Introduction to Programming
Computer programming is the process of writing **code**

**Code** is executable program instructions that are interpreted by computers to perform specific actions
The World of Computer Logic

Most computers operate on **binary logic** – that is, they utilize bits to perform complex operations

A **bit** is a basic unit of information that can only be one of two values – 0 or 1

Multiple bits can be interpreted together to form larger units of information; for example, 8 bits form a **byte**

Example of a byte: 0 0 1 0 1 1 0 1

These series of bits can be used to represent numbers
A **number representation** is a writing notation for numbers. In everyday, we typically use the decimal number representation to count. We count 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. We can show larger numbers by adding these digits together; for example, combining 4 and 2 produces 42.

However, computers do not use the decimal number system. They operate on **binary logic** – they only use 0’s and 1’s. This is known as the **binary number system**. It follows the same logic as the decimal number system. As such, it is important to understand how numbers can be represented using binary.
The **binary number system** is a base 2 number representation. Each digit in a binary number is a bit.

<table>
<thead>
<tr>
<th>Binary</th>
<th>Decimal Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>0</td>
</tr>
<tr>
<td>001</td>
<td>1</td>
</tr>
<tr>
<td>010</td>
<td>2</td>
</tr>
<tr>
<td>011</td>
<td>3</td>
</tr>
<tr>
<td>100</td>
<td>4</td>
</tr>
<tr>
<td>101</td>
<td>5</td>
</tr>
<tr>
<td>110</td>
<td>6</td>
</tr>
<tr>
<td>111</td>
<td>7</td>
</tr>
</tbody>
</table>

Table 1: Binary number system

Since all digits are either a “0” or “1”, it can be difficult to interpret what a number is at first glance. It can be useful to compare the number to the decimal number we are more familiar with.

Each digit in a binary number can be read as $2^n$, where $n$ represents the total number of digits in the binary number.

The rightmost digit is known as the least significant bit (LSB) and the leftmost digit is known as the most significant bit (MSB).
Conversion: Binary to Decimal

- To convert from binary to decimal, use a base of 2 and powers beginning with 0 at the LSB, counting upwards to the MSB.
- This technique is the same in decimal systems; it is similar to how “10” is 10 times larger than 1.
- For example, “2” in decimal can be represented in binary as “0010”.

Example: Convert \((0010)_2\) to decimal = \(2^3(0) + 2^2(0) + 2^1(1) + 2^0(0)\)
= \((8)(0) + (4)(0) + (2)(1) + (2)(0)\)
= \((2)_{10}\)
Conversion: Decimal to Binary

- To convert from decimal to binary, divide by 2. If dividing by an even number, carry a 0. If dividing by an odd number, carry a 1.
- Divide the remaining whole number by 2 and follow the same carry rules; repeat until the remaining whole number is 0.
- Build the binary sequence from bottom to top.

Example: Convert $(294)_{10}$ to binary $= (100100110)_{2}$

<table>
<thead>
<tr>
<th>Divide</th>
<th>Carry</th>
</tr>
</thead>
<tbody>
<tr>
<td>294 ÷ 2 = 147</td>
<td>0</td>
</tr>
<tr>
<td>147 ÷ 2 = 73.5</td>
<td>1</td>
</tr>
<tr>
<td>73 ÷ 2 = 36.5</td>
<td>1</td>
</tr>
<tr>
<td>36 ÷ 2 = 18</td>
<td>0</td>
</tr>
<tr>
<td>18 ÷ 2 = 9</td>
<td>0</td>
</tr>
<tr>
<td>9 ÷ 2 = 4.5</td>
<td>1</td>
</tr>
<tr>
<td>4 ÷ 2 = 2</td>
<td>0</td>
</tr>
<tr>
<td>2 ÷ 2 = 1</td>
<td>0</td>
</tr>
<tr>
<td>1 ÷ 2 = 0.5</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 2: Convert decimal to binary
Hexadecimal System

- Sometimes, it is useful to use a larger number representation
- **Hexadecimal** (a base 16 representation) is often used because it can represent a byte with a single character, and can be much quicker to read and understand
- This is a base 16, alphanumeric system which means that each digit can have one of sixteen different values (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F)

<table>
<thead>
<tr>
<th>Hexadecimal</th>
<th>Binary Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
</tr>
<tr>
<td>9</td>
<td>1001</td>
</tr>
<tr>
<td>A</td>
<td>1010</td>
</tr>
<tr>
<td>B</td>
<td>1011</td>
</tr>
<tr>
<td>C</td>
<td>1100</td>
</tr>
<tr>
<td>D</td>
<td>1101</td>
</tr>
<tr>
<td>E</td>
<td>1110</td>
</tr>
<tr>
<td>F</td>
<td>1111</td>
</tr>
</tbody>
</table>

Table 3: Hexadecimal number system
Common Programming Languages

There are many different languages that code can be written in. Each programming language varies in syntax (structure and format) and semantics (meaning).

