

1.40 Iteration (loops)

While, for, and do-while loops

Break statement

Continue statement

EE108 – Computing for Engineers

1

Overview

2

Aims

- ☐ Learn to use iteration (loops) correctly and choose the best approach for the problem at hand

Learning outcomes – you should be able to...

- ☐ Use the while statement
- ☐ Use the for statement
- ☐ Use the do-while statement
- ☐ Use the break and continue statements appropriately

19 October 2020

2

Iteration and the superloop architecture

3

Embedded systems often use **iteration** (looping) extensively

At the highest level, the embedded system architecture used in this module is the **superloop architecture**

- In the superloop architecture, there is one main loop in the programme which runs over and over again.
 - Within the main loop body the programme typically checks for input, does some processing, and finally generates some output
 - In Arduino C the code you write in the “loop” function body is what gets executed repeatedly in the main loop. (Behind the scenes there is for-loop in whose body the **loop** function is constantly being called.)
- The main loop is called the superloop only because it defines the high level flow of the programme
- Although our programmes to date have avoided it, most interesting programmes will contain other loops nested within the superloop body

19 October 2020

3

Iteration

4

A loop repeatedly executes a statement or block (called the **loop body**) until the loop terminates

A loop has a **controlling expression**

- The controlling expression is evaluated once for each execution of the loop body (called **an iteration** of the loop)

The value of the controlling expression determines whether the loop repeats or terminates

- If the controlling expression evaluates true, the loop repeats
- If it evaluates false, the loop terminates (i.e. completes) and the statements which come after the loop body in the programme begin to be executed

19 October 2020

4

The main loop related statements

5

while

- Used for loops whose **controlling expression is evaluated before the loop body** is executed

do-while

- Used for loops whose **controlling expression is evaluated after the loop body** is executed

for

- The for-loop is largely (*) just a more succinct version of a while-loop used when counting, iterating over arrays, etc.
- It allows the initialization of a counting variable and the update of that variable (usually a simple increment/decrement) to be written together rather than separately as for the while statement

** The point at which the update expression is evaluated (e.g. to increment the counting variable) is different in a for and a while loop so that when using the continue statement, a for-loop might behave differently to a similar while loop (see later slide)*

19 October 2020

5

The while loop

6

The while loop has the following syntax

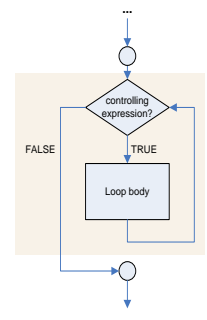
```
while ( controllingExpression )  
    loopBody
```

It works as follows:

1. The **controlling expression is evaluated before executing the loop body**
2. If the expression evaluates to true (non-zero) then the loop body is executed exactly once, otherwise go to step 4
3. When the end of the loop body is reached, go to step 1
4. If and when the controlling expression evaluates to false (zero) the loop terminates and the statements that follow the loop body are executed next

Things to note

- The loop body may be a single statement or a block surrounded by braces
- the loop body will not be executed even once if the controlling expression evaluates to false the first time it is checked



19 October 2020

6

While loop examples

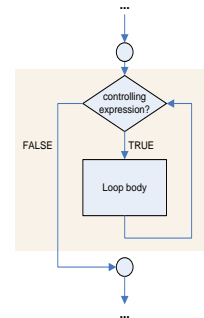
7

```
// calculate x to power of n, n>1

// initialize (previously declared) variables used by the loop
result = x;
i = 2;

while (i <= n) { // whether the loop body executes depends
                // whether i <= n is true or false
    result *= x;
    i++;        // update variable used in the loop expression
}

Serial.print("x to power of n is");
Serial.println(result);
```



Note: the current value of *i* in the controlling expression is first evaluated and used in the controlling expression. The value of *i* is then changed by side effect of the `--` operator – this is a common while loop idiom in C

```
i = NUM_ITERATIONS;
while (i-- > 0) { // when i is 0 this loop expression will evaluate to false,
                // and the loop will terminate
    Serial.print("iterations left: ");
    Serial.println(i); // compare this line with do-while version later.
}                    // what value will be printed?
Serial.println(i);    // what value will be printed here?
```

19 October 2020

7

Guidelines (rules) for well behaved loops

8

Initialize loop variables before the loop

- Variables must be initialized first before they are used in the controlling expression (e.g. `i=0`)

Ensure that the loop can terminate (unless you want an infinite loop)

- Ensure that the controlling expression can evaluate false under some condition – this will depend on how and when the variables affecting the controlling expression can change (see next point)

Update the variable(s) that affect the controlling expression, e.g.

