Learn to Code with Arduino

Learning to code in C++ (Arduino language) using minimal extra hardware

**Corey Rice**

Mayfield City Schools

Cleveland, Ohio, USA -- 2018

Updated 2019

# Intro to Arduino and C++

An Arduino is a complex object, with many parts rolled into one cheap package. The original four inventors of Arduino were based in Italy, and they designed it to be something they could teach to art students in a weekend. On the boards are a few main components:

1. The Microcontroller of the board [the largest black chip, which is the brains of the board]
2. Hardware to connect to a computer’s USB port [a small chip and USB-A port]
3. A voltage regulator for other power supplies [a 2.1mm center positive jack for up to 12Volts]
4. The pin headers to get direct electrical access to most of the Microcontroller’s Input/Output pins. [This is really what the entire board is built to give easy access to use]

There is a lot of fascinating behind-the-scenes work that the Arduino people have abstracted away. From a beginner’s perspective, all you need to do to use an Arduino is 1) hook it up to a computer’s USB port and 2) write code on the Arduino IDE to control its operation, and 3) upload the code to the board and see if it works as expected.

Here we will be learning the basics about the code to control an Arduino. There are lots and lots of fancy things that you can build with an Arduino (as a quick YouTube search will show) but most of these have relatively simple code to control them. So, to build up the needed skills to build advanced things, we will focus on the code-writing skills first. Because all complex actions you can see and Arduino achieve are only ever a fancy extension of switching the Arduino’s pins off and on at specific time intervals or checking voltages on input pins. That’s all a microcontroller can really do! But first, we will learn how to code in C++/ Arduino Language so that we know how to control the Arduino, before adding in the complexity of extra electronic components.

The computer language ‘C’ is very old (in computer terms) and C++ is derived from the original C language. Arduino’s language is basically just C++, with a few Arduino-hardware-specific keywords added in. In fact, Arduinos will run on C++ code in the IDE and all official libraries are written in C++ (you can tell by the ‘.cpp’ file extensions of the libraries).

The first thing to do when learning how to code in a new language, is to look at some examples. You want to fully understand each letter and character of the examples, so that you can use these to develop your own programs (Note: Arduino calls programs ‘sketches’).

## Things you will need to get started:

* A computer with a Windows, Mac or Linux-Based operating system & at least one free USB port
* An Arduino Uno (other versions will do, but it is best to start here)
* A USB cable that will connect between the computer and Arduino. This will typically be a standard USB (type B on one end and a type A) on the other end

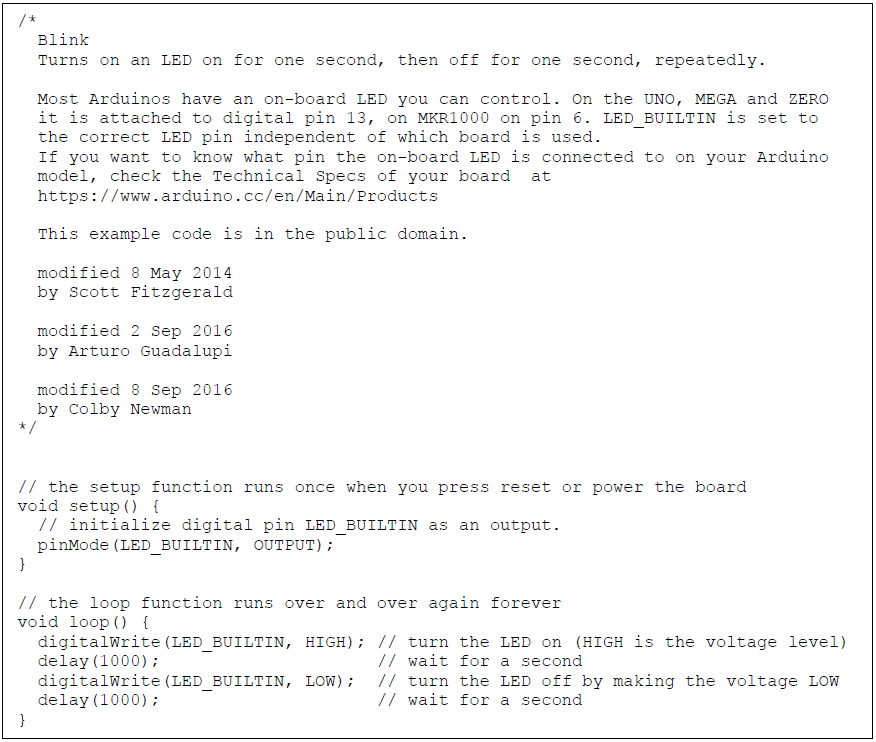
## Things to learn to a later time:

It is generally good for you to be aware of some of the related topics that will not be covered by this tutorial, but are very closely related, or the next steps in a larger course of study.

* Arduino libraries (prewritten code you can integrate into your own work)
* Basic Electronic circuits (to extend the functionality of the Arduino beyond the integrated parts)
* Battery use, care and circuit power consumption
* Advanced Electronic circuits (to push the limits of what you can do with an Arduino)
* Circuit design, non-Arduino chip programming, and board manufacturing (to advance beyond Arduino)

# Getting Oriented with Programming on the Arduino

## Looking at the basic ‘blink’ example sketch from Arduino

You can find this example sketch in the Arduino IDE by using the menu to navigate to FILE >> EXAMPLES >> BASICS >> BLINK

This area is a paragraph comment. You can tell because it has the /\* and \*/ around what it has to say.

You can see there are some one-line notes to help you understand the code. These are preceded by //

