

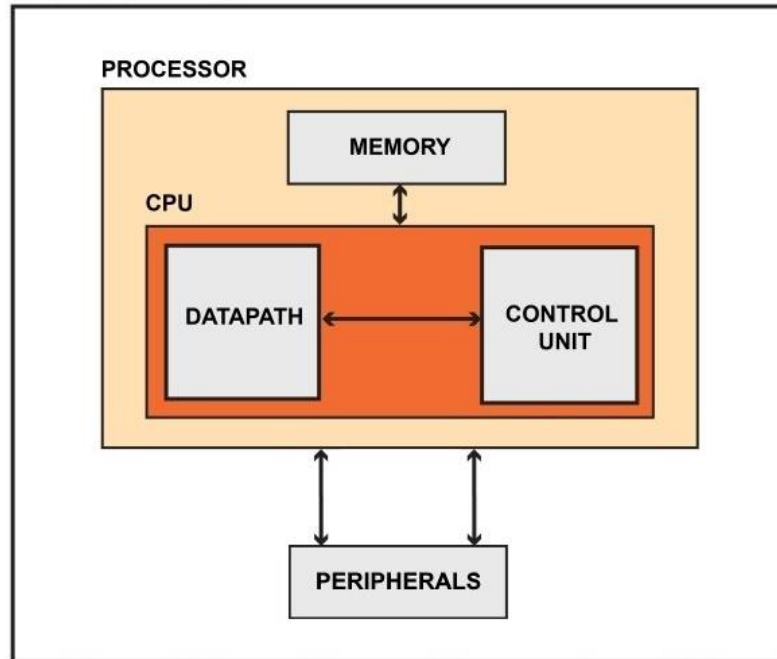
# Introduction to Biomedical Engineering

