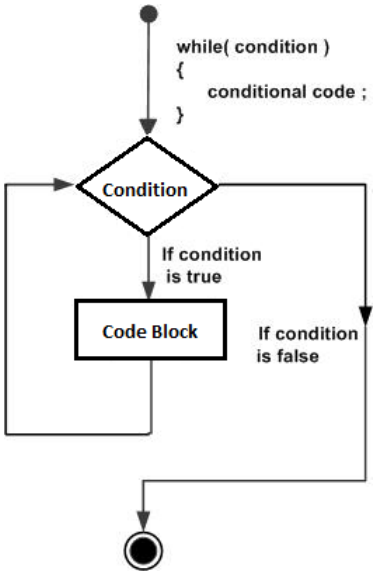

Figure 1: Flow Chart: while

Explanation

- Here, `statement(s)` may be one statement or a block.
- The boolean_expression is checked; if true the statement(s) are executed.
- Then the condition is checked again; the cycle repeats till the condition becomes false.
- When the condition becomes false, program control passes to the line immediately following the loop.
- In `while` loop the statement(s) may not execute even once, if the condition is `false` the first time.

Warning

The condition must change due to the effect of the loop body. Otherwise the loop will NEVER stop. In other words, there must be at least one statement in the loop body that affects the condition.

Example

A program to print numbers from 1 to 10.

```
#include <stdio.h>
int main() {
    int i = 1;
    while (i <= 10) {
        printf("%4d", i);
        i++;
    }
    printf("\n");
    return 0;
}
```

```
$
$ c99 Program-07-1.c
$ ./a.out
   1   2   3   4   5   6   7   8   9  10
$
```

Figure 2: Printing 1-10 in a while loop

Example

A program to print all even numbers within a given range.

```
#include <stdio.h>
int main() {
    int start_val;
    int end_val;

    printf("Enter The Range (Start and End Values):");
    scanf("%d %d", &start_val, &end_val);
    printf("The Even Numbers between %d and %d are:\n",
                            start_val, end_val);
    while (start_val <= end_val) {
```



```
11          if (start_val % 2 == 0)
12              printf("%4d", start_val);
13          start_val++;
14      }
15      printf("\n");
16      return 0;
17  }
```

```
$
$ c99 Program-07-2.c
$ ./a.out
Enter The Range (Start and End Values):12 24
The Even Numbers between 12 and 24 are:
  12  14  16  18  20  22  24
$
```

Figure 3: Printing a range of even numbers

For Statement

A **for** is a looping control structure that allows you to efficiently write a loop that needs to execute a specified number of times.

Syntax

```
for (initialization; condition; increment) {
    statement(s);
}
```

Explanation

Here is the flow of control in a **for** loop

1. The **initialization** step is executed first, and only once.
    - While this can be any executable statement, idiomatically it must be used to declare and initialize the loop control variable(s).

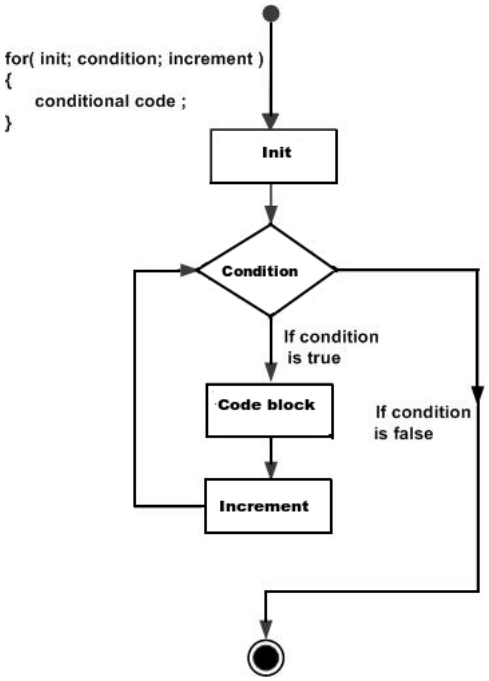

Figure 4: Flowchart: for

- It can even be an empty statement – but there must be a semicolon as the first non-space character inside the parentheses.

2. Next, the **condition** is evaluated. If it is `true`, the body of the loop is executed. If it is `false`, control directly passes to the next statement *after* the **for** loop.
   - In other words, a **for** loop may not execute the body of statements even once.
3. After the body of the **for** loop executes, control is transferred to the **increment** statement.
   - Once again, like the initialization statement, it can be any executable statement or empty. Idiomatically, this statement must modify loop control variable(s), thus controlling how many times the loop is executed.
   - We have called it increment; but it can be a decrement or any other statement that modifies the value of the index variable.
4. The condition is now evaluated again. If it is true, the loop executes and the process repeats itself (body of loop, then increment step, and then condition check). When the condition becomes false, the **for** loop terminates.

Example

Program to print the multiplication table of a given number.

```
#include <stdio.h>
int main() {
    int num;
    printf("Enter The Number: ");
    scanf("%d", &num);
    for (int i = 1; i <= 10; i++) {
        printf ("%2d * %2d = %3d\n", num, i , (num * i));
    }
    return 0;
}
```

```
$
$ c99 Program-07-3.c
$ ./a.out
Enter The Number : 8
 8 *  1 =   8
 8 *  2 =  16
 8 *  3 =  24
 8 *  4 =  32
 8 *  5 =  40
 8 *  6 =  48
 8 *  7 =  56
 8 *  8 =  64
 8 *  9 =  72
 8 * 10 =  80
$
```

Figure 5: Multiplication Table

Example

```
#include <stdio.h>
int main() {
    int limit;
    printf("Enter the size : ");
```



```
5       scanf("%d", &limit);
6       for (int i = 1; i <= limit; i++) {
7           for (int k = i; k <= limit; k++)
8               printf(" ");
9           for (int j = 1; j <= i; j++) {
10              printf("* ");
11          }
12          printf("\n");
13      }
14      return 0;
15  }
```

```
$
$ c99 Program-07-4.c
$ ./a.out
Enter the size : 5
    *
   * *
  * * *
 * * * *
* * * * *
$
```

Figure 6: Building a pattern

Break and Continue

There are two statements available in C, **break** and **continue** to change the control flow of a loop. Loops perform a set of operations repeatedly until certain condition is met but, it is sometimes desirable to skip some statements inside the loop or terminate the loop. In such cases, **break** and **continue** statements are used.

Break Statement

It is used to terminate the loop immediately after a certain condition encountered. This condition is different from the loop's condition. Thus the **break** statement is used with conditional **if** statement.

Example

Program to print the numbers between the given range, terminating the loop if it encounters a number that is a factor of 10.

```
#include <stdio.h>
int main() {
    int start_val;
    int end_val;
    printf("Enter the Start and End value : ");
    scanf("%d %d", &start_val, &end_val);
    while (start_val <= end_val) {
        printf("%4d", start_val);
        if (start_val % 10 == 0) {
            break;
        }
        start_val++;
    }
    printf("\n");
    return 0;
}
```

A natural question arises: 'Why use a separate condition? Why not combine it with loop condition?' We want to print the multiple of 10 *and then exit*. This means we need to check after printing.

More realistically, the condition for deciding on the **break** would depend on the computations made inside the loop and hence the checking has to be done after the relevant lines of code.

Continue Statement

The **continue** statement is used to skip over the rest of the loop iteration and directly jump to the loop control. Will start the next iteration immediately.

Example

Program to print the sum of all odd numbers between the given range.

```
#include <stdio.h>
```



```
$ c99 Program-07-5.c
$ ./a.out
Enter the Start and End value : 10 16
  10
$ ./a.out
Enter the Start and End value : 12 16
  12  13  14  15  16
$ ./a.out
Enter the Start and End value : 12 20
  12  13  14  15  16  17  18  19  20
$ ./a.out
Enter the Start and End value : 12 32
  12  13  14  15  16  17  18  19  20
$
```

Figure 7: Stop print at 10n

```
2  int main() {
3      int start_val;
4      int end_val;
5      int sum_odd = 0;
6      printf("Enter the Start and End values : ");
7      scanf("%d %d", &start_val, &end_val);
8      for (int num = start_val; num <= end_val; num++) {
9          if (num % 2 == 0) {
10             continue;
11         }
12         sum_odd += num;
13     }
14     printf("Sum Of Odd Numbers between %d and %d is %d\n",
15                                 start_val, end_val, sum_odd);
16     return 0;
17 }
```



```
$
$ c99 Program-07-6.c
$ ./a.out
Enter the Start and End values : 12 28
Sum Of Odd Numbers between 12 and 28 is 160
$ ./a.out
Enter the Start and End values : 12 11
Sum Of Odd Numbers between 12 and 11 is 0
$ ./a.out
Enter the Start and End values : 12 12
Sum Of Odd Numbers between 12 and 12 is 0
$ ./a.out
Enter the Start and End values : 13 13
Sum Of Odd Numbers between 13 and 13 is 13
$
```

Figure 8: Odd numbers: using Continue

do . . . while Loop

The **for** and **while** loops test the loop condition at the top of the loop. The do . . . **while** loop in C programming language checks its condition at the bottom of the loop.

A do . . . **while** loop is similar to a **while** loop, except that the body of the do . . . **while** loop is guaranteed to execute at least once.

Syntax

```
do {
    statement(s);
} while (boolean_expression);
```

Example

The following example illustrates how to use the do ... **while** loop.

```
1 #include <stdio.h>
2 int main() {
3     int i = 10;
4     do {
```


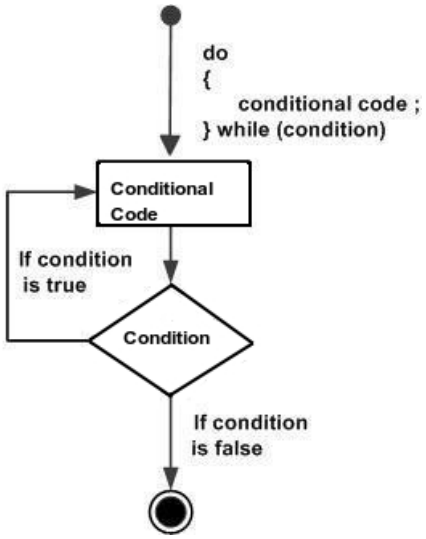

Figure 9: FlowChart: do ... while

```
5           printf("%4d", i);
6           i--;
7       } while (i > 0);
8       printf("\n");
9       return 0;
10  }
```

Figure 10: do while output

```
$
$ c99 Program-07-7.c
$ ./a.out
  10   9   8   7   6   5   4   3   2   1
$
```

Nested Loops

We can use one loop (of any kind) inside another loop (of any kind). This is known as *nested* loop. The syntax for some possible nested loops is given below:


```
for (init; condition1; increment) {
  statement(s);
  while (condition2) {
    statement(s);
  }
  statement(s);
}

do {
  statement(s);
  for (init; condition1; increment) {
    statement(s);
  }
  statement(s);
} while(condition2);
```

