A short introduction to Arduino Programming

Fauzan

July 13, 2016

1 What's the program?

Armed with an LED, you want to banish darkness at the push of a button. Because just using a battery and wires and sticky black tape is so fifth grade stuff, you are going to use the mighty Arduino as your savior. Congratulations, you have defined your **Problem**.

```
I want to banish darkness at the push of a button with the Arduino as my saviour.
```

You know what you want to do, but how are you going to do it? How is a *dumb someone else* (lets call him Dumbo) going to do it? Dumbo may have no moral obligations of banishing darkness and enlightening your path. Actually, Dumbo does not even know why darkness is evil or enlightenment is virtuous. You have to give line by line instructions to Dumbo, explaining what to do. Tricky, isn't it? This instruction set is the **Algorithm**. The algorithm to the above problem is given below.

```
Hey Dumbo!

Check the button.

If button is pressed and LED is OFF, turn LED ON.
If button is released and LED is ON, turn LED OFF.

Repeat lines 3-8.
```

But our Dumbo, the Arduino does not know about LEDs and buttons. It has I/O pins and knows how to read and set voltages. Let us try to write our algorithm in terms of pins and voltages.

```
#Button terminal A is at 5V pin
#Button terminal B is at pin 2
#LED cathode is at pin 13

Set pin 2 to input mode
Set pin 13 to output mode

Create a variable named "state"

state = value at pin 2

If state = HI, Set pin 13 to HI
```

```
If state = LO, Set pin 13 to LO

Goto line 10
```

We have another obstacle. Dumbos come in all shapes and sizes. Worst, hardly any of them are fluent in English. Dumbos understand languages which are, by today's standards, extraordinarily tedious for us to directly write in. Arduino understands hex code which looks like this.

```
:1001A000611108C0F8948C91209582238C938881EA
:1001B00082230AC0623051F4F8948C91322F30952A
:1001C00083238C938881822B888304C0F8948C913C
:1001D000822B8C939FBFDF91CF9108950F931F9334
:1001E000CF93DF931F92CDB7DEB7282F30E0F90110
:1001F000E859FF4F8491F901E458FF4F1491F90138
:10020000E057FF4F04910023C9F0882321F0698350
:100210000E948C006981E02FF0E0EE0FFF1FE25595
:10022000FF4FA591B4919FB7F8948C91611103C0D1
:1002300010958123 01C0812B8C939FBF0F90DF917C
:10024000CF911F910F910895CF93DF93282F30E026
:10025000F901E859FF4F8491F901E458FF4FD49117
:10026000F901E057FF4FC491CC2391F081110E9416
:100270008C00EC2FF0E0EE0FFF1FEC55FF4FA59127
:10028000B4912C912D2381E090E021F480E002C014
:1002900080E090E0DF91CF91089508950E94A7013A
```

However, we can **Program** in easier languages which can be translated to Dumbo language. Or, high level languages can be compiled to low level languages. Programs written in a high level language, C++ can be compiled to a hex file, which is then uploaded to the arduino. You can also write code in Assembly, a low level language, and compile it to hex.

The Arduino sketch for our algorithm looks like this. It will be explained in further sections.

```
const int buttonPin = 2;
const int ledPin = 13;

int buttonState = 0;

void setup() {
  pinMode(ledPin, OUTPUT);
  pinMode(buttonPin, INPUT);
}

void loop() {
  buttonState = digitalRead(buttonPin);
  if (buttonState == HIGH) {
    digitalWrite(ledPin, HIGH);
  } else {
    digitalWrite(ledPin, LOW);
  }
}
```

You can not upload the above code directly to the Arduino. It has to be compiled to a lower

level language (C++ is a high level language). For that we need a **Compiler**. Some people have combined an editor, compiler, uploader and other nice things into a single package, the Arduino IDE.

Note that the IDE is not necessary to work with, but recommended. You can use your own language, editor, compiler and uploader.

2 Setup the Arduino IDE

2.1 Download and install the IDE

The relevant packages can be obtained from `www.arduino.cc/en/Main/Software`. In the time the package downloads, you can go through the next section.