Common programming languages include:
- C, C#, C++
- Java, JavaScript
- Python

Arduino uses a variation of C/C++

We will focus on learning Arduino C, the programming language for Arduino hardware.
Serial Output

Communicating between the computer and Arduino is done by a serial link.

Creating a serial link allows the computer to receive information stored on the Arduino.

It is important that the computer and the Arduino have matching baud rates.

Baud rates are determined by how many symbols are being transmitted per unit of time.

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

Figure 3: Example of setting up a serial link with a 9600 baud rate.
The **serial monitor** allows users to display text, numbers, and other symbols on the computer screen.

It serves as the primary way of viewing your code’s output.

Can be used to debug code by adding additional serial outputs at key points within the code (such as before and after a series of mathematical operations).

Figure 4: Arduino Serial Monitor display
Variables

Variables are data values typically saved in memory that can be changed based on code execution.

Variables consist of three primary parts – the data type, the variable name, and the variable value.

```
int Var = 42;
```
Data Types

There are many different types of variables that store different kinds of data. The type of data stored within a variable depends on the variable’s data type.

Data types are typically declared before the name of the variable. They define how you intend to use data and let the computer know how much room to set aside in memory. The amount of memory set aside is measured in bytes.

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Keyword</th>
<th>Bytes</th>
<th>Range of Values</th>
<th>Numeric (N) or Alphanumeric (A)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>int</td>
<td>2</td>
<td>-32768 to 32767</td>
<td>N</td>
</tr>
<tr>
<td>Character</td>
<td>char</td>
<td>1</td>
<td>0 to 255</td>
<td>N</td>
</tr>
<tr>
<td>String</td>
<td>String</td>
<td>varies</td>
<td>varies</td>
<td>A</td>
</tr>
<tr>
<td>Boolean</td>
<td>bool</td>
<td>1 bit</td>
<td>0 or 1</td>
<td>N</td>
</tr>
<tr>
<td>Floating</td>
<td>float</td>
<td>4</td>
<td>-3.4 x 10^{38} to 3.4 x 10^{38}</td>
<td>N</td>
</tr>
<tr>
<td>Array</td>
<td>name[]</td>
<td>varies</td>
<td>varies</td>
<td>A or N (depends on array type)</td>
</tr>
</tbody>
</table>

Table 4: Data type specifications
2’s Complement

2’s complement is how negative numbers are stored. The MSB gives the sign of the number. 0 means the number is positive; 1 means the number is negative. To convert from a positive number to a negative number, invert all bits and then add 1.

\[
28 = 0001
grightarrow 1110
grightarrow 1110 0100
\]

\[
28 = 0001 1100
\]

Invert -> 1110 0011
Add 1 -> 1110 0100

-28 = 1110 0100
Operators are one of the most common ways of manipulating the value of a variable. They represent a functional operation such as adding or subtracting.

Common types of operators include:

- Arithmetic
- Logical
- Conditional
- Bitwise
- Comparison
Arithmetic operators are mathematical functions that take two operands, perform a calculation, and provide a result.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Adds two operands</td>
<td>A + B = C</td>
</tr>
<tr>
<td>-</td>
<td>Subtracts second operand from first</td>
<td>A − B = C</td>
</tr>
<tr>
<td>*</td>
<td>Multiplies operands</td>
<td>A * B = C</td>
</tr>
<tr>
<td>/</td>
<td>Divides dividend by divisor</td>
<td>B / A = C</td>
</tr>
<tr>
<td>%</td>
<td>Modulus operator: Remainder of quotient</td>
<td>B % C = D</td>
</tr>
<tr>
<td>++</td>
<td>Increments integer by 1</td>
<td>++A = A + 1</td>
</tr>
<tr>
<td>--</td>
<td>Decrements integer by 1</td>
<td>--A = A - 1</td>
</tr>
</tbody>
</table>

Table 5: Arithmetic operators
Logical Operators

- Logical operators use the laws of Boolean logic to compare two conditions and provide one result if true and another if false.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;&amp;</td>
<td>AND – If both operands are nonzero, the condition is true; otherwise, it is false</td>
<td>If A = 1 and B = 0, then A &amp;&amp; B = false</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>!</td>
<td>NOT – If a condition is true, then !condition is false</td>
<td>If A</td>
</tr>
</tbody>
</table>

Table 6: Logical operators
Conditional Operators

- A conditional operator will return one value if a condition is true and another if a condition is false.

- Most operators are conditional by nature because they compare entities and then proceed one way if a particular condition is met and another way if it is not.

