Contents

Structure:

setup()

This is the function called on when the sketch starts and will run only once after startup or reset. You can use it to start variables, pin modes, or the use of libraries (specific terms you can download for extra functionality).

loop()

The loop function requires the Arduino microcontroller board to repeat a function multiple times, continuously or until a certain variable or condition is met. You will set the condition for it to stop the loop or you will have it loop continuously until you detach the Arduino from the power source or turn it off.

CONTROL STRUCTURES

Control structures show how an input will be received. Just like the name implies, various inputs regarding control determine how your data will be read. Provisional language will also be considered in data analysis. Popular and various control structures are mentioned below.

If

This is what links a condition or input to an output. It means that *if* a certain condition has been met, a specific output or response of the microcontroller will occur. For example, *if* the thermometer to which the microcontroller is attached measures more than 75 degrees Fahrenheit, you might write the code to direct the Arduino to send a signal to your air conditioning unit to turn on to decrease the temperature back to 75 degrees.

If…Else

This is like the *If* conditional, but it specifies another action that the microcontroller will take if the condition for the first action is not met. This gives you an option of performing two different actions in two different circumstances with one piece of code.

While

This is a loop that will continue indefinitely until the expression to which it is connected becomes false. That is, it would perform a certain function until a parameter is met and the statement that is set as the condition is made false.

Do… While

This is like the *while* statement, but it always runs at least once because it tests the variable at the end of the function rather than at the beginning.

Break

This is an emergency exit of sorts from a function of the microcontroller. It is used to exit a *do*, *for,* or *while* loop without meeting the condition that must be met to exit that part of the functionality.

Return

This is the way to stop a function, and it returns a value with which the function terminated to the calling function or the function that is asking for the information.

Goto

This piece of code tells the microcontroller to move to another place, not consecutive, in the coded program. It transfers the flow to another place in the program. Its use is generally discouraged by C language programmers, but it can definitely simplify a program.

SYNTAX

; (semicolon)

This is used as a period in the English language: it ends a statement. Be sure, however, that the statement closed by the semicolon is complete, or else your code will not function properly.

{ } (curly braces)

These have many complex functions, but the thing you must know is that when you insert a beginning curly brace, you *must* follow it with an ending curly brace. This is called keeping the braces balanced and is vital to getting your program working.

// (single-line comment)

If you would like to remind yourself or tell others something about how your code functions, use this code to begin the comment and make sure that it only takes up one line. This will not transfer to the processor of the microcontroller but rather will live in the code and be a reference to you and anyone who is reading the code manually.

/* */ (multi-line comment)

This type of comment is opened by the */*,* and it spans more than one line. It can itself contain a single line comment but cannot contain another multi-line comment. Be sure to close the comment with **/* or else the rest of your code will be considered a comment and not implemented.

#define

This defines a certain variable as a constant value. It gives a name to that value as a sort of shorthand for that value. These do not take up any memory space on the chip so they can be useful in conserving space. Once the code is compiled or taken together as a program, the compiler will replace any instance of the constant as the value that is used to define it.
NOTE: This statement does NOT use a semicolon at the end.

#include

This is used to include other libraries in your sketch, that is, to include other words and coding language in your sketch that would not otherwise be included. For example, you could include AVR C libraries or many tools, or pieces of code, from the various C libraries.
NOTE: Do NOT add the semicolon at the end of this statement, just as you would exclude it from the *#define* statement. If you do include a semicolon to close the statement, you will receive error messages and the program will not work.

ARITHMETIC OPERATORS

Just as the name implies, arithmetic operators' complete codes through use of mathematical symbols. Each symbol connects one line of code to another. When looking for an output resulting in measured values, be sure to check your Arduino setup. Connecting wire with Arduino in the wrong voltage receptors may lead to negative or irrelevant values.

= (assignment operator)

This assigns a value to a variable and replaces the variable with the assigned value throughout the operation in which it appears. This is different than == which evaluates whether two variables or a variable and a set value are equal. The double equal signs function more like the single equal sign in mathematics and algebra than the single equal sign in the Arduino IDE.

