Arduino Programming

Introduction to Arduino

- Arduino is a prototype platform (open-source) based on an easy-to-use hardware and software. It consists of a circuit board, which can be programed (referred to as a microcontroller) and a ready-made software called Arduino IDE (Integrated Development Environment), which is used to write and upload the computer code to the physical board.
- Arduino boards are able to read analog or digital input signals from different sensors and turn it into an output such as activating a motor, turning LED on/off, connect to the cloud and many other actions.
- You can control your board functions by sending a set of instructions to the microcontroller on the board via Arduino IDE (referred to as uploading software).
- Arduino does not need an extra piece of hardware (called a programmer) in order to load a new code onto the board. You can simply use a USB cable.
- Arduino IDE uses a simplified version of C++, making it easier to learn to program.
- Arduino provides a standard form factor that breaks the functions of the micro-controller into a more accessible package.

Arduino Coding Basics

- Arduino IDE (Integrated Development Environment) allows us to draw the sketch and upload it to the various Arduino boards using code.
- The code is written in a simple programming language similar to C and C++.
- The basics to start with Arduino programming.

Brackets

There are two types of brackets used in the Arduino coding, which are listed below:

1. Parentheses ( )

- The parentheses brackets are the group of the arguments, such as method, function, or a code statement. These are also used to group the math equations.

2. Curly Brackets { }

- The statements in the code are enclosed in the curly brackets. We always require closed curly brackets to match the open curly bracket in the code or sketch.

  Open curly bracket- ' { '

  Closed curly bracket - ' } '

Line Comment

There are two types of line comments, which are listed below:

1. Single line comment
2. Multi-line comment

// Single line comment

- The text that is written after the two forward slashes are considered as a single line comment. The compiler ignores the code written after the two forward slashes. The comment will not be displayed in the output. Such text is specified for a better understanding of the code or for the explanation of any code statement.
- The // (two forward slashes) are also used to ignore some extra lines of code without deleting it.

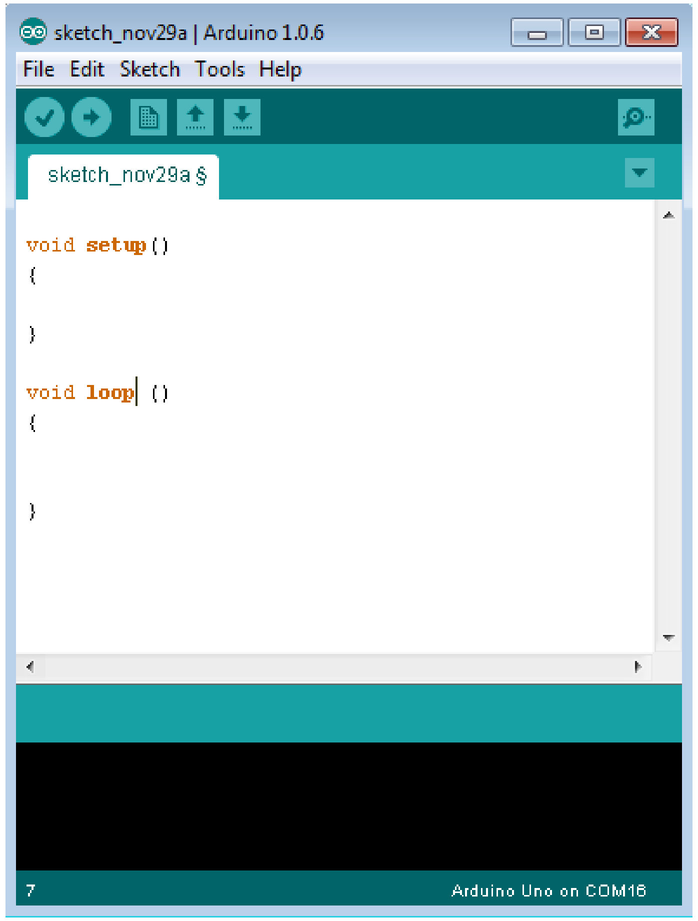
/ * Multi - line comment */

- The Multi-line comment is written to group the information for clear understanding. It starts with the single forward slash and an asterisk symbol (**/ ***). It also ends with the **/ ***. It is commonly used to write the larger text. It is a comment, which is also ignored by the compiler.

Arduino Program Structure

- Arduino programs can be divided in three main parts: **Structure, Functions** and **Values** (variables and constants).

- Software structure consist of two main functions –
  - Setup( ) function
  - Loop( ) function



```
void setup()
{

}

void loop ()
{


}
```

Setup( ) function

```
Void setup ( ) {

}
```

The setup() function is called when a sketch starts. Use it to initialize the variables, pin modes, start using libraries, etc. The setup function will only run once, after each power up or reset of the Arduino board.

Loop( ) function

```
Void Loop ( ) {

}
```

After creating a setup() function, which initializes and sets the initial values, the loop() function does precisely what its name suggests, and loops consecutively, allowing your program to change and respond. Use it to actively control the Arduino board.

Arduino Serial |Serial.begin()

Accura Tequipment

Serial.begin ( )

- The serial.begin( ) sets the baud rate for serial data communication. The **baud** rate signifies the data rate in bits per second.

- The default baud rate in Arduino is **9600 bps (bits per second**). We can specify other baud rates as well, such as 4800, 14400, 38400, 28800, etc.

- The Serial.begin( ) is declared in two formats, which are shown below:
  - begin(speed)
  - begin(speed, config)

  Where,
    - **serial**: It signifies the serial port object.
    - **speed**: It signifies the baud rate or bps (bits per second) rate. It allows ***long*** data types.
    - **config**: It sets the stop, parity, and data bits.

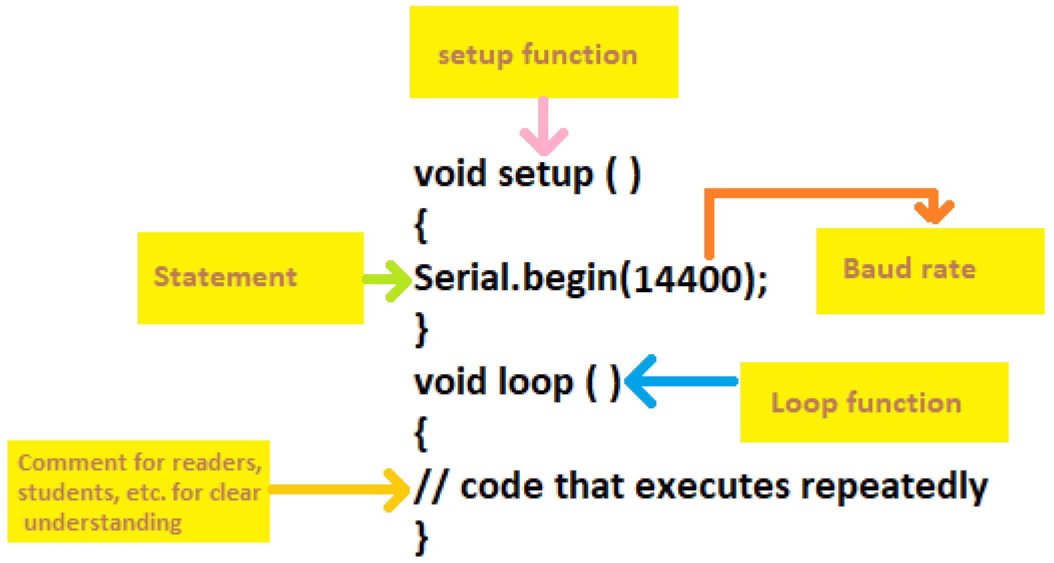
Example 1:

```
void setup ( )
{
Serial.begin(4800);
}
void loop ( )
{

}
```

- The serial.begin (4800) open the serial port and set the bits per rate to 4800. The messages in Arduino are interchanged with the serial monitor at a rate of 4800 bits per second.

Example 2:



```
void setup ( )
{
Serial.begin(14400);
}
void loop ( )
{
// code that executes repeatedly
}
```

Arduino Data Types

- The data types are used to identify the types of data and the associated functions for handling the data. It is used for declaring functions and variables, which determines the bit pattern and the storage space.
- The data types that we will use in the Arduino are
  - void Data Type
  - int Data Type
  - Char Data Type
  - Float Data Type
  - Double Data Type
  - Unsigned int Data Type
  - short Data Type
  - long Data Type
  - Unsigned long Data Type
  - byte data type
  - word data type

void Data Type

- The void data type specifies the empty set of values and only used to declare the functions. It is used as the return type for the functions that do not return any value.
- Example:

```
int a = 3;
void setup( )
{
.   //
}
void loop ( )
{
.
.
}
```

Int Data Type

- The integer data types are the whole numbers like 5, -6, 10, -123, etc. They do not have any fractional part. The integer data types are represented by **int**. It is considered as the primary data type to store the numbers.

- The size of int is 2 bytes (16 bits).