Example:

```cpp
if (expression1) a = a1;  // Test this first
else if (expression2)  a = a2;  // If above was false, test this
else a = a3;               // If above was also false, do this
```
Bitwise Operators

- Bitwise operators are similar to logical operators, except they compare individual bits instead of the entire operand.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>Bitwise AND – If both bits are nonzero, the condition is true; otherwise, it is false</td>
<td>If a = 1 and b = 0, then a &amp; b = false</td>
</tr>
<tr>
<td></td>
<td>Bitwise OR – If either bit is nonzero, the condition is true; otherwise, it is false</td>
<td>If a = 1 and b = 0, then a</td>
</tr>
<tr>
<td>^</td>
<td>Bitwise XOR – If both bits are different, the condition is true; otherwise, false</td>
<td>If a = 1 and b = 1, then a ^ b = false</td>
</tr>
<tr>
<td>~</td>
<td>Bitwise NOT – Inverts all bits of a number</td>
<td>If D = 0110, then ~D = 1001</td>
</tr>
</tbody>
</table>

Table 7: Bitwise operators
Comparison Operators

- Comparison operators are used to compare two operands
- These are typically found nested within a function

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>==</td>
<td>Equal to</td>
<td>$x == y$ (x is equal to y)</td>
</tr>
<tr>
<td>!=</td>
<td>Not equal to</td>
<td>$x != y$ (x is not equal to y)</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than</td>
<td>$x &lt; y$ (x is less than y)</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than</td>
<td>$x &gt; y$ (x is greater than y)</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal to</td>
<td>$x &lt;= y$ (x is less than or equal to y)</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than or equal to</td>
<td>$x &gt; y$ (x is greater than or equal to y)</td>
</tr>
</tbody>
</table>

Table 8: Comparison operators
A function is a code segment in a program that contains instructions the computer will use to perform a task.

To define a function:

- Specify a data type for the return
- Provide a unique name followed by a set of parenthesis
- After the parenthesis, put the instructions that need to be executed inside a set of brackets

```
void setup () {
  <insert instructions>
}
```

Figure 5: Structure of a void function
Void

- **Void** is a special data type used for declaring a function that is not expected to return any information.
- Arduino uses two void functions to get you started; the main setup runs one time when the program begins, followed by a loop that runs continuously thereafter.

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

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

Figure 6: Image of the Arduino start screen
Conditional Statements: If/Else

- An if statement proceeds one way if a condition is met and another way if it is not met.

```cpp
void setup() {
    Serial.begin(9600);   
}
void loop() {
    int A = 15;
    int B = 10;
    if (A >= B) {
        Serial.println (A – B);
    }
    else if ( (A < B) && (B != 0) ) {
        Serial.println (B + A);
    }
}
```

Figure 7: In this example, if A is greater than or equal to B, then A – B will print. Otherwise, if B is not zero then B + A will print.

```cpp
void setup() {
    Serial.begin(9600);   
}
void loop() {
    int A = 5;
    int B = 20;
    if (B >= A) {
        Serial.println (B - A);
    }
    else if ( (A < B) && (B != 0) ) {
        Serial.println (B + A);
    }
}
```

Figure 8: In this example, if B is greater than or equal to A, then B – A will print. Otherwise, if B is not zero then B + A will print.
Loops

- A **loop** is useful when repetitive operations are being performed because the instructions will repeat until a particular condition is met.

  - Some loop commands **pretest**, which means they test for a condition at the beginning of the loop.

  - Other loop commands **posttest**, which means they test for a condition at the end of the loop.
A **for loop** executes repeatedly and increments a counter variable until the conditional statement is no longer true (pretest condition).

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

void loop() {
  for (int i = 0; i <= 15; i++)  {
    Serial.println(i);
  }
}
```

**Output** → 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

Figure 9: A for loop that counts from 0 to 15. The variable i starts at 0 and increments every loop until the condition stated in the loop is no longer valid.
A **while loop** will only run when the conditional statement is true (pretest condition)

```cpp
while (carrier < 0) { Serial.print (carrier++); }
```

**Figure 10**: A while loop that counts from 0 to 19
A do/while loop only checks for a condition after some other action has occurred (posttest condition)

```cpp
int x = 0;
void setup() Serial.begin(9600);
void loop() {
    do {
        Serial.print("Waiting…");
        Serial.println(x++);
    } while (x < 10);

