Iteration / Loops - do-while

CSC100 / Introduction to programming in C/C++ - Spring 2025

Marcus Birkenkrahe

March 29, 2025

Contents

1 README

- This script introduces C looping structures.
- This section is based on chapter 4 in Davenport/Vine (2015) and chapter 6 in King (2008).
- Practice workbooks, input files and PDF solution files in GitHub

2 Loops

- A **loop** is a statement whose job is to repeatedly execute over some other statement (the **loop body**).

- 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 is non-zero), the loop continues to execute.
- C provides three iteration statements: `while`, `do`, and `for`

3 The `do` statement

- The `do` statement has the general form

  ```
  do /statement/ while ( /expression/ ) ;
  ```

- It's like a `while` statement whose controlling expression is tested *after* each execution of the loop body.
- When a `do` statement is executed, the loop body is executed first, (at least once), and then the controlling *expression* is evaluated.
- If the value of the *expression* is non-zero, the loop body is executed again and the expression is evaluated once more.
- Execution of the `do` statement terminates when the controlling *expression* has the value `0` (*FALSE*) **after** the loop body has been executed (or if you abort the execution with `break`).
- Always use braces `{...}` around *all* `do` statements, because otherwise it can be mistaken for a `while` statement.

4 Simple example

- How many times will this loop print "hello"

```
#define N 5
int i = 1;
do {
  puts("hello");
  i++;
 } while (i<N);
```

```
hello
hello
hello
hello
```

- Analyse using pseudocode:

```
#define N 5 // SET N to 5
int i = 1; // SET i to one
do {  // REPEAT
  puts("hello"); // PRINT "hello"
  i++;  // ADD 1 to i
 } while (i<N); // UNTIL i is less than N
```

```
hello
hello
hello
hello
```

- To clarify, print the loop counter:

```
#define N 5 // SET N to 5
int i = 0; // SET i to zero
do {  // REPEAT
  printf("hello %d\n",i++); // PRINT "hello" and ADD 1 to i
 } while (i<N); // UNTIL i is less than N
```

```
hello 0
hello 1
hello 2
hello 3
hello 4
```

5 Example: Calculating the number of digits in an integer

- `do` is handy for loops that must execute at least once.

- Let's write a program that calculates the number of digits in an integer entered by the user.
- Sample output:

```
Enter a nonnegative integer: 656
The number has 3 digits(s).
```

- Our strategy ("algorithm"): *'digits'* correspond to 'base 10' - if we divide the input by 10 repeatedly until it becomes 0 (via integer truncation), the number of divisions performed is the number of digits.

```
656 / 10 => 65 (remainder 6/10)
65  / 10 => 6  (remainder 5/10)
6   / 10 => 0  (remainder 6/10)
```

- In code:

```
printf("656 / 10 = %d (remainder %d/10)\n", 656/10, 656%10);
printf("65 / 10 = %d (remainder %d/10)\n", 65/10, 65%10);
printf("6 / 10 = %d (remainder %d/10)\n", 6/10, 6%10);


656 / 10 = 65 (remainder 6/10)
65 / 10 = 6 (remainder 5/10)
6 / 10 = 0 (remainder 6/10)
```

- Starter code: onecompiler.com/c/43d9uyzy7

```
// Input: non-negative integer n
  // Output: number of digits of n

  // SET digits to zero
  // SET input n

  // PRINT "Enter a non-negative integer"
  // GET input n
  // PRINT input n

  // REPEAT
```

```
    // divide input n by 10
    // add result to digits
  // UNTIL n is greater than 0

  // PRINT "The number has n digit(s)."
```

- Code solution:

```
// Input: non-negative integer n
// Output: number of digits of n

// SET digits to zero
int digits=0;
// SET input n
int n;

// PRINT "Enter a non-negative integer"
printf("Enter a non-negative integer: ");
// GET input n
scanf("%d",&n);
// PRINT input n
printf("%d\n",n);

do { // REPEAT
  // divide input n by 10
  n /= 10;
  // add result to digits
  digits++;
 } while (n>0);  // UNTIL n is greater than 0

// PRINT "The number has n digit(s)."
printf("The number has %d digit(s).\n",digits);


Enter a non-negative integer: 656
The number has 3 digit(s).
```

- Sample input:

```
echo 656 > ../data/dowhile
cat ../data/dowhile
```

- It is important to check **edge cases**. Why does the code **not** work for the input `6565656565`?
- Answer:

```
puts("Does 6565656565 lead to integer overflow?");
if (6565656565 > 2147483647) puts("Yes");
else puts("No");
```

```
Does 6565656565 lead to integer overflow?
Yes
```

> `int` is actually a so-called *signed integer*, a 32-bit datum that encodes integers in the range `[-2147483647,2147483647]`. Any integer larger than this will not work - we have to use long integer types (`long int`) and a different conversion specifier (`%ld`).

- Test: Modify the program so that it runs for long integers, too (bonus exercise).

6 Counting down

- Do you remember how to count down from 10 to 1 using `while`?

```
int i = 10;
while(i>0) printf("%d ",i--);
```

```
10 9 8 7 6 5 4 3 2 1
```

- Challenge: Turn this loop into a `do` loop.
  1. Pseudocode first (to understand how the logic changes)
  2. Code the long version first. Test.
  3. Code the concise version last.
- Solution:
  1. Pseudocode for `while`:

```
// SET loop counter to 10
// WHILE i is greater than 0
   // PRINT i
   // SUBTRACT 1 from i
// END WHILE
```

Pseudocode for `do`:

```
// SET loop counter to 10
// REPEAT
   // PRINT i
   // SUBTRACT 1 from i
// UNTIL i is greater than 0
```

2. Code: long version

```
int i = 10; // SET loop counter to 10
do {// REPEAT
  printf("%d ",i); // PRINT i
  i--; // SUBTRACT 1 from i
 } while (i > 0); // UNTIL i is greater than 0

10 9 8 7 6 5 4 3 2 1
```

3. Code: concise version

```
int i = 10;
do printf("%d ",i--);
while (i > 0);

10 9 8 7 6 5 4 3 2 1
```

7 Summing numbers

- How would you sum up a sequence of positive integers using `do`?
- The `while` pseudocode (home assignment):

```
BEGIN
    DECLARE sum = 0
    DECLARE num

    PRINT "Enter integers (0 to terminate). "
```

```
    READ num
    PRINT num

    WHILE num NOT equal to 0 DO
        ADD num to sum
        READ num
        PRINT num
    END WHILE

    PRINT "The sum is " + sum
END
```

- How would the `do` pseudocode look like? Think first what the main difference between `while` and `do` loops is.

```
BEGIN
    DECLARE sum = 0
    DECLARE num = 0

    PRINT "Enter integers (0 to terminate)."

    REPEAT
        READ num
        PRINT num
        ADD num to sum
    UNTIL num is NOT zero

    PRINT "The sum is " + sum
END
```

- What are the changes?

    1. Don't need to initialize `num` anymore.
    2. `do` loop will run at least once.
    3. During the first run, get the first number.
    4. Afterwards, add the number to the sum.

- Implementing this is another bonus assignment!