2.1.1 Windows

Download and run the installer.

2.1.2 Linux

Download the `arduino-1.X.X-linuxXX.tar.xz` archive. Open the terminal and run the below commands one by one. Navigate to the `Downloads` directory, extract the archive, move it to `opt` directory and execute script to create desktop shortcut, menu item and file associations.

```
cd Downloads
tar -xvf arduino-1.X.X-linuxXX.tar.xz
sudo mv arduino-1.X.X /opt
cd /opt/arduino-1.X.X/
./install.sh
```

2.1.3 Browser

Go to `create.arduino.cc/editor/`

2.2 Test

Open the IDE. Connect the Arduino to the computer using a USB A to USB B cable. Under File, Examples, 01.Basics, select Blink. Under Tools, Board, select the proper board. Under Tools, Port, select a port. Below the menu bar, click on the second circle with an arrow. Check the Arduino board. An LED should be blinking.

If you are getting errors, reset the Arduino board and reconnect it to the computer.

3 Variables and operations

3.1 Variables

You can store data in **variables**. There are different types of variables: integers, decimal numbers, strings, characters, booleans, etc. Before you can use a variable, you have to create it by writing a definition statement. All statements end with a `;`.

```
int apples;
```

The above statement creates a variable called `apples` of type `integer`. All `integer`s are granted `2 bytes` of memory in Arduino. Presently, the memory contains a `garbage value`[1]. You can change the value stored in `apples` with the help of an assignment statement.

```
apples = 7;
```

You can define and assign value to a variable in a single statement.

```
int apples = 7;
```

You can define different types of variables.

```
int apples = 5;
float applejuice = 4.5;
char initial = 'A';
bool isfruit = true;
```

3.2 Expressions

You can do mathematical operations in the sketch.

```
1+2;          //Evaluates to 3
5-9;          //Evaluates to -4
apples*5;     //Evaluates to 28, since apples = 7
24/4;         //Evaluates to 6
sizeof(int);  //Evaluates to 2
```

`+`, `-`, `*`, `/` and `sizeof` are examples of operators. All operators need one or more arguments or operands.

Another operator is `%`, the `modulo` operator.

```
1%2;          //Evaluates to 0
2%2;          //Evaluates to 1
3%2;          //Evaluates to 0
13%apples;    //Evaluates to 6
```

What will this expression evaluate to?

```
1+5*3%4;
```

While all operators have a precedence level, it is not easy to remember the order. We can instead use brackets to control the order of evaluation. The below expression is much easier to understand.

```
1+((5*3)%4);
```

[1] You will see later that static and global variables are by default initialized to zero.

There are also boolean operators, AND `&&`, OR `||` and NOT `!`.

```
1&&0;          //Evaluates to 0
true||0;       //Evaluates to 1
!3;            //Evaluates to 0
!false;        //Evaluates to 1
apples||false; //Evaluates to 1
```

In boolean logic, zero is equivalent to `false`, and any non zero value is equivalent to `true`. You can also compare using `==`, `!=`, `>`, `>=`, `<` and `<=`.

```
5 == 5;           //Evaluates to 1;
apples != (8-1);  //Evaluates to 0;
5 <= -5;          //Evaluates to 0;
```

As you may have guessed, variables can be assigned as well used as arguments in, an expression.

```
float applejuice = apples * 0.9; //Creates applejuice and sets it to 4.5
```

3.3 Memory

When you define a variable, a chunk of memory is reserved and a **pointer** to the memory location is returned to the variable. The pointer for `some_var` is given by `&some_var`. The value stored at the memory location pointed to by `some_ptr` is given by `*some_ptr`.