- Minimal range: -32768 to 32767 or - (2^ 15) to ((2 ^ 15) - 1)

- Some boards like Arduino Zero and MKR1000 (SAMD boards), and Arduino Due, the int data type stores the value of 4 bytes or 32 bits. The Minimal range in such case would be - (2^ 31) to ((2 ^ 31) - 1) or -2,147,483,648 to 2,147,483,647.

- The negative numbers are stored in the form of 2's complement, where the sign bit or the highest bit is flagged as the negative number.

The syntax is used as:

int var = val;

where,

var= variable

value = the value assigned to the variable

For example,

int a;

int b = 3;

Any variable or identifier becomes an integer variable and can hold only integer values.

Char Data Type

- The char datatype can store any number of character set. An identifier declared as the char becomes a character variable. The literals are written inside a single quote.

- The char type is often said to be an integer type. It is because, symbols, letters, etc., are represented in memory by associated number codes and that are only integers.

- The size of character data type is **minimum of 8 bits**. We can use the byte data type for an unsigned char data type of 8 bits or 1 byte.

  For example, character ' A ' has the ASCII value of 65.

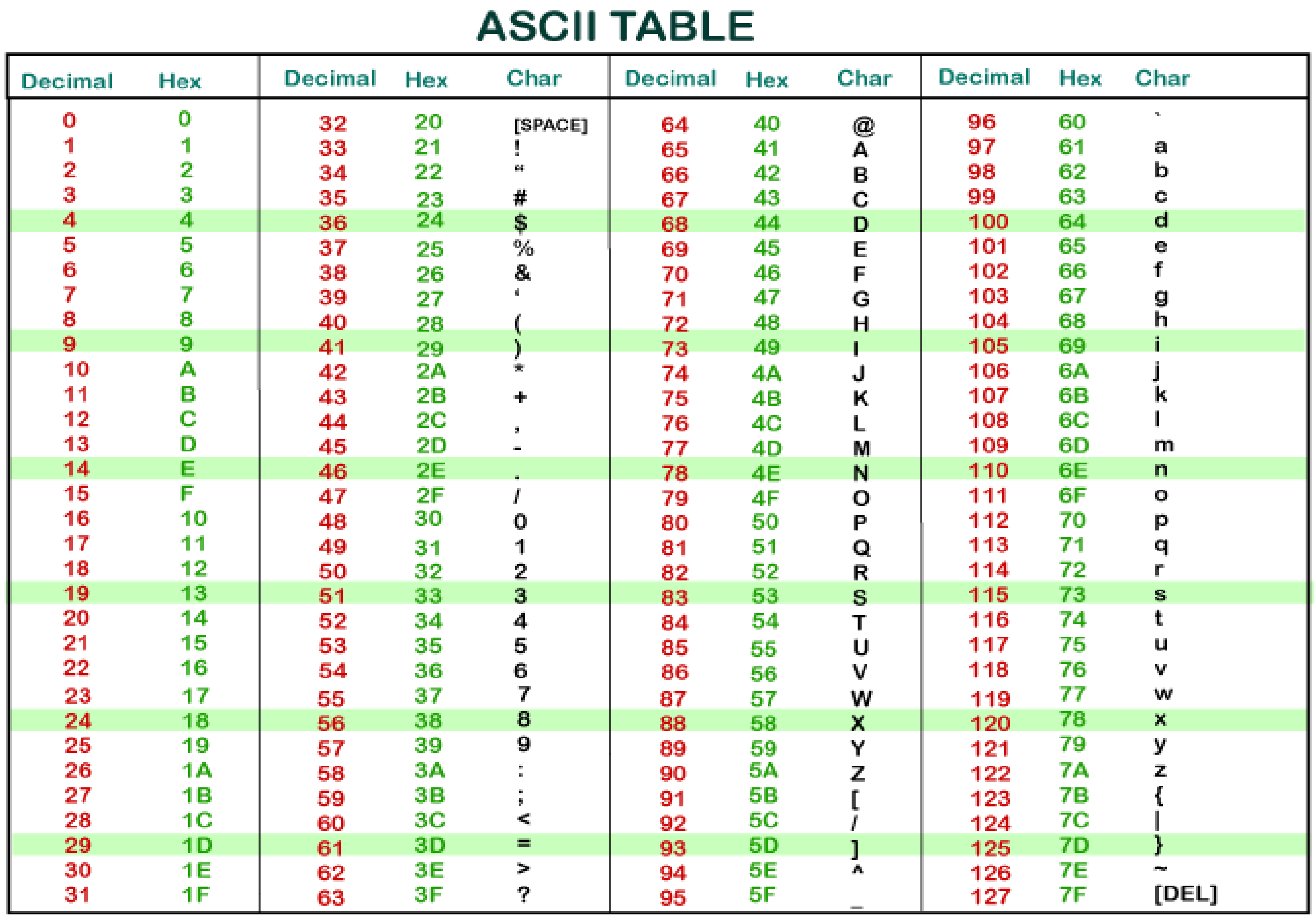
  The syntax is:

  char var = val;

  where,

  var= variable

  val = The value assigned to the variable.



ASCII TABLE

Decimal	Hex	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	32	20	[SPACE]	64	40	@	96	60	`
1	1	33	21	!	65	41	A	97	61	a
2	2	34	22	“	66	42	B	98	62	b
3	3	35	23	#	67	43	C	99	63	c
4	4	36	24	$	68	44	D	100	64	d
5	5	37	25	%	69	45	E	101	65	e
6	6	38	26	&	70	46	F	102	66	f
7	7	39	27	‘	71	47	G	103	67	g
8	8	40	28	(	72	48	H	104	68	h
9	9	41	29	)	73	49	I	105	69	i
10	A	42	2A	*	74	4A	J	106	6A	j
11	B	43	2B	+	75	4B	K	107	6B	k
12	C	44	2C	,	76	4C	L	108	6C	l
13	D	45	2D	-	77	4D	M	109	6D	m
14	E	46	2E	.	78	4E	N	110	6E	n
15	F	47	2F	/	79	4F	O	111	6F	o
16	10	48	30	0	80	50	P	112	70	p
17	11	49	31	1	81	51	Q	113	71	q
18	12	50	32	2	82	52	R	114	72	r
19	13	51	33	3	83	53	S	115	73	s
20	14	52	34	4	84	54	T	116	74	t
21	15	53	35	5	85	55	U	117	75	u
22	16	54	36	6	86	56	V	118	76	v
23	17	55	37	7	87	57	W	119	77	w
24	18	56	38	8	88	58	X	120	78	x
25	19	57	39	9	89	59	Y	121	79	y
26	1A	58	3A	:	90	5A	Z	122	7A	z
27	1B	59	3B	;	91	5B	[	123	7B	{
28	1C	60	3C	<	92	5C	/	124	7C	|
29	1D	61	3D	=	93	5D	]	125	7D	}
30	1E	62	3E	>	94	5E	^	126	7E	~
31	1F	63	3F	?	95	5F	_	127	7F	[DEL]

Accura Tequipment

Float Data Type

- A number having the fractional part and a decimal part is considered as a floating-point number.
- For example, 4.567 is a floating-point number. The number 13 is an integer, while 13.0 is a floating-point number.
- Due to their greater resolution, fractional numbers are used to approximate the contiguous and analog values.
- Floating point numbers can also be written in the exponent form. The numbers can be as large as 3.4028235E+38 and as small as -3.4028235E+38. The size of float data types is 4 bytes or 32 bits.

  The syntax is:

  float var = val;

  where,

  var= variable

  val = The value assigned to the variable

Double Data Type

Accura Tequipment

- The double data type is also used for handling the decimal or floating-point numbers. It occupies twice as much memory as float. It stores floating point numbers with larger precision and range. It stands for double precision floating point numbers.

- It occupies 4 bytes in ATmega and UNO boards, while 8 bytes on Arduino Due.

The syntax is:

```
double var = val;
where,
    var= variable
    val = The value assigned to the variable
```

Unsigned int Data Type

- The unsigned int stores the value upto 2 bytes or 16 bits. It stores only positive values. The range of unsigned int data type is from 0 to 65,535 or 0 to ((2 ^ 16) - 1).
- Arduino Due stores the unsigned data value of 4 bytes or 32-bits.
- The difference between Unsigned and signed data type is the sign bit. The int type in Arduino is the signed int. In a 16-bit number, 15 bits are interpreted with the 2's complement, while the high bit is interpreted as the positive or negative number. If the high bit is '1', it is considered as a negative number.

The syntax is:

unsigned **int** var = val;

where,

var= variable

val = The value assigned to the variable

short Data Type

- The short is an integer data type that stores two bytes or 16-bit of data.
- The range of short data types is from -32768 to 32767 or - (2^ 15) to ((2 ^ 15) - 1).
- The ARM and ATmega based Arduino's usually stores the data value of 2 bytes.

The syntax is:
short var = val;
where,
var = variable
val = the value assigned to the variable

long Data Type

- The long data types are considered as the extended size variables, which store 4 bytes (32 -bits).
- The size ranges from -2,147,483,648 to 2,147,483,647.
- While using integer numbers, at least one of the numbers should be followed by L, which forces the number to be a long data type.