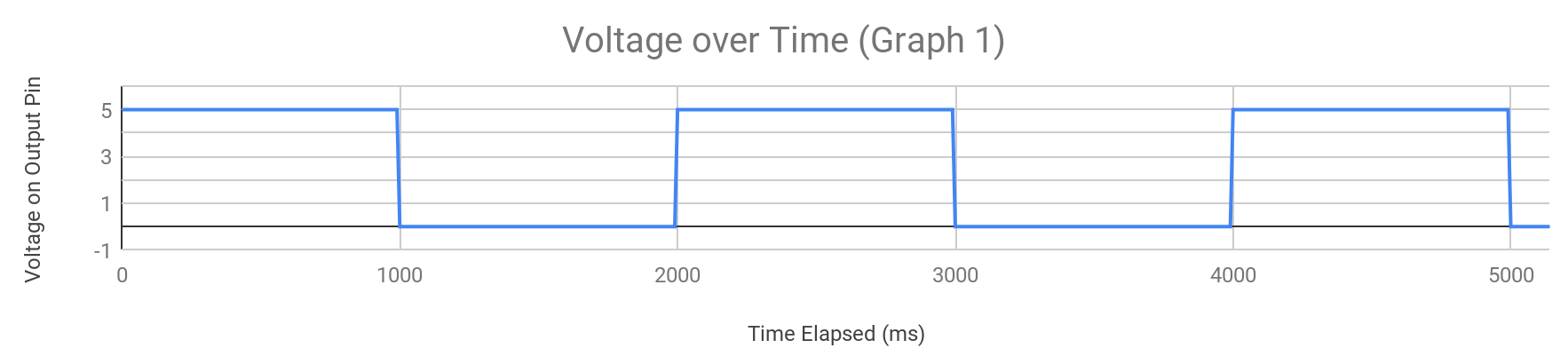
Comments in a program are often the best way to tell how the program works. It is good practice to have them in your code, and all the official example sketches have helpful comments present. You will want to write your own comments as well, so you don’t forget what parts of your program are doing. This may seem silly as a new programmer, but it is necessary as programs become increasingly complex. It is best to get in the habit of writing good comments now…

The program above has two functions that run. The *void setup()* and *void loop()* functions. These must be in every Arduino sketch and there will be much to say about them in the future. For now you need to know what the comments above tell you: the setup gets things started, and the loop runs forever while the Arduino is powered.

This program is designed to blink an LED, so it implements a few types of functions to do so. The pinMode() function sets up the use of the Arduino’s pin for the LED, and this is why it is in the void setup(). pinMode() requires two inputs that go in the parenthesis. The first input tells pinMode which pin it will be working on [LED\_BUILTIN, in this case] and the second input has two options: INPUT and OUTPUT. The LED will be used as an output in this sketch since the microcontroller will be sending power to it, rather than using the pin to read a voltage. As we progress, knowing the difference between an input and output for the Arduino will be critical, as will be remembering to set up all of your outputs with pinMode(). If an output isn’t set up, it usually won’t work right; or won’t work at all.

The other functions are delay() which is actually pretty easy. It just has the microcontroller sit and do nothing for the specified amount of milliseconds. The input number is how many thousandths of a second the Arduino is to wait.

The last function in this first example sketch is digitalWrite(). The digitalWrite() function is what controls a digital output pin. It will set up the output pin to either deliver 5 volts to be output from the pin, or tie it to ground/zero volts. Without trying to go too far into the quagmire of explaining circuits: supplying a voltage is what determines if the LED is on or not. If 5V are present on the output pin (and the other side of the LED circuit is tied to ground) then current will flow and the LED and its resistor are on. If the output pin is tied to ground (and the other side is also tied to ground) no current will flow through the LED and resistor, and the LED will not glow.

Getting digitalWrite() to control the 5 volt output is relatively easy. The function has two inputs. The first input tells the digitalWrite() function what pin it will be controlling, and the second input tells the function what state to write to the pin. There are only two options to control the pin, but many synonyms to describe those options. Above, “HIGH” is used to indicate on and “LOW” is used to indicate off. However, on can also be indicated by using any of these options: true, 1, HIGH and off can be indicated by any of these options: false, 0, LOW. There are many ways that you can tell the function to write the output on or off, but they all correspond to a binary digital logic ‘on’ or ‘off.’ 

## Next steps: Working to Understand Code

Once you have identified all of the keywords and logical steps of the code, the next thing that you will need to do is create a cohesive mental model of the program’s operation. For many people learning to code this is an assumed step in the process of understanding. However, it is critical to say here that anyone who *wants to* improve at coding will need to start to think in computational ways.

Everything outlined in the section above was the ‘how’ of the code. Explaining what functions like digitalWrite() actually do in the program are the first steps to understanding, but now we need to focus on the ‘why’ of the program. An inquisitive learner should see this program for the first time and start asking questions like: “why is there a delay after the digitalWrite() function?” Why does the program only need to digitalWrite() the pin HIGH and LOW once to make it blink continuously?” “What would happen if the delay times were much smaller, or if they were unequal?” Questions like these approach the ‘why’ of the program. They allow the programmer to understand the reasons that certain actions are taken, and in their precise order or logical sequence.

The first steps to getting to understand the ‘why’ of a program is to understand the ‘how.’ You need to know what the functions do, before you can get to understanding why they were included in the program. Once you have the ‘how’ (at least mostly) covered, you can get on to understanding the ‘why’ of the program. Once you can understand the ‘why’ of some programs, you will eventually grow in skill until you understand things well enough to chart your own ‘why.’ This will be simple at first. Maybe you want a program so that when you press a button, a light turns on. This example would be a simple ‘why’ that will inform the functions you implement to get to your own ‘how’ in a program of your own creation.

At first, you will need to consume the content of many programs. It will involve working to understand their ‘how’s and ‘why’s. Playing with the parameters of the program until you understand every part, and how a change impacts the behavior of the program. This is important to do while the programs you work with are small, because making such manipulations is more challenging in larger programs. Eventually, you will develop in skill enough until it takes less effort to understand first the ‘how’ and then the ‘why’ of a new program. Simultaneously, you will get better at designing you own ‘why’s and then ‘how’s of programs you want to write on your own. Practice will be the only thing that can really advance this skill. You will need to read programs, manipulate programs and write your own programs to get better. Sometimes you will get things wrong. But after some re-working and some learning, you will get things back on track. Experience gained from overcoming your own programming struggles is what makes you a better programmer.

## Getting the Sketch to the Arduino

The **Arduino IDE** looks like the image to the right on a Windows 10 computer, at the time of this writing. It is a program that you download, for free, from Arduino.cc to get code on to your Arduino via USB cable. There are some [unofficial](https://chrome.google.com/webstore/detail/chromeduino2/llclpgogfbmiicabgcfbndeokekmggpm?hl=en) and [official](https://chrome.google.com/webstore/detail/chromeduino2/llclpgogfbmiicabgcfbndeokekmggpm?hl=en) versions that run on chromebooks as well. No doubt, it will have changed a bit before you get to work. There are a few important things to point out on the IDE interface if you want to write code, and get it going.

First is the menu across the top. Look at the options and see what is going on. In this image, the ‘Tools’ section is open so that you can see two important settings. First, you need to notice the “Board:” setting about half-way down. This is the area where you select the type of Arduino you’re working with. [If you’re not sure what type of Arduino it is, that it is printed on the board itself. If you can’t find the board type, the safest bet is to go with the Arduino/Genuino Uno. It’s the most common type.] The board selection makes sure that keywords like ‘LED\_BUILTIN’ work properly and everything compiles for the right chipset.