Section 3: Microcontrollers/Arduino

Lecture 3.1: Basics of coding for digital controller

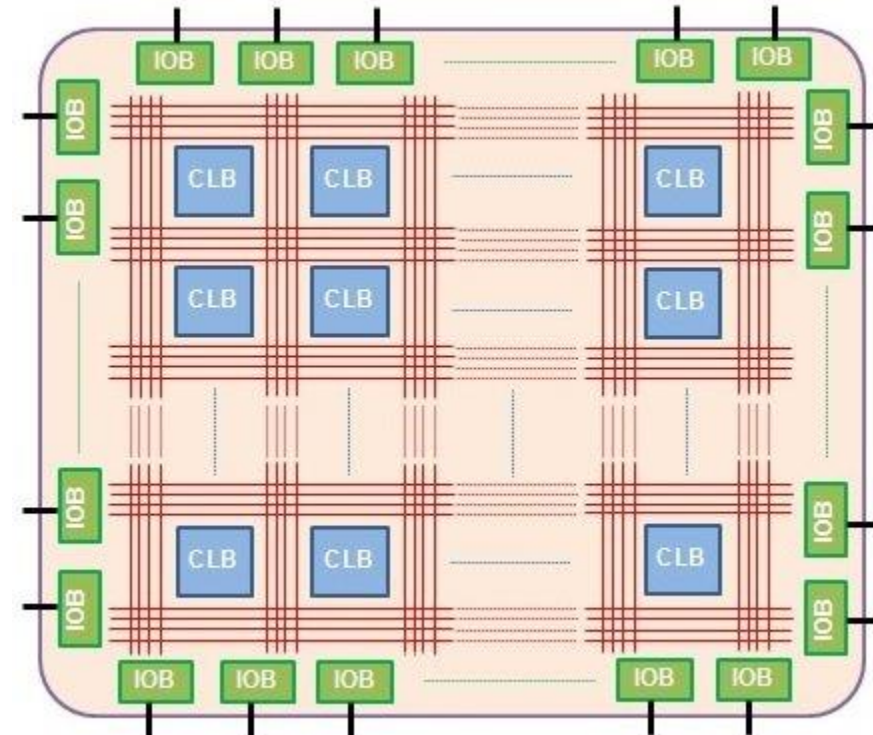
# Embedded Programming

Microcontroller



- Universal
- Versatile
- High-level
- SOFTWARE

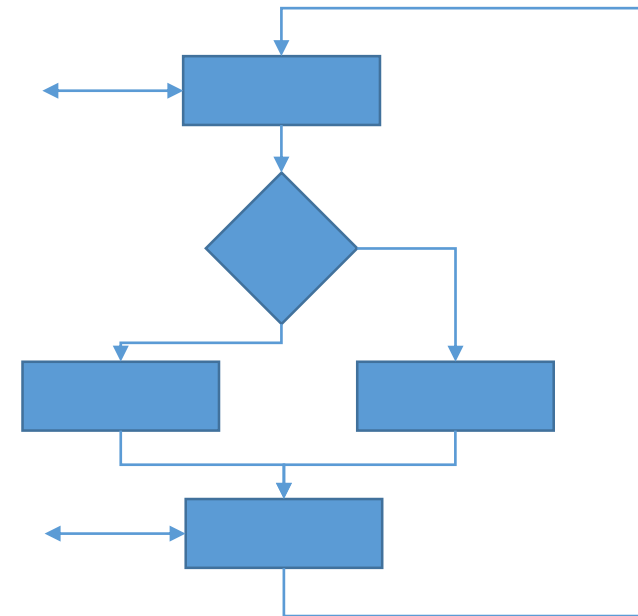
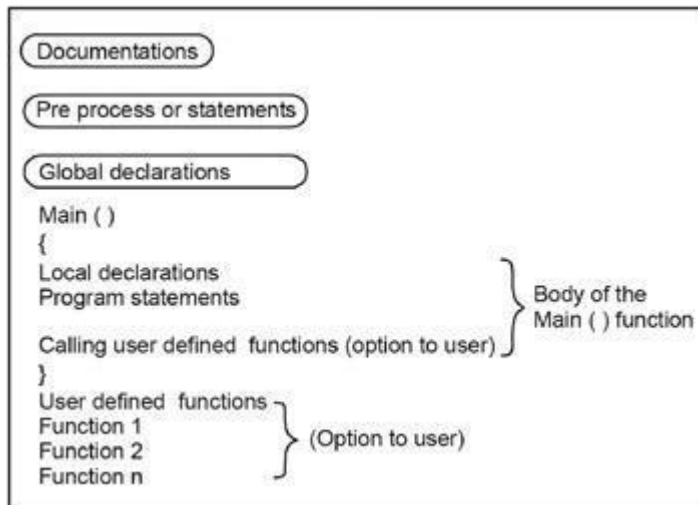
Field-programmable gate array (FPGA)



- Quick
- Reliable
- Parallel
- HARDWARE

# Embedded Programming for microcontrollers

- C language was developed by Dennis Ritchie in 1969. It is a collection of one or more functions, and every function is a collection of statements performing a specific task.
- C language is a middle-level language as it supports high-level applications and low-level applications.



# Embedded Programming for microcontrollers

- Some chip-specific and periphery-specific code is already there
- Mid-level programming (you do not have to know internals, although this knowledge helps code optimisation)