- You might modify the variable by side effect in the controlling expression
- You might explicitly modify the variable in the loop body (or the update-expression of a for-loop)
- You might read a hardware supplied value that you know can change under some circumstances
 - E.g. `digitalRead(SW2_PIN)` will change depending on whether the SW2 button is open or closed/pressed.
- Most commonly, you would modify a variable used in the controlling expression on every iteration. You can however change only on every Nth loop or some other condition being true

```
while (i < 10) {
    ...
    i++;
}
```

```
while (i++ < 10) {
    ...
}
```

```
while (i < 10) {
    ...
    if (++j == 5) {
        j = 0;
        i++;
    }
}
```

19 October 2020

8

Contd – rules in practice

9

```
// calculate value to power of n, n>1

// GUIDELINE 1: initialize variables used in loop
result = x;
i = 2;

// GUIDELINE 2: ensure the controlling expression can evaluate false
// (it will eval to false when i > n)
while (i <= n) {
    result *= x;

    // GUIDELINE 3: ensure variables affecting the controlling expression are
    // updated at some point in loop body or by side effect of the controlling
    // expression itself. (Here, the variable is changed at the end of the loop body)
    i++;
}
```

19 October 2020

9

The do-while loop

10

The do-while loop has the following syntax

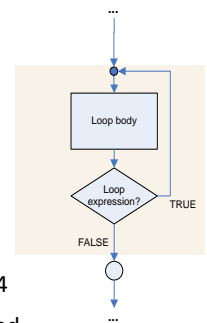
```
do
    loopBody
while ( controlling Expression ) ;
```

It works as follows:

1. Execute the loop body once
2. **Evaluate the controlling expression after executing the loop body**
3. If the expression evaluates to true (non-zero) then go to step 1, otherwise go to step 4
4. If and when the controlling expression evaluates to false (zero) the loop terminates and the statements following the loop are executed

Things to note

- **A do-while loop body will always be executed at least once**, even when controlling expression is initially false
- always use braces around loop body in do-while – it is more readable and less error prone



19 October 2020

10

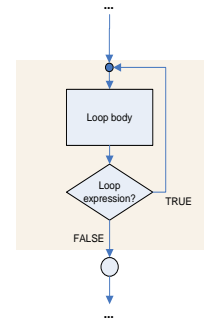
do-while example

11

```
...
// print analog value once and repeat every second
// as long as button is pressed
do {
    switchState = digitalRead(SW1_PIN);
    potValue = analogRead(POTENTIOMETER_PIN);

    Serial.print("analogue value is ");
    Serial.println(potValue);
    delay(1000); // delay 1000 ms
} while (switchState == SW_CLOSED);
```

```
// compare to while-loop and for-loop versions to see how the value
// within the loop body and number of iterations differs
i = NUM_ITERATIONS;
do {
    Serial.print("iterations left: ");
    Serial.println(i); // compare what value will be printed with the
                      // while-loop and for-loop versions
} while ( i-- ); // if i is currently 0 this expression will evaluate
                // to false (and then decrement i before the next statement)
Serial.println(i); // Q: what value will be printed here?
```



19 October 2020

11

The for loop

12

The for loop has the following syntax

```
for ( initExpr ; controllingExpression ; updateExpr )
    loopBody
```

It works as follows:

1. First *initExpr* is evaluated. Any valid expression may be used and its value is discarded, only the side effects are important. Usually this expression assigns an initial value to a variable used in the *controlling expression*.
2. Like the while loop, **the controlling expression is evaluated before executing the loop body**
3. If the expression evaluates to true (non-zero) then the loop body is executed once, otherwise go to step 6
4. Unlike a while-loop, when the end of the loop body is reached, the *updateExpr* is evaluated. The value of the *updateExpr* is discarded and only the side effects are relevant. Usually *updateExpr* just updates the value of a variable used in the *controllingExpression*.
5. Go to step 2
6. If and when the controlling expression evaluates to false (zero) the loop terminates and the statements following the loop body are executed

Things to note

- Any combination of *initExpr*, *controllingExpr*, or *updateExpr* may be empty!