The next thing to notice is the ‘Port’ section. In the image it is grayed out, because there was not an Arduino plugged in at the time of the screenshot. If you have one plugged in, it should automatically select the correct port. However, you may occasionally need to go in here and help the computer select the correct port. USB ports are referred to in different ways on different operating systems, so you may have to try a few options before you really connect to your Arduino. Windows sometimes gives phantom COM1 & COM3 ports. It will be fine – play around until you are comfortable.

The circular button near the top left with a checkmark is the compile button. This button will take the code you have written in C++ and compile it for the Arduino board, but not send it to the Arduino board. This can be very helpful if you want to test your code, without having an Arduino plugged in to the computer. You will be able to see if there are any compile errors in the code and be able to address them. The notes and error messages that come up during the compilation of a sketch will be shown in the black field near the bottom of the Arduino IDE.

The circular button with a right arrow will both compile your sketch and upload it to the connected board, via the selected port. This is probably going to be the button you click most often on the Arduino IDE. It gets your programs into machine language, and communicates that to the microcontroller.

The last important button for us to investigate at this time is the square one in the top right, that has a magnifying glass inside. This will open the Serial Monitor. The serial monitor opens up a link to the USB port attached to the Arduino. This is mainly used to send and receive messages or information between the Arduino and a computer. [Although occasionally, if the Serial Monitor window is open, it will block your ability to upload a new version of the sketch to the Arduino.] The Serial Monitor is a great tool to use to debug your programs. You can have an Arduino send messages to a computer to see how the program is functioning. You can also access the Serial Monitor through the hotkeys CTRL+SHIFT+M. The Serial Plotter is also worth noting and it can be accessed by using CTRL+SHIFT+L. The Serial plotter is a derivative of the serial monitor, and it can graph properly formatted numbers coming out of the Arduino as well. This is easier than writing your own graphing software to run on a computer while the Arduino spills out simple data.

\*\*The colorful text in the code area is there to help show different recognized words in the Arduino IDE. It’s helpful.

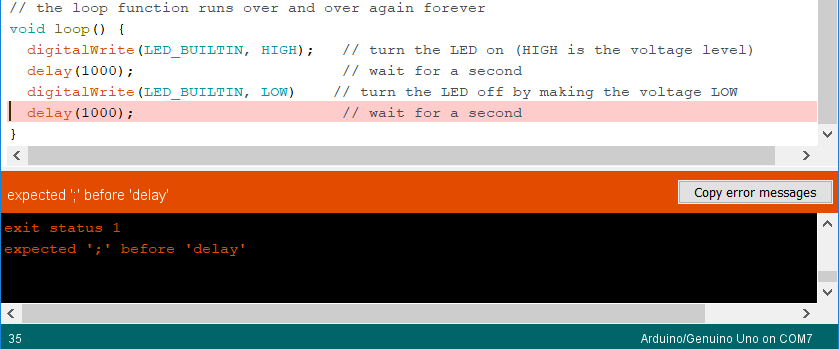
## Compilation Error Messages

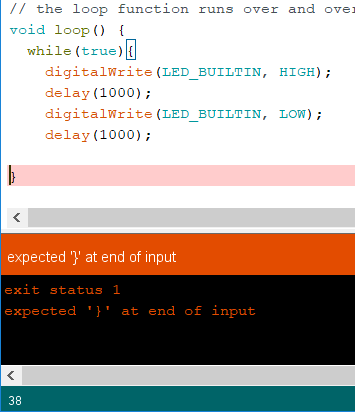
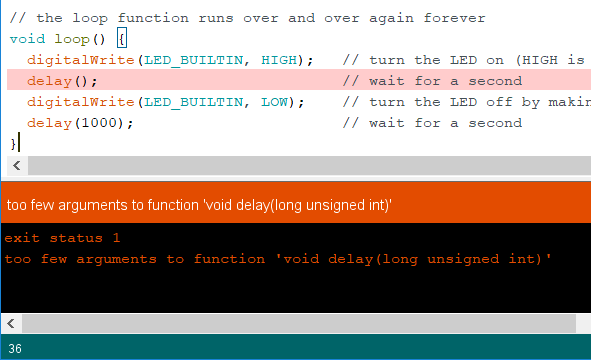
As you write your code, you are creating a set of instructions for the computer to follow so that it can complete the task you desire. The **compiler** is a behind-the-scenes part of the Arduino IDE that translates your code to a series of binary 0s and 1s that the microchip can understand. Machines cannot understand code directly. They must have this translation from C++ to machine code to work. In the past, programmers had to write code in harder-to-learn machine and assembly languages that were much closer to what the microchips (or a computer) can understand. The coding languages you will use to program on any modern platform were created several generations after those early programming languages.

Think of the compiler as your own personal translator. It keeps you from needing to learn machine language; and, therefore can be thought of as a helpful friend. (But a little needy.) All of the **syntax** of your code must be just right for the compiler to translate from C++ to machine language. This means that your curly brackets must be in place, and lines of code should end in a semicolon. The syntax must be right for the compiler to work. The compiler will not make guesses to try and help you. It can only identify places where the syntax is wrong, but it does not know how to fix them.

The fact that the compiler can only identify certain problems has a few implications. First, it can only identify problems with the ‘how’ of your programs, but not the ‘why’s. The compiler will be able to tell if your functions or syntax is not typed correctly, but it will not be able to tell if your program will do what you want. Second, the compiler can only guess as to where the problems are. It will be helpful and try to identify the line of code that a problem is in; but a single error message may represent a problem that spans several lines of code. It will be up to you to figure this out.

All in all, the compiler is a great tool you should be very thankful to have (so you can use higher-level languages) but it will take some time to practice using and optimizing your work with the compiler as a new friend in your programming.

