Basics of Arduino Programming

In this tutorial, we will cover the fundamental aspects of Arduino programming, including the structure of an Arduino program, variables, data types, operators, conditional statements, and loops.

Structure of an Arduino Program

An Arduino program, also known as a sketch, has a specific structure. Here's a basic outline:

```
// Libraries
#include <LibraryName.h>

// Constants
#define LED_PIN 13

// Global variables
int globalVariable = 0;

void setup() {
  // Setup code runs once when the Arduino is powered on or reset
  pinMode(LED_PIN, OUTPUT);
}

void loop() {
  // Loop code runs continuously after setup
  digitalWrite(LED_PIN, HIGH);
  delay(1000);
  digitalWrite(LED_PIN, LOW);
  delay(1000);
}
```

- **Libraries** : Include any external libraries your program might use.
- **Constants** : Define constants using #define.
- **Global Variables** : Declare variables that need to be accessible in both setup() and loop() functions.
- **setup() Function** : Code inside setup() runs once when the Arduino is powered on or reset. Initialize pin modes and perform setup tasks here.
- **loop() Function** : Code inside loop() runs continuously after setup(). This is where your main program logic goes.

Variables

Variables are containers used to store and manipulate data. They have a specific data type that determines the type of data they can hold, such as integers, floating-point numbers, characters, or booleans.

Variables must be declared before they are used. Declaration specifies the type of data the variable will hold. Here's an example:

```
// Declaration of an integer variable
int age;
// Declaration of a floating-point variable
float temperature;

// Initialization
// Assigning a value to the integer variable
age = 25;
// Assigning a value to the floating-point variable
temperature = 98.6;

// Declaration and initialization in one line
int age = 25;
float temperature = 98.6;
```

Variables can have different scopes

- **Local Variables** : Declared inside a specific function and only accessible within that function.
- **Global Variables** : Declared outside of any function and accessible throughout the entire program.

```
int globalVariable = 0;  // Global variable

void setup() {
  int localVariable = 10;  // Local variable
}

void loop() {
}
```

Constants

Constants are variables whose values should not be changed during program execution. They are declared using #define or the const keyword

```
#define MAX_VALUE 100    // Define a constant
const int MIN_VALUE = 0; // Declare a constant with the const keyword
```

Operators

Operators perform operations on variables. Common operators include arithmetic, comparison, and logical operators

```
int a = 5;
int b = 2;
```

```
int sum = a + b;  // Addition
int difference = a - b;  // Subtraction
int product = a * b;  // Multiplication
int quotient = a / b;  // Division
int remainder = a % b;  // Modulo (remainder)
```

1. **Arithmetic Operators** Perform basic mathematical operations.
    - **Addition (+)** : Adds two values.
    - **Subtraction (-)** : Subtracts the right operand from the left operand.
    - **Multiplication (*)** : Multiplies two values.
    - **Division (/)** : Divides the left operand by the right operand.
    - **Modulo (%)** : Returns the remainder of the division.
2. **Comparison Operators** Compare two values and return a Boolean result.
    - **Equal to (==)** : Checks if two values are equal.
    - **Not equal to (!=)** : Checks if two values are not equal.
    - **Greater than (>)** : Checks if the left operand is greater than the right operand.
    - **Less than (<)** : Checks if the left operand is less than the right operand.
    - **Greater than or equal to (>=)** : Checks if the left operand is greater than or equal to the right operand.
    - **Less than or equal to (<=)** : Checks if the left operand is less than or equal to the right operand.
3. **Logical Operators** Perform logical operations and return a Boolean result.
    - **Logical AND (&&)** : Returns true if both conditions are true.
    - **Logical OR (||)** : Returns true if at least one of the conditions is true.
    - **Logical NOT (!)** : Returns true if the condition is false and vice versa.
4. **Bitwise Operators** Manipulate individual bits in binary representations of integers.
    - **Bitwise AND (&)** : Performs a bitwise AND operation.
    - **Bitwise OR (|)** : Performs a bitwise OR operation.
    - **Bitwise XOR (^)** : Performs a bitwise XOR (exclusive OR) operation.
    - **Bitwise NOT (~)** : Flips the bits of a binary number.
    - **Left Shift (<<)** : Shifts the bits of the left operand to the left by a specified number of positions.
    - **Right Shift (>>)** : Shifts the bits of the left operand to the right by a specified number of positions.
5. **Assignment Operator** Assigns a value to a variable.
    - **Assignment (=)** : Assigns the value on the right to the variable on the left.

IMP NOTE : Bitwise Operators

Bitwise AND and OR operations are often used for set and reset operations, especially when dealing with individual bits in registers or variables.