You can also create a variable that can store a pointer.

```
int* ptrA;    //ptrA can store a pointer to a memory location that stores an integer
char* ptrC;   //ptrC ... character
```

Usually, the assignment is not done directly, but by usng the `&` on some other variable.

```
ptrA = &apples;
ptrC = &initial;
```

3.4 Types, typecasting and overflows

You know that the expression `6/2;` will evaluate to `3`. But what about `6/4;`? No, it won't evaluate to `1.5`, but to `1`. You can see that `1`, `2`,..., `6` are all `int`s, but `1.5` is a `float`. In these two expressions, both the operands are of type `int`. The operation is conducted in the "highest" data type of the operands, so is the type of the result of the evaluation. In this particular case, the fractional part is dropped to convert `1.5` to an `int`.

```
1/7;       //0    (int)
6/4;       //1    (int)
6.0/4;     //1.5  (float)
6/4.0;     //1.5  (float)
6/(float)4 //1.5  (float)
```

In the third and fourth example, one of the operands is a `float`, which also happens to be the highest type. So, all the operands are first converted to, and the result is given, in `float` type. In the last example, a **cast operator** has been used, which converts the `int 4` to a `float`. Either method can be used.

`int` type variables have 2 bytes of memory, and can only hold numbers in the range -32768 – 32677. If the result of some calculation exceeds this range, **overflow** occurs. `32677+1;` evaluates to `-1`, as `32677` and `1` are `int`s, so is the result. `(long)32677+1;` will evaluate to `32678`, a long integer. `long`s have 4 bytes of memory and can store in -2,147,483,648 – 2,147,483,647.

Although, `float`s can store numbers as large and small as $\pm 3.4028235 \times 10^{\pm 38}$, they should be used sparingly, as floating point math on the Arduino takes up a lot of memory and time.

3.5 Arrays

Arrays are a group of variables where each variable has an index number. Arrays are useful for storing similar data. You can declare an array in the below ways:

```
int myInts[5]; //Uninitialised
int senVal[3] = {6, 3, 9}; //Initialised
int myChar[] = {'a', 'b', 'c', 'd'}; //Automatically creates array of size of number of
    elements
char myFruit[6] = "apple"; //Creates a C string
```

If you want to use a character array as a string, the element after the last useful character should be `\0`. All strings have a `\0` at the end, hence, `apples` string requires at least a 5+1, 6 sized character array.

Array indices start from zero. Thus, the last element of an array of size `n` has index `n-1`. You can access an array element by its index:

```
myInt[3] = 43; //Assigns 43 to fourth element
senVal[2];     //Evaluates to 9, not 3
myChar[0];     //Evaluates to 'a'
myFruit[5];    //Evaluates to '\0'
```

4 Sketches

Code that you write in the IDE is a sketch. It is not the same as C++ code, but very similar. Any sketch is made up of statements. All statements must end with a `;`. Arduino sketches have `.ino` extension. All sketches have `abc/abc.ino` as the directory structure.

On opening the IDE, you will see the below snippet in the code window:

```
void setup() {
  // put your setup code here, to run once:

}

void loop() {
  // put your main code here, to run repeatedly:

}
```

These are two **functions**. The `setup` function is run once when the Arduino is powered up or reset. Then, the `loop` runs repeatedly until the Arduino is powered down.

4.1 Hello World

Let us now write a program that will print `Hello World` on our computer screen. Our computer and the Arduino are two separate computers. We will need to communicate between the two. Before we can do so, we need to set the speed of communication. Add the following line to the setup function.

```
3 Serial.begin(9600);
```

`Serial.begin()` is a function which configures the Arduino to send data through the serial port at a baud rate which is given as an argument. This function takes one argument of integer type. We have provided `9600` as an argument. Thus, the Arduino will send data at a rate of 9600 baud.

Now we can send data which will printed on our screen. Add the following line below `Serial.begin()`, inside the setup function.

```
4 Serial.println("Hello World");
```

`Serial.println()` is a function that takes a value and sends it through the serial port. In this case, we want to send a string. A string is enclosed within two `"`. `Hello World`, and not `"Hello World"`, will be sent through the serial port.

Our code looks like this:

```
1 void setup() {
2   // put your setup code here, to run once:
3   Serial.begin(9600);
4   Serial.println("Hello World");
5 }
6
7 void loop() {
8   // put your main code here, to run repeatedly:
9 }
```

Now we are ready to upload! Click the round upload button below the menubar. This will compile the code and upload it to the Arduino.