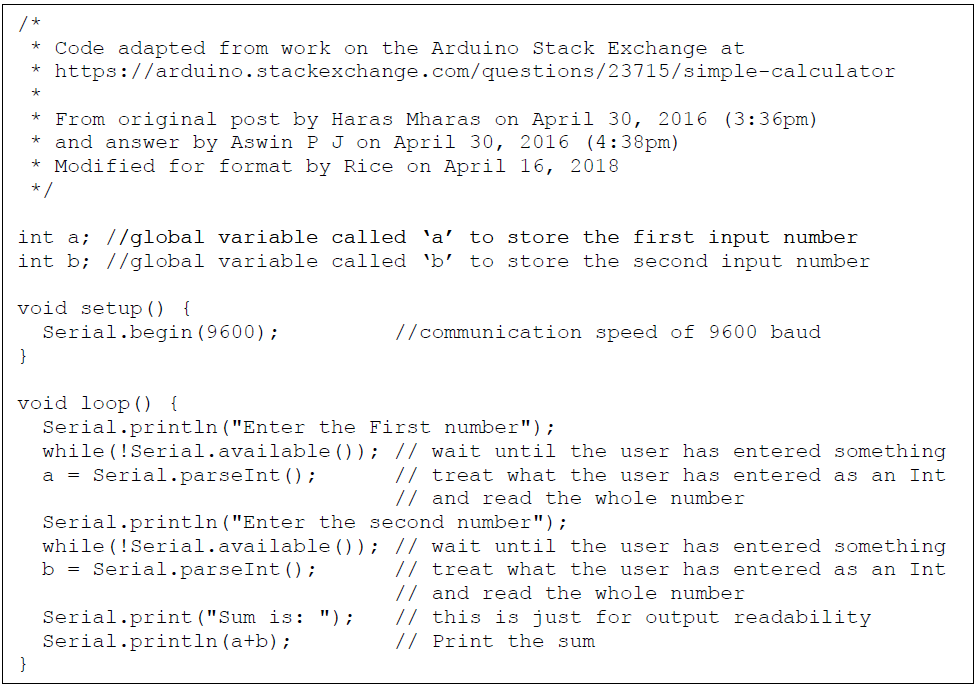
**Error messages** show up in the black field at the bottom of the Arduino IDE. The image to the left shows a typical error message that a beginner may see when rearranging the ‘blink’ example sketch. The compiler identified the first syntactic error it found with a red highlight on the line of code, and then gave a message as to the expected cause in the black field at the bottom of the IDE window. 

The error message to the left is the result of a missing curly bracket. You can see the red highlight is close, but not exactly on the right spot. 

To the right, delay() was not given a time to wait. This is too few inputs for the delay() function. The compiler lets you know, since it cannot translate it into machine code.

These are just a few examples of the errors that the compiler might give you when you are working on programs. You should always read these, and consider them hints to make your programs work in the way that you want them to.

## Serial.print messages back to the computer

This sketch is significantly different from the first example investigated. Many of the same parts are there, such as the ‘void setup()’ and ‘void loop()’ but there are also several new parts to understand. 

The most important of new item presented in the sketch (and the reason why we are looking at this particular program) is the ‘Serial’ communication. **Serial** is a library that was written to be included in the standard distribution of Arduino, and has many functions. Here, we will look at only a few of the functions in Serial, and we will use them repeatedly hereafter.

**Serial.begin()** is a good place to start understanding Arduino’s serial function. But first, we need to know a little more about serial communication. Serial communication is a very simple way for different computer chips to talk to each other. Serial communication consists of sending and receiving a series of 1’s and 0’s needed to communicate in binary, at a preassigned clock speed that both the sender and receiver of information must agree upon. In the case above this speed is 9600 baud (baud is not exactly “bits per second,” but the comparison is appropriate). This communication speed will need to be revisited later. Serial.begin() just initializes the serial communication between the Arduino and the computer, while determining the agreed-upon baud rate for the data transfer. The information is sent as voltages (like in Graph 1).

**Serial.print()** and **Serial.println()** are almost the same function, and very likely the functions you will use most often with Arduino. These are how you push information from the Arduino itself to the Computer. All that these functions do is print out the values that are passed into them. One thing that is important to note is that if you want the Serial.print() function to put out readable words, you need to place that text inside of quotes to indicate it is a string of characters (more on strings to come). The only difference between Serial.print() and Serial.println() is that the Serial.println() version will start a new line of text when it is done. The same could be accomplished by adding a “\n” to the end of any Serial.print() message, since this is just being done behind the scenes in Serial.println().

## Computational Logic --- The pieces and parts of thinking with a machine

Now we are ready for the actual coding. Once you are reasonably familiar with the environment you are using to write code, you can appropriately shift your focus to the code itself. As C++ is explored, we will find that there are many ways to express an idea. Many of these practices used to express an idea are common features of code, and you will see they apply in the code you find in C++ from someone else’s work, and even in other programming languages.

The fist distinction to draw is between 1) ‘procedural’ and 2) ‘object oriented’ code. **Procedural programming** is great for beginners because it is a series of commands, written in the order they are to be executed. This is not only good for beginners, it is how all code started. If learning how to write procedural code is like learning how to add and subtract in math, then learning object oriented programming is like learning how to multiply and divide. **Object oriented programming** will be covered later, but you should know that it is how all Arduino Libraries are written and is very good at handling things that your code needs to do repeatedly, or in an unpredictable order (usually based on sensor inputs).

As we begin to understand coding, the **if() logic** is the most important piece to understand. Everything else is based on the digital logic of if statements, even the physical hardware of the Arduino and computer itself. If() is a piece of logic that evaluates a statement. In the case to the left, that statement is the boolean variable named ‘a.’ If the variable is true, the if() logic will perform the next line of code and Serial.println() will push “true” to the serial monitor. If the variable were false, the if() logic would not execute the code inside the curly brackets, and instead move on to whatever is after the brackets. The if() logic takes as an input whatever is in parenthesis. In this case, it is the [boolean](#ry2zxjrodesn) variable named ‘a’ and the logic evaluates if the statement in the parentheses is true. [The thing in the parenthesis can be as complex as you like. Here it is quite simple.] Here, the if() condition will be evaluated as true, and then the program does whatever is in the following curly brackets of the if() function.

## Variables

If you are going to perform computational logic, you need something to make logical decisions about. Hence, variables. In C++ there are several variable types, and these all exist for a range of interesting reasons that are related to how a computer (of any size) allocates memory in bits and bytes. While getting started, you don’t need to know the deeper reason for why there are so many types of variables. (Languages like Python deal with this for you.) All you need now is to understand is how to implement the common variables, a summary of their pros and cons, and the limits of their use.

