1.50 Functions

EE108 – Computing for Engineers

1

Overview

Aims

- Learn to use functions to simplify programme structure and allow code to be reused

Learning outcomes – you should be able to…

- Write and interpret function definitions
- Use (call) your own functions and library functions
- Write and interpret function declarations/prototypes

2

Abstraction and decomposition

To make code readable and understandable, it is best to divide it up into self contained chunks of about 25 lines or so

If you have a complex programme, you can simplify it by breaking the problem into different levels of detail– start with the big picture, e.g.

```
// the following is pseudocode – it is not C code

Superloop:
  checkButtonState()
    if displayPulseModeActive:
      readAnalogueInput()
      detectPeakSignal()
      if peakDetected:
        flashThePulseLED()
        calculateAveragePulseRate()
        showPulseOnLCDDisplay()

// what do you think this programme is supposed to do?
```


Contd.

After the big picture, start to work out the low level "under-the-hood" details, e.g.

```
// the following is pseudocode – it is not C code

// Now we define some of the under-the-hood details…
showPulseOnLCDDisplay:
  set cursor to home location in display
  lcdPrintString("Pulse is ")
  lcdPrintNumber(pulseValue)

// we assume lcdPrintString and lcdPrintNumber are detailed elsewhere
```

So in this simple example we have at least 3 levels of decomposition

- Top level – superloop
- Second level – showPulseOnLCDDisplay
- Third level – lcdPrintString, lcdPrintNumber

Each level should be reasonably straightforward to understand on its own

- We use **functions** to help us decompose the code in this manner with real C code

Code reuse

In many programmes we need to repeat certain calculations or actions over and over

- We've seen that one way to repeat things is to use a loop—but this only works if we want to repeat the calculations/actions immediately after each other
- What if we want to do the same thing at some other point in our programme? For example

```
// the following is pseudocode – it is not C code

main:
  // we want to flash the LEDs fast at this point to indicate system has
  // started up successfully

  // do some things

  while (recalibratingSystem):
    // we want to flash the LEDs slowly at this point
    // do some calibration stuff

  …

// it is likely that much of the code to flash the LEDs (whether slowly or
// fast) will be essentially identical
```


Contd.

*A **function** in C is a reusable block of code that has been given a name and that is designed to solve a particular problem*

Functions are the building blocks of C programmes

- Functions facilitate abstraction/decomposition
- Functions enable code reuse
  - *Functions are only defined once and there is only one copy of the code resulting in smaller programmes*

Functions

7

All C programmes have at least one function called **main**

- The function main is required in all C programmes and it is the first function to run when the programme starts
- This function is there, but hidden "under the hood" in Arduino C so you must not have a function called main in Arduino C sketches

Most C programmes have other functions besides main. These may be custom functions that you write yourself or **library functions**

- Library functions are simply functions that someone else has already written and packaged for your benefit

We have already used some library functions in Arduino C, e.g. millis, delay, delayMicroseconds, digitalRead, digitalWrite, etc.

We will now focus on custom functions that you write yourself…

7

Contd.

8

We will examine the following concepts

- **Function definition**
  - This defines the implementation code of the function
  - We'll also look at the *function signature* which specifies the *return type* and *parameters*
  - We'll look at *pure functions* vs. *functions with side effects*
- **function call**
  - This is used to invoke or execute a function
  - We will look at *arguments* vs. *parameters.* (Any argument expressions specified in the function call's argument list are evaluated first and the resulting values are assigned to the function parameters.)
  - If the function call appears in an expression or assignment statement, the value of the function is the *return value* from the function
- **function declaration**
  - This is used to declare the function signature somewhere before the function is called (even if the function definition is located elsewhere)

8

Functions overview

9

9

Function **definition** - overview

10

All functions must be defined somewhere before they can be used/called/executed

The function definition tells the compiler about the function implementation, i.e. what the function does and how it does it

The general form of a **function definition** in C is:

```
returnType functionName ( parameterList ) {
   functionBody
}
```

```
/**
 * trival example to add two numbers
 */
int add(int a, int b) {
  int result;

  result = a + b;
  return result;
}
```

Return type

Parameter list

Function body enclosed by braces

10

Function **definition** – overview contd.