+ (addition)

This does what you might expect it would do: it adds two values, or the value to a variable, or two to a fixed constant. One thing that you must take into account is that there is a maximum for variable values in the C programming languages. This means that, if your variable maxes out at 32,767, then adding 1 to the variable will give you a negative result, -32,768. If you expect that the values will be greater than the absolute maximum value allowable, you can still perform the operations, but you will have to instruct the microcontroller what to do in the case of negative results. In addition, as well as in subtraction, multiplication, and division, you place the resulting variable on the left and the operation to the right of the = or ==.
Also, another thing to keep in mind is that whatever type of data you input into the operation will determine the type of data that is output by the operation. We will look at types of data later, but for example, if you input integers, which are whole numbers, you will receive an answer rounded to the nearest whole number.

- (subtraction)

This operation, like the addition sign, does what you would expect: it subtracts two values from each other, whether they both are variables, or one is a constant value. Again, you will have to watch out for values greater than the maximum integer value. Remember to place the resulting variable on the left of the equal sign or signs, and the operation on the right.

* (multiplication)

With multiplication especially, you will need to be careful to define what happens if the value you receive from the operation is greater than the greatest allowable value of a piece of data. This is because multiplication especially grows numbers to large, large values.

/ (division)

Remember to place the resulting variable on the left of the operation, and the values that you are dividing on the right side of the operation.

% (modulos)

This operation gives you the remainder when an integer is divided by another integer. For example, if you did *y = 7 % 5*, the result for y would be 2, since five goes into seven once and leaves a remainder of 2. Remember, you must use integer values for this type of operation.

COMPARISON OPERATORS

Comparison operators compare the values from the left side of the equation to the right. If the left operator does not have the same units as the right, it is still possible to use these operators, but the results may be unpredictable (Arduino.cc).

== (equal to)

This operator checks to see if the data on the left side of the double equal signs match the data on the right side, that is, whether they are equal. For example, you might ask the pin attached to the temperature gauge *t == 75*, and if the temperature is exactly 75 degrees, then the microcontroller will perform a certain task, whether it be turning off the heating or cooling or turning off a fan.

!= (not equal to)

This is the mirror image of the previous operation. You could just as easily write a program to test *t != 75* and set up the microcontroller to turn on a heating lamp, turn on a fan, or ignite the wood in the fireplace if this statement is true. Between == and *!=*, you can cover all the possible conditions that input might give your microcontroller.

< (less than)

If this statement is true, then you can program a certain response from your microcontroller, or, in other words, program output for such input.
> (greater than)

INPUT

In the input state, a digital pin will require very little of the processing power and energy from the microcontroller and battery. Instead, it is simply measuring and indicating to the microcontroller its measurements.

OUTPUT

These are very good at powering LED's because they are in a low impedance state, meaning they let the energy flow freely through them without much resistance. Output pins take their directions from the microcontroller once it has processed the information given by the input pins, and the output pins power whatever mechanism will perform the intended task.

INPUT_PULLUP

This is what mode you will want to use when connected to a button or a switch. There is a lot of resistance involved in the INPUT_PULLUP state. This means that it is best used for Boolean-like situations, such as a switch either being on or off. When there are only two states and not much in between, use INPUT_PULLUP.

LED_BUILTTIN

true

In a Boolean sense, any integer that is not zero is true. One is true, 200 is true, -3 is true, etc. This would be the case when a statement matches reality. One of your pins might be testing a value, and the statement is trying to match *y != 35*, so if the pin receives information that the value of *y* is 25, then the statement *25 != 35* is true.

false

This is part of a Boolean Constant, meaning that a statement is false, or that its logic does not match reality. For example, you could have a statement, *x > 7* and the value the microcontroller receives for x is 3. This would make the statement *false*. It would then be defined as 0 (zero).

integer constants

These are constants that are used by the sketch directly and are in base 10 form, or integer form. You can change the form that the integer constants are written in by preceding the integer with a special notation signifying binary notation (base 2), the octal notation (base 8), or hexadecimal notation (base 16), for example.

floating point constants

These save space in the program by creating a shorthand for a long number in scientific notation. Each time the floating-point constant appears, it is evaluated at the value that you dictate in your code.

