ME 120: Arduino Variables

Gerald Recktenwald
gerry@pdx.edu

October 14, 2019

1 Introduction

Variables store data that may change while an Arduino sketch is running. Variables are helpful, and in many cases necessary, for tasks such as storing sensor readings, keeping track of the state of a switch (e.g. high or low/open or closed), or counting the number of times a loop has been repeated.

Variables are classified by the type of data they store. Arduino programmers can choose from several types of variables. The most frequently used are "int" and "float". An "int" variable stores integers (2, 0, -7, 20342) whereas a "float" stores *floating point* values, i.e. numbers with fractional parts (3.1415, -7.654).

We begin with a simple tutorial to show how variables are created and used. In the second part of this chapter, we discuss the features of variable types and arrays more systematically. After working through the tutorial you will be ready to use variables in your Arduino sketches.

2 Basic Variable Operations: Define, Assign, and Recall

Using a variables involves three operations

1. Define the variable: specify its name and type
2. Assign a value to the variable
3. Recall and use the value stored in the variable

The following three lines show a typical example of the declaration, assignment and use of variables:

```
int  sensorPin, sensorReading;          //  define sensorPin and sensorReading as ints

sensorPin = 4;                          //  store a value in sensorPin

sensorReading = analogRead(sensorPin);  //  use value in sensorPin; store data in sensorReading
```

Before using a variable to store a value, both the *name* and *type* of the variable must be specified. The first line in the preceding code declares `sensorPin` and `sensorReading` to be `int` variables. All `int` variables store integer values between -32768 and +32767. The names `sensorPin` and `sensorReading` are arbitrary. We could have written `int a, b;`, but it is good practice to use variable names that in some way describe how the variable is used or what kind of information (beyond just the type) that the variable stores.

The `sensorPin = 4;` statement stores 4 in the memory location named `sensorPin`. The `sensorReading = analogRead(sensorPin);` statement assigns the value returned by the built-in `analogRead` function to the variable named `sensorReading`.

It's best to think of the assignment statement as a two-step process. The first step is to evaluate the right hand side of the expression. The second step is to store the result into the variable on the left hand side of the assignment operator. This idea is discussed in more detail in the next section.

2.1 The Assignment Operator

The statement:

```
x = 3;
```

assigns the value 3 to the variable named `x`. The equals sign is the assignment operator. When the statement is executed, 3 is stored in `x`, it does mean that 3 and `x` are the same. At some later point in the program execution, a different value can be stored in `x`.

The conceptual difference between assignment and equality is important, and can trip up novice programmers. To understand the concept, you can replace the equal sign with an arrow pointing from left to right.

$$\texttt{x} = \texttt{3} \qquad \text{means} \qquad \texttt{x} \leftarrow \texttt{3}$$

The term on the right side of the equals sign determine what is stored in the variable name appearing on the left side of the equals sign. Consider the following sequence of statements:

```
x = 3;
y = x;
x = 5;
```

When the last statement is executed, 5 is stored in `x` and 3 is stored in `y`. The second statement `y = x;` does not mean that `y` is always equal to `x`. Instead it means that the *current* value of `x` is stored in `y`.

For more complex assignment statements, e.g.,

```
t = 1.23;
s = sin(t)*cos(t);
```

the order of operations are

1. Evaluate the expression on the right hand side of the equals sign.
2. Assign the result to the variable on the left hand side of the equals sign.

Therefore, in the expression `s = sin(t)*cos(t);`, the value of `t` must be defined by a previous statement. Given a value of `t`, the values of `sin(t)` and `cos(t)` are computed. Next the product `sin(t)*cos(t)` is evaluated. Finally, the result of `sin(t)*cos(t)` is stored in the memory location assigned to `s`.

2.2 Using Variables to Store Constants

Huh?