The syntax is:
long var = val;
where,
var= variable
val = The value assigned to the variable

For example:

long speed = 186000L;

Unsigned long Data Type

- The unsigned long data types are also considered as the extended size variables, which store 4 bytes (32 -bits).

- It does not store negative numbers like other unsigned data types, which makes their size ranges from 0 to 4,294,967,295 or (2^32 - 1).

The syntax is:

unsigned **long** var = val;

where,

var= variable

val = The value assigned to the variable

For example:

unsigned **long** currenTtime;

byte

- 1 byte = 8 bits.

- It is considered as an unsigned number, which stores values from 0 to 255.

  The syntax is:
  byte var = val;
  where,
  var= variable
  value = the value assigned to the variable

For example,

byte c = 20;

word

- It is considered as an unsigned number of 16 bits or 2 bytes, which stores values from 0 to 65535.

The syntax is:

word var = val;

where,

var= variable

val = The value assigned to the variable

For example:

word c = 2000;

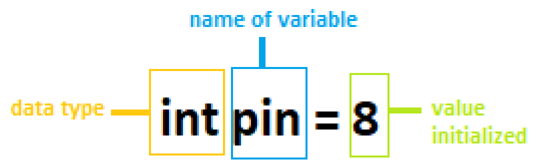
Arduino Variables

- The variables are defined as the place to store the data and values. It consists of a name, value, and type.

- The variables can belong to any data type such as int, float, char, etc

For example:

int pin = 8;



- Here, the **int** data type is used to create a variable named **pin** that stores the value **8**. It also means that value 8 is initialized to the variable **pin**.

- If we have not declared the variable, the value can also be directly passed to the function.

Advantages of Variables

The advantages of the variables are :

- We can use a variable many times in a program.
- The variables can represent integers, strings, characters, etc.
- It increases the flexibility of the program.
- We can easily modify the variables. For example, if we want to change the value of variable LEDpin from 8 to 13, we need to change the only point in the code.
- We can specify any name for a variable. For example, greenpin, bluePIN, REDpin, etc.

Variables Scope

The variables can be declared in two ways in Arduino, they are:

- Local variables
- Global variables

Local Variables

- The local variables are declared within the function. The variables have scope only within the function. These variables can be used only by the statements that lie within that function.

For example,

```
void setup()  {
        Serial.begin(9600);
}
void loop()  {
        int x = 3;
        int b = 4;
        int sum = 0;
        sum = x + b;
        Serial.println(sum);
}
```

Global Variables

- The global variables can be accessed anywhere in the program. The global variable is declared outside the setup() and loop() function.

For example,

Consider the below code.

```
int LEDpin = 8;
void setup()  {
        pinMode(LEDpin, OUTPUT);
}
void loop()  {
        digitalWrite(LEDpin, HIGH);
}
```

- We can notice that the LEDpin is used both in the loop() and setup() functions.

Arduino constants

- The constants in Arduino are defined as the predefined expressions. It makes the code easy to read.

The constants in Arduino are defined as:

Logical level Constants

- The logical level constants are **true** or **false**.
- The value of true and false are defined as 1 and 0. Any non-zero integer is determined as true in terms of Boolean language. The true and false constants are type in lowercase rather than uppercase.

LED_BUILTIN Constant

- The Arduino boards have built-in LED connected in series with the resistor. The particular pin number is defined with the constant name called LED_BUILTIN.
- Most Arduino boards have the LED_BUILTIN connected to Pin number 13.

Pin level Constants

- The digital pins can take two value **HIGH** or **LOW**.
- In Arduino, the pin is configured as INPUT or OUTPUT using the pinMode() function. The pin is further made HIGH or LOW using the digitalWrite() function.

HIGH

- The board includes two types of voltage pins to provide HIGH value, they are, 5V and 3V
- Some boards include only 5V pins, while some include 3.3V
- The pin configured as HIGH is set at either 5V or 3.3V.
- The pins are configured at the 5V or 3.3V depending on:
  - for voltage > 3.0V (presented at 5V pin)
  - for voltage > 2.0V (presented at 3.3V pin)

LOW

- The pin configured as **LOW** is set at 0 Volts.
- The pins are configured at the 5V or 3.3V depending on:
  - for voltage < 1.5V (presented at 5V pin)
  - for voltage < 1V (presented at 3.3V pin)

Constant Keyword

- The name **const** represents the constant keyword. It modifies the behavior of the variables in our program. It further makes the variable as 'read-only'.
- The variable will remain the same as other variables, but its value cannot be changed.
- It means we cannot modify the constant.

  For example,

  const int a =2; //....

  a = 7;   // illegal - we cannot write to or modify a constant
- The **const** keyword is considered superior compared to the **#define** keyword because it obeys the rules of the **variable scope**.
- We can either use **const** or **#define** in the case of **strings** and **numeric** constants. But we can only use **const** for **arrays**.

#define

- The #define in Arduino is used to give a name to the constant value. It does not take any memory space on the chip.

- At the compile time, the compiler will replace the predefined value in the program to the constants with the defined value.

The syntax is:

```
#define nameOFconstant value
```

where,

nameOFconstant: It is the name of the macro or constant to define

value: It includes the value assigned to the constant or macro.

For example,

```
#define LEDpin 12  // It is the correct representation of #define
```

- The #define does not require any semicolon. Hence, we do not need to specify any semicolon after the #define. Otherwise, the compiler will show errors.

Arduino Operators

- The operators are widely used in Arduino programming from basics to advanced levels.

- The operators are used to solve logical and mathematical problems. For example, to calculate the temperature given by the sensor based on some analog voltage.

- The types of Operators classified in Arduino are:
    1. Arithmetic Operators
    2. Compound Operators
    3. Boolean Operators
    4. Comparison Operators
    5. Bitwise Operators

Arithmetic Operators

- There are six basic operators responsible for performing mathematical operations in Arduino, they are:

Assignment Operator (=)

  - The Assignment operator in Arduino is used to set the variable's value. It is quite different from the equal symbol (=) normally used in mathematics.

Addition (+)

  - The addition operator is used for the addition of two numbers. For example, P + Q.

Subtraction (-)

  - Subtraction is used to subtract one value from the another. For example, P - Q.

Multiplication (*)

  - The multiplication is used to multiply two numbers. For example, P * Q.

Division (/)

  - The division is used to determine the result of one number divided with another. For example, P/Q.

Modulo (%)

  - The Modulo operator is used to calculate the remainder after the division of one number by another number.

Order of mathematical operations:

- The order of operations considered by the Arduino while performing calculation:
  1. Parentheses ( )
  2. Multiplication, division, and modulo
  3. Addition and subtraction
- If there are multiple operations next to each other, they will be computed from left to right.

Example:

```
int c;
void setup ( )  {
            Serial.begin( 9600 );
}
void loop ( )  {
            c = 2 * 3 / (2 + 1)  + 4;
            Serial.println(c);
}
```

OUTPUT: 6

the number inside parentheses will be calculated first

$$b = 2 * 3 / \boxed{(2 + 1)} + 4$$

$$b = \boxed{2 * 3} / \boxed{3} + 4$$

As discussed above, from left to right calculations will be performed i.e. first multiplication and then division

$$b = 6 / 3 + 4$$

$$b = 2 + 4$$

$$b = 6$$

Compound Operators

- The compound operators perform two or more calculations at once.
- The result of the right operand is assigned to the left operand.

Let's consider a variable **b**.

b + +

Here, b = b + 1. It is called the **increment operator**.

b + =

For example, b + = 4. It means, b = b+ 4.

b - -

Here, b = b - 1. It is called as the **decrement operator**.

b - =

For example, b - = 3. It means, b = b - 3.

b * =

For example, b * = 6. It means, b = b * 6.

b / =

For example, b / = 5. It means, b = b / 5.

b % =

For example, b % = 2. It means, b = b % 2.

Now, let's use the above operators with two variables, b and c.

- b + = c (b = b + c)
- b - = c (b = b - c)
- b * = c (b = b * c)
- b / = c (b = b / c)
- b % = c (b = b % c)

We can specify any variable instead of b and c

Comparison Operators

- The comparison operators are used to compare the value of one variable with the other.
- The comparison operators are

less than (<)

The less than operator checks that the value of the left operand is less than the right operand. The statement is true if the condition is satisfied.

greater than (>)

The less than operator checks that the value of the left side of a statement is greater than the right side. The statement is true if the condition is satisfied.

For example, a > b.

If a is greater than b, the condition is true, else false.

equal to (= =)

It checks the value of two operands. If the values are equal, the condition is satisfied.

For example, a = = b.

The above statement is used to check if the value of a is equal to b or not.

not equal to (! =)

It checks the value of two specified variables. If the values are not equal, the condition will be correct and satisfied.

For example, a ! = b.

less than or equal to (< =)

The less or equal than operator checks that the value of left side of a statement is less or equal to the value on right side. The statement is true if either of the condition is satisfied.

For example, a < = b

It checks the value of a is less or equal than b.

greater than or equal to (> =)

The greater or equal than operator checks that the value of the left side of a statement is greater or equal to the value on the right side of that statement. The statement is true if the condition is satisfied.