### Types of Variables (table)

These are the most common types of variables that a new programmer may use in Arduino/C++ but is not a comprehensive list. A quick online search will give a more comprehensive set, if you are ready. [Arduino variables…](https://www.arduino.cc/reference/en/)

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Description | Biggest Advantages | Biggest Limitations |
| [bool](https://www.arduino.cc/reference/en/language/variables/data-types/bool/) | A boolean variable: only be 0 or 1 | Smallest memory footprint | Only two possible values |
| [char](https://www.arduino.cc/reference/en/language/variables/data-types/char/) | A 1-byte [ASCII character](https://en.wikipedia.org/wiki/ASCII#Overview) | Low memory use, English | Limited set of variables |
| [string](https://www.arduino.cc/reference/en/language/variables/data-types/string/) | An array of char symbols | To make words & more | Difficult for logic use |
| [int](https://www.arduino.cc/reference/en/language/variables/data-types/int/) | Stores numbers -32,768 to 32,767 | Low memory use | Can ‘overflow’ max value |
| [long](https://www.arduino.cc/reference/en/language/variables/data-types/long/) | # from -2,147,483,648 to 2,147,483,647 | Store large +/- numbers | Can ‘overflow’ /4-bytes |
| [unsigned long](https://www.arduino.cc/reference/en/language/variables/data-types/unsignedlong/) | # from 0 to 4,294,967,295 | Store very large numbers | ‘overflow’/ no neg/ 4-by |
| [float](https://www.arduino.cc/reference/en/language/variables/data-types/float/) | Numbers w/decimals | Decimals | ‘overflow/Accuracy/4-by |

Each of the variables listed in the table merits further investigation. At some point, you should also become familiar with all of the variables available in Arduino & C++ programming, but this set will suffice for now.

### Bool, Synonyms, Math and Overflow

If we just look at ‘**bool**’ we can understand why more depth is necessary. Although ‘bool’ can only have a zero or a one for its value, these two numbers have many **synonyms in a program**. For example, the following list of items would all be acceptable representations of 1 for a bool: ‘HIGH’ ‘true’ ‘1’ The following list would all be acceptable representations of 0 for a bool: ‘LOW’ ‘false’ ‘0’.

It is important to know that if you have a bool with a value of zero and you add one, the sum is one. This may seem obvious. We all know this **basic math**: 0+1=1 However, the extension of this in computer science seems bizarre to many newcomers: if you add one to one, within a bool, the sum is zero: 1+1=0 This doesn’t make mathematical sense, but it is our first example of **variable overflow.** Since the ‘bool’ can only have values of zero or one in the single bit of memory for a ‘bool,’ the mathematical sum of two is too large for the allowed values. This means the variable would overflow back to the bottom of the set of allowed values. Interestingly, it will overflow by the amount of extra values beyond the set, so 1+2=1 and 0+2=0 and 0+3=1 These three simple additions each make no mathematical sense, but they are normal ‘overflow’ behavior that any memory-restricted variable exhibits. Several variables in the table above warned against overflow, because it can be frustrating. Often, people just learning to code will think about math instead of memory...

### Implementing Variables

Creating and using a variable is a two-step process in Arduino & C++, but these two steps are often done back-to-back. First, you must **declare** a variable. This means you state the type and name of the variable, such as these examples:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| bool a; | int amount; | int numTrack; | long t; | char L; |

Convention for naming variables is that 1) the names are short, 2) don’t start with a capital letter, and 3) never have spaces. Although variable names tend to be short, you may make them longer to make your code more readable. If this is that case, any time there is a second word (avoid more than that) the second word is designated with a capital letter rather than a space between words.

The second step to implementing a variable is assigning a value. **Assignment** is done with a single equals sign. So assigning values for each of the variables declared in the one-line table above could like the table below:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| a = true; | Amount=7; | numTrack = 12418; | t=1920349820; | L = b; |

Each of these variable assignments uses a single equal sign, but it is worth noting that the spaces on either side of the equal sign do not matter. The variable must come first, and whatever it is assigned to be comes second.

When implementing a variable, both the declaration and assignment must happen before it is used, but these are often done in a single line.

## Math and More

It takes an unusually long time to get oriented in programming. That can scare some people, but it shouldn’t. Once the basics are learned, the actual act of programming isn’t too hard. Now it's time to introduce a programming feature that should feel familiar: MATH.

Computers do math. There are some interesting conversations to have about the mechanisms of how computers do math, but that is for another time. Right now, just think of a computer (at any level) as a calculator. If you have two numbers, you can add, subtract, multiply, divide or find the modulo. Probably only that last operation is new to you.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| + | Addition | - | Subtraction | \* | Multiplication | / | Division | % | Modulo |
| 35 + 4 = 39 | | 35 – 4 = 31 | | 35 \* 4 = 140 | | 35 / 4 = 8.75 | | 35 % 4 = 3 | |
| **Sum:** One number and another number together | | **Difference:** One number, with the following amount removed | | **Product:** The total when there are {first number} groups of the second amount present | | **Quotient:** when the first amount is broken up by the second amount | | **Remainder:** leftover from division, stopping at integer quotient amounts | |

Many programming languages will have these five basics built-in. There are other math operations (like exponents or linear functions) that can be explored further, but they do not need to be explained as we get started.

## Variable scope

Variables in most languages, including arduino’s version of C++, have a useful/ odd property called [scope](https://www.arduino.cc/reference/en/language/variables/variable-scope--qualifiers/scope/). This property describes where and how the variable can be used, based on where it was created. If a variable as a type of ‘bucket’ to hold information, then think of scope as the rules where that bucket can be used. A variable made outside of any function in a program is a global variable, which can be used by all the functions of the program. If a variable is declared within a specific function of a program, then it is a local variable that can only be used in that function. Making a local variable may seem silly at first, but it is actually best practice, whenever possible. That’s because it means you can’t accidentally change the value in another function, and you can re-use the variable name in other (separate) functions.

# 

# 

# More to Learn about Programming

There is a lot to learn as you start with any new programming language. However, you will find that once you know one language, it is much easier to pick up a second. This pattern also holds: the more languages you learn, the faster you can get acquainted with a new one. But mastering any language can take years.

Here is a link to an article on some prime languages to learn, as long as the link holds: <https://hackernoon.com/5-programming-languages-every-master-developer-should-learn-a3929a8c6f69>

Specifically, the above article recommends this paid resource for learning C and C++: <https://www.udemy.com/c-programming-for-beginners/?ranMID=39197&ranEAID=JVFxdTr9V80&ranSiteID=JVFxdTr9V80-VIGgdrBZCMzdqmzLe6flUg&LSNPUBID=JVFxdTr9V80>

However, this is a promising resource that is still under development: <https://www.learn-cpp.org/>

And many universities (including the University of Cincinnati) use an online textbook such as this one to teach their students intro-level coding: <https://www.zybooks.com/catalog/programming-c-plus-plus/>

However, all of this pales in comparison to the traditional method of learning in the internet age: 1) read random websites you find in a relevant search, 2) try it on your own computer and 3) keep track of what works and what doesn’t.