Before we can see anything on the screen, we need to set up a serial monitor. Thankfully, it is included in the IDE. Click on the magnifying glass icon on the right end below the menubar. Select a baud rate equal to the one at which Arduino is sending us data, ei., 9600. You will see this on the screen:

```
1 Hello World
```

4.2 Blink an LED

Let us blink the onboard LED, which is attached to pin 13. Our algorithm will be:

```
Set pin 13 to Output

Set pin 13 to HI
Set pin 13 to LO
Goto line 3
```

The equivalent sketch will be:

```
void setup() {
  pinMode(13, OUTPUT);
}

void loop() {
  digitalWrite(13, HIGH);
  digitalWrite(13, LOW);
}
```

We have used two new functions in our sketch.

`pinMode()` is a function that takes two parameters. The first parameter is the pin number, and the second is the mode of the pin. Pin mode can be `INPUT`, `OUTPUT` or `INPUT_PULLUP`. We want to use our pin as `OUTPUT`.

`digitalWrite` also takes two parameters, the first is the pin number. The second is the logic level, it can be `HIGH` (5V)[2] or `LOW` (0V).

Upload the sketch.

What is happening here? The Arduino is switching on, then off, and on and off again and again as fast as it can, faster than we can comprehend. The speed of blinking is only limited by the processor.

If want to blink the LED every second, there is a function `delay()` that can help us. Add this line after you set the pin high, and after you set the pin low.

```
delay(1000);
```

`delay()` takes one parameter, the number of milliseconds to wait. When the Arduino encounters this statement, it waits for the specified interval.

Upload the code and check the LED now.

4.3 A timer

Let us create a timer that will count the number of seconds for us. Our algorithm will be:

```
Create a variable named sec_elapsed

sec_elapsed = 0
Set serial communication baud rate

Wait one second
sec_elapsed = sec_elapsed + 1
Print sec_elapsed

```

[2]On 3.3V boards, it is 3.3V

```
10 Goto line 6
```

The equivalent Arduino sketch will be:

```
1  int sec_elapsed;
2
3  void setup() {
4    Serial.begin(9600);
5    sec_elapsed = 0;
6  }
7
8  void loop() {
9    Serial.print(sec_elapsed);
10   Serial.println(" seconds elapsed");
11   delay(1000);
12   sec_elapsed += 1;
13 }
```

You can see on line 9, how we have passed `sec_elapsed` as an argument to the `Serial.print` function. Also we have used `Serial.print` on line 9, and `Serial.println` on line 10. `Serial.println` also creates a new line and returns the cursor to the initial position, the same effect the enter key perfoorms in a text editor.

On line 12 we have used `+=` operator to increment `sec_elapsed` by 1. `some_var += other_var` is a shorthand for `some_var = some_var + other_var`. If you need to increment a variable by `1`, you can simply write `some_var++;` .

Another subtle point is the leading space in the string on line 10.

5 Arduino functions

Arduino provides a lot of core functions, very few of the basic ones are explained below.

5.1 Serial communication

- `Serial.begin()` : Set the serial communication baud rate. Takes one argument, the baud rate. Typically, you need to set it only once.
- `Serial.print()` : Prints data to the serial port. Takes a string or variable as an argument. You can also specify the format of a variable as a second, optional argument.
- `Serial.println()` : Does the same as above, in addition, appends carriage return `\r` and newline `\n` characters to the printed string.

5.2 Digital I/O

These functions work on all pins.

- `pinMode()` : Set the mode of the pin. Takes two arguments, first is the pin number, the second is the mode. Pin mode can be `INPUT`, `OUTPUT` or `INPUT_PULLUP`. Setting this is only necessary if you want to use the pin digitally.

`INPUT`: This is the default state of the pins. The pins have impedance (100M) and hence allow very less current to not disturb the circuit being read, and to avoid accidental shorting. How ever, pins in this state which are not connected to other circuits are susceptible to pick up electrical noise and report apparently random logic levels. This can be avoided by using pullup or pulldown resistors.

`OUTPUT`: Pins are in low impedance and can support current upto 40 mA. Be very careful and use a series resistor, as short circuits can damage the pin transistor or even the processor.