For example, a > = b

It checks the value of a is greater or equal than b. If either of the condition satisfies, the statement is true.

Bitwise Operators

- The Bitwise operators operate at the **binary level**. These operators are quite easy to use.
- There are various bitwise operators. Some of the popular operators are :

bitwise NOT (~)

- The bitwise NOT operator acts as a complement for reversing the bits.

For example, if b = 1, the NOT operator will make the value of b = 0.
Let's understand with another example.

```
0 0 1 1 // Input or operand 1   ( decimal value 3)
1 1 0 0 // Output ( reverses the input bits ) decimal value is 12
```

bitwise XOR (^)

- The output is 0 if both the inputs are same, and it is 1 if the two input bits are different.

For example,

```
1 0 0 1  // input 1 or operand 1
0 1 0 1  // input 2
1 1 0 0 // Output ( resultant - XOR)
```

bitwise OR (|)

- The output is 0 if both of the inputs in the OR operation are 0. Otherwise, the output is 1. The two input patterns are of 4 bits.

```
For example,
1 1 0 0  // input 1 or operand 1
0 0 0 1  // input 2
1 1 0 1 // Output ( resultant - OR)
```

bitwise AND (&)

- The output is 1 if both the inputs in the AND operation are 1. Otherwise, the output is 0. The two input patterns are of 4 bits.

```
For example,
1 1 0 0  // input 1 or operand 1
0 1 0 1  // input 2
0 1 0 0 // Output ( resultant - AND)
```

bitwise left shift (< <)

- The left operator is shifted by the number of bits defined by the right operator.

bitwise right shift (> >)

- The right operator is shifted by the number of bits defined by the left operator.

Boolean operators

- Boolean operators in Arduino programming are essential for making logical decisions based on true or false conditions.
- These operators, namely AND (&&), OR (||), and NOT (!), allow us to combine multiple conditions and evaluate whether they are true or false. By doing so, we can control the flow of our Arduino program based on logical outcomes.

- **AND Operator (&&)**

The AND operator returns true if both operands are true. Otherwise, it returns false. It requires both conditions for the entire expression to be true. If any of the operands are false, the entire expression will evaluate to false.

Syntax: operand1 && operand2

- **OR Operator (||)**

The OR operator returns true if at least one of its operands is true , otherwise, it returns false. It requires only one condition for the entire expression to be true. If both operands are false, the entire expression will evaluate to false.

Syntax: operand1 || operand2

- **NOT Operator (!)**

The NOT operator is a unary operator that returns the opposite of its operand. If the operand is true, it returns false. If the operand is false, it returns true.

Syntax: !operand

Arduino Array

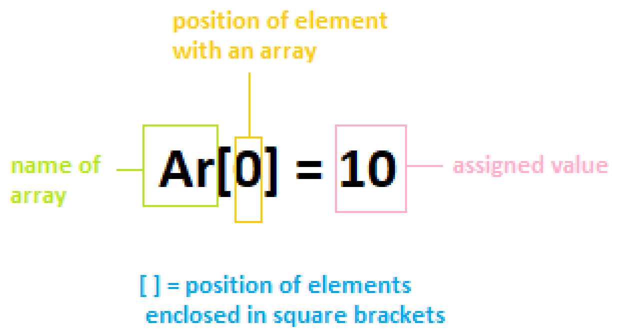
- The arrays are defined as the data structures that allow multiple values to be grouped together in a simple way.

- The array is normally built from the data types like **integer, reals, characters, and boolean**. It refers to a named list of finite number (n) of similar data elements.

- The set of consecutive numbers usually represent the elements in the array, which are **0, 1, 2, 3, 4, 5, 6,.......n.**

- The array in Arduino is declared with the integer data type. It is also defined as the collection of variables, which is acquired with an index number.

- The array is represented as:


position of element with an array
name of array
Ar[0] = 10
assigned value
[ ] = position of elements enclosed in square brackets

- We can specify any name according to our choice. The array name is the individual name of an element.

Array Declaration

- There are different methods to declare an array in Arduino, which are
  1. We can declare the array without specifying the size.
     For example: int myarray[ ] = { 1, 4, 6, 7 } ;
  2. We can declare the array without initializing its elements.
     For example: **int** myarray[5];
  3. We can declare the array by initializing the size and elements.
     For example: **int** myarray[8] = { 1, 4, 7, 9, 3, 2 , 4};

Features of Array

- The elements of the array can be characters, negative numbers, etc.

  For example,

  ```
  int myarray[ 4 ] = { 1, -3, 4};
  char myarray[ 6] =  " Hi " ;
  ```

- The size of the array should not be less than the number of elements.

  For example,

  ```
  int myarray[5 ] = { 1, 4, 6, 7 } ; can be written as int myarray[8 ] = { 1, 4, 6, 7 } ;
  But, it cannot be written as int myarray[ 2] = { 1, 4, 6, 7 } ;
  ```

- The total elements, while specifying the char type should be (n - 1), where n is the size of the array. It is because one element is required to hold the null character in the array.

  For example,

  ```
  char abc[8 ] = " Arduino";
  ```

Access of array in Arduino

- The array in Arduino has zero index. It means that the first element of the array is indexed as 0.

  For example: myvalue[0] = = 1, myvalue[1] = = 2, . . .

- The last element of the array will be n-1, where n is the declared size of an array.

  For example:

  ```
  ARarduino[6] = { 1, 4, 7, 6, 11, 15};
  // ARarduino[4] contains value 11
  // ARarduino[5] contains value 15
  //ARarduino[6] is invalid
  ```

Loop Arrays

- The arrays can be inside the loop. For each element of an array, the loop counter acts as an index element for that array.

Example: Printing the sum of all elements

```
const int sizeOFarray = 5;                    // constant variable indicating size of array
int b[sizeOFarray] = {10, 20, 30, 40, 50}; // five values initialized to five elements of an array
int sum = 0;

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

void loop ()   {
  // sum of array b
   for ( int i = 0; i < sizeOFarray; i++ )
     sum += b[ i ];                       // here, sum = sum + b[i]
    Serial.print('Sum of total elements of an array:') ;
   Serial.print(sum) ;  // It will print the sum of all elements of an array
}
```

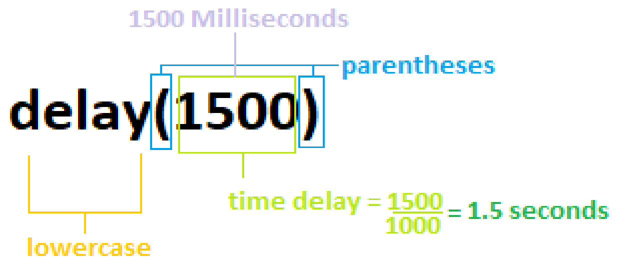
OUTPUT: Sum of total elements of an array: 150

Arduino Delay

- Arduino Delay specifies the **delay( ) function** used in the Arduino programming.
- The delay( ) function pauses the program or task for a specified duration of time. The time is specified inside the open and closed parentheses in milliseconds.

  Where,

  1 second = 1000 milliseconds



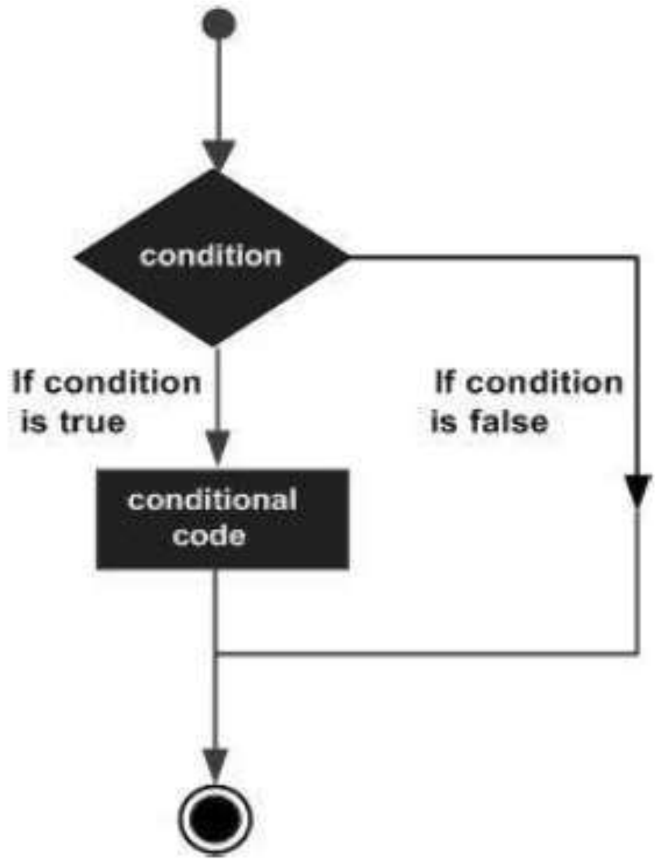
- The program waits for a specified duration before proceeding onto the next line of the code. The delay( ) function allows the **unsigned long** data type in the code.