Before using variables to perform more complex operations, we introduce the common practice of using a variable to hold a value that does not change while a sketch is running. In this case, the variable is a constant. There are two main reasons for using a variable to store constant values. First the *name* of the variable helps us understand the significance of the value. In the preceding example we could have written

```
sensorReading = analogRead(4);
```

instead of

```
sensorReading = analogRead(sensorPin);
```

Either form is correct and either form can be successfully used in an Arduino sketch. However, the name `sensorPin` is suggestive of the programmer's intention as well as the physical requirement of the circuit. If the sensor is a photo-resistor, an even better name would be `photosensorPin`.

The second reason for using a variable to store a constant is that the variable makes it easier to update a program with higher confidence of not introducing hard-to-find bugs. Suppose that you need to rewire your circuit so that the sensor is now attached to pin 0 instead of pin 4. After making the physical change in the circuit, you change the sketch in just one place

```
sensorPin = 0;
```

and if the program was working before, it is likely to continue working. This is especially important if the value of `sensorPin` is needed in several places in your code.

The most important reason for using a variable is that it allows values to change as the program is running, which leads to the obvious

> Use a variable whenever you need to store values that will change as the program is running..

The other reason to use variables, especially for program values that don't change is that it makes a lot of sense to

> Use variables (instead of constants) to make your program easier to understand and change.

2.3 Declaring and Initializing Variables

Suppose you were going to use a variable called `LEDpinRed` to store the pin assignment for an LED that you want to turn on and off. You could declare the variable with

```
int  LEDpinRed;
```

Now that `LEDpinRed` is defined, at some later part in the program we can assign a value to `LEDpinRed`

```
LEDpinRed = 5;
```

and after a meaningful value is assigned to `LEDpinRed`, we can use that variable on the right hand side of an expression:

```
pinMode( LEDpinRed,  OUTPUT );
```

This expression uses the built-in `pinMode` function to configure the `LEDpinRed` (currently, pin number 5) as an output. There is no assignment operator and no left-hand-side. Nonetheless, the expression `pinMode( LEDpinRed, OUTPUT );` cannot be evaluated unless all of the variables inside the parenthesis (1) have been previously defined and (2) have been assigned meaningful values.

The `pinMode` function is discussed in the chapter on digital input and output. For now, the essential idea is that the *value* stored in `LEDpinRed` is used as the first argument to the `pinMode` function.

Note that `OUTPUT` is a constant that is pre-defined by the programmers who created the Arduino IDE. The convention is that names in all-capital letters are constants available to all programs. Whatever you do, **do not** attempt to change the value of `OUTPUT` by reassigning it.

Now, if `LEDpinRed` is one of the variables-that-are-really-constant, you could also declare it and assign it in one line like this

```
int  LEDpinRed = 5;
```

This form of assignment is called an *initialization*, and can only be used once when a variable is declared. Of course, since you created the variable `LEDpinRed`, you can change its value later.

It is good practice to add a comment statement to explain the meaning of variable names

```
int x = 2;        //  x is a global variable

void setup() {

  int y,z;        //  y and z are local variables

  y = 2*x;        //   Value of x is available to be used
  x = 3;          //   A new value can overwrite the previous value stored in x
  z = x/y;        //   z uses the most recent values of x and y

  Serial.begin(9600);
  Serial.print("x, y, z = ");
  Serial.print(x);
  Serial.print(",  ");
  Serial.print(y);
  Serial.print(",  ");
  Serial.println(z);
}

void loop() { }   // loop is empty on purpose
```

Listing 1: Sketch to demonstrate global and local variables. In this sketch, `x` is global, while `y` and `z` are local variables in the `setup` function.

```
int  LEDpinRed = 5;    //  LED to indicate that the heater is on
```

The text starting with `//  LED to indicate ...` is called an in-line comment statement. The `//` and all text to the end of the line is ignored by the Arduino IDE. The text is only there for humans to read. Note that we did not write

```
int  LEDpinRed = 5;    //  red LED
```

which has a comment statement that does not add any information about the *purpose* of the pin assignment. The name of the variable conveys the idea that this pin probably has something to do with a red LED.

If you have multiple assignment and initializations you can combine them like this

```
int  LEDpinRed = 5, LEDpinYellow =6;
```

although this form makes it hard to use in-line comment statements.