`INPUT_PULLUP`: Pins are pulled up by inbuilt 20K resistors to `HIGH`. There is no corresponding pulldown state for a pin inbuilt in Arduino. Pin 13 cannot used in this state, as it is connected to ground via an LED and a seres resistor, which pull the voltage down to 1.7V.

- `digitalWrite()` Takes two argument, first is the pin number and the second is the logic level, `HIGH` or `LOW`. If the pin is not set to `OUTPUT` mode before writing `HIGH` to it, the internal pull up will be activated and act as a current limiting resistor.
- `digitalRead()` Takes one argument, the pin to read the logic level from. Returns the level (`HIGH` or `LOW`). You can read pins both in `INPUT` or `OUTPUT` mode. If the pin is not connected to any circuits to pullup/pulldown resistor, it will pick up noise.

5.3 Analog I/O

- `analogRead` Takes one argument, the pin number. Returns the voltage at the pin in the form an integer between 0 and 1023. This range can be mapped to 0 to 5V.[3] to yield the actual voltage. Again, the pin will pick up noise if not properly connected. This function works only on analog pins (`A0...`).
- `analogWrite` Takes two arguments, the pin number and voltage mapped to 0–255 corresponding to 0–5V. This actually generates a PWM signal with a duty cycle to simulate the needed voltage. This function works on analog (`A0...`) and PWM (`~`) pins.

6 Control structures

Many times, we need to execute a code snippet repeatedly or on satisfaction of a condition. Control structures can control the flow of a sketch.

6.1 if...else

The `if...else` structure is used to check a condition and execute corresponding statements.

6.1.1 Simple if

```
1 if ( /*condition*/ )
2     // do something
```

[3] or 3.3V. You can set your own by using analogReference(). Google it.

If the condition expression inside the parenthesis is true, the following statement or block is executed. Multiple statements enclosed within braces is a block.

```
if ( /*condition*/ )
{
    // do this
    // and this
}
```

In the below snippet, the second statement is always executed. It does not belong to the if structure. Only the first statement belongs to the if structure.

```
if ( /*condition*/ )
    // do thing A
    // do thing B
```

An example:

```
if ( senVal >= 50 )   //No semicolon here!
{
    digitalWrite( ledPin, HIGH );
    counter++;
}                     //No semicolon here!
```

6.1.2 Branching

```
if ( /*condition*/ )
    // do something
else
    // do this
```

If the condition evaluates to `false` or `0`, The statement(s) after `else` is(are) executed.
You can supply `else` with another `if` or `if...else` structure.
In the below snippet, the second statement is executed if and only if conditions A is false and B true. Condition C is not checked. The third statement is executed iff conditions A and B both are false, and C is true. The fourth statement is executed iff all three conditions are false.

```
if ( /*condition A*/ )
    // do thing A
else if ( /*condition B*/ )
    // do thing B
else if ( /*condition C*/ )
    // do thing C
else
    // do thing D
```

An example:

```
if ( senVal >= 50 )
{
    digitalWrite( led1Pin, HIGH );
```

```
    counter1++;
}

else if (senVal >= 25) //What would happen if we checked this before senVal>=50?
{
    digitalWrite( led2Pin, HIGH );
    counter2++;
}

else
    counter0++;
```

6.2 switch

The switch structure is used when you want to execute particular statements if the value of a variable matches one of the values supplied in the structure.

```
switch ( some_var )
{
    case val_A:
        //1 do something when some_var equals val_A
        break; //exits the structure
    case val_B:
        //2 do this when some_var equals val_B
        //3 and this too
        break;
    default:
        //4 do this when some_var matches none of the above values.
        break;
}
```

The `break` statement is necessary to avoid the control to fall through all the statements below the satisfied case. If the second break statement was absent, and `some_var` was equal to val_B, statements 2, 3 and 4; all would have been executed. When the break statement is encountered, the control exits the switch structure.

6.3 while and do. . . while

While loops are used to execute code repeatedly until a given condition is `true` or `1`.