A Collection of Exercises to Build Your Skills

Starting on the next page will be a litany of things that will test and develop your skills in C++

# Getting Started with Exercises

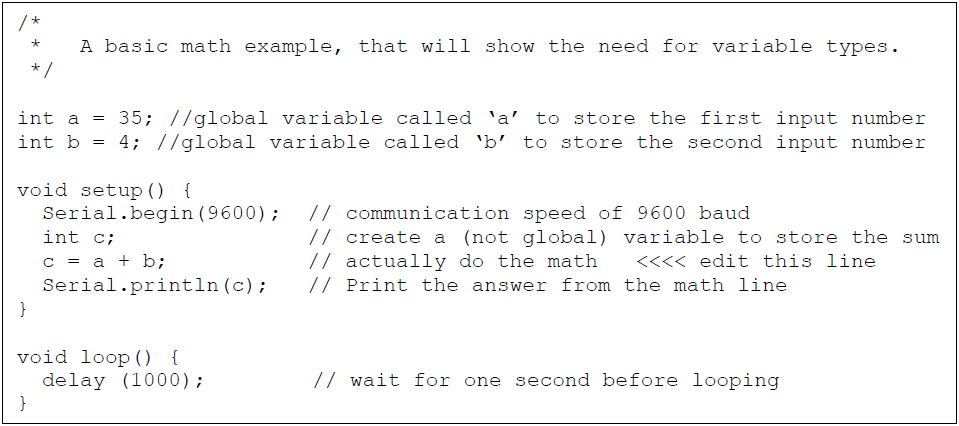
What follows ia whole series of exercises that will help you wrestle with these concepts of computer programming. Complete the assignments in order with an Arduino attached to your computer. You will need to take your time and read everything in order to answer the questions that are proposed in these exercises. Often, the very best strategy will be to copy and paste (or retype) the example code from these assignments into the Arduino IDE, then upload the code to an arduino and see what pops out on the Serial monitor. For many of the questions in the exercises, they will ask what is the output of a given program, and there is no better way to confirm this for yourself than to see the program in action!

Also, once you’ve got the program loaded onto an Arduino and in the Arduino IDE, it is much easier to tinker with the code to see how it works. This tinkering is the core of the process that nearly every great programmer has used to figure out how code works. The next great programmer could be you, and this the very start of your journey!

## Calculations Performed by a Microcontroller

THe first ever calculator looked something like the image to the right. it is now a museum piece, and depended on discrete vacuum tubes that had to be hot to work properly. Your Arduino microcontroller has silicon transistors in place of those vacuum tubes, and can fit many thousand into a small-ish [Integrated Circuit](https://en.wikipedia.org/wiki/Integrated_circuit) package. Giving the small microcontroller far more computing power than the machine shown here, and [about as much as was embedded in our first missions into space](https://www.nasa.gov/mission_pages/voyager/multimedia/vgrmemory.html#.W86K1EtKiMo). Technology has come a long way...

One of [first examples in this document](#_gjdgxs) was a math sketch. The program did addition in the background, and the user would input numbers, and get an answer. This example is certainly math, but there were several as-of-yet unexplained elements that *obfuscate* the meaning of what is going on. Now it is time to explore some mathematical programs, where there is more opportunity for a new programmer to control the program’s operation and manipulate things.



Type this program into the IDE. The code must be types correctly. Then upload the program and start the serial monitor.

This program doesn’t do much. It only adds two numbers and then prints the outcome to the computer, from the Arduino. This will be the first of our test programs. Try changing the numbers to confirm that it does the addition that you expect. You can make the numbers anything you would like, but there are some limits.

If you input any number, the math should work. Provided that the numbers (a, b and c) all meet two conditions: 1) they are in the range of -32,768 to 32,767 and 2) they do not include any fractions or decimals. Even with these two limitations, you’ve got something exciting. You are having a microchip do math and telling the result to your computer.

*Time to celebrate this success!!! You did it! Code was typed by you, compiled by an IDE, executed by a microchip, and then communicated back to your computer. This is the first step in making programs.*

The next step to developing this simple calculator program is to remove the limitations. The limitation for the size of the numbers is all based on how the memory inside the microchip works. If you want a bigger number, you can make your variables “unsigned int” or unsigned integers. This trick shifts the range of allowed numbers so that they are all in the positives. However, making a variable unsigned doesn’t make it any larger than a 16-bit (2-byte) number. If you need a larger number you can use a ‘long’ to make the memory space for the number 32-bit (4-bytes) or ranging from -2,147,483,648 to 2,147,483,647. Making an ‘unsigned long’ will also shift your range, which could help, provided you don’t need negative numbers.

As soon as division is started, or if a decimal or fraction is input as a or b, you will also find that the calculator doesn’t work as expected. To have division work and include any decimal answers, these must be initiated as ‘float’ variables. Larger values can be obtained with a ‘double’ in the exact same way as int and long, from above. However, float and double variables lead to slower computations (by microseconds) and have the distinct disadvantage of introducing rounding errors that can be hard to anticipate. For these reasons, it is good practice to avoid using float and double in program design. For simple testing, it is not a major issue.

### **Exercise 1:** [**Using the Arduino for Math**](https://docs.google.com/document/d/1yx46KKK7U1yMhBJkQ-8U75BRr3drNjmaE2GL_BETU5I/edit?usp=sharing)

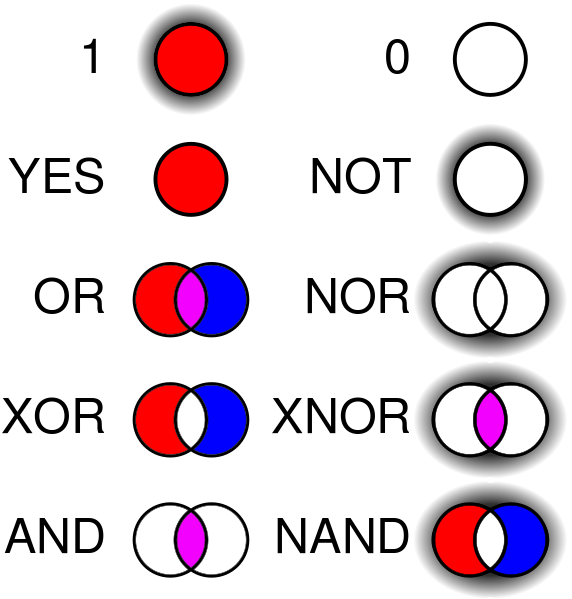
Lots of quick experiments can be done with a simple Arduino sketch that will have you understand the basics of how the programming works. As evidence for these experiments, you will need to collect a series of screenshots to show that you have completed them. You will also need to get an actual photograph of how you’ve connected the Arduino to the computer with a USB cable. The laptop’s webcam or a phone can get the Arduino’s photo, but screenshots should be taken with the operating system’s utility to do so. On Windows, the easiest option is to use the “Snipping Tool.” Save all these screenshots, so you can submit them to show your work.

