ME 120: User-defined Functions in a Nutshell

Gerald Recktenwald
gerry@pdx.edu

October 24, 2021

Why Use User-Defined Functions

User-defined functions are blocks of code that perform a task such as evaluating a formula, making a sensor reading, deciding whether to turn on a light, or making a robot move. User-defined functions are just C/C++ functions. The "User-defined" adjective merely indicates that the function is not the required `setup` and `loop` functions that are required in all Arduino sketches.

In a general sense, a C/C++ function is no different than a sequence of statements. However, that simple description – a function as a block of code – does not capture the true benefit of using functions. Functions, when designed well and used appropriately, have many advantages:

- Functions encapsulate a task in a way that allows it to be reused in similar, but different situations.
- The variables local to a function are in a separate memory space, which allows reuse of simple names without worry that those variables will affect code outside the function.
- Functions can be updated, e.g., by fixing bugs or improving efficiency, and as long as the input and output parameters don't change, the the code that uses the function does not need to change.
- Using functions simplifies code and logic in the main part of a program. Using a function is like thinking with high-level concepts or *chunking*.
- Code that uses the benefits of functions described in the preceding bullets can be easier to read and to maintain.

Components of a User-defined Function

- **Name of the function**
  - ▷ Follow the rules for naming variables.
  - ▷ Do not use the names of built-in functions.
  - ▷ Do not use the names of libraries you have included.
  - ▷ Recommended: use names that include verbs to suggest how the function acts on data, e.g., `sensorAverage`, `updateDisplay`, `accelerometerStart`.
- **Input arguments**
  - ▷ Specify *type* and *name of the local variable*
  - ▷ In parenthesis immediately after the function name.
  - ▷ The *type-name* pairs are separated by commas.
    Example:

```
void updateDisplay(float timeVal, float Tin, float Tout) { ... }
```

- **Return type**
  - ▷ *At most, only one* value can be returned from a function

- ▷ Any valid Arduino variable type *or* `void`
- ▷ Functions with `void` type do not return any values
- ▷ Values returned are in the argument of `return`
- ▷ Usually the call to the `return` function is in the last line of the user-defined function

Examples:

```
int ambientLight() { ... }

float sensorAverage(int inputPin, int Nave) { ... }
```

- **Function body**
  - ▷ Any sequence of valid Arduino statements
  - ▷ Uses input arguments as local variables
  - ▷ Can use global variables

Simple Example

Listing 1 shows simple functions for adding two numbers. `add_two_int` adds two integer values. `add_two_float` adds two floating point values. The type of the input parameters matters and the type of output matters. Therefore, the `add_two_int` has two `int` inputs and returns an `int`. The `add_two_float` has two `float` inputs and returns a `float`.

Running the `add_two` sketch produces the following output in the Serial Monitor.

```
Add two ints:
k, m, n =  5,  7,  12

Add two floats:
x, y, z =  13.25,  -4.84,  8.41
```

Scope of Variables

Scope of a variable refers to the extent to which it is visible in a sketch. Variables can have global or local scope. A global variable is visible in all functions in a code. In most cases[1] a global variable can be changed by code in any function contained in the sketch. A local variable is defined within a function block delineated by a starting bracket, `{`, and an ending bracket, `}`. When a variable is defined inside a function it is local to that function, and cannot be changed by code in other functions.

A key advantage of putting code in a C/C++ functions is to isolate the code from other parts of a sketch. The computations in a function are local to that function. A function can depend on input parameters that change the calculations it performs. That means we can

- Use local variables that have simple names with meaning in the context of the function;
- Apply the same function to different inputs from the main (or calling) function;

[1] The exception is when a variable is defined with the `const` keyword.

- Make the main program shorter and easier to read by hiding complex calculations in the function.

The `sensorAverage` function in Listing 2 is reused for two different measurements. The local variables `pin`, `nread`, `ave` and `sum` make sense within the scope of `sensorAve`. The variables in the `loop` function are descriptive and have meaning inside `loop`.

```
//  File:  add_two.ino
//
//  Demonstrate a user-defined function to add two values

void setup() {

  int   k,m,n;
  float x,y,z;

  k = 5;
  m = 7;
  n = add_two_int(k,m);

  x = 13.25;
  y = -4.84;
  z = add_two_float(x,y);

  Serial.begin(9600);
  Serial.println("\nAdd two ints:");
  Serial.print("k, m, n =");
  Serial.print("  ");   Serial.print(k);
  Serial.print(",  ");  Serial.print(m);
  Serial.print(",  ");  Serial.println(n);

  Serial.println("\nAdd two floats:");
  Serial.print("x, y, z =");
  Serial.print("  ");   Serial.print(x);
  Serial.print(",  ");  Serial.print(y);
  Serial.print(",  ");  Serial.println(z);
}

// ------------------------------------------------
void loop() {  }   //  Loop is empty on purpose

// ------------------------------------------------
int add_two_int( int a, int b) {

  int s;    //  Local variable used to accumulate sum

  s = a + b;
  return(s);
}

// ------------------------------------------------
float add_two_float( float a, float b) {

  float s;   //  Local variable used to accumulate sum

  s = a + b;
  return(s);
}
```

Listing 1: Arduino sketch that demonstrates user-defined functions that add two `ints` and add `floats`.

```
//  File: average_analog_input.ino
//
//  Demonstrate use of a user-defined function to compute
//  the average of an input sensor reading

void setup() {
  Serial.begin(9600);
}

void loop() {

  int  t_sensor_pin=3, p_sensor_pin=5;
  float temperature, pressure;

  temperature = sensorAverage(t_sensor_pin, 25);
  pressure    = sensorAverage(p_sensor_pin, 15);

  Serial.print(temperature);
  Serial.print("  ");
  Serial.println(pressure);
}

// ------------------------------------------------
//  Read and average an analog input channel.
//      pin is the analog input pin.
//      nread is the number of readings to average.
float sensorAverage(int pin, int nread) {
  float ave, sum;

  sum = 0.0;
  for (int i = 1; i <= nread; i++) {
    sum += analogRead(pin);
  }
  ave = sum / float(nread);
  return (ave);
}
```

Listing 2: Arduino sketch that demonstrates reuse of a user-defined to read and average an analog input channel.