6.3.1 Simple while

The condition is checked before every iteration. If true, the following statement or block is executed and again condition is checked and so on. If false, the statement/block is skipped and control goes ahead.

```
while ( /*condition*/ )
{
    //do something
}
```

6.3.2 do. . . while

It is same as the above while loop, but the condition is not checked before the first iteration. Thus, the statement/block is guaranteed to execute at least once.

```
do
{
    //do something
}
while ( /*condition*/ )
```

6.4 for

For loops are used to execute a code snippet a number of times.

```
for ( /*initialization*/; /*condition*/; /*increment*/)
{
    //do something
}
```

The initialization is executed just once in the beginning. The condition is checked. If true, the statement/block is executed, and then the increment is executed. Again the condition is checked, and so on.

The below code will print all integers from 1 to 100.

```
for ( int i = 0; i<100; i++) //You can define a variable inside
{
    Serial.println(i+1);
}
```

The below code prints all powers of 2 upto 5000.

```

int i;     //You can define a variable outside.

for (i = 2; i<=5000;)
{
    Serial.println(i);
    i*=2   //You can increment i here too.
}
```

All the three parameters are optional in a for loop, but the two `;` must be present.

7 Functions

If you need to re use a code many times at different places in a sketch, it is better to create a function, that when called, will execute the code. This helps in debugging, modifications can be made at just one place.

You are already defining and using setup and loop functions. These two are required functions. Only they are called when the sketch is executed. Thus, we can define our own functions outside setup and loop. But to use them, they must be called inside either setup or loop.

7.1 Definition

This is how a definition of a function that returns the product of two numbers supplied as arguments or parameters looks like.

```
int myProduct (int num1, int num2)
{
    int result;
    result = num1 * num2;

    return result;
}
```

Generalized, a function definition looks like this:

```
[return type] [function name] ([argument 1 type] [argument 1 name], ... )
{
    //do something

    return [some data of return type];
}
```

The formal parameter you define in the parenthesis of function definition are just placeholders for the arguments which will be supplied when the function will be called. All variables defined in the parenthesis are replaced by a local variable copy or value (which ever is given) of supplied argument throughout the function.

If we don't need our function to return anything, return type can be set to `void`.

If a function expects no arguments, leave the parenthesis empty, do not omit them.

```
void myGreeting ()
{
    Serial.println("Hello World");
}
```

A definition can be put anywhere outside of other functions, including, setup and loop.

7.2 Call

To use a function, you can call it anywhere in your sketch by it's name. You must supply an equal number of arguments of matching variable type as in the definition.

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

    int num = 4;
    int product;

    product = myProduct(num, 5); //Stores 20 in product
    //You can pass both variables as well as data as parameters
    myGreeting();     //Prints Hello World to the Serial port
}

```

```
13 void loop ()
14 {
15     ...
16 }
17
18 //Function definitions
19
20 int myProduct (int num1, int num2)
21 {
22     int result;
23     result = num1 * num2;
24
25     return result;
26 }
27
28 void myGreeting ()
29 {
30     Serial.println("Hello World");
31 }
```

7.3 Scope

The area where a variable exists and can be seen is known as the scope of a variable. The scope depends on the place where a variable is defined. There are three places where you can define variables:

Local variables defined inside a function or block.

Global variables defined outside of all functions.

Formal parameters defined within the parenthesis of function definitions.

7.3.1 Local variables

All variables defined inside a function body or block are local to the function or block. They can be accessed only inside the function or block. They do not exist outside.

```
1  void setup()
2  {
3      Serial.begin(9600);
4      int var_a = 4 //var_a is local to setup function
5                    //It's scope ends with setup function
6  }
7
8  //There is no var_a here...
9
10 void loop()
11 {
12     //...nor here
13     Serial.println(var_a);
14     //This code will not compile
15 }
```

```
void setup()
{
    int var_a = 4
}

//There is no var_a here

void loop()
{
    //var_a of setup does not exist here
    //In fact you can declare another var_a that belongs to loop function
    int var_a = 7;
    //Ths code will compile
}
```

At the end of the block or function, the memory occupied by local variables is deallocated.

7.3.2 Global variables

All variables defined outside all function are global. Global variables have global scope, they exists and can be accessed directly or modified anywhere in the sketch.