The ***returnType*** of a function is the type of value that the function returns, e.g. int, float, unsigned char, long long, etc.

If a function will not return any value its return type must be specified as `void`

Conversely, if a function has a return type that is not void, then it must explicitly return a value of that type (using the return statement)

Examples…

```
/**
 * add two numbers
 * (must return an int)
 */
int add(int a, int b) {
  int result;

  result = a + b;
  return result;
}
```

```
/**
 * Switch on a LED
 * return type is void so
 * no return value
 */
void switchOffLeds(void) {
  digitalWrite(LED1, LOW);
  digitalWrite(LED2, LOW);
  // note: no return value
}
```


Function **definition** – overview contd.

The *parameterList* in the function definition is a comma separated list of parameters that accept values passed from the caller that may be needed by a function to do its work, or to modify its behaviour in some way

The general syntax is:

Type1 param1 , Type2 param2 , … , TypeN paramN

If the function takes no parameters, the *parameterList* should really be specified as `void`

```
/**
 * light an LED
 * @param pin which LED pin to light
 * @param on  switch LED on if true,
 *            off otherwise
 */
void lightLed(int pin, boolean on) {
  digitalWrite(pin, on ? HIGH : LOW);
}
```

```
/**
 * Switch on a LED
 * no parameters required
 */
void switchOffLeds(void) {
  digitalWrite(LED1, LOW);
  digitalWrite(LED2, LOW);
}
```

Function **definition** – overview contd.

Looking at the function body in particular…

```
returnType functionName ( parameterList ) {
   variableDeclarations

   statements
}
```

The ***function body*** (the block of code in braces) is the code required to actually do the work or computation of the function

- In C89, any local variables required by the function to do its work, must be placed at the top of the function body and before all other statements. (See *variableDeclarations* above.)

⚠ Warning: Arduino C (like C++) allows you to declare variables anywhere throughout the function body, but you will lose marks in EE108 if you do this, since we want to comply with C89

- The primary work of a function is implemented by the statements which follow any local variable declarations.

Function definition examples


```
/**
 * Example 1: a function to calculate the magnitude squared of
 * two integers and return the result
 * (This function has 2 parameters and returns a value)
 */
int magSquared(int a, int b) {
  int result;

  result = a * a + b * b;
  return result;
}
```

```
/**
 * Example 2: a function to switch off LED1 and LED2
 * (This function has a side effect -- quenching the LEDs. It has
 * no parameters and does not return a value.)
 */
void switchOffLeds(void) {
  digitalWrite(LED1, LOW);
  digitalWrite(LED2, LOW);
  // note: no return value
}
```

Function definition summary

Summary: the function definition …

- tells the compiler everything about how a function actually works – in particular, it provides the function body code which actually implements the calculations or actions of the function
- specifies how the function should be called when you would like to make use of its reusable functionality (the definition specifies the function's name, return type and parameter list)
- is never executed unless the function is called

Self test questions

Q. Define a function that blinks LED1 once, switching it on for 100 ms. This function needs no parameters or return value (similar to switchOffLeds in examples).

```
// function Definition
```

Self test questions

Q. Define a function which clamps an integer to a specified max value. The function should take 2 int parameters: the value to clamp and the max value permitted (see magSquared in examples). The function should return the original value if it is less than the max or return the max permitted value otherwise. E.g. if the max value is 10, then the function should return 5 if the value is 5 but it should return 10 if the value is 15.

```
// function Definition
```


Function **call** - overview

To actually make use of a function and execute it's code, you must call the function

When a function call is encountered in code…

- the current position in the programme code is saved (pushed onto the call stack)
- any argument expressions (see later) are evaluated and the resulting values become the parameter values passed into the function
- Next, the processor starts to execute the body code of the function being called
- Finally, when the called function returns, the programme resumes where it left off (by popping the location to resume off the call stack)

Function **call** - overview

To use/call/invoke a function, write the function name followed by the **argument list** surrounded by parentheses

We'll explore the argument list in more detail a bit later

functionName (*argExpr1* , *argExpr2* , … , *argExprN*)

Any function may be used as a simple statement on its own by calling it as follows:

```
switchOffLeds(); // a function with no parameters has no arguments
digitalWrite(LED1, HIGH); // a function with 2 parameters requires 2 arguments
magSquared(x, y); // when called this way, any return value is ignored
add(x, y); // when called this way, any return value is ignored
```


Function **call** – overview contd.