Arduino Control Statements

- Decision making structures require that the programmer specify one or more conditions to be evaluated or tested by the program.

- It should be along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

- The general form of a typical decision making structure found in most of the programming languages



Arduino If statement

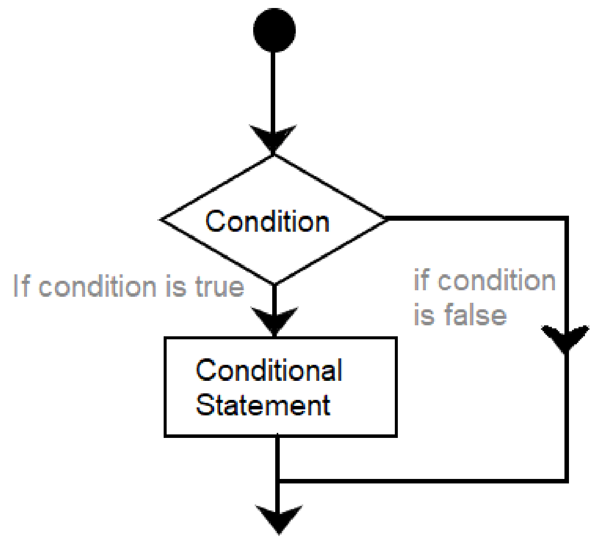
- The if ( ) statement is the conditional statement, it checks for the condition and then executes a statement or a set of statements.

- If the condition is False, it comes out of the if ( ) statement. If the condition is true, the function is performed.

- The if ( ) statement is written as:

```
if ( condition)
{
// include statements
// if the condition is true
// then performs the function or task specified inside the curly braces
}
```

Here, condition = It includes the boolean expressions, that can be true or false.



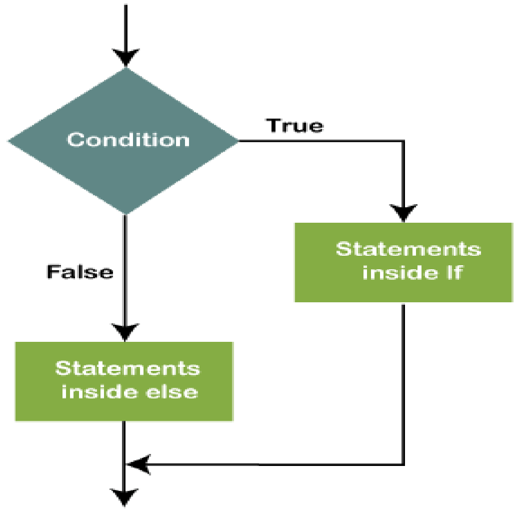
Arduino if-else and else-if

- The else and else-if both are used after specifying the if statement. It allows multiple conditions to be grouped.

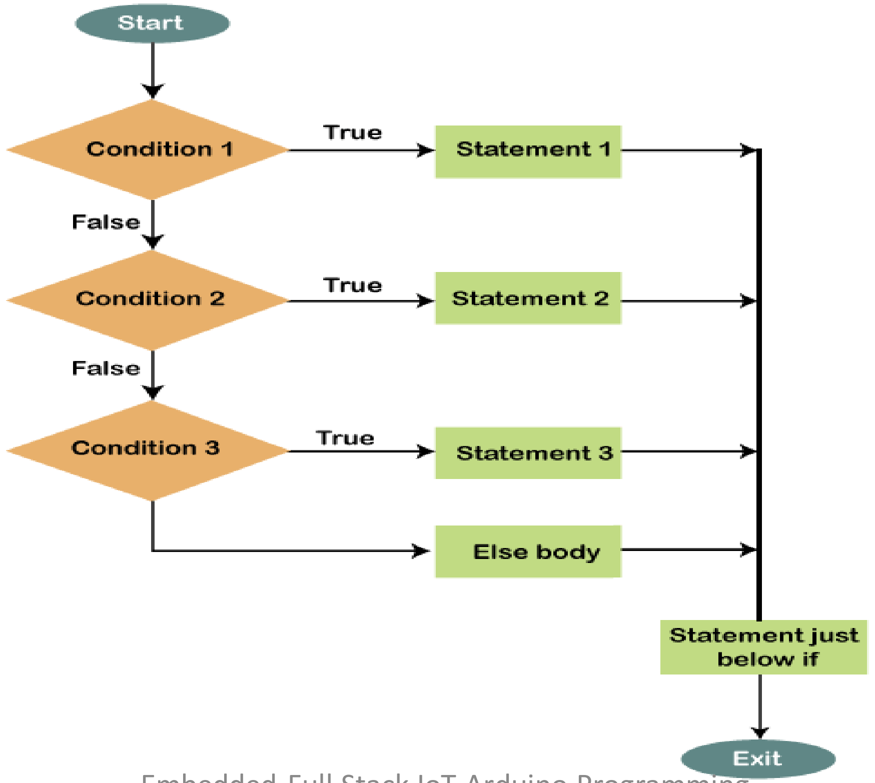
If else

- The if-else condition includes if ( ) statement and else ( ) statement. The condition in the else statement is executed if the result of the If ( ) statement is false.

```
if (condition)
{
// statements
}
else
{
//statements
}
```



- The else( ) statement can also include other if statements. Due to this, we can run multiple statements in a single program.
- The statements will be executed one by one until the true statement is found. When the true statement is found, it will skip all other if and else statements in the code and runs the associated blocks of code.



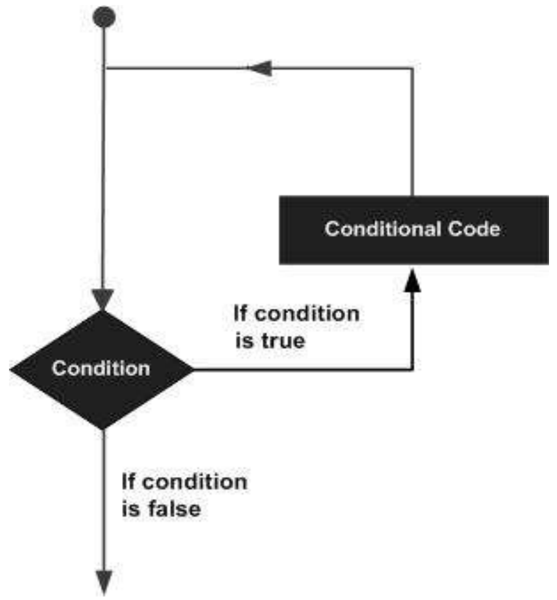
Else if

The else if statement can be used with or without the else ( ) statement. We can include multiple else if statements in a program.

```
if (condition)
{
// statements
}
else if ( condition)
{
// statements
// only if the first condition is false and the second is true
}
else
{
//statements
}
```

Arduino Loops

- Programming languages provide various control structures that allow for more complicated execution paths.
- A loop statement allows us to execute a statement or group of statements multiple times and following is the general form of a loop statement in most of the programming languages



Arduino for Loop

- The statements inside the curly brackets under for loop are executed repeatedly according to the specified condition. An increment counter in the for loop is used to increment or decrement the loop repetitions.

- The for statement is commonly used for repetitive task or operation or to operate on the group of data/pins in combination with arrays.

The syntax is:

```
for (initialization; condition; increment)
{
\\ statements
}
```

Where

- **initialization**: It is defined as the initialization of the variable.
- **condition**: The condition is tested on every execution. If the condition is **true**, it will execute the given task. The loop ends only when the condition becomes **false**.
- **increment**: It includes the increment operator, such as i + +, i - - , i + 1, etc. It is incremented each time until the condition remains true.

Example:

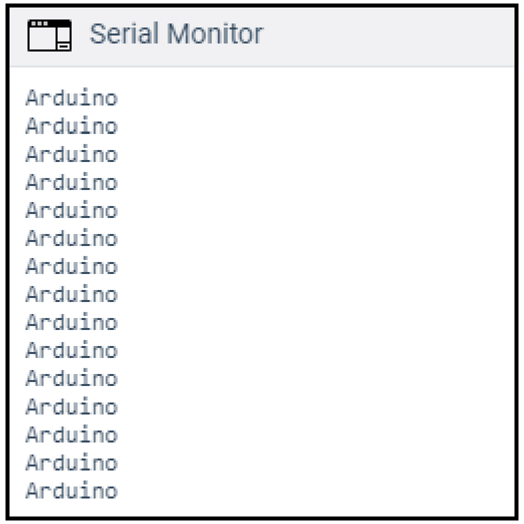
- **for** (i = 0 ; i < 5 ; i + +) // this statement will execute the loop for five times.   The values of i will be from 0 to 4.
- **for** (i = 0 ; i < = 5 ; i + +) //this statement will execute the loop six times. The values of i will be from 0 to 5.

- If we do not want to execute the for loop again and again. Then, we can insert the for loop in the void setup( ) function.