## Boolean Logic

There are many equivalent truths in the world, but computers heed no ambiguity. Computers use the logic of semiconductor systems to do calculations and evaluate conditions to determine one outcome or another.

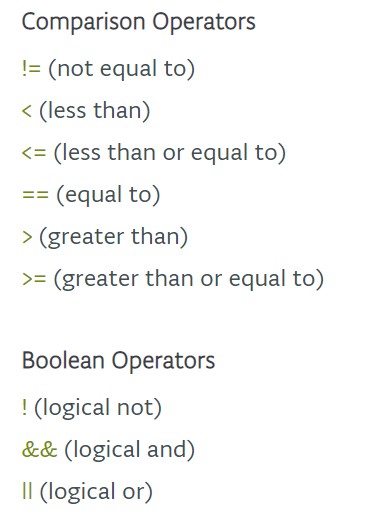
A TRUE statement is a logical 1, which corresponds to the physical ‘voltage on’ state of the computer.

A FALSE statement is a logical 0, which corresponds to the physical ‘voltage off’ state of the computer.

There are many people willing to [explain Boolean Logic to you on the internet](https://youtu.be/7dvqfpXEjdg?t=1m5s), since it is a foundational concept for all of computer science, in its many forms.

The image to the left is a diagram that hopes to explain all of the relationships that are implied by the boolean operators.

Arduino treats this boolean logic with a series of operators:

[](https://www.arduino.cc/reference/en/#functions)

Complete the following assignment where you must show your understanding of boolean operators by interpreting a program that is pre-written for you, and then creating your own programs that must meet certain requirements.

### **Exercise 2:** [**Boolean Logic Investigation**](https://drive.google.com/open?id=1NmfKzc3n3OIt16tKmcTWmfz--GL6uoTq0m0nMb_u-o8)

Lots of quick experiments can be done with a simple Arduino sketch that will have you understand the basics of how the programming works. As evidence for these experiments, you will need to collect a series of screenshots to show that you have completed them. You will also need to get an actual photograph of how you’ve connected the Arduino to the computer with a USB cable. The laptop’s webcam or a phone can get the Arduino’s photo, but screenshots should be taken with the operating system’s utility to do so. On Windows, the easiest option is to use the “Snipping Tool.” Save all these screenshots, so you can submit them to show your work.

## 

## 

# Built-in Functions to use in Programs

Functions are the main ‘things’ that a program runs on. Functions can be large and gangly, or they can be small and used in concert. There are many reasons *why* you would implement certain functions in a program, and certain style choices to be made with regard to *how* you implement them.

Some functions are embedded in the language, such as the ‘While Loop,’ ‘For Loop,’and the ‘Switch Case’ functions. Also, in Arduino the ‘void setup()’ and ‘void loop()’ functions are always defined (but they would not be predefined in all C++ programs, outside Arduino applications). It is also possible and often powerful for you to be able to write your own functions that can be called within your program.

## While Loop Function

### **Exercise 3:** [**While Loop Investigation**](https://drive.google.com/open?id=1X6GeWDHsB66cOg9c5vx8PZajWSIQCWQq-j4h41-CnpY)

This is a function that will loop forever, WHILE a condition is met.

**bool a = true;**  // assign the variable ‘a’ to be true

**while(a){** // while function evaluates if ‘a’ is true

**delay(1000);** // wait one second

**a = false;** // reassign the variable ‘a’ to be false

**}**

The while loop shown above will only run once. First the function checks to see if the thing in the parentheses is true. If the thing in the parentheses is true, then it runs the code inside the following curly brackets. It would loop through the code in the curly brackets forever, until the next time that the statement in parentheses is evaluated to be false. For this code, the variable a will be false the second time the code runs, because it is reassigned within the while loop. So the while loop shown here will only run once.

**int a = 0;**  // assign the variable ‘a’ to be zero

**while(a<100){** // while function evaluates if ‘a’ is less than 100

**delay(1000);** // wait one second

**a = a + 3;** // add three to the variable ‘a’

**}**

The example above is also a while loop, but the condition is different. The variable ‘a’ is now a number, and the while() loop will continue until value of the variable ‘a’ is more than or equal to 100. Each time the loop runs, the value of ‘a’ is increased by three, so it is only a matter of time until this while() loop is exited. How many times would this run? \*\*It is important to note that the variable ‘a’ is not re-zeroed inside the while() loop. If it was, this loop could be stuck forever.

## Arrays for Storing Data Lists

Although arrays are not functions, they are an important type of data that become really useful anytime you want to store a list of information. In the real world, people do this all the time. They are called ‘lists’ and you know all about them. Some lists are for the most successful songs ([Billboard](https://www.billboard.com/articles/news/hot-100-turns-60/8468142/hot-100-all-time-biggest-hits-songs-list)) some lists are for the top-valued companies ([Fortune 500](http://fortune.com/fortune500/)) and some lists are for what to get at the grocery store.

**int ledPins[] = {** // this example array is holding int values

**2, 7, 4, 6, 5, 3** // these are the actual listed values

//0, 1, 2, 3, 4, 5 <<< the positions of the array are shown

**};** // an array of pin numbers to which LEDs are attached

**int x = ledPins[3];** // the new variable ‘x’ now equals 6 from above

### **Exercise 4:** [**Arrays Investigation**](https://drive.google.com/open?id=1XIq-oSZLFyaSX1a_GuZ5SmE-PoBeo2RDer0gv-li2os)

An array is a way to store a bunch of information in an organized way. These are important to learn before you move on to the For() loop function, as it is commonly implemented.

## For Loop Function

### **Exercise 5:** [**For Loop Investigation**](https://drive.google.com/open?id=15F3AaOB35tLVnhHUGKW5c-qcC-TqFE6BnCVgHMzwtsU)

A for loop will iterate a value each time the function’s code runs, within a limited range.