To use/call/invoke a function, write the function name followed by the **argument list** surrounded by parentheses

functionName (*argExpr1* , *argExpr2* , … , *argExprN*)

Any function that returns a value may be used in an assignment expression (or indeed any expression). The function is first executed and then its return value is used in the expression.

```
z = add(x, y);  // use in an assignment expression statement

if ((add(x, y) < 10) && (x > 2))  // use in the testExpr of an if-statement
  …

// z = switchOffLeds(); // ILLEGAL because the function doesn't return a value
```

Function **declaration** - overview

A function's definition or declaration must come before any call to that function

The function declaration

- Tells the compiler (and you) how the function should be called
- It must be located before the function call and is generally added near the top of a file (or in an **include file**)
- Using a declaration gives you flexibility with the order and location in which function definitions are subsequently written and called
- It is a requirement to have function declarations when the function definitions are in other C files

```
// function declaration
int add(int a, int b);

// loop function calls the
// add function
void loop() {
  int i = 2;
  int j = 3;
  int k;
  …
  // function call
  k = add(i, j);
  …
}

// function definition
int add(int a, int b) {
  return a + b;
}
```

Note: identical

Always put function declarations at the top of your C file (or in an include file)

Function **declaration** - overview

A function declaration (also called a **function prototype**) is just the function signature (i.e. return type, function name, and parameter type list).

It is usually written identically to the function definition with the function body code left out.

returnType functionName (parameterList) ;

Side note: For the function declaration parameter list, C permits the parameter names to be omitted, but we do not recommend this.

Example declarations…

```
// Note: Declarations should go near the top of the file, before all function
// definitions. I tend to write them in the same order as the function
// definitions appear in the file.
void switchOffLeds(void);
int magSquared(int a, int b);

// here's an example declaration from Arduino.h (part of the arduino library)
void delayMicroseconds(unsigned int us);
```

Self test questions

Q. Declare, define, and use a function that blinks LED1 once, switching it on for 100 ms. This function needs no parameters or return value (similar to switchOffLeds in examples).

```
// Function Declaration


void loop() {
  // Function call: blink LED1


  delay(1000);
}

// function Definition – copy from prev relevant self test question
```


Self test questions

Q. Declare, define, and use a function which clamps an integer to a specified max value. The function should take 2 int parameters: the value to clamp and the max value permitted (see magSquared in examples). The function should return the original value if it is less than the max or return the max permitted value otherwise. E.g. if the max value is 10, then the function should return 5 if the value is 5 but it should return 10 if the value is 15.

```
// Function Declaration


void loop() {
  int x = random(50);

  // Function call: clamp x to 20 (inclusive) or less


  Serial.println(x);
}

// function Definition – copy from prev relevant self test question
```

Functions in more detail

25

25

Function definition conventions and style

26

Recall: For C89, you must declare variables at the top of a block of code

- In the case of a function, this is at the top of the function body.

For readability, leave a blank line between the variable declarations and the remainder of the function body

Put a header comment above every function

- at a minimum this should be a one line summary of what the function is intended to do
- The particular style of block comment shown used in the examples (starting with a /**) is compatible with tools that can automatically generate documentation and help files from the comments (e.g. doxygen) – a point we may return to later

26

Contd.

The following function header comment is compatible with documentation tools such as doxygen and is also fairly clear to human readers:

Each @param line is used to describe a parameter and the @returns line is used to describe what return values the function gives . It is a good habit to document your functions like this.

```
/**
 * clamp the value of x to a specified range
 *
 * @param x    the value to be clamped
 * @param min  the minimum value of x allowed
 * @param max  the maximum value of x allowed
 * @returns    the value of x limited to the range min to max
 */
int clamp(int x, int min, int max) {
  if (x < min)
    return min;
  else if (x > max)
    return max;
  else
    return x;
}
```

Side note: This is a good example of a cascaded if-statement

Function declaration, call, definition order

C requires that A function's definition or declaration must come before any call to that function. It is better to use declarations than reorder the function definitions.