Example:

```
int i;
void setup ( )
{
  Serial.begin(9600);
  for ( i = 0 ; i < 15 ; i ++ )
  {
   Serial.println(  "Arduino");
  }
}
void loop ( ) {
}
```

OUTPUT



Serial Monitor

```
Arduino
Arduino
Arduino
Arduino
Arduino
Arduino
Arduino
Arduino
Arduino
Arduino
Arduino
Arduino
Arduino
Arduino
Arduino
```

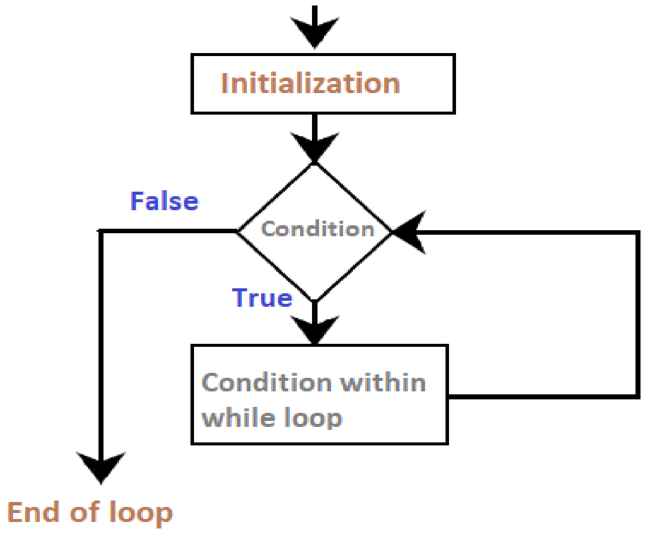
Arduino while loop

- The while loop() is the conditional loop that continues to execute the code inside the parentheses until the specified condition becomes false.

- The while loop will never exit until the tested condition is changed or made to stop. The common use of a while loop in Arduino includes sensor testing, calibration (calibrating the input of sensor), variable increment, etc.

The syntax is:

```
while (condition)
{
// code or set of statements
}
```



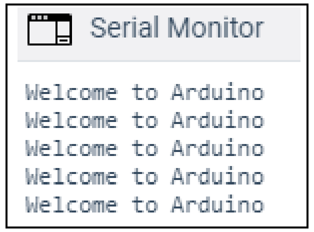
where,
condition: It specifies the boolean expression, which determines the condition to be true or false.

- In order to change the flow of the program, we need to change the specified condition inside the parentheses of while loop. The process is much like the if statement.

Example

```
int a = 0;
void setup()
{
  Serial.begin(9600);
while( a < 5)
{
  Serial.println("Welcome to Arduino");
  a = a + 1;
}
}
  void loop()
  {
  }
```

OUTPUT



```
Welcome to Arduino
Welcome to Arduino
Welcome to Arduino
Welcome to Arduino
Welcome to Arduino
```

do...while

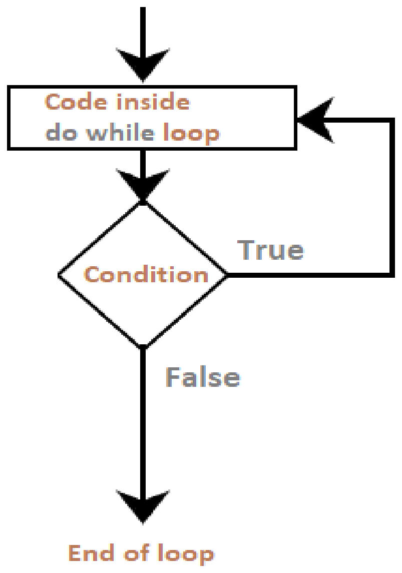
- The working of the do-while loop is similar to the while loop.
- The condition inside the do-while will execute at least once. It is because the condition is tested at the end of the loop instead of the beginning.

The syntax is:

```
do
{
// code or set of statements
} while (condition);
```

where,

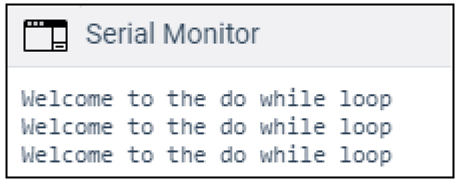
condition: It specifies the boolean expression, which determines the condition to be true or false.



Example:

```
int a = 0;
void setup()
{
  Serial.begin(9600);
do
{
  Serial.println("Welcome to the do while loop");
  a = a + 1;
} while( a < 3);
}
  void loop()
  {
  }
```

OUTPUT



```
Welcome to the do while loop
Welcome to the do while loop
Welcome to the do while loop
```

Arduino switch case

- The switch case controls the flow of the program by executing the code in various cases. A switch statement compares a particular value of a variable with statements in other cases. When the statements in a case matches the value of a variable, the code associated with that case executes.

- The break keyword is used at the end of each case. For example, if there are five cases, the break statements will also be five. The break statement exits the switch case.

- The switch statement without a break will continue to execute all the cases until the end. Hence, it is essential to include a break statement at the end of each case.

Example:

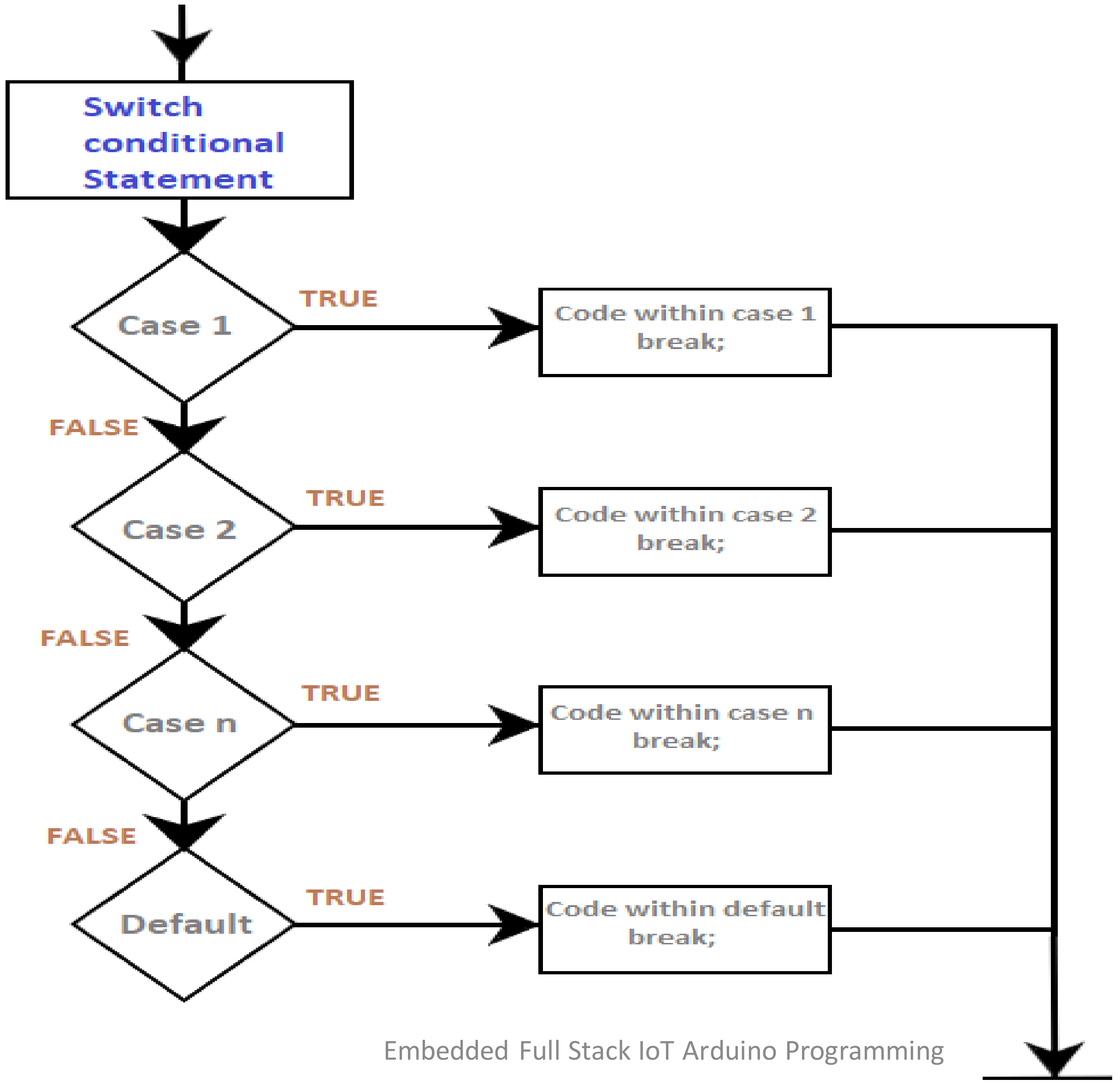
```
switch(variable)
{
case 1:
// statements related to case1
break;
case 2:
// statements related to case2
break;
. .
case n:
// statements related to case n
break;
default:
// it contains the optional code
//if nothing matches in the above cases, the default statement runs
break;
}
```

where,

variable: It includes the variables whose value will be compared with the multiple cases

value: It consists of a value to compare. These values are constants. The allowed data types are int and char.