19 October 2020

12

The for loop

13

```
// calculate value to power of n, n>1

// compare this to previously shown while-loop version
result = x;

for (i = 2; i <= n; i++)
    result *= x;

Serial.print("value to power of n is");
Serial.println(result);
```

```
// compare to while-loop and do-while-loop versions

for (i = NUM_ITERATIONS; i > 0; i--) { // when i reaches 0 loop will terminate
    Serial.print("iterations left is ");
    Serial.println(i); // compare what value will be printed with the
                       // while-loop and do-while versions
}
Serial.println(i); // what value will be printed here?
```

19 October 2020

13

Contd.

14

Note: all the following examples have exactly the same effect...

```
// empty initExpr
int i = 2;

for ( ; i <= n; i++)
    ...
```

```
// empty initExpr and updateExpr
int i = 2;

for ( ; i <= n; ) {
    ...
    i++; // NOTE: variable in controlling expr must still be updated, so do it here
}
```

```
// empty controllingExpression (for indeterminately long loops)
// (pretty uncommon thing to do)
int i;

for (i = 2 ; ; i++) {
    ...
    if (i > n)
        break; // we'll see the break statement in more detail shortly
}
```

19 October 2020

14

A for loop can generally be converted into an equivalent while loop (and vice versa)

```
for (initExpr;
    controllingExpression;
    updateExpr)
    loopBody
```



```
initExpr;
while ( controllingExpression ) {
    loopBody
    updateExpr;
}
```

```
// blink LED (on and off) 10 times
for (i = 10; i > 0; i--) {
    digitalWrite(LED1, HIGH);
    delay(100); // delay 100 ms
    digitalWrite(LED1, LOW);
    delay(100); // delay 100 ms
}
```



```
// blink LED (on and off) 10 times
i = 10;
while (i > 0) {
    digitalWrite(LED1, HIGH);
    delay(100); // delay 100 ms
    digitalWrite(LED1, LOW);
    delay(100); // delay 100 ms
    i--;
}
```

Self test questions

Q. Write a loop that simulates tossing a coin and terminates when heads have come up 10 times

```
unsigned int numHeads;

// to simulate a coin toss, use the expression (random(2) == 0)
// which we'll consider to be heads when true and tails when false
...
// TODO - toss 10 heads (may require more than 10 tosses!)
```


Self test questions

17

Q. Calculate the number of bits that are 1 in the variable “value” using (a) a while-loop and (b) a for loop. (For example, binary 0011 0001 has three bits that are 1).

*(Hint: use the arduino function **bitRead**(value, bitIndex) to check each bit in value. The parameter value is the value to be checked and bitIndex identifies which binary digit to check, with index 0 being the least significant bit)*

```
unsigned char value;  
int numOnes;  
...  
// TODO - calculate number of bits which are 1 in value
```

19 October 2020

17

Self test questions

18

Q. A simple way to communicate a value between computer systems is by transmitting bits one after the other and encoding each bit as a voltage on an output pin (or an LED on/off value for optical communications). The voltage or LED state must be held for a short time – e.g. at 1000 bits per second, the duration of each bit must be 1 ms.

Use a loop to iterate over the bits in the variable value (starting at the MSB); set LED1 high or low according to the bit value and hold the value for one bit duration (1 ms).