Best, most flexible

```
int add(int a, int b);

void loop() {
  int i = 2;
  int j = 3;
  int k;

  k = add(i, j);
}

int add(int a, int b) {
  return a + b;
}
```

Function declaration comes before function call – all is good

OK, but inflexible

```
int add(int a, int b) {
  return a + b;
}

void loop() {
  int i = 2;
  int j = 3;

  int k = add(i, j);
}
```

No function declaration, but Function definition comes before function call – OK, but inflexible for larger programmes

Do not do:
Only works in Arduino C

```
void loop() {
  int i = 2;
  int j = 3;
  int k;

  k = add(i, j);
}

int add(int a, int b) {
  return a + b;
}
```

Function definition comes after function call – in general the compiler won't even know this function exists, so it won't work.

Function declaration revisited

Normally a function declaration looks just like a function definition without any function body. This makes it simple to write a declaration simply by copying and pasting the definition and then deleting the function body. E.g.

Function definitions

```
int add(int a, int b) {
  return a + b;
}
```

```
void switchOffLeds(void) {
  digitalWrite(LED1, LOW);
  digitalWrite(LED2, LOW);
}
```

Declarations with parameter names [recommended]

```
int add(int a, int b);
```

```
void switchOffLeds(void);
```

In fact, it is also legal for the function declaration to leave out the parameter names – the compiler only cares about knowing the parameter types, e.g.

Declarations without parameter names [not recommended]

```
int add(int, int);
```

```
void switchOffLeds(void);
```

⚠ ***Although legal, declarations without parameter names are NOT recommended.*** *It is far more informative to a reader of the code and less error prone to simply copy the signature from the function definition without editing out the parameter names.*

Function declarations contd.

As a developer you do not need to know HOW every function that you use works. You only need to know WHAT it does.

- E.g. Most likely you do not know exactly how the delayMicroseconds function is actually implemented but you know what it does from the name

Although it is beneficial to be supplied with documentation, very often you can infer enough information from the function declaration to be able to call the function correctly because the declaration tells you:

- Will the function return a value, and if so, what type is it?
- Does the function require parameters? How many, what are their types and, if good parameter names have been chosen, what is each parameter for?

E.g.

```
int add(int a, int b);
```

From the declaration above we can guess that this is a function to add two numbers, both of which must be int types. Furthermore the function returns an int. Therefore we know we can write things like the following

```
// assume, x, y, z declared and initialised as int types
add(x,y);
z = add(x,y);
z = add(x,1) + add(y,4);
```

Self test questions

Q. Based only on the function declarations and comments below, write code snippets to demonstrate calling each of the following functions

```
// Declarations
// led is the led number. The constants HIGH (on) or LOW (off) should be
// used for the state
void digitalWriteBarLed(int barLedNum, int state);

int max(int a, int b); // returns the maximum of a or b
…
// Qa. Call the appropriate function to switch on bar LED 4


// Qb. The variable xMax keeps track of the maximum value of x encountered
// so far. If a new value of x is bigger than xMax, update xMax.
// Write the appropriate snippet to do this (only needs 1 line of code).


// Qc. If the maximum of variables x and y is greater than THRESHOLD,
// delay for 1 ms. Write a code snippet to do this.
```


Function parameters revisited

Parameters are used to pass values that may be needed by a function to do its work, or to modify its behaviour in some way

We say that a function is parameterized, when it's general behaviour is the same but it's specific result or side effect depends on the value of some inputs you give it (namely, the parameters).

Consider the following mathematical functions:

$$f(x) := x^2, \qquad g(x) := x + 1$$

$$a = f(2) + g(3)$$

$$z = 4; \quad b = f(z)$$

$$z = 5; \quad c = g(f(z))$$

What values would you expect a, b, and c to have?