Flowchart of the switch case:


Switch conditional Statement
Case 1
TRUE
Code within case 1 break;
FALSE
Case 2
TRUE
Code within case 2 break;
FALSE
Case n
TRUE
Code within case n break;
FALSE
Default
TRUE
Code within default break;

- Implementing a switch case is somewhat easier than if statements.

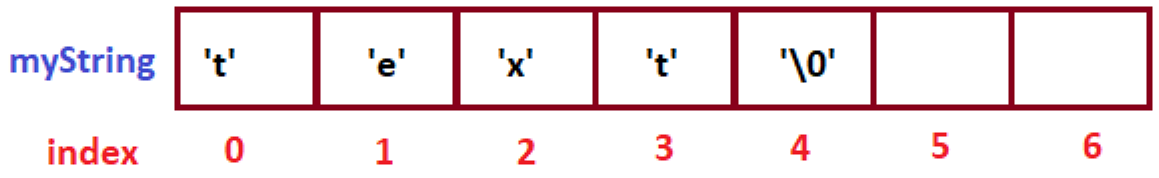
- It is recommended to use switch cases instead of if statement when multiple conditions of a non-trivial expression are being compared.

- The if statement allows us to choose between the two options, TRUE or FALSE. We can also use multiple if statements for more than two cases. The switch case allows us to choose between various discrete options.

Arduino String

- The string is a data type that stores text rather than the integer values. It is similar to other data types such as integer, float, etc., in Arduino.
- The string is an array of characters or a set of characters that consists of **numbers, spaces,** and **special characters** from the ASCII table.

  Example: **char** myString[len] = "text";



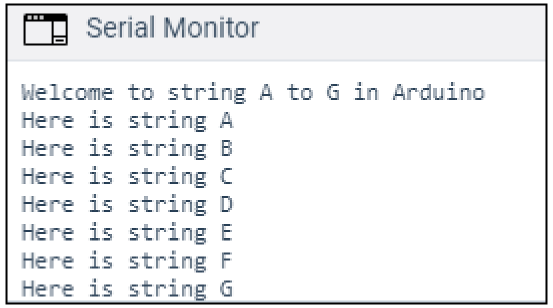
- We create a string using the double quotes like the word "**text**" specified above, the compiler automatically creates an element out of each character. It further appends the **null character** to the end.
- The null character has the value **0** in the ASCII table. It is defined using two characters (\ and \0). The backslash (\) represents the special character when used in conjunction with other characters. The backslash is also known as an **escape** character. The null character is added by the compiler to terminate the string.
- Strings are always declared inside the double quotes, while characters are declared inside single quote

Array of Strings

- We can specify the array of strings while working with large projects, such as LCD display, etc.
- The array names are also considered as pointers. The data type marked with an asterisk is defined as pointers.
- For example, char*. To define an array of arrays, we actually need pointers.

```
char *StrAB[] = {"Welcome to string A to G in Arduino", "Here is string A", "Here is string B",  "Here is string C", "Here is string D",
 "Here is string E",   "Here is string F", "Here is string G" };
void setup() {
 Serial.begin(9600);
}
void loop() {
  for (int i = 0; i < 8; i++)
  {
    Serial.println(StrAB[i]);
    delay(1000);
  }
}
```

OUTPUT


Serial Monitor

```
Welcome to string A to G in Arduino
Here is string A
Here is string B
Here is string C
Here is string D
Here is string E
Here is string F
Here is string G
```

Arduino String Object

- An object is like a variable, **which points to a memory location that holds some data**. The functions associated with the object are called **member functions**.
- We can make the objects to perform some actions.

Examples:

- The **begin(), print(),** and **println()** are the functions that are declared using the serial object. The period after the Serial (print(), etc.) specifies that we are accessing some members within the serial object. The members can be either a function or a variable.
- The parentheses after the print() and println() function determines the function being called in the serial. These functions cannot be declared alone in the global scope in the code. Thus, they are declared with the serial object as Serial.print() and Serial.println().

String Object

- The String object allows us to store and perform actions on an array of characters. The String object takes more memory than the regular String.
- The String object is always displayed with the uppercase 'S'. It produces an instance of the String class.
- It can be constructed from different data types, which are listed below:
  - a constant integer using a specific base
  - a long integer using a particular base
  - an instance of a String object
  - a constant character enclosed within a single-quotes
  - A constant string of characters enclosed within the double-quotes.
  - float and double

Arduino Functions

- The functions allow a programmer to divide a specific code into various sections, and each section performs a particular task. The functions are created to perform a task multiple times in a program.

- The function is a type of procedure that returns the area of code from which it is called.

- For example, to repeat a task multiple times in code, we can use the same set of statements every time the task is performed.

Advantages of using Functions

- It increases the readability of the code.
- It conceives and organizes the program.
- It reduces the chances of errors.
- It makes the program compact and small.
- It avoids the repetition of the set of statements or codes.
- It allows us to divide a complex code or program into a simpler one.
- The modification becomes easier with the help of functions in a program.

- The Arduino has two common functions **setup()** and **loop(),** which are called automatically in the background. The code to be executed is written inside the curly braces within these functions.

  - **void setup()** - It includes the initial part of the code, which is executed only once. It is called as the **preparation block**.

  - **void loop()** - It includes the statements, which are executed repeatedly. It is called the **execution block**.

- Sometimes, we need to write our own functions.

Function Declaration

The method to declare a function is listed below:

Function return type

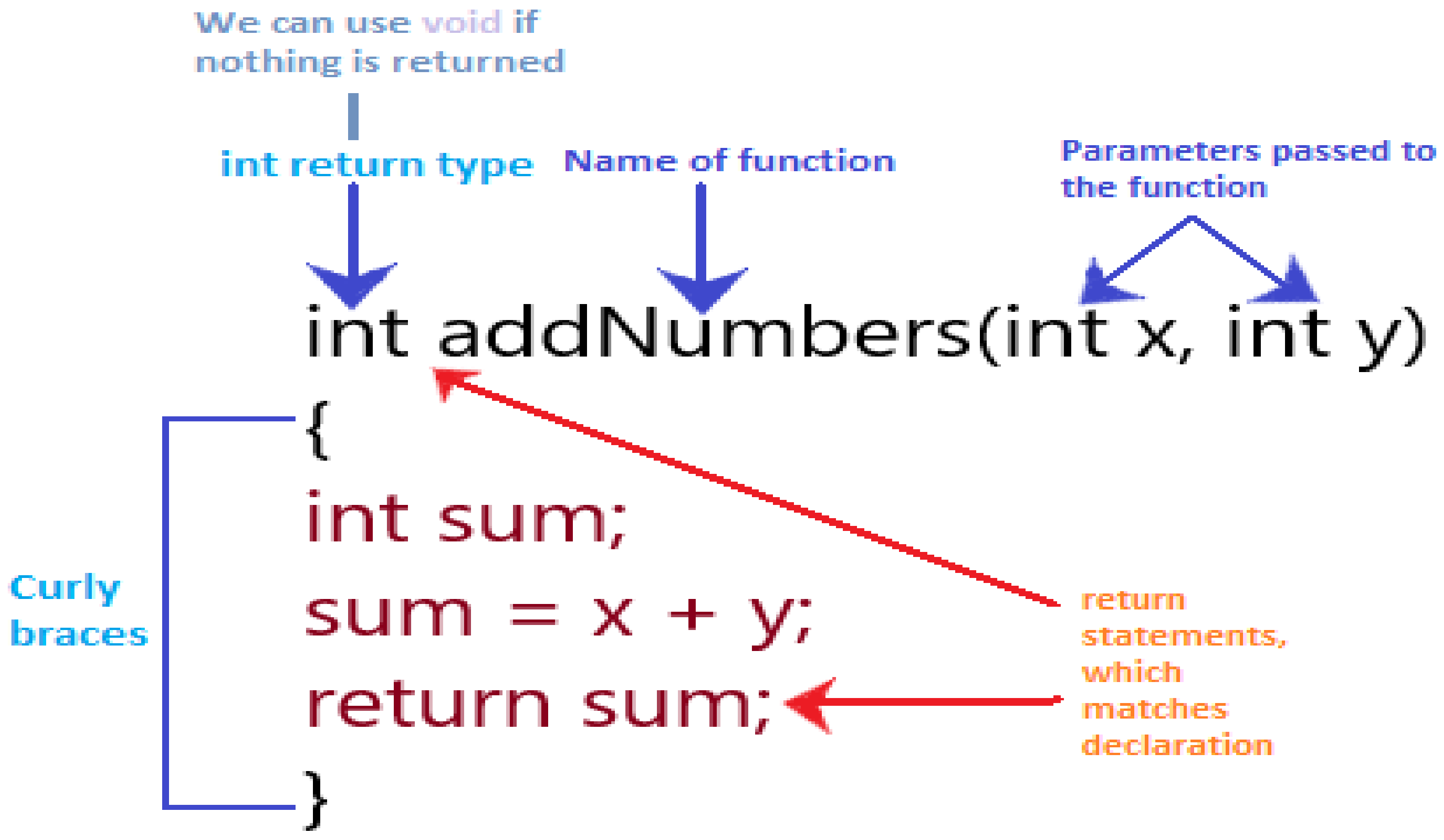
- We need a return type for a function. For example, we can store the return value of a function in a variable.
- We can use any data type as a return type, such as **float, char**, etc.