**for (int i = 0; i < 10; i=i+1) {** // this is a complex step, see below

// first make a variable ‘i’ with a starting value of zero

// run this function as long as ‘i’ is valued less than 8

// each time this function loops raise the value of ‘i’ by 1

**delay(1000);** // wait one second

**Serial.print(i);**  // serial print the variable ‘i’

**Serial.println(“ second(s) passed”);** // serial print some words

**}**

This function is slightly more complicated than a while loop, but it is similar in spirit. It makes a variable and will run for a determined amount of times based on the conditions that are set

## Switch Case Function

The [switch case function](https://www.arduino.cc/reference/en/language/structure/control-structure/switchcase/) matches a set of outcomes to a set of defined inputs to the function. This is probably the hardest to initially understand, of the predefined functions in the C++ Language. Also, it can be entirely built from a complex constellation of if statements. The reason that the Switch Case function exists is to make it easier to implement this kind of logic in a program (which is also why the while and for loop exist). However, the switch case will not be directly covered in this tutorial at this time, but here is an example to consider….

// do something different depending on the range value:

**switch (range) {** // range can have values zero to 3

**case 0:** // your hand is on the sensor

**Serial.println("dark");**

**break;** // break ends the case

**case 1:** // your hand is close to the sensor

**Serial.println("dim");**

**break;**

**case 2:** // your hand is a few inches from the sensor

**Serial.println("medium");**

**break;**

**case 3:**  // your hand is nowhere near the sensor

**Serial.println("bright");**

**break;**

**}**

### 

# More stuff that needs added in the Document

## User-Defined Functions

### **Exercise 6:** [**User Defined Functions**](https://drive.google.com/open?id=151a-8bPBegYnFrjYqnZV9GONivUKKi6wx4slR-HA8OI)

It is possible to write your own functions, with all levels of complexity for any application you can imagine. This is especially useful if you have a program that needs to do the same thing repeatedly. writing that repeated function once, can make your code more understandable and maintainable.

### Function Definition

A function is defined the same way every time, but there are some important pieces and parts that you may have overlooked in creating functions, up until now. An important part of any function is its name. Carefully naming functions can make them easier to understand. Just looking at the first example program, you can see two functions that are given clear and concise names, that make them somewhat easier to understand:

// the setup function runs once when you press reset or power the board

**void setup() {**

// initialize digital pin LED\_BUILTIN as an output.

**pinMode(LED\_BUILTIN, OUTPUT);**

**}**

// the loop function runs over and over again forever

**void loop() {**

**digitalWrite(LED\_BUILTIN, HIGH);** // turn the LED on (HIGH is 5 volts)

**delay(1000);** // wait for a second

**digitalWrite(LED\_BUILTIN, LOW);**  // turn the LED off by applying 0 volts

**delay(1000);** // wait for a second

**}**

The names ‘setup’ and ‘loop’ (highlighted in green) may have been set by Arduino, but they also make it pretty clear what these do. Well selected names can make a function more understandable and easier to use. What a function does is the most important part of a function, and that is all defined by what is inside the curly brackets of the function.

### Function Inputs and Outputs

Functions can (but do not have to) take variables as **inputs** and also return them as **outputs**. The examples in the example directly above on this page can show where the input and output are located while setting up a function. The inputs that a function expects to receive are placed inside of the parentheses. in the example above, neither function is expecting any inputs, so the parentheses are left blank. The example function below (named modulo) is expecting two input values, and when the function is defined, these variable types need declared [as shown in blue below]. Within the curly brackets of the function, the variables are then understood to have the names and types declared as the function was defined. When the function is called, the inputs just need to be given [purple] and the function does the rest.

**int modulo(int x, int m) {**  //this line defines the function ‘modulo’

**return x % m ;** //this is what the function does & returns

**}**

//this line calls the function modulo and prints the value returned

**Serial.println(modulo(10,3));** //this Serial.println will show “1”

The a function may or may not return a value as an output. if a function does not return a value, the type of the function [shown in orange] is given as void. If a function is going to return a value, the type needs described as the function is defined, as shown in the second example on this page. The ‘modulo’ function also needs to have a return line of code at the end of the function, to designate what exactly is going to be returned by the function. In the example shown here, the modulo calculation is done in that line of code and then returned as the output of the function. That’s why the Serial monitor will show the value of 1 (the modulo answer) for this function.

## Variable Scope

### **Exercise 7:** [**Variable Scope**](https://drive.google.com/open?id=1D5G4_87WC6caLrNuwmmT66rwmr-v92lM0WzIWZ-1oIY)

Variables in most languages, including arduino’s version of C++, have a useful/ odd property called [scope](https://www.arduino.cc/reference/en/language/variables/variable-scope--qualifiers/scope/). This property describes where and how the variable can be used, based on where it was created. If a variable as a type of ‘bucket’ to hold information, then think of scope as the rules where that bucket can be used. A variable made outside of any function in a program is a global variable, which can be used by all the functions of the program. If a variable is declared within a specific function of a program, then it is a local variable that can only be used in that function. Making a local variable may seem silly at first, but it is actually best practice, whenever possible. That’s because it means you can’t accidentally change the value in another function, and you can re-use the variable name in other (separate) functions.

A little more information about variables and pointers: <https://dev.to/mindsers/pointers-in-c-c-4b52>

## 

## Code Readability & Maintainability

more needs added here (this was previously not an exercise)

### **Exercise 8:** Readability & Maintainability

more needs added here (this was previously not an exercise)

## 

## Object Oriented Programming

[Adafruit did a great job explaining this concept...](https://learn.adafruit.com/multi-tasking-the-arduino-part-1/a-classy-solution)

### **Exercise 9:** [**Object Oriented Programming**](https://drive.google.com/open?id=1-ON9_857Sb_qSPOU15IMkT9xT3XV1k8-if_DOvVK2n0)

Object Oriented Programming [OOP] is a more advanced level of code. Reading and writing code in this manner requires understanding many (if not all) of the previous concepts, and builds upon them. OOP can take a tedious, repetitive program and simplify it down to just the core of what needs to be described. OOP also generally helps programs become more readable, since it consolidates the code into smaller spaces.

## Arduino Libraries

more needs added here (this was previously not an exercise)

### **Exercise 10:** [**Arduino Libraries**](https://drive.google.com/open?id=1AaMAyP9Bj686-ChzFm8GkzEi9YxopyRgBY9YNVAda9k)

more needs added here (this was previously not an exercise)