2.4 Variable Scope

The *scope* of a variable refers to the locations in the program where that the variable can be retrieved, i.e., used on the right-hand side of an equal sign), or set, i.e., used on the left hand side of an equal sign.

global A variable with global scope is available to all program functions.

- The value of the variable can be used in any function.
- The value of the variable can be changed in any function.

Global variables are "shared" by all program functions.

local A variable with local scope is only available *within* the function where it is defined.

Listing 1 shows how a global variable (`x`) is defined outside the body of any functions in the sketch. Therefore, `x` has a global scope. The value of `x` is visible to statements inside of the `setup` function.

Table 1: Integer and integer-like types in Arduino sketches.

	Integer variables
`int`	16 bit signed integer with values in the range -32,768 to 32,767 on 16-bit architectures like the Arduino Uno and Leonardo. On the Arduino Due, `int` variables are 32 bit, meaning that their range is the same as a `long` on an Arduino Uno
`unsigned int`	16 bit integer with values in the range 0 to 65,535
`long`	32 bit signed integer with values in the range -2,147,483,648 to 2,147,483,647
`unsigned long`	32 bit integer with values in the range 0 to 4,294,967,295
	Integer-like variables
`boolean`	one-bit values, i.e., `boolean` variables can store either 1 or 0. `boolean` variables save memory and are used in situations where a value is only going to be 1 or 0, which corresponds to the logical values TRUE and FALSE.
`byte`	an unsigned 8-bit integer. `byte` values are limited to the range 0 to 255
`word`	an unsigned 16-bit integer value. A `word` variable is equivalent to a `unsigned int` variable.

3 Basic Variable Types

For simple programs, we can classify variables as belonging to one of three main groups

- integers
- floating point numbers
- characters

In C or C++, these main groups have variants. Although Arduino sketches are ultimately processed by a C++ compiler, we will only describe variables commonly used in Arduino sketches.

3.1 Integer or integer-like variables

Table 1 lists the four variable types that are used to count items (a natural use of integers) or to perform arithmetic where the results are always integers. There are three additional types that act like unsigned integers.

All variables have a limited range, which means that there is a finite lower limit and a finite upper limit that can be stored in a variable of a given type. For 16-bit microcontrollers like the Arduino Uno and Leonardo an `int` variable is stored in 16 bits. For 32-bit microcontrollers like the Arduino Due, the `int` is 32 bits, meaning that an `int` variable on a Due has the same range as a `long` on an Arduino Uno.

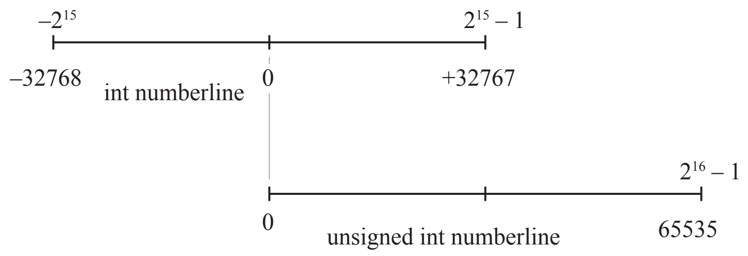

Figure 1: Number lines for the `int` and `unsigned int` variable types.

Figure 1 is a graphical representation of the `int` and `unsigned int` data types. An `unsigned int` has the same number of possible values as an `int`, but the `unsigned int` numberline is shifted so that the first value is zero. The relationship between a `long` and an `unsigned long` is analogous to the relationshiop between and `int` and an `unsigned int`.

General recommendations

1. Use `long` integers to work with quantities that can take on large values, e.g. long time measurements with `millis()` or `micros()`.
2. Use unsigned integers when you are certain that the value begin stored will never be negative *and* when you need the extra range. Note that it is probably better to switch from `int` to `long` than to switch from `int` to `unsigned int` when you are worrying about providing sufficient magnitude to store a value.

3.2 Floating Point Variables

For Arduino Uno, Leonardo and earlier boards based on the ATMEL microcontroller, there is only one unique type of floating point value, the `float`.