Function name

- It consists of a name specified to the function. It represents the real body of the function.

Function parameter

- It includes the parameters passed to the function. The parameters are defined as the special variables, which are used to pass data to a function.
- The actual data passed to the function is termed as an argument.


We can use void if nothing is returned
int return type
Name of function
Parameters passed to the function
Curly braces
return statements, which matches declaration

```
int addNumbers(int x, int y)
{
int sum;
sum = x + y;
return sum;
}
```

Arduino I/O Functions

- To interface the Arduino board with different integrated chips, sensors, LEDs, and other peripherals different functions are used for the input and output.

- Similarly, to run the compiled code on the Arduino board these functions are also used. These input and output functions also define the inputs and outputs of the Arduino program

Input/output functions

- There are five different types of functions that are used in Arduino for configuring its inputs and outputs. They are:

  - pinMode() function
  - digitalRead() function
  - digitalWrite() function
  - analogRead() function
  - analogWrite() function

pinMode() function

Accura Tequipment

- For connecting the peripherals to the Arduino board its pins are assigned to each device that has to be connected to the Arduino board.
- The pin number is assigned in the Arduino code using the pin mode function.
- The pin mode function has two arguments: one is the pin number, and the other is the mode of the pin.
- The pin modes are further divided into three types.
  - INPUT : It defines the respective pin that will be used as an input for Arduino.
  - OUTPUT : This mode is used when instruction is to be given to any connected device.
  - INPUT_PULLUP : This mode is also used to assign input state to the pin. By using this mode the polarity will be reversed of the given input for example if the Input is high that will mean the device is off and if the input is low that means the device is on. This function works with the help of internal resistors that are built in Arduino.

Syntax : **pinMode(pin-number, mode-of-pin);**

digitalRead() and digitalWrite() functions

Accura Tequipment

- There are usually 14 digital pins in the Arduino Boards which can be used for the read and write functions. **When the status of any specific pin is to be known then the digitalRead() function is used.** This function is a return type function as it will tell the status of the pin in its output.

- Similarly, **when a state is to be assigned to any pin then a digitalWrite() function is used.** The digitalWrite() function has two arguments one is the pin number and other is the state that will be defined by the user.

- Both functions are of Boolean type so, only two types of states are used in digital write function one is high and the other is low.

- To use digitalRead() and digitalWrite() functions the following syntax should be used:

```
digitalRead (pin-number);
digitalWrite(pin-number , state);
```

analogRead() and analogWrite() functions

Accura Tequipment

- The Arduino boards having analog ports which can be used by these analog read and write functions.
- The analogRead() function will read the state of the analog pin and will return a value in the form of numbers in the range from 0 to 1024 for 10 bits resolution and for 12 bits resolution the range will be 0 to 4095.
- The bit resolution is the analog to digital conversion so for 10 bit the range can be calculated by 2^10 and for 12 bits it will be 2^12 respectively. However, **to assign a state to any analog pin on the Arduino board the function analogWrite() is used.** It will generate the pulse modulation wave and the state will be defined by giving its duty cycle that ranges from 0 to 255
- The main difference between the analog and digital functions is that the digital defines the data in the form of either high or low whereas the analog gives the data in the form of a duty cycle of pulse width modulation.

The syntax of the analogue read and write is:

```
analogRead(pin-number);
analogWrite(pin-number , value-of-pin);
```

Programming Nodemcu ESP8266 with Arduino IDE

- The Nodemcu ESP8266 Development Board can be easily programmed using the Arduino IDE since it is easy to use.

- All you need is the latest version of the Arduino IDE, a USB cable and the **Nodemcu ESP8266** board itself.

Arduino Nodemcu ESP8266 Board Installation:

- Install the latest version of the Arduino IDE
- Copy the Nodemcu ESP8266 URL Link:

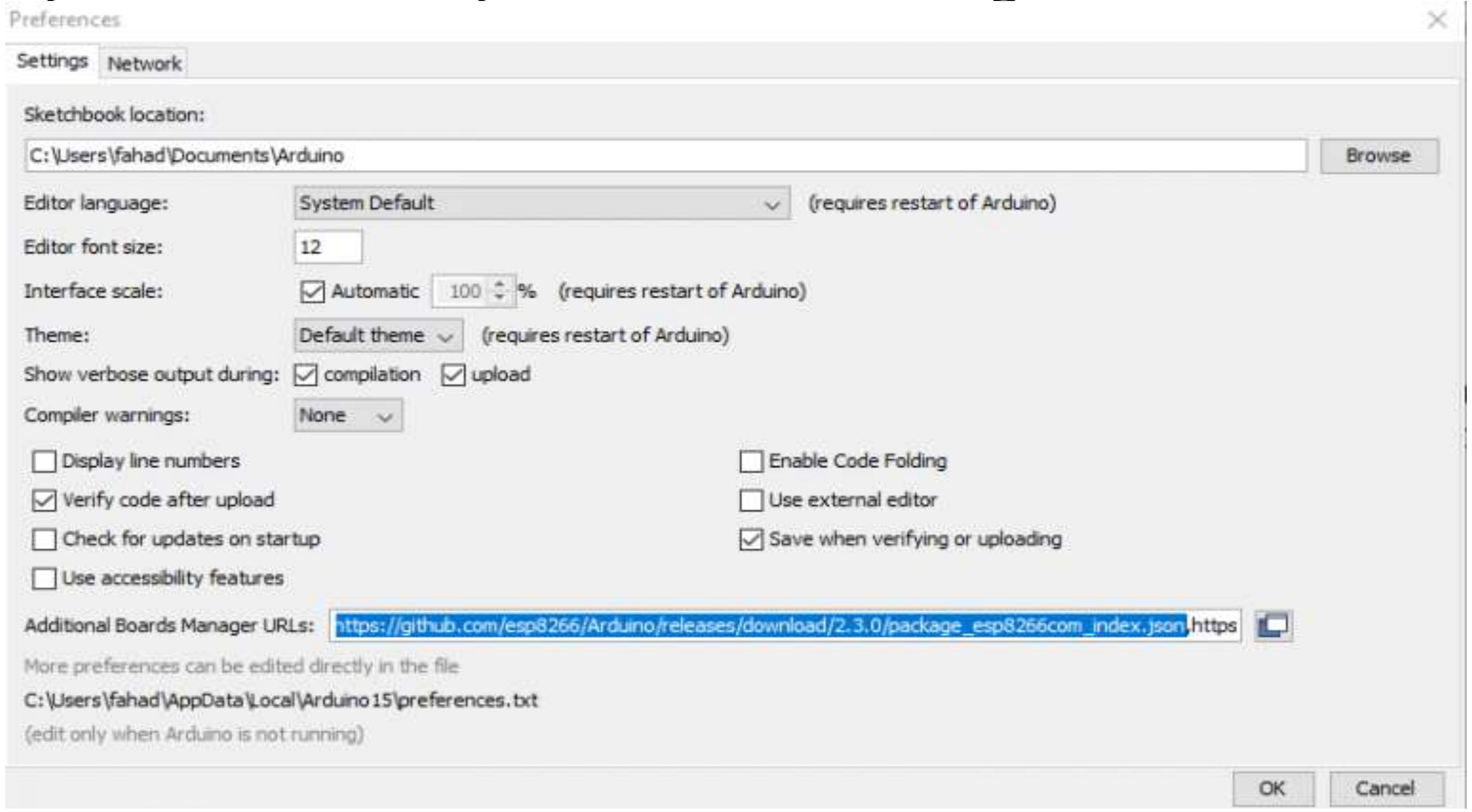
  http://arduino.esp8266.com/stable/package_esp8266com_index.json

Step1:

Open your Arduino IDE

Step2:

Click on the preferences and paste the URL link given above.



Preferences

Settings | Network

Sketchbook location:
C:\Users\fahad\Documents\Arduino

Editor language: System Default (requires restart of Arduino)
Editor font size: 12
Interface scale: [✓] Automatic 100 % (requires restart of Arduino)
Theme: Default theme (requires restart of Arduino)
Show verbose output during: [✓] compilation [✓] upload
Compiler warnings: None

- [ ] Display line numbers
- [✓] Verify code after upload
- [ ] Check for updates on startup
- [ ] Use accessibility features
- [ ] Enable Code Folding
- [ ] Use external editor
- [✓] Save when verifying or uploading

Additional Boards Manager URLs: https://github.com/esp8266/Arduino/releases/download/2.3.0/package_esp8266com_index.json,https

More preferences can be edited directly in the file
C:\Users\fahad\AppData\Local\Arduino15\preferences.txt
(edit only when Arduino is not running)

Paste the link in the additional Boards Manager URLs box and click on the OK button.