(Hint: use a loop and the bitRead function to test the individual bit values)

```
const int BIT_MS = 1;  
unsigned char value;  
...  
// TODO - transmit bits of value (MSB first) via LED1
```

19 October 2020

18

Infinite loops

19

An infinite loop is a loop that will repeat indefinitely

- It normally occurs when the controlling expression always evaluates true
- In the case of a for-loop it also occurs if there is no controlling expression specified
- Traditionally we don't use a do-while loop for infinite loops

We use infinite loops when we want to repeat forever (e.g. for the superloop in embedded system programmes)

```
// all of the following mean loop
// forever, i.e. infinite loop

while (true) { // preferred version
  ...
}

while (1) {
  ...
}

while (47) { // don't do this
  ...
}
```

```
// loop forever (i.e. infinite loop)
// because there is no controlling
// expression

for (;;) {
  ...
}
```

19 October 2020

19

busy-wait loops

20

In embedded systems there are two main reasons that it is useful to write loops which do nothing useful in the loop body

- To implement a “busy waiting” delay—this implements a delay simply as a result of the processor spending time to evaluate the controlling expression a (usually large) number of times
- To [busy] wait for some specific event or condition to happen

To write a loop body which does nothing use one of the following idioms

- Write the loop body using the **null statement** which is simply a semicolon on its own
- Write the loop body using the **continue statement** which is more readable and has the same effect (since the loop body is otherwise empty)

```
// e.g. using the null statement
// 1. A simple delay
for (i=0; i<10000; i++)
  ; // the null statement

// 2. busy waiting for something
while (digitalRead(SW1_PIN) == LOW)
  ; // the null statement
```



Prefer this more readable idiom

```
// e.g. using the continue statement
// 1. simple A delay
for (i=0; i<10000; i++)
  continue;

// 2. busy waiting for something
while (digitalRead(SW1_PIN) == LOW)
  continue;
```

19 October 2020

20

Nested loops

21

It is perfectly reasonable to nest loops within one another.

A nested (or inner) loop is treated like any other statement that appears within the body of the enclosing (outer) loop.

This means that the outer loop body only progresses when the inner loop runs to completion. (And the inner loop must run to completion on every iteration of the outer loop.)

```
// blink an LED the correct number of times for the digit
for (digit = 0; digit < 10; digit++) { // iterate over digits
  Serial.print("digit is ");
  Serial.println(digit);

  // nested loop -- count the number of blinks to show
  for (blink = 0; blink < digit; blink++) {
    digitalWrite(LED1, HIGH);
    delay(100);
    digitalWrite(LED2, LOW);
    delay(100);
  }

  delay(1000); // delay 1 second
}
```

19 October 2020

21

Break and continue

22

break

- We've previously seen break used with switch statements. The usage here is similar.
- **The break statement exits the loop body immediately** without executing any remaining statements in the loop body and without evaluating the controlling expression again
- After a break statement, execution proceeds with the statements that immediately follow the loop body
- Using break, saves the need for additional if-statements in the loop body (see examples later)

continue

- Unlike break, which exits the loop completely, continue merely **skips to the next iteration of the loop**, skipping over any remaining statements in the in the current iteration of the loop body
- After a continue statement, any following statements in the loop body are skipped and execution proceeds to evaluating the update expression (for-loops only) or the controlling expression (in while and do-while loops)
- Like break, the continue statement saves the need for additional if-statements in loop body (see examples later)

19 October 2020

22

Break (breaking out of a loop)

23

The **break** statement is used to **exit the innermost enclosing loop (while/do/for)**

- It is used to terminate a loop at a point other than beginning/end of the loop body
- It is also useful when there are several different conditions under which the loop can terminate or the conditions under which it terminates are not directly related to the main business of the loop

When using **break**, it is often necessary to check how the loop terminated.

- Did it terminate normally because its controlling expression was false?
- Did it terminate due to a **break** statement, which is usually indicated by a variable used in the controlling expression having a value that would have allowed the controlling expression to be true

```
// find smallest divisor of n
for (d = 2; d < n; d++)
    if (n % d == 0)           // any remainder?
        break;               // divisor found!

// recheck loop expression to see how loop terminated
if (d >= n) {                 // normal termination
    ...                       // no divisor found
} else {                     // terminated via break
    ...                       // divisor is d
}
```

19 October 2020

23

Break (breaking out of a loop) contd.