We can see that you can pass literal values (or constants), or variables, or other mathematical functions as arguments to mathematical functions. You can do the same with functions in C.

Arguments vs. parameters

Arguments are the expressions that a caller uses when making a function call

- **all arguments are evaluated before the function body code is executed**, and the resulting values are assigned to the function parameters

Parameters are the named data values seen by the function body when it starts executing

- A function body never knows where it was called from
- A function body never "sees" the original argument expressions, it only ever sees the values that result from evaluating the argument expressions
- There is no relationship between parameter names and argument names or expressions. Argument names and parameter names are usually different.
- However, argument data types must match or support implicit conversion to the function parameter data types

Arguments vs. parameters example


```
int add(int a, int b) { // function definition with parameter list
  return a + b;
}

void loop() {
  int x = 10;
  int y = 5;
  int z;

  z = add(1, 2);       // literal value arguments
  // run add function body with a=1, b=2 -- returns 3, z = 3

  z = add(x, y);       // variables as arguments
  // run add function body with a=10, b=5 -- returns 15, z = 15

  z = add(x*10, x/y);  // expressions as arguments
  // run add function body with a=100, b=2 -- returns 102, z = 102

  z = add(add(x, 10), -add(y, 10)); // function calls as arguments
  // evaluate first argument
  //   run add function body with a=10, b=10 -- returns 20
  // evaluate second argument
  //   run add function body with a=5, b=10 -- returns 15
  // outermost add function call
  //   run add function body with a=20, b=-15 -- returns 5, z = 5
  …
```

return values and side effects

35

A function has a **side effect** if it modifies any program state outside the function's own scope or has any observable effect on the outside world besides returning a value

- Changing a global variable value is a side effect
- Other side effects include lighting an LED, printing text to serial output, delaying for some period of time, etc.

"Pure" functions compute and return a value and have no side effects

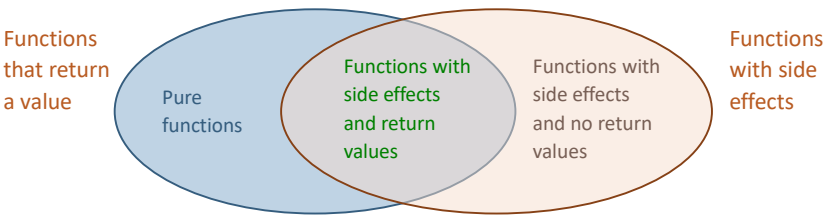
- Such functions are frequently used to calculate a value (usually based on some input parameters)
- This is just like the typical mathematical definition of a function
  - E.g. compute the average of two numbers, or compute the nth power of a number
- A pure function must return a value

In contrast to pure functions, many useful functions do have side effects

- A function with side effects may return a value but it is not required
  - You must decide at implementation time whether returning a value is appropriate to your function. Once you have decided, the function must be consistent (either always returning a value or never doing so).

return values and side effects

36



When we look at calling functions we will see that:

- **Functions that return a value can be used in any expression** just like a variable
- **Functions that do not return a value cannot be used in an expression** – they must be used as a standalone function call statement and this results in imperative style programming: do this, do that, etc.

Functions return values in expressions

A function that returns a value (i.e. the return type is not void) may be called wherever a value could be directly used in an expression.

Functions are evaluated in the order needed for the expression (e.g. according to operator precedence).

In particular, function calls whose return values will be passed into other function calls are executed early enough to provide those values.

```
int add(int a, int b) { // function definition with parameter list
  return a + b;
}

void loop() {
  int x = 12;
  int y;

  y = 145 - add(x, 10) * 2; // function call used in expression

  if (add(x, 2) < 100) { // function call used in relational expression
    …
  }

  y = add(27, add(x, 13)); // function call used as argument to function call
                           // first evaluate add(x,13) then use result for
}                          // add(27, …)
```