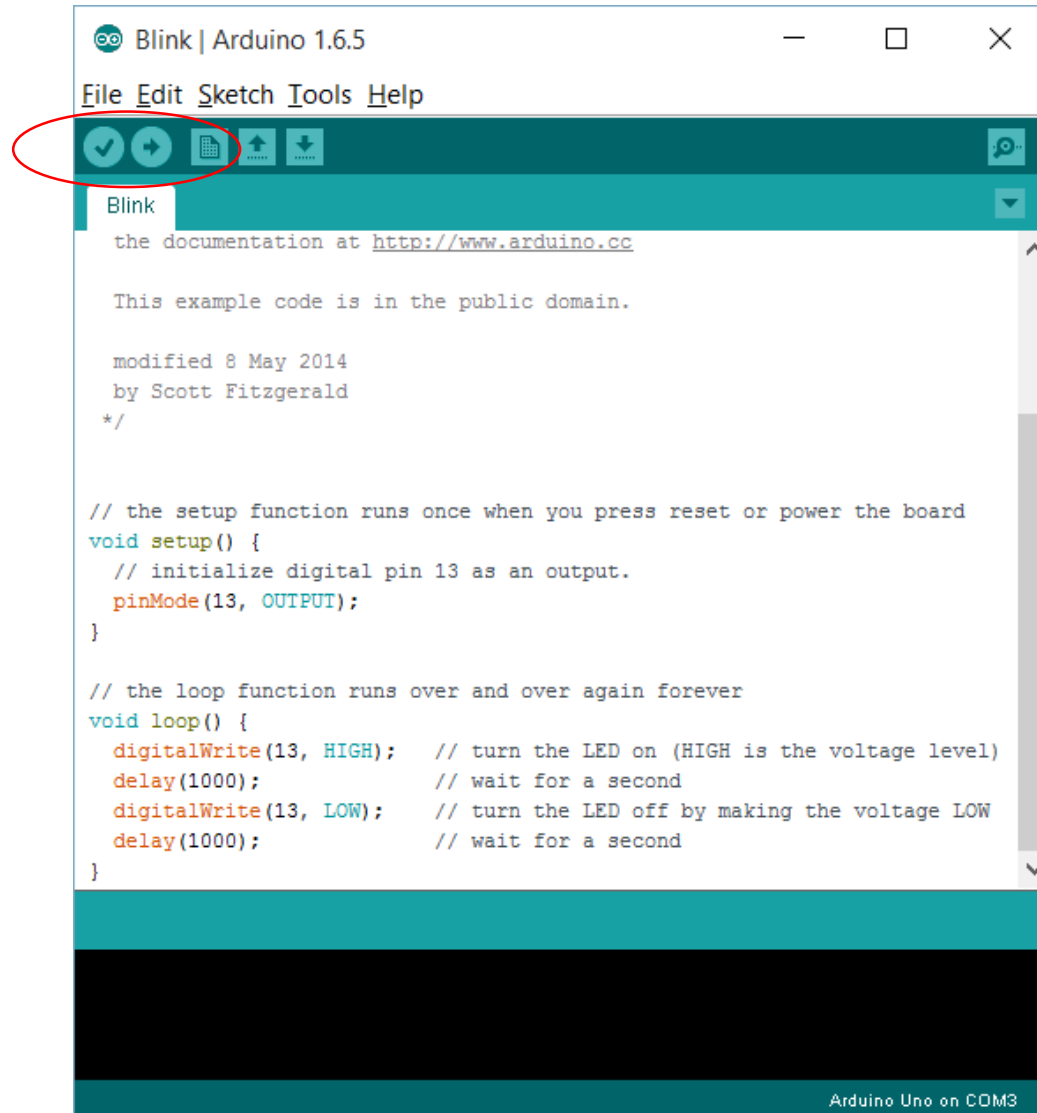
# Arduino basics



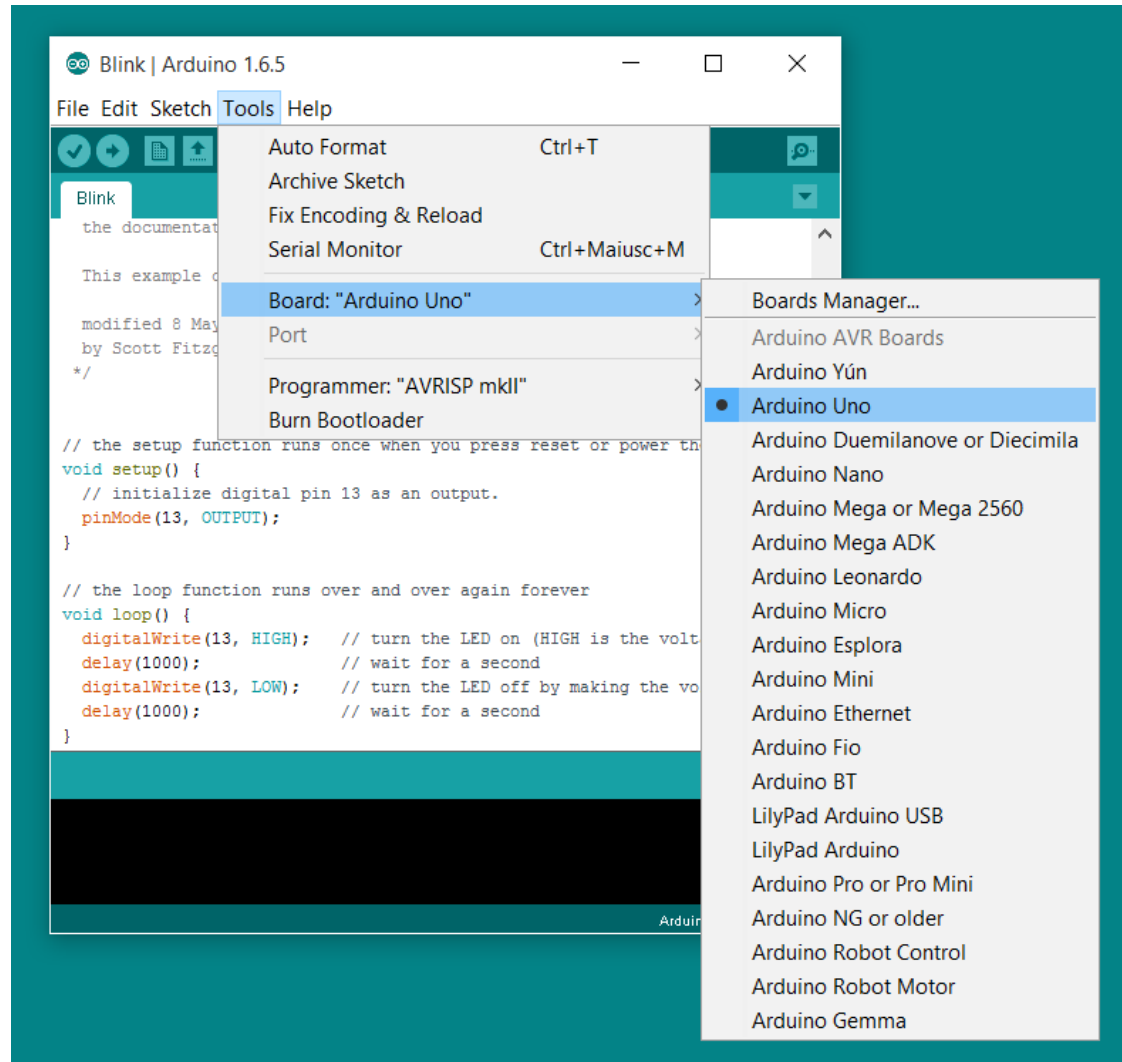
## Basic structure

Arduino program is called **sketch**

# Arduino IDE overview



# Getting started



Select your serial port



Upload the example program

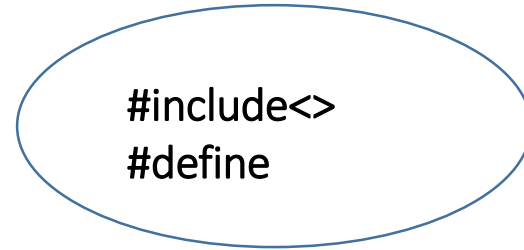


See what it is doing



Read the code to  
analyse how it works

# Arduino program structure



#include<>  
#define

Pre processor directives

- Include other code written elsewhere
- Define parameters/structures used in the code

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

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



# Lets see what the basic program does

// 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);  
}
```

Semicolon separates the lines of code!

// the loop function runs over and over again forever

```
void loop() {  
  digitalWrite(LED_BUILTIN, HIGH); // turn the LED on (HIGH is the voltage level)  
  delay(1000);                     // wait for a second  
  digitalWrite(LED_BUILTIN, LOW);  // turn the LED off by making the voltage LOW  
  delay(1000);                     // wait for a second  
}
```

# Lets see what the basic program does

// 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);

**}**

Do not have to memorise the  
LED port address, and  
prepare dozen of other  
hidden functions, just say  
what you want!

// the loop function runs over and over again forever

**void loop() {**

  digitalWrite(LED\_BUILTIN, HIGH); // turn the LED on (HIGH is the voltage level)

  delay(1000); // wait for a second

  digitalWrite(LED\_BUILTIN, LOW); // turn the LED off by making the voltage LOW