24

The **break** statement is used to **exit the innermost enclosing loop (while/do/for)**

- Innermost means with nested loops it only breaks out of the one that directly encloses the **break** statement, and not any outer loops

```
// outer loop
for (loops = 0; loops < 10; loops++) {
    n = random(100);

    Serial.print("\nfactors <= sqrt of ");
    Serial.println(n);

    // inner loop
    for (div = 1; div < n; div++) {
        if ((div * div) > n)           // <= sqrt ?
            break;                   // exit inner loop only

        if ((n % div) == 0)           // is it a factor?
            Serial.println(div);      // end of inner loop body
    }                                // end of inner loop body
    Serial.println("end of factors"); // end of outer loop body
}
```

Once break
from innermost
loop is
executed, the
next statement
to execute is
this one

19 October 2020

24

Continue (skipping remainder of loop body)

25

The **continue** statement is used to **skip the remaining statements of the loop body only in the current iteration** of a loop – it does not exit the loop

- After the continue statement is executed
 - first evaluate the update expression (for-loop only)
 - Then evaluate the controlling expression to see whether to repeat or terminate the loop
 - *In a while loop there are additional issues (see following slide)*

```
for (i=0; i<N; i++) {  
    if (cond1) {  
        doFirstThing();  
        if (cond2) {  
            doSecondThing();  
        }  
    }  
}  
  
for (i=0; i<N; i++) {  
    if (!cond1)  
        continue;  
    doFirstThing();  
    if (!cond2)  
        continue;  
    doSecondThing();  
}
```

When there are many nested conditions, the continue statement may help to make the code a bit clearer

19 October 2020

25

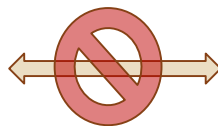
Contd.

26

None of the statements that follow the continue statement in the loop body are executed if the continue executes

- In a while-loop, this can introduce subtle bugs if a variable used by the loop expression is normally incremented at the end of the loop body since this will be skipped by continue
- In a for-loop, the problem generally does not occur since variables are updated in the *updateExpr* outside the loop body and this will be executed even after a continue statement

```
i = 0;  
while (i<N) {  
    if (someCondition1) {  
        doFirstThing();  
        if (someCondition2) {  
            doSecondThing();  
        }  
    }  
    i++;  
}
```




```
i = 0;  
while (i<N) {  
    if (!someCondition1)  
        continue; // PROBLEM!  
    doFirstThing();  
    if (!someCondition2)  
        continue; // PROBLEM!  
    doSecondThing();  
    i++; // not executed after continue  
}
```

19 October 2020

26

The fix for the problem on the previous slide is to update the necessary variables before calling continue. Sometimes this may require repeating some update expression code.



```

i = 0;
while (i < N) {
    if (someCondition1) {
        doFirstThing();
        if (someCondition2) {
            doSecondThing();
        }
    }
    i++;
}

i = 0;
while (i < N) {
    if (!someCondition1) {
        i++; // update expression
        continue; // OK!
    }
    doFirstThing();
    if (!someCondition2) {
        i++; // update expression
        continue; // OK!
    }
    doSecondThing();
    i++; // original update expression
}

```

Can you think of another way to eliminate this problem???

19 October 2020

27

ADVANCED TOPIC: A do loop that executes once only

In some rare and advanced situations (writing MACROS) it useful to define a block of code that can act as if it was a single statement by ending with a semicolon

The idiom/trick to achieve this is a do-while loop whose loop expression is false. The do-loop body will always executed exactly once – You don't need to write code like this for EE108, but just recognise what this means if you see it written

```

do {
    ...
} while (FALSE);

```

19 October 2020

28

Self test questions

29

Q. Write a loop that simulates tossing a coin and terminates when heads have come up 10 times. Use break to exit the loop if 3 tails in a row are thrown.

```
// to simulate a coin toss, use the expression: coinToss = random(2)
// test if it is a head by using the expression: (coinToss == 0)
// which evaluates to true for heads and false for tails
...
// TODO
```

19 October 2020

29

Self test questions

30

Q. Write a loop that simulates throwing 2 dice. Terminate the loop when double sixes have been thrown (using break). After the first die is thrown, if it is not a six, skip throwing the second die (and the remainder of the loop) using continue.

```
// to simulate a single die throw, use the expression: dieValue = random(6)
// TODO
```

19 October 2020

30