    Serial.println("done");
    while(1) {};
}
```

Figure 11: A do/while loop that counts from 0 to 9
Leaving Comments

- If you can fit your comment on one line, then simply type two backslashes followed by your text.
- If you need more room, then use a backslash and asterisk combination to comment over multiple lines.
- You can highlight a block of information and press ctrl + backslash to comment the entire block.

// Leave a one-line comment like this

/* Use as many lines as needed in order to provide enough information for someone else to understand your code */
Figure 12: This an example of good commenting. Notice the comments explaining each step and the use of white space to help the user understand the code.
Bad Comments

Figure 13: This an example of bad commenting. The lack of comments make the code difficult for a user to follow and understand the purpose of this function.
Version Control

• While developing software, it is important to track the changes made within your code. This is accomplished by version control.

• **Version Control** is the practice of managing and recording changes to software or other frequently changed documents.

• Without version control, changes are more frequently lost, miscommunicated, or duplicated.

• Version control helps facilitate effective communication in development teams.
In large software projects, version control is often handled by a version control system developed by a third party. A version control system (VCS) is a software program that creates and tracks multiple versions of a codebase on a server.

Some examples of VCS software include GitHub, Subversion, and BitKeeper.

Figure 14: Example of a GitHub repository for a large software program
Working Copies and Branches

In development, it is often useful for multiple programmers to edit software at the same time.

The initial copy of the software that the programmers begin with is called the **working copy** or **baseline**. The edited software that each programmer creates is known as a **branch**. Multiple branches may exist at the same time. A branch may become the working copy when the team agrees to shift to the new branch for further development work.

When a programmer is finished with his or her changes to the branch, they may compile a **change list** which summarizes all changes made to the software.
File Tags

In version control, a **file tag** is a series of numbers or letters that designate the version of an existing document or software. File tags often include a version number that is incremented with any changes or the date the file was modified.

The system for updating the file tag is defined during project creation and is followed throughout the lifetime of the project.

Figure 15: Example of a file tag system for iterations of a document
Function Version History

• It is useful to track changes of a function. This can be done by implementing a change log inside the code.

```c
String MakeFileName() {
  /******** Function MakeFileName ******************************
  
  * Creates a filename using the date and time returned from the clock on the Adafruit
  * GPS shield. The SD library is limited to FAT file structure and 8.3 format filenames.
  * The filename is returned as the function value and takes the following form:
  *   DDDHHMMSS.txt
  
  * Note that the Adafruit GPS unit must be fully functional for this function to
  * return a rational filename. However, the Adafruit GPS unit battery backup keeps
  * the time current and a GPS lock is not necessary for a correct filename.
  * 
  * Note: This version (v02a) will return a long filename as YMDDDHHMMSS.txt
  
  // Version history:
  
  * v01a-TGG-190316:
  * Initial version of function
  
  * v02a-TGG-190517:
  * Using SdFat to handle large SD volumes. The library can also handle long filenames.
  * So add the year and month to the filename.
  
  ******************************************************/
```

Figure 16: Example of a change log for a function. After the description of a function, include version history. This will tell a user when any changes were made and what those changes were.
Sketch Version History

- Like functions, sketch changes should be documented. This should be done in the beginning of the sketch.

```
* Version history:

* v0lg-TGG-190307:
* This is the initial version of this code. Includes reading the Adafruit GPS via Serial1, keeping the last NMEA sentence in a string variable, identifying the NMEA sentence type, if the NMEA is a GGA or RMA then sending the sentence to the MTT4BT and parsing it.
* Finally, write to the serial monitor some of the parsed GPS information.

* NOTE: Need to make certain that one is sending serial data to the MTT4BT (i.e. PORTASerial, and PORTBSerial) at a baud rate much higher than reading form the GPS unit. Otherwise, characters will be lost during the GPS read. In this version, both MTT4BT ports are set to 19200 baud and the GPS is at 9600 baud. If you need to change these baud rates, then the GPS and/or the MTT4BT needs to be reconfigured with the new baud rate.

* v02a-TGG-190315:
* Added MakeGPSStatus function to produce a GPS status record for logging. Test writing the GPS status record to PORTA on XTM. Changed how the startup message is composed and printed. This version is fully functional.

* v02b-TGG-190317:
* Includes basic code for writing log data to the SD card. Add MakeFileName and MakeTimeStamp functions. Note that filename appears to be limits to a 8.3 format and the filename must be a char array rather than a String object.

* v02c-TGG-190317:
```

Figure 17: Example of version history for a sketch. Every time the sketch is worked on, a new section is added describing the changes that were made.
Troubleshooting Your Code

- Check syntax
- Check punctuation: semicolons, brackets and parenthesis must be placed correctly
- Ensure correct placement of conditional statements and loops
- Use correct data types
- Make sure global and local variables are accessible to the appropriate functions
Good Bookkeeping

- There is typically more than one way of writing a program to accomplish a particular task; as such, programmers tend to have their own styles

- It is good practice to write your code in a manner that is easy for you to navigate through and clear enough for others to understand

- Practice taking advantage of whitespace, utilize control characters, identify variables and functions using descriptive names, and always comment your code

- Work to establish good habits while you are learning
References

- For a list of Arduino keywords, visit https://www.arduino.cc/reference/en/
- C++ in 24 hours, Sams Teach Yourself (6th Edition)
- Images: