

# Chapter 6

# Loops

## Iteration Statements

- C's iteration statements are used to set up loops.
- A *loop* is a statement whose job is to repeatedly execute some other statement (the *loop body*).
- In C, every loop has a *controlling expression*.
- Each time the loop body is executed (an *iteration* of the loop), the controlling expression is evaluated.
  - If the expression is true (has a value that's not zero) the loop continues to execute.

## Iteration Statements

- C provides three iteration statements:
  - The `while` statement is used for loops whose controlling expression is tested *before* the loop body is executed.
  - The `do` statement is used if the expression is tested *after* the loop body is executed.
  - The `for` statement is convenient for loops that increment or decrement a counting variable.

## The `while` Statement

- Using a `while` statement is the easiest way to set up a loop.
- The `while` statement has the form  
`while ( expression ) statement`
- *expression* is the controlling expression; *statement* is the loop body.

## The `while` Statement

- Example of a `while` statement:

```
while (i < n) /* controlling expression */  
    i = i * 2; /* loop body */
```

- When a `while` statement is executed, the controlling expression is evaluated first.
- If its value is nonzero (true), the loop body is executed and the expression is tested again.
- The process continues until the controlling expression eventually has the value zero.

## The `while` Statement

- A `while` statement that computes the smallest power of 2 that is greater than or equal to a number `n`:

```
i = 1;
while (i < n)
    i = i * 2;
```

- A trace of the loop when `n` has the value 10:

<code>i = 1;</code>	<code>i</code> is now 1.
Is <code>i &lt; n</code> ?	Yes; continue.
<code>i = i * 2;</code>	<code>i</code> is now 2.
Is <code>i &lt; n</code> ?	Yes; continue.
<code>i = i * 2;</code>	<code>i</code> is now 4.
Is <code>i &lt; n</code> ?	Yes; continue.
<code>i = i * 2;</code>	<code>i</code> is now 8.
Is <code>i &lt; n</code> ?	Yes; continue.
<code>i = i * 2;</code>	<code>i</code> is now 16.
Is <code>i &lt; n</code> ?	No; exit from loop.

## The `while` Statement

- Although the loop body must be a single statement, that's merely a technicality.
- If multiple statements are needed, use braces to create a single compound statement:

```
while (i > 0) {  
    printf("T minus %d and counting\n", i);  
    i--;  
}
```

- Some programmers always use braces, even when they're not strictly necessary:

```
while (i < n) {  
    i = i * 2;  
}
```

## The `while` Statement

- The following statements display a series of “countdown” messages:

```
i = 10;
while (i > 0) {
    printf("T minus %d and counting\n", i);
    i--;
}
```

- The final message printed is T minus 1 and counting.



## The `while` Statement

- Observations about the `while` statement:
  - The controlling expression is false when a `while` loop terminates. Thus, when a loop controlled by `i > 0` terminates, `i` must be less than or equal to 0.
  - The body of a `while` loop may not be executed at all, because the controlling expression is tested *before* the body is executed.
  - A `while` statement can often be written in a variety of ways. A more concise version of the countdown loop:

```
while (i > 0)
    printf("T minus %d and counting\n", i--);
```

## Infinite Loops

- A `while` statement won't terminate if the controlling expression always has a nonzero value.
- C programmers sometimes deliberately create an *infinite loop* by using a nonzero constant as the controlling expression:

```
while (1) ...
```

- A `while` statement of this form will execute forever unless its body contains a statement that transfers control out of the loop (`break`, `goto`, `return`) or calls a function that causes the program to terminate.

## Program: Printing a Table of Squares

- The `square.c` program uses a `while` statement to print a table of squares.
- The user specifies the number of entries in the table:

```
This program prints a table of squares.
```

```
Enter number of entries in table: 5
```

```
1      1
2      4
3      9
4     16
5     25
```

### square.c

```
/* Prints a table of squares using a while statement */  
  
#include <stdio.h>  
  
int main(void)  
{  
    int i, n;  
  
    printf("This program prints a table of squares.\n");  
    printf("Enter number of entries in table: ");  
    scanf("%d", &n);  
  
    i = 1;  
    while (i <= n) {  
        printf("%10d%10d\n", i, i * i);  
        i++;  
    }  
  
    return 0;  
}
```

## Program: Summing a Series of Numbers

- The `sum.c` program sums a series of integers entered by the user:

```
This program sums a series of integers.
```

```
Enter integers (0 to terminate): 8 23 71 5 0
```

```
The sum is: 107
```

- The program will need a loop that uses `scanf` to read a number and then adds the number to a running total.

## Chapter 6: Loops

### sum.c

```
/* Sums a series of numbers */

#include <stdio.h>

int main(void)
{
    int n, sum = 0;

    printf("This program sums a series of integers.\n");
    printf("Enter integers (0 to terminate): ");

    scanf("%d", &n);
    while (n != 0) {
        sum += n;
        scanf("%d", &n);
    }
    printf("The sum is: %d\n", sum);

    return 0;
}
```

## The do Statement

- General form of the do statement:  
`do statement while ( expression ) ;`
- When a do statement is executed, the loop body is executed first, then the controlling expression is evaluated.
- If the value of the expression is nonzero, the loop body is executed again and then the expression is evaluated once more.

## The do Statement

- The countdown example rewritten as a do statement:

```
i = 10;
do {
    printf("T minus %d and counting\n", i);
    --i;
} while (i > 0);
```

- The do statement is often indistinguishable from the while statement.
- The only difference is that the body of a do statement is always executed at least once.



## The `do` Statement

- It's a good idea to use braces in *all* `do` statements, whether or not they're needed, because a `do` statement without braces can easily be mistaken for a `while` statement:

```
do
```

```
    printf("T minus %d and counting\n", i--);  
while (i > 0);
```

- A careless reader might think that the word `while` was the beginning of a `while` statement.

## The `for` Statement

- The `for` statement is ideal for loops that have a “counting” variable, but it’s versatile enough to be used for other kinds of loops as well.

- General form of the `for` statement:

```
for ( expr1 ; expr2 ; expr3 ) statement
```

*expr1*, *expr2*, and *expr3* are expressions.

- Example:

```
for (i = 10; i > 0; i--)  
    printf("T minus %d and counting\n", i);
```

## The `for` Statement

- The `for` statement is closely related to the `while` statement.
- Except in a few rare cases, a `for` loop can always be replaced by an equivalent `while` loop:

```
expr1;  
while ( expr2 ) {  
    statement  
    expr3;  
}
```

- *expr1* is an initialization step that's performed only once, before the loop begins to execute.

## The `for` Statement

- *expr2* controls loop termination (the loop continues executing as long as the value of *expr2* is nonzero).
- *expr3* is an operation to be performed at the end of each loop iteration.
- The result when this pattern is applied to the previous `for` loop:

```
i = 10;
while (i > 0) {
    printf("T minus %d and counting\n", i);
    i--;
}
```

## The `for` Statement

- Studying the equivalent `while` statement can help clarify the fine points of a `for` statement.

- For example, what if `i--` is replaced by `--i`?

```
for (i = 10; i > 0; --i)
    printf("T minus %d and counting\n", i);
```

- The equivalent `while` loop shows that the change has no effect on the behavior of the loop:

```
i = 10;
while (i > 0) {
    printf("T minus %d and counting\n", i);
    --i;
}
```

## The `for` Statement

- Since the first and third expressions in a `for` statement are executed as statements, their values are irrelevant—they're useful only for their side effects.
- Consequently, these two expressions are usually assignments or increment/decrement expressions.

## for Statement Idioms

- The `for` statement is usually the best choice for loops that “count up” (increment a variable) or “count down” (decrement a variable).
- A `for` statement that counts up or down a total of  $n$  times will usually have one of the following forms:

***Counting up from 0 to  $n-1$ :***      `for (i = 0; i < n; i++) ...`

***Counting up from 1 to  $n$ :***      `for (i = 1; i <= n; i++) ...`

***Counting down from  $n-1$  to 0:***      `for (i = n - 1; i >= 0; i--) ...`

***Counting down from  $n$  to 1:***      `for (i = n; i > 0; i--) ...`

## for Statement Idioms

- Common `for` statement errors:
  - Using `<` instead of `>` (or vice versa) in the controlling expression. “Counting up” loops should use the `<` or `<=` operator. “Counting down” loops should use `>` or `>=`.
  - Using `==` in the controlling expression instead of `<`, `<=`, `>`, or `>=`.
  - “Off-by-one” errors such as writing the controlling expression as `i <= n` instead of `i < n`.



## Omitting Expressions in a `for` Statement

- C allows any or all of the expressions that control a `for` statement to be omitted.
- If the *first* expression is omitted, no initialization is performed before the loop is executed:

```
i = 10;
for (; i > 0; --i)
    printf("T minus %d and counting\n", i);
```

- If the *third* expression is omitted, the loop body is responsible for ensuring that the value of the second expression eventually becomes false:

```
for (i = 10; i > 0;)
    printf("T minus %d and counting\n", i--);
```

## Omitting Expressions in a `for` Statement

- When the *first* and *third* expressions are both omitted, the resulting loop is nothing more than a `while` statement in disguise:

```
for (; i > 0;)
    printf("T minus %d and counting\n", i--);
```

is the same as

```
while (i > 0)
    printf("T minus %d and counting\n", i--);
```

- The `while` version is clearer and therefore preferable.

## Omitting Expressions in a `for` Statement

- If the *second* expression is missing, it defaults to a true value, so the `for` statement doesn't terminate (unless stopped in some other fashion).
- For example, some programmers use the following `for` statement to establish an infinite loop:

```
for ( ; ; ) ...
```

## The Comma Operator

- On occasion, a `for` statement may need to have two (or more) initialization expressions or one that increments several variables each time through the loop.
- This effect can be accomplished by using a *comma expression* as the first or third expression in the `for` statement.
- A comma expression has the form  
*expr1* , *expr2*  
where *expr1* and *expr2* are any two expressions.

## The Comma Operator

- A comma expression is evaluated in two steps:
  - First, *expr1* is evaluated and its value discarded.
  - Second, *expr2* is evaluated; its value is the value of the entire expression.
- Evaluating *expr1* should always have a side effect; if it doesn't, then *expr1* serves no purpose.
- When the comma expression  $++i, i + j$  is evaluated,  $i$  is first incremented, then  $i + j$  is evaluated.
  - If  $i$  and  $j$  have the values 1 and 5, respectively, the value of the expression will be 7, and  $i$  will be incremented to 2.

## The Comma Operator

- The comma operator is left associative, so the compiler interprets

$i = 1, j = 2, k = i + j$

as

$((i = 1), (j = 2)), (k = (i + j))$

- Since the left operand in a comma expression is evaluated before the right operand, the assignments  $i = 1$ ,  $j = 2$ , and  $k = i + j$  will be performed from left to right.

## The Comma Operator

- The comma operator makes it possible to “glue” two expressions together to form a single expression.
- Certain macro definitions can benefit from the comma operator.
- The `for` statement is the only other place where the comma operator is likely to be found.

- Example:

```
for (sum = 0, i = 1; i <= N; i++)  
    sum += i;
```

- With additional commas, the `for` statement could initialize more than two variables.

## Program: Printing a Table of Squares (Revisited)

- The `square.c` program (Section 6.1) can be improved by converting its `while` loop to a `for` loop.



### square2.c

```
/* Prints a table of squares using a for statement */

#include <stdio.h>

int main(void)
{
    int i, n;

    printf("This program prints a table of squares.\n");
    printf("Enter number of entries in table: ");
    scanf("%d", &n);

    for (i = 1; i <= n; i++)
        printf("%10d%10d\n", i, i * i);

    return 0;
}
```

## Exiting from a Loop

- The normal exit point for a loop is at the beginning (as in a `while` or `for` statement) or at the end (the `do` statement).
- Using the `break` statement, it's possible to write a loop with an exit point in the middle or a loop with more than one exit point.

## The `break` Statement

- The `break` statement can transfer control out of a `switch` statement, but it can also be used to jump out of a `while`, `do`, or `for` loop.
- A loop that checks whether a number `n` is prime can use a `break` statement to terminate the loop as soon as a divisor is found:

```
for (d = 2; d < n; d++)  
    if (n % d == 0)  
        break;
```

## The `break` Statement

- After the loop has terminated, an `if` statement can be used to determine whether termination was premature (hence `n` isn't prime) or normal (`n` is prime):

```
if (d < n)
    printf("%d is divisible by %d\n", n, d);
else
    printf("%d is prime\n", n);
```

## The `break` Statement

- The `break` statement is particularly useful for writing loops in which the exit point is in the middle of the body rather than at the beginning or end.
- Loops that read user input, terminating when a particular value is entered, often fall into this category:

```
for (;;) {  
    printf("Enter a number (enter 0 to stop): ");  
    scanf("%d", &n);  
    if (n == 0)  
        break;  
    printf("%d cubed is %d\n", n, n * n * n);  
}
```

## The `break` Statement

- A `break` statement transfers control out of the innermost enclosing `while`, `do`, `for`, or `switch`.
- When these statements are nested, the `break` statement can escape only one level of nesting.

- Example:

```
while (...) {  
    switch (...) {  
        ...  
        break;  
        ...  
    }  
}
```

- `break` transfers control out of the `switch` statement, but not out of the `while` loop.

## The `continue` Statement

- The `continue` statement is similar to `break`:
  - `break` transfers control just past the end of a loop.
  - `continue` transfers control to a point just before the end of the loop body.
- With `break`, control leaves the loop; with `continue`, control remains inside the loop.
- There's another difference between `break` and `continue`: `break` can be used in `switch` statements and loops (`while`, `do`, and `for`), whereas `continue` is limited to loops.

## The `continue` Statement

- A loop that uses the `continue` statement:

```
n = 0;
sum = 0;
while (n < 10) {
    scanf("%d", &i);
    if (i == 0)
        continue;
    sum += i;
    n++;
    /* continue jumps to here */
}
```



## The `continue` Statement

- The same loop written without using `continue`:

```
n = 0;
sum = 0;
while (n < 10) {
    scanf("%d", &i);
    if (i != 0) {
        sum += i;
        n++;
    }
}
```

## The goto Statement

- The `goto` statement is capable of jumping to any statement in a function, provided that the statement has a *label*.

- A label is just an identifier placed at the beginning of a statement:

*identifier* : *statement*

- A statement may have more than one label.
- The `goto` statement itself has the form

`goto identifier ;`

- Executing the statement `goto L;` transfers control to the statement that follows the label *L*, which must be in the same function as the `goto` statement itself.

## The goto Statement

- If C didn't have a `break` statement, a `goto` statement could be used to exit from a loop:

```
for (d = 2; d < n; d++)
    if (n % d == 0)
        goto done;
done:
if (d < n)
    printf("%d is divisible by %d\n", n, d);
else
    printf("%d is prime\n", n);
```

## The goto Statement

- The `goto` statement is rarely needed in everyday C programming.
- The `break`, `continue`, and `return` statements—which are essentially restricted `goto` statements—and the `exit` function are sufficient to handle most situations that might require a `goto` in other languages.
- Nonetheless, the `goto` statement can be helpful once in a while.

## The goto Statement

- Consider the problem of exiting a loop from within a `switch` statement.
- The `break` statement doesn't have the desired effect: it exits from the `switch`, but not from the loop.
- A `goto` statement solves the problem:

```
while (...) {
    switch (...) {
        ...
        goto loop_done;    /* break won't work here */
        ...
    }
}
loop_done: ...
```

- The `goto` statement is also useful for exiting from nested loops.

## The Null Statement

- Accidentally putting a semicolon after the parentheses in an `if`, `while`, or `for` statement creates a null statement.

- Example 1:

```
if (d == 0);                                /*** WRONG ***/  
    printf("Error: Division by zero\n");
```

The call of `printf` isn't inside the `if` statement, so it's performed regardless of whether `d` is equal to 0.

- Example 2:

```
i = 10;  
while (i > 0);                                /*** WRONG ***/  
{  
    printf("T minus %d and counting\n", i);  
    --i;  
}
```

The extra semicolon creates an infinite loop.

## The Null Statement

- Example 3:

```
i = 11;
while (--i > 0);           /*** WRONG ***/
    printf("T minus %d and counting\n", i);
```

The printf statement is executed only once; the message printed is:

```
T minus 0 and counting
```

- Example 4:

```
for (i = 10; i > 0; i--);   /*** WRONG ***/
    printf("T minus %d and counting\n", i);
```

Again, the loop body is executed only once, and the same message is printed as in Example 3.