  delay(1000); // wait for a second

**}**

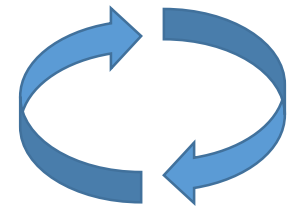
# Lets see what the basic program does

// 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 the voltage level)  
  delay(1000);                     // wait for a second  
  digitalWrite(LED_BUILTIN, LOW);  // turn the LED off by making the voltage LOW  
  delay(1000);                     // wait for a second  
}
```



# Lets see what the basic program does

// 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 the voltage level)  
  delay(1000);                     // wait for a second  
  digitalWrite(LED_BUILTIN, LOW);  // turn the LED off by making the voltage LOW  
  delay(1000);                     // wait for a second  
}
```

# Lets see what the basic program does

// 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 the voltage level)  
  delay(1000);                     // wait for a second  
  digitalWrite(LED_BUILTIN, LOW);  // turn the LED off by making the voltage LOW  
  delay(1000);                     // wait for a second  
}
```

# Lets see what the basic program does

// 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 the voltage level)  
  delay(1000);                     // wait for a second  
  digitalWrite(LED_BUILTIN, LOW);  // turn the LED off by making the voltage LOW  
  delay(1000);                     // wait for a second  
}
```

# Lets see what the basic program does

// 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 the voltage level)  
  delay(1000);                     // wait for a second  
  digitalWrite(LED_BUILTIN, LOW);  // turn the LED off by making the voltage LOW  
  delay(1000);                     // wait for a second  
}
```

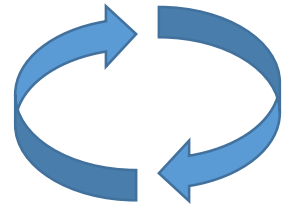
# Lets see what the basic program does

// 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);  
}
```

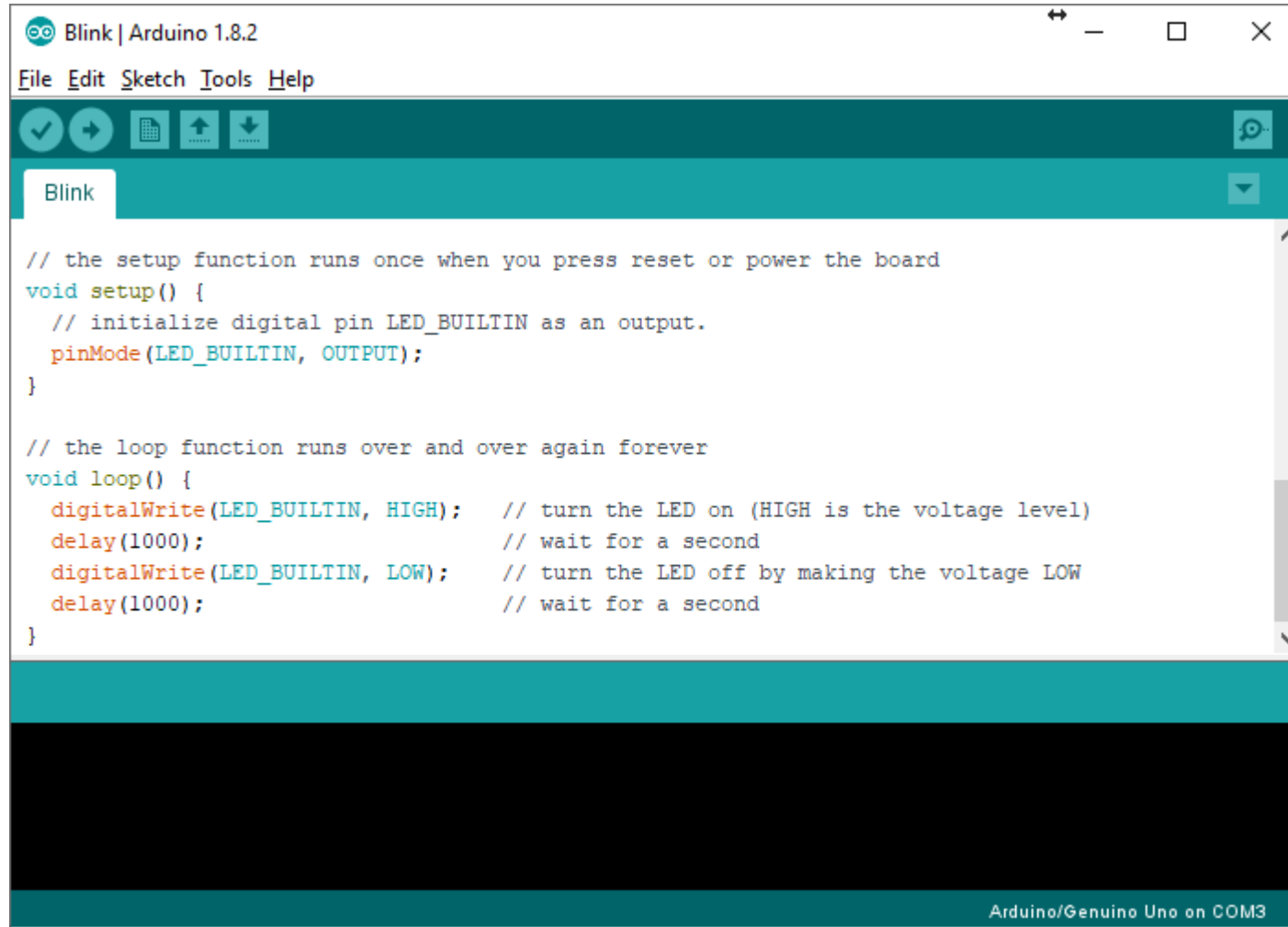
// the loop function runs over and over again forever

```
void loop() {  
  digitalWrite(LED_BUILTIN, HIGH); // turn the LED on (HIGH is the voltage level)  
  delay(1000);                     // wait for a second  
  digitalWrite(LED_BUILTIN, LOW);  // turn the LED off by making the voltage LOW  
  delay(1000);                     // wait for a second  
}
```





# Arduino IDE is helping with colours



The screenshot shows the Arduino IDE window titled "Blink | Arduino 1.8.2". The menu bar includes File, Edit, Sketch, Tools, and Help. Below the menu bar is a toolbar with icons for checking, running, uploading, and downloading. A tab labeled "Blink" is active. The main text area contains the following C++ code:

```
// 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 the voltage level)
  delay(1000);                      // wait for a second
  digitalWrite(LED_BUILTIN, LOW);  // turn the LED off by making the voltage LOW
  delay(1000);                      // wait for a second
}
```

The bottom status bar indicates "Arduino/Genuino Uno on COM3".

# Useful (necessary) programming concepts

- Data types
- Arrays
- Loops
- Functions
- Logical statements
- Interrupts


Stick to these if you do not know what you are doing



## More sophisticated/OOP:

- Pointers
- Classes and encapsulation/polymorphism/inheritance

Read about these if you are serious about the application



# Data types

Note: **signed** variables allow both positive and negative numbers, while **unsigned** variables allow only positive values.

- **boolean** (8 bit) - simple logical true/false
- **byte** (8 bit) - unsigned number from 0-255
- **char** (8 bit) - signed number from -128 to 127. The compiler will attempt to interpret this data type as a character in some circumstances, which may yield unexpected results
- **unsigned char** (8 bit) - same as 'byte'; if this is what you're after, you should use 'byte' instead, for reasons of clarity
- **word** (16 bit) - unsigned number from 0-65535
- **unsigned int** (16 bit)- the same as 'word'. Use 'word' instead for clarity and brevity
- **int** (16 bit) - signed number from -32768 to 32767. This is most commonly what you see used for general purpose variables in Arduino example code provided with the IDE
- **unsigned long** (32 bit) - unsigned number from 0-4,294,967,295. The most common usage of this is to store the result of the millis() function, which returns the number of milliseconds the current code has been running
- **long** (32 bit) - signed number from -2,147,483,648 to 2,147,483,647
- **float** (32 bit) - signed number from -3.4028235E38 to 3.4028235E38. Floating point on the Arduino is not native; the compiler has to jump through hoops to make it work. If you can avoid it, you should. We'll touch on this later.

# Variable. Scope. Qualifiers

Variable has to be declared. Depending on where you declare it, it will have different scope.

```
int gPWMval; // any function will see this variable
```

```
void setup() { // ...  
}
```

```
void loop() {  
  int i; // "i" is only "visible" inside of "loop"  
  float f; // "f" is only "visible" inside of "loop"  
  // ...
```

```
  for (int j = 0; j < 100; j++) {  
    // variable j can only be accessed inside the for-loop brackets  
  }  
}
```

# Variable scope qualifiers

## const

```
const float pi = 3.14;  
float x;  
// ....  
x = pi * 2; // it's fine to use consts in math  
pi = 7; // illegal - you can't write to (modify) a constant
```

## static

used to create variables that are visible to only one function. However unlike local variables that get created and destroyed every time a function is called, static variables persist beyond the function call, preserving their data between function calls.

## volatile

A variable should be declared volatile whenever its value can be changed by something beyond the control of the code section in which it appears, such as a concurrently executing thread. In the Arduino, the only place that this is likely to occur is in sections of code associated with interrupts, called an interrupt service routine

# Arrays

```
int myInts[6];
```

```
int myPins[] = {2, 4, 8, 3, 6};
```

```
int mySensVals[6] = {2, 4, -8, 3, 2};
```

```
char message[6] = "hello";
```

```
mySensVals[0] = 10;
```

```
x = mySensVals[4];
```

# Loops

- If you are doing repetitive task

```
for (int i = 0; i <= 255; i++) {  
    // do something repetitive 256 times  
    analogWrite(PWMPin, i);  
}
```

```
int var = 0;  
while (var < 256) {  
    // do something repetitive 256 times  
    var++;  
}
```

```
int x = 0;  
do {  
    delay(50); // wait for sensors to stabilize  
    x = readSensors();  
    // check the sensors  
} while (x < 100);
```

# Functions

- If you need to use the same action several times under different conditions
  - Can take input variables
  - Can return output variables

```
void loop() {  
    int a=1;  
    int b=2;  
    int x = myFun( a, b);  
}
```

```
int myFun (int a, int b) {  
    int result;  
    result = a + b;  
    return result;  
}
```



# Functions

- Some pre-built functions are already there
- You can get plenty of help on the website <https://www.arduino.cc/reference/en/>

## Digital I/O

`digitalRead()`  
`digitalWrite()`  
`pinMode()`

## Analog I/O

`analogRead()`  
`analogReference()`  
`analogWrite()`

## Zero, Due & MKR Family

`analogReadResolution()`  
`analogWriteResolution()`

## Advanced I/O

`noTone()`  
`pulseIn()`  
`pulseInLong()`  
`shiftIn()`  
`shiftOut()`  
`tone()`

## Time

`delay()`  
`delayMicroseconds()`  
`micros()`  
`millis()`

## Math

`abs()`  
`constrain()`  
`map()`  
`max()`  
`min()`  
`pow()`  
`sq()`  
`sqrt()`

## Trigonometry

`cos()`  
`sin()`  
`tan()`

## Characters

`isAlpha()`  
`isAlphaNumeric()`  
`isAscii()`  
`isControl()`  
`isDigit()`  
`isGraph()`  
`isHexadecimalDigit()`  
`isLowerCase()`  
`isPrintable()`  
`isPunct()`  
`isSpace()`  
`isUpperCase()`  
`isWhitespace()`

## Random Numbers

`random()`  
`randomSeed()`

## Bits and Bytes

`bit()`  
`bitClear()`  
`bitRead()`  
`bitSet()`  
`bitWrite()`  
`highByte()`  
`lowByte()`

## External Interrupts

`attachInterrupt()`  
`detachInterrupt()`

## Interrupts

`interrupts()`  
`noInterrupts()`

## Communication

Serial  
Stream

## USB

Keyboard  
Mouse

# Logical statements

```
if (x > 120) digitalWrite(LEDpin, HIGH);
```

```
if (x > 120) digitalWrite(LEDpin, HIGH);
```

```
if (x > 120) { digitalWrite(LEDpin, HIGH); }
```

```
if (x > 120) {  
    digitalWrite(LEDpin1, HIGH);  
    digitalWrite(LEDpin2, HIGH);  
}
```

```
if (condition1) {  
    // do Thing A  
}  
else if (condition2) {  
    // do Thing B  
}  
else {  
    // do Thing C  
}
```

# Other bits worth knowing

## Sketch

`loop()`

`setup()`

## Control Structure

`break`

`continue`

`do...while`

`else`

`for`

`goto`

`if`

`return`

`switch...case`

`while`

## Further Syntax

`#define` (define)

`#include` (include)

`/* */` (block comment)

`//` (single line comment)

`;` (semicolon)

`{ }` (curly braces)

## Arithmetic Operators

`%` (remainder)

`*` (multiplication)

`+` (addition)

`-` (subtraction)

`/` (division)

`=` (assignment operator)

## Comparison Operators

`!=` (not equal to)

`<` (less than)

`<=` (less than or equal to)

`==` (equal to)

`>` (greater than)

`>=` (greater than or equal to)

## Boolean Operators

`!` (logical not)

`&&` (logical and)

`||` (logical or)

## Pointer Access Operators

`&` (reference operator)

`*` (dereference operator)

## Bitwise Operators

`&` (bitwise and)

`<<` (bitshift left)

`>>` (bitshift right)

`^` (bitwise xor)

`|` (bitwise or)

`~` (bitwise not)

## Compound Operators

`%=` (compound remainder)

`&=` (compound bitwise and)

`*=` (compound multiplication)

`++` (increment)

`+=` (compound addition)

`--` (decrement)

`-=` (compound subtraction)

`/=` (compound division)

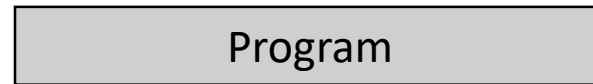
`^=` (compound bitwise xor)

`|=` (compound bitwise or)

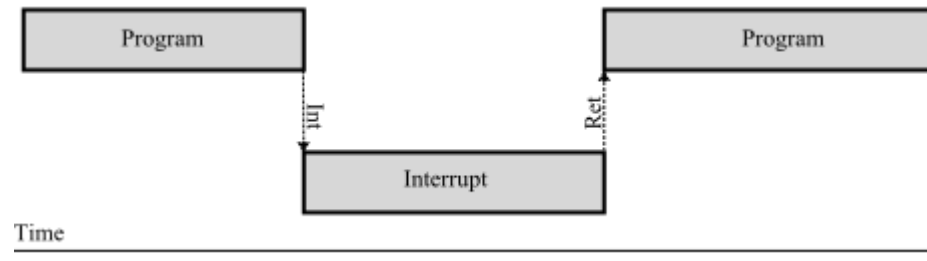
# Microcontroller-specific tools

- Program flow and interrupts

Normal flow



Interruption



# Interrupts

- Interruptions are useful:
  - making things happen automatically in microcontroller programs
  - can help solve timing problems
- Examples:
  - Catch the pulses from a rotary encoder, so that program never misses a pulse
  - Trying to read a sound sensor that is trying to catch a click
  - Infrared slot sensor (photo-interrupter) trying to catch a coin drop

# Using interrupts

```
attachInterrupt(digitalPinToInterrupt(pin), ISR, mode )
```

```
const int ledPin = 13;  
const int interruptPin = 2;  
volatile byte state = LOW;
```

```
void setup() {  
    pinMode(ledPin, OUTPUT);  
    pinMode(interruptPin, INPUT_PULLUP);  
    attachInterrupt(digitalPinToInterrupt(interruptPin), blink, CHANGE);  
}
```

```
void loop() {  
    digitalWrite(ledPin, state);  
}
```

```
void blink() {  
    state = !state;  
}
```

# Interrupt Service Routines

- ISRs are special kinds of functions that have some unique limitations most other functions do not have. An ISR cannot have any parameters, and they shouldn't return anything.
- Typically global variables are used to pass data between an ISR and the main program. To make sure variables shared between an ISR and the main program are updated correctly, declare them as volatile.

# Example: pelican crossing algorithm

- Given: Traffic light, zebra crossing, button that pedestrian can press
- Algorithm:
  - Always green to cars unless button is pressed
  - If button is pressed, turn red to cars, green to pedestrian
  - Wait 1 minute, then un-press the button, and return to Always green to cars
- We will simplify for 1 built-in LED (**on** indicate safe to cross and stop for cars, **off** indicates green for cars and red to pedestrians) and 1 built-in button



# Example: pelican crossing algorithm (no interrupts)

```
// constants won't change. They're used here to
// set pin numbers:
const int buttonPin = 2;    // the number of the pushbutton pin
const int ledPin = 13;      // the number of the LED pin

// variables will change:
int buttonState = 0;        // variable for reading the pushbutton status

void setup() {
  // initialize the LED pin as an output:
  pinMode(ledPin, OUTPUT);
  // initialize the pushbutton pin as an input:
  pinMode(buttonPin, INPUT);
}

void loop() {
  // read the state of the pushbutton value:
  buttonState = digitalRead(buttonPin);

  // check if the pushbutton is pressed.
  // if it is, the buttonState is HIGH:
  if (buttonState == HIGH) {
    // turn LED on:
    digitalWrite(ledPin, HIGH);
    delay(60000); // 1 minute = 60 seconds
  } else {
    // turn LED off:
    digitalWrite(ledPin, LOW);
  }
}
```

Can you spot the reliability/inflexibility issue?

- What happens if button is pressed during LED on?
- What happens if we would like to let cars go for a while after pedestrian crossed, to avoid traffic jam?

# Example: pelican crossing algorithm (interrupts)

```
// constants won't change. They're used here to
// set pin numbers:
const int buttonPin = 2;    // the number of the pushbutton pin
const int ledPin = 13;     // the number of the LED pin

// variables will change:
volatile byte buttonState = LOW; // variable for reading the pushbutton status

void setup() {
  // initialize the LED pin as an output:
  pinMode(ledPin, OUTPUT);
  // initialize the pushbutton pin as an input:
  pinMode(buttonPin, INPUT_PULLUP);
  → attachInterrupt(digitalPinToInterrupt(buttonPin), GoPedestrian, CHANGE);
}

void loop() {
  // check if the pushbutton is pressed.
  // if it is, the buttonState is HIGH:
  if (buttonState == HIGH) {
    // turn LED on:
    → buttonState = LOW; // Now this is LOW and ready to be turned HIGH again on interrupt
    digitalWrite(ledPin, HIGH);
    delay(60000); // 1 minute = 60 seconds

  } else {
    // turn LED off:
    digitalWrite(ledPin, LOW);
  }
}

void GoPedestrian() {
  buttonState = HIGH;
}
```

Spot the  
difference  
with previous!

This is happening EVERY  
time the button is pressed

# Thank you for your attention!

*Some graphic material used in the course was taken from publicly available online resources that do not contain references to the authors and any restrictions on material reproduction.*