```
int x = 3;

void setup()
{
    Serial.begin(9600);
    //x exists here...
    Serial.println(x); //x has global scope, can be accessed anywhere
}

//...and here...

void loop()
{
    //...and here too
    Serial.println(x++);
}
```

An interesting fact is that when a local variable of the same name as global variable is defined, the more local variable takes preference over the global variable in that scope. You can check by printing `x` to serial port in the setup and loop functions.

```
int x = 3;

void setup()
{
    Serial.begin(9600);
    int x = 7;
    //For this scope, ei., this setup function, the new local definition of x takes
        preference over the global definition.
    x++;
```

```
9     //Any changes you make to x here are local.
10    Serial.println(x); //Prints 8 to the serial port
11 }
12
13 //Here x is the global variable
14
15 void loop()
16 {
17    //Here x is the global variable
18    Serial.println(x); //Prints 3 to the serial port
19 }
20
21 //Here x is the global variable
```

7.3.3 Formal parameters

These are the parameters you define in a function declaration and use them in the function body. They are also treated as local variables and have local scope, and take preference over global variables.

```
1  int a = 7, b = 12;
2
3  void setup ()
4  {
5     Serial.begin(9600);
6     myFunction(3, 5);     //Prints 3 and 5
7     Serial.println(a)     //Prints 7
8     Serial.println(b)     //Prints 12
9  }
10
11 void loop ()
12 {
13    ...
14 }
15
16 //Function definitions
17
18 void myFunction(int a, int b)
19 {
20    Serial.println(a);
21    Serial.println(b);
22 }
```

7.4 Passing variables: by value and by reference

7.4.1 Pass by value

When we pass a variable as an argument to a function, a local copy of the variable is created. What ever the function does with this local copy has no effect on our variable. This is known as 'passing by value'.

```
void myFunction(int num)
{
    num++;
    Serial.println(num);
}
```

```
int apples = 7;

Serial.println(apples);  //Prints 7
myFunction(apples);      //Prints 8
Serial.println(apples);  //Prints 7, as the increment was done on the local copy of
    apples, not apples itself
```

If you want a function to update a variable, you cannot pass the variable as a parameter. You can either assign the return value of a function to the variable...

```
void setup ()
{
    int num = 6;
    num = myIncrementer(num, 3);
}

void loop ()
{
    ...
}

int myIncrementer (int var, int inc)
{
    var += inc;
    return var;
}
```

...or declare the variable as global so that it can be seen and modified inside the function.

```
int num = 6;

void setup ()
{
    myIncrementer(3);
}

void loop ()
{
    ...
}

int myIncrementer (int inc)
{
    num += inc;
}
```

In the first method, you need the function to return and also assign it to the variable. In the second function you need a global variable. Both of these can be avoided by using pointers.

7.4.2 Pass by reference

When passing a variable by value to a function, a local copy of the variable is made for the function. This means that our original variable and the copied variable do not share the same memory space. They are alloted different chunks of memory. Thus any changes you make in the copied variable are made at the second memory location, and the original memory location remains intact.

```
1  void setup ()
2  {
3      Serial.begin(9600);
4      int a = 5;
5      Serial.println(&a);   //Prints memory pointer of a
6      myFunction(a);        //Prints poiter of local copy of a
7  }
8
9  void loop ()
10 {
11     ...
12 }
13
14 void myFunction (int var)
15 {
16     Serial.println(&var);
17 }
```

Thus, you can pass a pointer to a function and access the value pointed to by the pointer.

```
1  void setup ()
2  {
3      Serial.begin(9600);
4      int a = 5;
5      Serial.println(a);
6      myFunction(&a);
7      Serial.println(a);
8  }
9
10 void loop ()
11 {
12     ...
13 }
14
15 void myFunction (int* ptr)
16 {
17     (*ptr)++;  //increment the value pointed by ptr
18 }
```

If you run the above sketch, you will see that the original variable `a` has been modified.