`float` a 32 bit floating point value in the range -3.4028235E+38 to 3.4028235E+38

There is a `double` type in the Arduino IDE and language, but in implementation the `double` and `float` have the same precision on earlier Arduino boards.

The Arduino Due uses a 32-bit microcontroller. The `double` type on the Due is 8 bytes or 64 bits.

3.3 Character Variables

`char` A single character occupying 1 byte (8 bits).

`unsigned char` An unsigned data type occupying 1 byte (8 bits). An `unsigned char` is equivalent to a `byte`. Use `byte` instead of `unsigned char` to make your intention known.

4 Idiosyncrasies of Using Variables

4.1 Mixed Calculations with `int` and `float` Variables

Computations are a basic part of programming. Here is a simple example that includes the variable declaration:

```
float  x,y,z;

x = 5.0;
y = 7.0;

z = x/y;
```

When this sequence of statements is executed, the value stored in `z` is approximately 0.7143. Repeating the calculations with integers produces a different result:

```
int  x,y,z;

x = 5;
y = 7;

z = x/y;
```

When the calculations are performed with `int` variables, the value stored in `z` is zero.

Rules of integer arithmetic are discussed in more detail in Section integer_arithmetic. A very brief primer on floating point arithmetic is given in Section floating_point_arithmetic.

4.2 When Do Variables Retain Their Value?

The ideas in this section require an understanding of variable scope. You may want to reread Section 2.4 before continuing.

> **Short version:** only global and static variables retain values on subsequent calls to the `loop` function[1].

The `variable_retain_value.ino` sketch in Listing 2 demonstrates how different variable declarations influence whether values are retained on subsequent calls to the `loop` function. We focus the difference between global variables, local variables and local variables declared with the `static` qualifier. Although the variables in this example are all `int` types, the results apply to all other types, e.g., `long`, `float` `char`, etc. Running the `variable_retain_value.ino` script produces the following output

```
i, j, iglob, istat = 1  1  1  1
i, j, iglob, istat = 1  1  2  2
i, j, iglob, istat = 1  1  3  3
i, j, iglob, istat = 1  1  4  4
i, j, iglob, istat = 1  1  5  5
i, j, iglob, istat = 1  1  6  6
i, j, iglob, istat = 1  1  7  7
i, j, iglob, istat = 1  1  8  8
i, j, iglob, istat = 1  1  9  9
...
```

[1]There are lots of details buried in this simple rule. First, a *global* variable is defined in the header (recommended) and outside the body of any function in the sketch. A `static` variable is define as such, e.g., `static int c`. Without getting too lost in the details, just remember that most variables (except global and `static` *forget* the values they held during the last time the function was called. In general, this is a *good thing*.

In `varible_retain_value.ino`, the `i` and `j` variables are local to the `loop` function. One consequence is that values stored in `i` and `j` are not retained from one call of `loop` to the next. Effectively, `i` and `j` are created anew each time `loop` is called.

The `iglob` variable is global, which means that its value is visible to all functions defined in the sketch. The value of `iglob` is also retained on subsequent calls to `loop`.

The `istat` variable is an `int` that is local to `loop`, and as such, the value of `istat` is only visible inside `loop`. Since `istat` is declared with the additional `static` qualifier, its value *is* retained on subsequent calls to `loop`.

```
//  File:  variable_retain_value.ino
//
//  Demonstrate how variables do not retain their value unless
//  they are declared as global or static within a function.

int  iglob=0;   //  Global variable will retain value

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

void loop() {

  static int  istat=0;
  int i=0, j;            //  Bad idea to leave j un-initialized

  // -- Increment all variables
  i++;
  j++;
  iglob++;
  istat++;

  // -- Print values of variables in loop()
  Serial.print("i, j, iglob, istat = ");
  Serial.print(i);
  Serial.print("  ");
  Serial.print(j);
  Serial.print("  ");
  Serial.print(iglob);
  Serial.print("  ");
  Serial.println(istat);

  delay(1000);   //  slow down the output
}
```

Listing 2: Sketch to demonstrate which variable types retain values.