Returning a value from a function

A function which does some calculation and has a return value…

- **Must define the *returnType*** , i.e. the type of value that the function returns, such as int, float, etc.
- **Must have a return statement** in the function body. The return statement specifies an expression that must evaluate to a value of the correct *returnType*. The syntax is:

```
return expression ;
```

```
/**
 * calculates the (integer truncated)
 * average of two integers
 * (a "pure" function with a return
 * value and no side effects)
 */
int truncatedAverage(int a, int b) {
  int average;

  average = (a + b) / 2;
  return average;
}
```

```
// same function written more
// succinctly by putting the
// calculation expression directly
// into the return statement


int truncatedAverage(int a, int b) {
  return (a + b) / 2;
}
```

Using **return** to exit a function early

The return statement can be used to return (exit) from the middle of a function body without executing the remaining statements (similar to using `break` to exit the middle of the a loop body)

- A function with a return value must use **`return`** ***expression*** **`;`**
- A function whose return type is void cannot return a value or expression and must use **`return;`** with no expression instead to exit the function from the middle

⚠ Beware: **you cannot mix the with-expression and without-expression forms of return in a single function** – the function is either required to return a value or it is not

Function must return a value of type int

```
/**
 * calculate x to an integer power
 */
int toPower(int x, int power) {
  int result;
  if (power == 0) {
    // anything to power of zero is 1
    return 1;
  } else {
    result = x;
    …
    return result;
  }
}
```

Function must not return a value

```
/**
 * a modified digitalWrite function
 */
void digitalWriteBarLed(int ledNum,
                        int value) {
  // check is LED number valid
  if ((ledNum < 0)
      || (ledNum >= NUM_BAR_LEDS))
    return;

  digitalWrite(BAR_LED_1_PIN + ledNum,
               value);
}
```


Self test questions

Q5. Define a function with 2 int parameters t1, t2, and an int return type to calculate the time, T, that it would take 2 people to do a task compared to the times, t1 and t2, that it would take each of them individually (see equation below). In the case that the denominator is zero, the function should return zero. Give the function an explanatory name.

$$T = t_1 t_2 / (t_1 + t_2)$$

Q6. Define a function with two integer parameters, a and b, but no return value to switch on LED1 when a is largest, and LED2 otherwise. If either parameter is negative return from the function immediately without lighting either LED.

Summary

41

Functions are used to create reusable blocks of code

Function parameters are used to allow some customization of the function behaviour, side effects, and outputs or results

To use the reusable code in a function, you simply call the function

In particular, functions that return a value can be used (called) wherever a value (literal, constant, or variable) could appear in an expression

A function declaration or definition must appear before the first call of the function in the code

A function declaration specifies the function signature only (i.e. the return type, function name, and parameter list)

A function definition specifies both the function's signature and its body implementation

Self test questions

42

Q1. Define a function with no parameters or return value to switch on LED1 and LED2. Place a header comment on your function.

Q2. Define a function with no parameters and no return value to delay 2.5 ms using delay for the whole ms part and and delayMicroseconds for the fractional ms part. Place a header comment on your function.

Self test questions

Q3. Define a function to switch on one of the two LEDs on the launchpad mainboard. Use one parameter (to specify which LED) – perhaps a number or boolean value, and no return value.

Q4. Define a function with two integer parameters, a and b, but no return value to switch on LED1 when a is largest, and LED2 otherwise.

Self test questions

*Q. Write a function to calculate the number of bits that are 1 in an **unsigned char**. (For example, binary 0011 0001 has three bits that are 1). Hint: remember bitRead(value, bitNum).*

```
// TODO – function definition
    .
```

```
// TODO – call the function to check if 0x5A has an even number of ones
```

Self test questions

Q1. Declare, define, and use a function which simulates a coin toss and evaluates true if heads were thrown, and false otherwise. (This a function which computes and returns a value.)

```
// Declaration


// Definition
// Hint: simulate a coin toss as in previous examples









// Use – light LED1 if heads were thrown
```


Self test questions

Q2. Write a function which does not return until heads have been thrown a specified consecutive number of times. Under the hood you will need a loop, one or more counters, and you will need to call the simulated coin toss function developed in Q1. (Note: this function has a side effect – delaying – but does not compute a value).

```
// Function Declaration


// Function Definition









…
// Function Call – light LED1 if heads were thrown 5 times in a row
```