1. Set Operation:

```
// Set a specific bit (e.g., 3rd bit)
int value = 0b11011010;  // Binary representation: 11011010
value |= (1 << 2);       // Set 3rd bit
```

2. Reset Operation

```
// Reset a specific bit (e.g., 4th bit)
int value = 0b11011010;  // Binary representation: 11011010
value &= ~(1 << 3);      // Reset 4th bit
```

Data types

Arduino supports various data types for handling different kinds of data. Here is a list of commonly used data types in Arduino

1. Primitive Data Types

```
// int: Integer data type. Typically 16 bits.
int myInteger = 42;

// long: Long integer data type. Typically 32 bits.
long myLong = 123456789;

// float: Single-precision floating-point data type.
float myFloat = 3.14;

// double: Double-precision floating-point data type.
double myDouble = 3.14159265359;

// char: Character data type. Represents a single ASCII character.
char myChar = 'A';

// boolean: Boolean data type. Represents true or false.
boolean myBoolean = true;
```

2. Derived Data Types

```
// String: String class for working with text.
String myString = "Hello, Arduino!";
```

3. Arrays

```
// Collection of variables of the same type.
int myArray[5] = {1, 2, 3, 4, 5};
```

4. Pointer Types

```
// Holds the memory address of another variable.
int* myPointer = &myInteger;
```

5. Special Types

```
// Represents the absence of type
void setup() {
    // Code for setup
}

// nullptr_t: Represents a null pointer.
nullptr_t myNullPtr = nullptr;
```

6. Integer Types with Specific Bit Lengths

```
// uint8_t: Unsigned 8-bit integer.
uint8_t myUInt8 = 255;

// int8_t: Signed 8-bit integer.
int8_t myInt8 = -128;

// uint16_t: Unsigned 16-bit integer.
uint16_t myUInt16 = 65535;

// int16_t: Signed 16-bit integer.
int16_t myInt16 = -32768;

// uint32_t: Unsigned 32-bit integer.
uint32_t myUInt32 = 4294967295;

// int32_t: Signed 32-bit integer.
int32_t myInt32 = -2147483648;
```

7. Other Types

```
// size_t: Represents the size of an object. Typically used for indexing and
loop counters.
size_t mySize = 10;
```

```
// byte: Equivalent to uint8_t. Used for working with bytes of data.
byte myByte = 0xAA;
```

8. User-Defined Types

```
// enum: User-defined enumeration type.
enum Day {Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday};

// A structure allows you to group different data types under a single name.
struct Point {
    int x;
    int y;
};
Point myPoint;
myPoint.x = 10;
myPoint.y = 20;


// If you're working with object-oriented programming (OOP), you can create
classes to encapsulate data and behavior.
class Rectangle {
    public:
        int width;
        int height;

    int calculateArea() {
        return width * height;
    }
};
Rectangle myRect;
myRect.width = 5;
myRect.height = 10;
int area = myRect.calculateArea();
```

Conditional statements

Conditional statements in Arduino allow you to control the flow of your program based on certain conditions. Here are the basic conditional statements:

1. If Statement

The `if` statement is used to execute a block of code if a specified condition is true.

```
int sensorValue = analogRead(A0);

if (sensorValue > 500) {
  // Code to be executed if the sensor value is greater than 500
  digitalWrite(LED_PIN, HIGH);
}
```

2. If-else Statement

The if-else statement allows you to specify two blocks of code: one to be executed if the condition is true and another if it's false.

```
int buttonState = digitalRead(buttonPin);

if (buttonState == HIGH) {
  // Code to be executed if the button is pressed
  digitalWrite(LED_PIN, HIGH);
} else {
  // Code to be executed if the button is not pressed
  digitalWrite(LED_PIN, LOW);
}
```

3. If-else if-else Statement

You can use the else if statement to check multiple conditions in sequence.

```
int sensorValue = analogRead(A0);

if (sensorValue < 100) {
  // Code to be executed if the sensor value is less than 100
  digitalWrite(LED_PIN, HIGH);
} else if (sensorValue < 500) {
  // Code to be executed if the sensor value is between 100 and 500
  digitalWrite(LED_PIN, LOW);
} else {
  // Code to be executed if the sensor value is greater than or equal to 500
  digitalWrite(LED_PIN, HIGH);
}
```

Loop Statements in Arduino

Loop statements in Arduino allow you to repeatedly execute a block of code. The primary loop structure is the `loop()` function.

1. Basic Loop Structure

The basic structure of the Arduino sketch includes the `setup()` function, executed once at the beginning, and the `loop()` function, continuously executed in a loop.

```
void setup() {
  // Code to run once
```

```
}

void loop() {
  // Code to run repeatedly
}
```

2. For Loop

The for loop allows you to repeat a block of code a specific number of times.

```
for (int i = 0; i < 5; i++) {
  // Code to be repeated 5 times
  digitalWrite(LED_PIN, HIGH);
  delay(500);
  digitalWrite(LED_PIN, LOW);
  delay(500);
}
```

3. While Loop

The while loop repeats a block of code as long as a specified condition is true.

```
int counter = 0;

while (counter < 3) {
  // Code to be repeated 3 times
  digitalWrite(LED_PIN, HIGH);
  delay(500);
  digitalWrite(LED_PIN, LOW);
  delay(500);

  counter++;
}
```

4. Do-While Loop

The do-while loop is similar to the while loop but guarantees that the code inside the loop is executed at least once.

```
int buttonState;

do {
  // Code to be executed at least once
  buttonState = digitalRead(buttonPin);
  delay(1000);
```

```
  } while (buttonState == LOW);
```