DATA TYPES

Data types refer to the type of data received in each of the programming setups you apply. Data received by Arduino are sent to your program of choice to determine various outcomes. Some examples are listed below.

Void

This is used in a function declaration to tell the microcontroller that no information is expected to be returned with this function. For example, you would use it with the *setup()* or *loop()* functions.

Boolean

Boolean data holds one of two values: true or false. This could be true of any of the arithmetic operator functions or of other functions. You will use *&&* if you want two conditions to be true simultaneously for the Boolean to be true, *//* if you want one of two conditions to be met, either one setting off the output response, and ! for not true, meaning that if the operator is *not* true, then the Boolean is true.

Char

This is a character, such as a letter. It also has a numeric value, such that you can perform arithmetic functions on letters and characters. If you want to use characters literally, you will use a single quote for a single character, *'A'* and a double quote for multiple characters, *"ABC"* such that all characters are enclosed in quotes. This means the microcontroller will output these characters verbatim if the given conditions are met. The numbers -128 to 127 are used to signify various signed characters.

Unsigned Char

This is the same as a character but uses the numbers 0 to 255 to signify characters instead of the “signed” characters which include negatives. This is the same as the byte datatype.

Int

Integers are how you will store numbers for the most part. Because most Arduinos have a 16-bit system, the minimum value is -32,768 and the maximum value of an integer is 32,767. The Arduino Due and a few other boards work on a 32-bit system, and thus can carry integers ranging from -2,147,483,648 to 2,147,483,647. Remember these numbers when you are attempting arithmetic with your program, as any numbers higher or lower than these values will cause errors in your code.

Unsigned Int

This yields the ability to store numbers from 0 to 65,535 on the 8-bit boards with which you will likely be working. If you have higher values than the signed integers will allow, you can switch to unsigned integers and achieve the same amount of range but all in the positive realm, such that you have a higher absolute value of the range.

Word

A word stores a 16-bit unsigned number on the Uno and on other boards with which you will likely be working. In using the Due and the Zero, you will be storing 32-bit numbers using words. Word is essentially the means by which integers and numbers are stored.

Long

If you need to store longer numbers, you can access 4-byte storage, or 32-bit storage in other words, using the long variable. You simply follow an integer in your coded math with the capital letter *L.* This will achieve numbers from –

2,147,483,648 to 2,147,483,647.

Unsigned Long

The way to achieve the largest numbers possible and store the largest integers possible is to direct the microcontroller using the unsigned long variables. This also gives you 32 bits or 4 bytes to work with, but being unassigned the 32nd bit is freed from indicating the positive or negative sign in order to give you access to numbers from 0 to 4,294,967,295.

Short

This is simply another way of indicating a 16-bit datatype. On every type of Arduino, you can use short to indicate you are expecting or using integers from -32,768 to 32,767. This helps free up space on your Due or Zero by not wasting space on 0's for a small number and by halving the number of bits used to store that number.

Float

A float number is a single digit followed by 6 to 7 decimal places, multiplied by 10 to a power up to 38. This can be used to store more precise numbers or just larger numbers. Float numbers take a lot more processing power to calculate and work with, and they only have 6 to 7 decimals of precision, so they are not useful in all cases. Many programmers actually try to convert as much float math to integer math as possible to speed up the processing. In addition, these take 32 bits to store versus the normal 16 bits, so if you're running low on storage, try converting your float numbers to integers.

Double

This is only truly relevant to the Due, in which doubling allows for double the precision of a float number. For all other Arduino boards, the floating-point number always takes up 32 bits, so floating does nothing to increase precision or accuracy.